Implementierung und Vergleich zweier Skelettierungsalgorithmen für Baum-Punktwolken

Jannis Schaake

Bachelorarbeit

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Studiengang B. Sc. Mediendesigninformatik

13. Juli 2023

Autor

Jannis Schaake
******
******
******
E-Mail: jannis.schaake@stud.hs-hannover.de

Erstprüfer

Prof. Dr. Adrian Pigors
Hochschule Hannover
Fakultät IV, Abteilung Informatik
Ricklinger Stadtweg 120
30459 Hannover

Zweitprüfer

Ruben Hohndorf
Graswald GmbH
Berckhusenstraße 89
30625 Hannover

Selbständigkeitserklärung

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

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieser Bachelorarbeit werden zwei Skelettierungsalgorithmen für Baum-Punktwolken implementiert und miteinander verglichen. Darüber hinaus werden zwei Erweiterungen für einen der beiden Algorithmen präsentiert.

In diesem Kapitel werden zunächst die Hintergründe zur Skelettierung von Baum-Punktwolken beleuchtet, um darauf aufbauend die Motivation für die Untersuchung dieser Thematik zu erörtern (**Abschnitt 1.1**). Anschließend werden Probleme und Herausforderungen betrachtet, die für die Skelettierung zu berücksichtigen sind, und es wird das übergeordnete Ziel dieser Arbeit definiert (**Abschnitt 1.2**). Nach einer Einordnung der vorliegenden Arbeit in den Forschungskontext (**Abschnitt 1.3**) folgt ein Überblick über den Aufbau der Arbeit (**Abschnitt 1.4**).

1.1 Hintergrund und Motivation

Um einen umfassenden Einblick in die Hintergründe und die Relevanz der Thematik zu erhalten, wird im Folgenden zunächst eine grundlegende Einführung gegeben, die eine Herleitung der theoretischen Konzepte liefert und damit einen fundierten Ausgangspunkt für die Skelettierung von Baum-Punktwolken darstellt.

Virtuelle Welten haben in den letzten Jahrzehnten in einer Vielzahl von Anwendungen eine immer größere Bedeutung erlangt. Ob in der Spieleentwicklung, Filmproduktion oder anderen Produkten der Unterhaltungsindustrie, in Simulationen oder Visualisierungen, im Bildungssektor mit Lern- und Trainingsprogrammen oder in medizinischen Anwendungen: Virtuelle Welten sind heute allgegenwärtig und nicht mehr aus unserem Alltag wegzudenken. Dies begründet sich nicht nur in der wachsenden Nachfrage nach realistischen und immersiven Anwendungen, sondern auch in den kontinuierlichen Fortschritten der Computertechnik. Die zunehmende Rechenleistung hat zu erheblichen Verbesserungen in der Darstellung von und Interaktion mit virtuellen Welten geführt, insbesondere in Echtzeit-Anwendungen. Früher war dies aufgrund begrenzter Rechenkapazitäten und Speicherressourcen nur eingeschränkt realisierbar. Durch die Steigerung der Rechenleistung ergeben sich zudem neue Perspektiven in der Softwareentwicklung. Während in der Vergangenheit die Limitierungen der Hardware bedeutende Faktoren darstellten, sind diese heutzutage weniger präsent. Stattdessen liegt das Hauptaugenmerk auf der effizienten Bewältigung komplexer Software-Probleme [AFM15].

Virtuelle Welten, die eine authentische Darstellung der realen Welt anstreben, erfordern in der Regel verwandte Objekte aus der Realität. Wichtige Bestandteile vieler virtueller Welten sind unter anderem Darstellungen von Vegetation und Pflanzen. Diese können in unterschiedlichen Szenarien auftreten, sowohl in Innenräumen als auch in der freien Natur. Anwendungsgebiete reichen von der Spieleindustrie über Filmproduktionen hin zu Produkt- oder Architekturvisualisierungen. Grundsätz-

lich sind Pflanzen ein wesentlicher Bestandteil, um Szenen lebhafter erscheinen zu lassen. Livny et al. [Liv+10] betonen, dass virtuelle Welten, die keine Vegetation aufweisen, oft leblos und künstlich wirken. Dabei spielt es keine Rolle, ob diese im Fokus steht oder nur im Hintergrund erscheint. Natürlich aussehende Pflanzen können im Allgemeinen einen erheblichen Einfluss auf die Wahrnehmung einer Szene haben (s. Abbildung 1.1).

Abbildung 1.1: Pflanzen in einer virtuellen Umgebung (Quelle: [Eas23])

Die Erzeugung solcher virtuellen Vegetationsszenen stellt aufgrund der komplexen und unregelmäßigen geometrischen Formen von Pflanzen eine besondere Herausforderung dar. Klassische Modellierungsverfahren, bei denen 3D-Objekte von einem Designer manuell erstellt und bearbeitet werden, bieten zwar eine nahezu uneingeschränkte Kontrolle während des Modellierungsprozesses, bedingen allerdings auch einen erheblichen Zeitaufwand und erfordern eine gewisse Expertise des Designers zum Erreichen eines akzeptablen Ergebnisses. Um den Prozess der Modellierung von Pflanzen zu vereinfachen, existieren automatisierte Verfahren, die eine computergesteuerte Generierung solcher Objekte ermöglichen. Insbesondere bei der Erzeugung einer breiten Vielfalt von Objekten gleicher oder auch verschiedener Pflanzenarten erweisen sich automatisierte Verfahren als äußerst hilfreich, da sie nicht den Aufwand der manuellen Modellierung erfordern.

Bei der automatischen Generierung von Pflanzen kann im Wesentlichen zwischen zwei Kategorien von Verfahren unterschieden werden: Die *prozedurale Generierung virtueller Pflanzen* und die *Rekonstruktion von Pflanzen aus der realen Welt* [Liv+10]. Bei der *prozeduralen Generierung* werden Pflanzen auf Basis von Regeln, die eine vereinfachte Beschreibung der Vorgänge in der Natur darstellen sollen, vollständig virtuell modelliert [AK14]. Ein Beispiel hierfür sind *L-Systeme*, die

erstmals im Jahr 1968 von Lindenmayer vorgestellt wurden [Lin68]. Lindenmayer erkannte, dass der Wachstumsprozess von Pflanzen zu einer Eigenschaft der Selbstähnlichkeit der resultierenden Verzweigungsstruktur führt, wie sie auch in Fraktalen zu finden ist. Basierend auf dieser Eigenschaft können Regeln abgeleitet werden, die das Wachstumsverhalten bestimmter Pflanzenarten beschreiben und deren geometrische Modellierung ermöglichen [PL90]. Ein Vorteil von prozeduralen Verfahren ist, dass diese schnell überzeugende Modelle für bestimmte Pflanzenarten generieren können, um damit eine virtuelle Welt zu erzeugen. Ferner wird hierfür keine zusätzliche Hardware benötigt, da die Pflanzen vollständig virtuell konstruiert werden. Ein wesentlicher Nachteil hingegen ist, dass für die Definition der Regeln häufig eine umfangreiche Anzahl von Parametern eingestellt werden muss. Laut Aiteanu und Klein [AK14] sei es zudem aufgrund von Limitierungen dieser Parameter schwierig, reale Pflanzen individuell zu reproduzieren. Darüber hinaus können kleine Änderungen an Parametereinstellungen zu erheblichen Änderungen des visuellen Gesamtergebnisses führen. Dies lässt sich durch die starke Abhängigkeit zwischen den Regeln und dem generierten Modell begründen. Weiterführend wurden Verfahren entwickelt, die das Problem der manuellen Eingabe von Parametern durch Skizzieren der groben Aststruktur [Che+08] oder Silhouette [Wit+09] von Pflanzen oder Bäumen lösen. Bei diesen Verfahren ist die Qualität des Ergebnisses beschränkt durch die Form der Eingabe und abhängig von der Genauigkeit der Nutzerskizzen.

In der zweiten Kategorie, der *Rekonstruktion realer Pflanzen*, werden Pflanzen aus der Natur auf Basis von Daten, die mit Hilfe unterschiedlicher Messtechniken erfasst wurden, digital rekonstruiert. Solche Verfahren haben durch die erhöhte Nachfrage nach realistischen Modellen an Bedeutung gewonnen [XGC07]. Der wesentliche Vorteil besteht darin, dass Pflanzen realitätsnah abgebildet werden, da die Eingabedaten zur Generierung direkt aus der Natur stammen und die morphologischen[1] Merkmale, also die äußere Gestalt, der erzeugten Modelle damit automatisch realen Pflanzen entsprechen [Wei+22a]. Dabei können sowohl fiktive (aber realitätsnahe) virtuelle Welten erzeugt werden als auch reale Abbilder existierender Orte. So finden diese Verfahren beispielsweise Anwendung in der Visualisierung von Bauplänen, bei denen existierende Objekte zusammen mit geplanten Strukturen dargestellt werden [AK14]. Die Qualität der Verfahren wird maßgeblich durch die Reproduzierbarkeit der gescannten Daten und die Genauigkeit des Reproduktionsprozesses bestimmt. Eine fehlerhafte oder inakkurate Nachbildung kann den Betrachter stören, insbesondere dann, wenn er mit der potenziell existierenden Umgebung vertraut ist [XGC07].

Zur Erfassung der natürlichen Daten können unterschiedliche Messtechniken zum Einsatz kommen. Einige Verfahren erlauben es, Pflanzen basierend auf Fotografien zu reproduzieren. Beispiele hierfür finden sich in den Arbeiten [NFD07], [MMD04], [Shl+01] oder [Zen+07]. Unter diesen Ansätzen existieren Modellierungs- und Rendering-Verfahren, die laut Xu et al. (2007) aufgrund ihrer Effizienz in Echtzeitsystemen bevorzugt wurden [XGC07]. Obwohl das Anfertigen von Fotografien zum Sammeln der Realdaten vergleichsweise unkompliziert ist, haben diese fotografiegestützten Verfahren einen entscheidenden Nachteil: Durch den eingeschränkten Informationsgehalt von 2D-Bildern ist es nicht immer möglich, ein exaktes und repräsentatives Ergebnis zu erzielen. Häufig gehen relevante Informationen durch Verdeckungen im Bild verloren. Darüber hinaus werden in Bildern lediglich Farbinformationen gespeichert, wohingegen räumliche Informationen nicht gegeben sind. Dies kann eine nachträgliche Interpretation der erfassten Objekte erschweren. Livny et al. wei-

[1] „Morphologie“ ist in der Biologie ein Begriff für die äußere Gestalt von Lebewesen [Spe23]

sen darauf hin, dass die Genauigkeit und Effizienz dieser Modellierungsverfahren grundsätzlich durch die Qualität und die Dichte der 3D-Punkte limitiert ist, die aus den Bildern rekonstruiert werden können [Liv+10].

Eine wesentlich bessere Grundlage bieten hingegen Laserscanning-Technologien wie LiDAR. Die NOAA (2023), die nationale Ozean- und Atmosphären-Behörde der USA, erklärt LiDAR wie im Folgenden:

> Lidar, which stands for Light Detection and Ranging, is a remote sensing method that uses light in the form of a pulsed laser to measure ranges (variable distances) to the Earth. These light pulses [...] generate precise, three-dimensional information about the shape of the Earth and its surface characteristics. [NOA23]

Hierbei bezieht sich die NOAA auf LiDAR im Kontext der geographischen Datenerfassung und Geländevermessung. Es sei jedoch darauf hingewiesen, dass LiDAR auch in anderen Bereichen Anwendung findet, um die geometrische Form von Objekten, wie etwa Pflanzen, zu erfassen. Dabei beschränken sich die Anwendungsgebiete nicht nur auf die reine Visualisierung für Stadtplanung oder Videospiele. Laserscanning-Technologien spielen auch eine wichtige Rolle in der Botanik, um beispielsweise die Biomasse individueller Pflanzen abzuschätzen, die Beziehung zwischen Vegetation und elektromagnetischer Strahlung in Strahlungstransfermodellen zu untersuchen oder um Verdunstung in ökophysiologischen Modellen zu quantifizieren [Wei+22a].

Folgende Technologien sind hinsichtlich ihrer Form der Messung zu unterscheiden: Stationäre Messsysteme in Bodennähe werden als TLS (*terrestrial laser scanning*) bezeichnet. Für Messungen aus der Luft kann ALS (*airborne laser scanning*) oder ULS (*UAV-borne laser scanning*) verwendet werden [Wei+22a]. Wie die Ergebnisse von Weiser et al. [Wei+22a] zeigen, ist es für eine umfassendere Abdeckung des gescannten Objekts sinnvoll, mehrere Messungen aus unterschiedlichen Standpunkten durchzuführen und die einzelnen Ergebnisse in einem weiteren Schritt, der als „Registrierung“ bezeichnet wird [BL08], zusammenzuführen.

Die Ausgabe eines LiDAR-Scanners ist typischerweise eine Punktwolke (s. Abbildung 1.2 (b)), die eine Menge von Punkten im Raum (oft mehrere Millionen [BL08]), mit weiteren Informationen zur Oberfläche des gemessenen Objekts, darstellt [Wei+22a]. Der wesentliche Vorteil gegenüber Fotografien besteht darin, dass die erzeugten Ausgaben direkt räumliche Informationen der Objekte enthalten. Dies bietet einen entscheidenden Vorteil bei der Rekonstruktion der geometrischen Formen. Da sich diese Arbeit mit der Rekonstruktion von Baumstrukturen befasst, beziehen sich die folgenden Ausführungen auf Punktwolken von Pflanzen, die eine baumartige Struktur aufweisen.

Ein fundamentaler Schritt zur Rekonstruktion der geometrischen Form von Bäumen aus Punktwolken ist die Extraktion eines Skeletts (auch Skelettierung, s. Abbildung 1.2 (c)) [Liv+10]. Bucksch et al. (2009) definieren ein Skelett als eindimensionale Beschreibung der Objektstruktur, wobei dieses als Kurven, geordnete Menge an Punkten oder Graphen repräsentiert sein kann [BLM09]. Es enthält topologische und, da es räumlich im Zentrum des Objekts eingebettet ist, geometrische Informationen wie Astpositionen oder Verzweigungswinkel [LBW17]. Zur Spezifikation der zentrierten Eigenschaft des Skeletts wird häufig die Definition einer *medialen Achse* herangezogen, die im Verlauf der Arbeit näher erläutert wird.

Der entscheidende Nutzen eines Skeletts zeigt sich darin, dass sich das Volumen oder die Oberfläche eines Baums auf einfache Weise rekonstruieren lässt (s. Abbildung 1.2 (d)). Darüber hinaus eröffnet die Skelettierung eine breite Palette weiterer Einsatzmöglichkeiten, von skelettbasierter Animation und physikalischen Simulationen über eine vereinfachte Analyse komplexer Baumstrukturen und Segmentierung des Baums in Komponenten bis hin zur virtuellen Navigation [CSM07].

Insgesamt lässt sich feststellen, dass die Skelettierung von Baum-Punktwolken eine entscheidende Rolle bei der Rekonstruktion realer Pflanzen spielt. Die Nachbildung von Bäumen aus der Natur liefert in der Regel authentischere Ergebnisse als prozedurale Verfahren, was durch die wachsende Nachfrage nach realistischen Modellen in virtuellen Welten von großer Bedeutung ist.

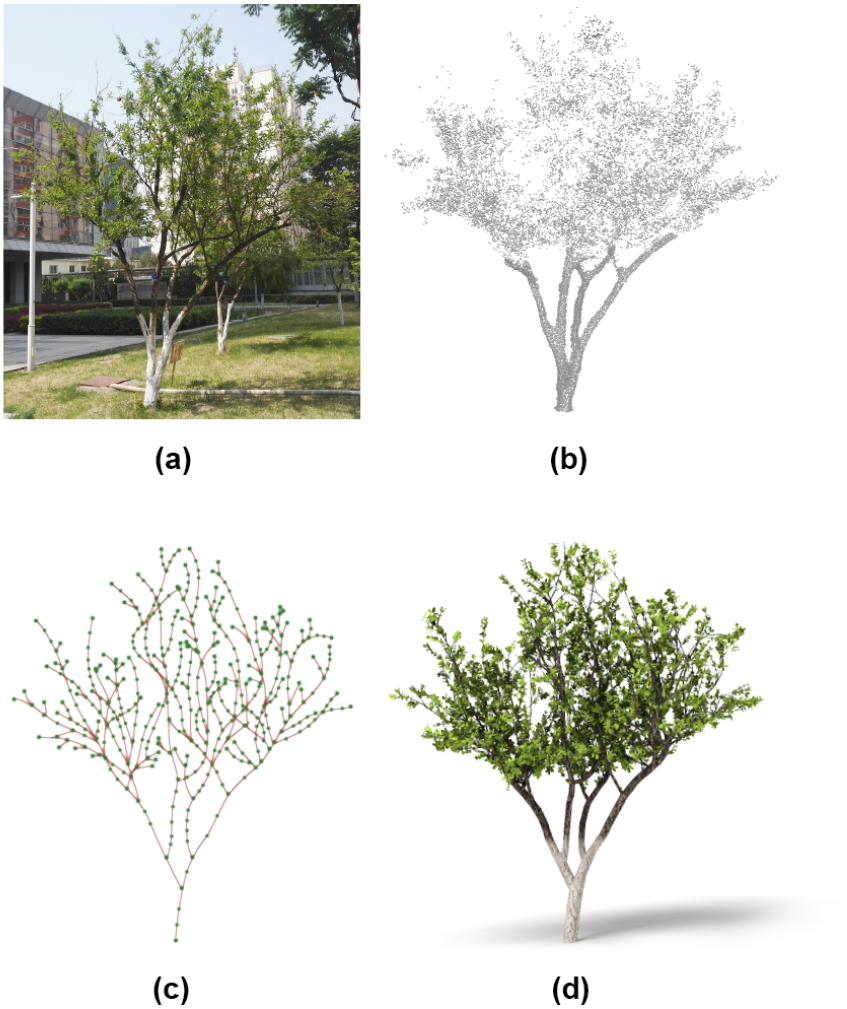

Abbildung 1.2: Rekonstruktion eines realen Baums. Realer Baum (a); Punktwolke (b); Skelett (c); Rekonstruierter Baum (d) (Quelle: (a), (b) und (d) aus [Liu+21])

1.2 Probleme und Zielstellung

Wie im vorherigen Abschnitt beschrieben, bildet die Skelettierung von Baum-Punktwolken eine wichtige Grundlage, um die geometrische Form natürlicher Bäume rekonstruieren zu können. Die Aufgabe, aus vorliegenden Punktdaten ein Skelett abzuleiten, ist jedoch nicht trivial: Punktwolken beinhalten zwar Positionen einzelner Punkte, die das signifikante Objekt beschreiben, es besteht jedoch keine Konnektivität zwischen ihnen. Während es für Menschen oft einfach ist, durch visuelle Inspektion Strukturen in der Punktwolke zu erkennen und zu interpretieren, stellt dies eine komplexe Aufgabe für Computer dar.

Darüber hinaus stellen die Limitierungen von Laserscanning-Technologien und die damit verbundenen Ungenauigkeiten der resultierenden Punktwolken einige algorithmische Herausforderungen an die Skelettierung. Bucksch et al. (2009) führen in ihrem Paper einige dieser Probleme an. Zunächst ist zu beachten, dass die sphärische Form der Datenerfassung durch TLS und die Überlagerung mehrerer Messungen zu einer inhomogenen Punktdichte in der Punktwolke führen können. Rauschen und systematische Fehler in der Punktwolke erschweren zudem die Extraktion eines Skeletts. Ferner sind die Daten durch Überdeckungen oft unvollständig [BLM09]. Laut Xu et al. (2007) ist es in bestimmten Anwendungsfällen nicht praktikabel, mehrere Scans eines Baums anzufertigen, was dieses Problem verstärkt. Zuletzt haben die Überdeckungen und die eingeschränkte Auflösung der Laserscanner zur Folge, dass kleine Zweige innerhalb der Baumkrone fehlen oder unzureichend repräsentiert sind. Dies kann die Rekonstruktion dieser feinen Strukturen aus der Punktwolke erheblich erschweren [XGC07]. Einige Skelettierungsalgorithmen haben daher den Fokus darauf gesetzt, lediglich die signifikanten Strukturen zu extrahieren, während feinere Strukturen prozedural synthetisiert werden [Liv+10].

Die Probleme werden weiterhin durch die Tatsache verstärkt, dass Bäume eine vergleichsweise komplexe Makrostruktur aufweisen (im Gegensatz z. B. zu Gebäuden, die aus eher einfachen geometrischen Formen bestehen). Hinzu kommt, dass Bäume nicht stationär sind und sich während des Scanprozesses im Wind bewegen können, was die Qualität der Ergebnisse zusätzlich beeinträchtigt [XGC07].

Abschließend sei anzumerken, dass mit Hilfe von LiDAR gescannte Punktwolken neben den obigen Problemen zusätzliche Herausforderungen mit sich bringen, die bei der Skelettierung zu berücksichtigen sind. Durch die Funktionsweise von Laserscanning-Technologien ist zu erwarten, dass die Punkte der Punktwolke, an denen der Laserstrahl reflektiert wurde, auf der Oberfläche der Äste liegen [GP04]. Diese Gegebenheit stellt eine Schwierigkeit dar, da das Skelett idealerweise entlang der medialen Achse konstruiert werden soll. Darüber hinaus ist zu beachten, dass Laserscanner in der Regel keine automatische Segmentierung der gescannten Objekte durchführen [Liv+10]. Dies kann dazu führen, dass ein einziger Scan mehrere Bäume oder andere Objekte enthält, die nicht relevant für die Extraktion des Skeletts sind oder diese sogar stören können.

Das Ziel dieser Arbeit besteht aus drei Teilen. Zunächst sollen zwei Skelettierungsalgorithmen für Baum-Punktwolken ausgewählt und in der Programmiersprache *Python* unter Berücksichtigung der beschriebenen Probleme implementiert werden. Dabei ist darauf zu achten, dass die von den Skelettierungsalgorithmen erzeugten Skelette die wesentlichen Baumstrukturen (Stamm, Äste und Zweige) der in den Punktwolken dargestellten Bäume hinreichend genau repräsentieren. Für die Berechnungsdauer beider Algorithmen wird eine Obergrenze von 30 Minuten festgelegt. Im zweiten Teil sollen

die Algorithmen hinsichtlich ihrer Genauigkeit und Laufzeit untersucht und verglichen werden, um Rückschlüsse auf die Qualität der Algorithmen ziehen zu können. Im letzten Teil soll einer der beiden Algorithmen erweitert werden, sodass zusätzliche Informationen über den Baum ermittelt und gespeichert werden. Für die Implementierung kann eine existierende Codebasis als Grundlage verwendet werden, sofern ihre Lizenz eine kommerzielle Weiterverwendung und Veröffentlichung des abgeleiteten Programms erlaubt.

1.3 Einordnung

Diese Arbeit entstand in Kooperation mit der Firma *Graswald GmbH*. Seit ihrer Gründung im Jahr 2021 hat sich Graswald auf die Produktion hochqualitativer und fotorealistischer 3D-Modelle von Pflanzen unterschiedlicher Art spezialisiert. Um diese möglichst detailgetreu zu modellieren, werden von dem Unternehmen Verfahren entwickelt, die unterschiedliche Techniken wie *LiDAR*, *Photometric Stereo*, *Structure-From-Motion* und *Neural Radiance Fields* kombinieren, um Pflanzen zu digitalisieren. Die genannten Techniken sind nicht Gegenstand dieser Arbeit und werden daher nicht weiter beleuchtet.

Graswald stellt auf ihrer Website eine Sammlung mit über 3.000 individuell angefertigten und texturierten 3D-Modellen von Pflanzen aus unterschiedlichen Ökosystemen zur Verfügung [Gra23a]. Die Modelle werden weltweit von mehr als 75.000 Nutzern, Künstlern und renommierten Produktionsfirmen für den Einsatz in Film- und Fernsehproduktionen, Videospielen und Architekturvisualisierungen verwendet [Gra23a]. Darüber hinaus entwickelt Graswald Werkzeuge und Integrationen für 3D-Programme und Game-Engines, wie *Autodesk Maya*, *Cinema 4D*, *3ds Max*, *Blender* und die *Unreal Engine*, die es Nutzern ermöglichen sollen, auf Basis der bereitgestellten 3D-Modelle effizient beeindruckende und realistische Vegetationsszenen zu erstellen. Ein bekanntes Werkzeug ist das Add-on *Gscatter* für Blender, das Benutzern Funktionen zur Generierung und Streuung von Gras und anderen Pflanzen auf einem Terrain bietet [Gra23b].

Bisherige Arbeiten des Unternehmens fokussierten sich auf die Modellierung kleinerer Pflanzenarten, wie Gräser, Blumen und Sträucher. Dies ist unter anderem auf die begrenzte Skalierbarkeit der verwendeten Technologien zurückzuführen. Um Landschaften in einem größeren Maßstab abzubilden, soll es zukünftig auch möglich sein, große Pflanzen und Bäume zu modellieren. Hierfür wird ein neues Verfahren entwickelt, das auf Basis von Ausgabedaten eines LiDAR-Scans (Punktwolken) die geometrische Form von natürlichen Bäumen automatisch nachbilden kann. Die Rekonstruktion des Baum-Skeletts aus den Scandaten, womit sich diese Arbeit auseinandersetzt, stellt dabei eine wichtige Grundlage dar. Weiterführende Ziele von Graswald sind unter anderem, mit Hilfe von künstlicher Intelligenz die Oberfläche der Bäume entlang der Baumskelette synthetisch zu konstruieren und aus den Skeletten Regeln abzuleiten, aus denen ohne großen Arbeitsaufwand mehrere Varianten der gleichen Baumart generiert werden können.

1.4 Aufbau der Arbeit

Im Folgenden wird ein Überblick über die Inhalte dieser Arbeit gegeben. **Kapitel 2** bietet einen Einblick in klassische und aktuelle Verfahren zur Skelettierung von Baum-Punktwolken. Basierend auf verschiedenen Auswahlkriterien wurden ein graphenbasierter und ein voxelbasierter Skelettierungsalgorithmus ausgewählt, deren grundlegende Funktionsweise erläutert wird. **Kapitel 3** beschäftigt sich mit der Implementierung der beiden Algorithmen und veranschaulicht die praktische Umsetzung anhand relevanter Codeausschnitte. In **Kapitel 4** werden die implementierten Algorithmen anhand verschiedener Datensätze ausgewertet. Dabei erfolgt ein detaillierter Vergleich hinsichtlich ihrer Genauigkeit und Effizienz. Abschließend werden in **Kapitel 5** zwei Erweiterungen des graphenbasierten Skelettierungsalgorithmus vorgestellt, die im Rahmen dieser Arbeit entwickelt wurden.

2 Grundlagen zur Skelettierung

In diesem Kapitel werden die theoretischen Grundlagen zur Skelettierung von Baum-Punktwolken behandelt. Dabei wird zunächst ein Einblick in die jüngsten Entwicklungen und den Stand der Technik gegeben, indem aktuelle Verfahren vorgestellt werden (**Abschnitt 2.1**). Im Anschluss erfolgt eine begründete Auswahl von zwei spezifischen Skelettierungsalgorithmen für Baum-Punktwolken, die im Rahmen dieser Arbeit implementiert werden sollen (**Abschnitt 2.2**). Es schließt sich eine Erläuterung der grundlegenden Konzepte und Funktionsweisen der beiden ausgewählten Skelettierungsalgorithmen an, um ein grundlegendes Verständnis für ihre Arbeitsweise zu vermitteln (**Abschnitte 2.3, 2.4**).

2.1 Stand der Technik

In den letzten Jahrzehnten wurden zahlreiche Algorithmen zur Skelettierung von Baum-Punktwolken entwickelt. Im Folgenden wird eine kleine Auswahl von aktuellen Verfahren aus verschiedenen Klassen vorgestellt.

Einige Verfahren basieren auf der Definition einer *medialen Achse* (*medial axis*) als Skelett. Der Begriff der medialen Achse hat seinen Ursprung in der 2D-Geometrie, wo er eine eindimensionale Struktur innerhalb zweidimensionaler Formen beschreibt. Die mediale Achse ist dabei definiert als die Menge aller Punkte innerhalb einer Form, die mindestens zwei nächstgelegene Punkte auf der Begrenzung (Kante) der Form haben [CSM07]. Der Mittelpunkt eines Kreises, der innerhalb der Form liegt und deren Begrenzung an mindestens zwei Punkten berührt, liegt auf der medialen Achse dieser Form (vgl. Abbildung 2.1) [BL08]. Die Definition der medialen Achse für 2D-Formen kann ebenso auf 3D-Formen angewendet werden, wobei die Menge aller Punkte gesucht sind, die mindestens zwei nächstgelegene Punkte auf der Oberfläche des Objekts haben. Dies resultiert in einer zweidimensionalen Struktur, weshalb diese auch als *mediale Fläche* (*medial surface*) bezeichnet wird [CSM07]. Da das Skelett ein Objekt als

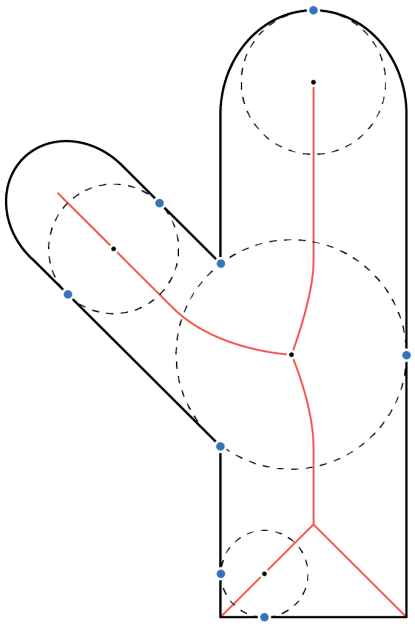

Abbildung 2.1: Mediale Achse (rot) einer 2D-Form. Kreise berühren die Form an mindestens zwei Kontaktpunkten (blau) (Quelle: Basierend auf [CSM07])

eindimensionale Struktur beschreiben soll, ist es bei Verfahren, die auf der Definition einer medialen Achse basieren, notwendig, die mediale Fläche in einem weiteren Schritt auf ein 1D-Skelett zu reduzieren [BLM12].

Es existieren verschiedene Verfahren, um die mediale Achse eines Objekts zu extrahieren. Sie kann beispielsweise mit Hilfe eines Voronoi-Diagramms oder durch Extraktion der lokalen Maxima in einem Abstandsfeld (*distance field*) approximiert werden [BL08; BLM12]. Eine weitere Möglichkeit zur Approximation der medialen Achse besteht in der Anwendung von morphologischen[1] Verdünnungsoperationen (*morphological thinning*) auf ein voxelrepräsentiertes Objekt. Hierbei wird die Punktwolke zunächst in ein dreidimensionales Raster übertragen und schichtweise Voxel von der Oberfläche des Objekts entfernt, bis ein Skelett übrig bleibt [BLM12]. Die Anwendung dieses Verfahrens auf Baum-Punktwolken wird von Gorte und Pfeifer [GP04] beschrieben.

Ein anderer Ansatz zum Extrahieren eines Skeletts aus einer gegebenen Baum-Punktwolke besteht darin, benachbarte Punkte in der Punktwolke durch Clustering zu gruppieren [BLM12]. Xu et al. [XGC07] beschreiben ein Verfahren, bei dem aus einer Punktwolke ausgehend von der Wurzel des Baums zunächst ein minimaler Spannbaum konstruiert wird und die Knoten des Spannbaums durch Quantisierung der Entfernung zum Wurzelknoten geclustert werden. Durch Verbinden benachbarter Clustermittelpunkte wird ein Skelett konstruiert. Da mit dieser Methode nur die groben Strukturen in der Punktwolke zuverlässig rekonstruiert werden können, werden zusätzlich feine Äste in der Baumkrone auf Basis von Grundkenntnissen über Bäume künstlich erzeugt. Li et al. [LBW17] stellen eine Erweiterung dieses Verfahrens vor, bei der durch eine nichtlineare Quantisierung feine Äste in der Baumkrone besser modelliert werden können.

Ein weiterer Ansatz bietet der Algorithmus von Livny et al. [Liv+10]. Dieser geht ebenfalls von einem minimalen Spannbaum aus, der in das Skelett transformiert wird. Der wesentliche Vorteil gegenüber den anderen Verfahren besteht in der Verwendung eines globalen Optimierungsverfahrens, wodurch die Qualität des Skeletts und die Stabilität des Algorithmus gegenüber ungenauen oder unvollständigen Punktwolkendaten verbessert werden können.

Bucksch und Lindenbergh [BL08] konstruieren zunächst einen initialen Graphen mit Hilfe eines Octrees, der die Punktwolke räumlich unterteilt. Anhand bestimmter Regeln werden benachbarte Octree-Zellen zusammengeführt, um den initial konstruierten Graphen auf das Skelett zu reduzieren. Durch die adaptive Raumaufteilung ist dieses Verfahren robust gegenüber inhomogenen Punktdichten und unvollständigen Punktwolken. Das Verfahren wurde in [BLM09] und [BLM12] weiterentwickelt, um die Ergebnisse zu optimieren.

In der Methode von Huang et al. [Hua+13] wird ein sogenanntes L_1-*mediales Skelett* (L_1-*medial skeleton*) konstruiert, das auf der Definition eines L_1-*Medians* basiert. Da der L_1-Median im Kontext dieser Arbeit nicht relevant ist, wird auf eine genaue Definition verzichtet. Während dieses Verfahren robust gegenüber signifikantem Rauschen, Ausreißern und unvollständigen Daten ist und Objekte beliebiger Form skelettieren kann, hat sich gezeigt, dass es für die Rekonstruktion natürlicher Bäume mit komplexer Struktur ungeeignet ist. Darauf aufbauend beschreiben Mei et al. [Mei+16] ein Verfahren, das diese Probleme durch eine Datenvervollständigung und ein iteratives Optimierungs-

[1] Der Begriff „morphologisch“ bezieht sich hier auf die äußere Form allgemeiner Objekte

verfahren löst, sodass auch aus den Punktwolken komplexer Bäume qualitative Skelette extrahiert werden können.

2.2 Auswahl der implementierten Algorithmen

In diesem Abschnitt wird die Auswahl der beiden Skelettierungsalgorithmen für Baum-Punktwolken erläutert, die in dieser Arbeit implementiert werden sollen. Die Auswahl wurde im Hinblick auf die beschriebene Zielsetzung getroffen. Die Algorithmen müssen technisch umsetzbar sein und die Anforderungen an die Genauigkeit der resultierenden Skelette und die Rechenzeit erfüllen (s. Kapitel 1.2). Die Komplexität der Algorithmen wurde ebenfalls berücksichtigt, um sicherzustellen, dass sie sowohl zeitlich realisierbar als auch hinreichend anspruchsvoll in der Implementierung sind.

Eine angestrebte Eigenschaft für beide Skelettierungsalgorithmen ist, dass die Skelettierung ausschließlich auf Basis von Punkt-Positionen erfolgt. Es existieren einige Verfahren, die für die Berechnung des Skeletts zusätzliche Informationen wie Oberflächennormalen oder Intensitätswerte für jeden Punkt benötigen. Solche Daten sind jedoch nicht immer verfügbar oder können nicht ohne weiteres ermittelt werden [Mei+16]. Darüber hinaus sollen die beiden ausgewählten Skelettierungsalgorithmen eine gute Vergleichbarkeit sowohl hinsichtlich ihrer grundsätzlichen Funktionsweise als auch der erzielten Ergebnisse bieten.

Basierend auf den oben genannten Kriterien wurden zwei Skelettierungsalgorithmen ausgewählt. Der erste Algorithmus ist der von Gorte und Pfeifer [GP04], bei dem das Skelett mittels morphologischer Operationen in einem 3D-Raster extrahiert wird. Als zweiter Algorithmus wurde der von Livny et al. [Liv+10] gewählt, der das Skelett direkt im kontinuierlichen Vektorraum konstruiert und dabei eine globale Optimierung anwendet. Zum besseren Verständnis wird in dieser Arbeit der erste Algorithmus entsprechend seiner Funktionsweise als „voxelbasierter“ und der zweite als „graphenbasierter“ Skelettierungsalgorithmus bezeichnet.

Die Auswahl der beiden Algorithmen begründet sich neben den genannten Voraussetzungen vor allem in ihrer unterschiedlichen Funktionsweise. Während der voxelbasierte Skelettierungsalgorithmus in einem 3D-Raster arbeitet und hauptsächlich auf lokalen Berechnungen basiert, arbeitet der graphenbasierte Algorithmus direkt im kontinuierlichen Vektorraum und verfolgt einen globalen Ansatz. Diese Unterschiede bilden eine solide Grundlage für die Vergleichbarkeit. Darüber hinaus erzeugen beide Algorithmen Skelette in Form von Graphen mit Knoten und Kanten, was den Vergleich der Ergebnisse erleichtert. Beide Algorithmen wurden als technisch sowie zeitlich realisierbar bewertet.

2.3 Voxelbasierter Algorithmus

Um ein Verständnis des voxelbasierten Algorithmus von Gorte und Pfeifer zu vermitteln, werden im Folgenden zunächst die erforderlichen Grundlagen erläutert und anschließend die Funktionsweise des Algorithmus erklärt. Für eine umfassende Beschreibung des Verfahrens sei auf [GP04] verwiesen.

2.3.1 Grundlagen

Die Grundidee dieses Verfahrens besteht darin, die Analyse der Punktwolke innerhalb eines dreidimensionalen Rasters durchzuführen, da auf diese Weise Konnektivität und Nachbarschaftsbeziehungen einfacher festgestellt werden können als zwischen den Punkten in der Punktwolke. Konzeptionell kann dieses Raster auch als diskreter Raum betrachtet werden, der aus einem dreidimensionalen Gitter aus kleinen Zellen, den sogenannten Voxeln (*volume pixel* [Col23]), besteht. Der diskrete Raum wird daher auch als Voxelraster oder Voxeldomäne bezeichnet [GP04].

Das Voxelraster ermöglicht die Modellierung von Volumendaten. Für jedes Voxel wird hierfür eine Information über den Inhalt des Teilraums, den es abdeckt, gespeichert. Die einfachste Möglichkeit ist die Speicherung von binären Informationen, wobei ein Voxel mit dem Wert „0“ einen leeren Raum und eines mit dem Wert „1“ ein gefülltes Volumen kennzeichnet (bzw. einen Teil des modellierten Objekts). In Bezug auf Punktwolken kann es nützlich sein, in jedem Voxel die Anzahl der enthaltenen Punkte zu speichern. Dieser Wert kann später als Maß für die Punktdichte innerhalb des Voxels einbezogen werden [GP04].

Ein dreidimensionales Voxelraster kann in horizontale Schichten, respektive Ebenen (*planes*, p), unterteilt werden. Weiterhin lässt sich jede dieser zweidimensionalen Ebenen in Zeilen (*lines*, l) und Spalten (*columns*, c) unterteilen (vgl. Abbildung 2.2). Die Position eines Voxels kann damit durch ein Tupel (p, l, c) definiert werden [GP04].

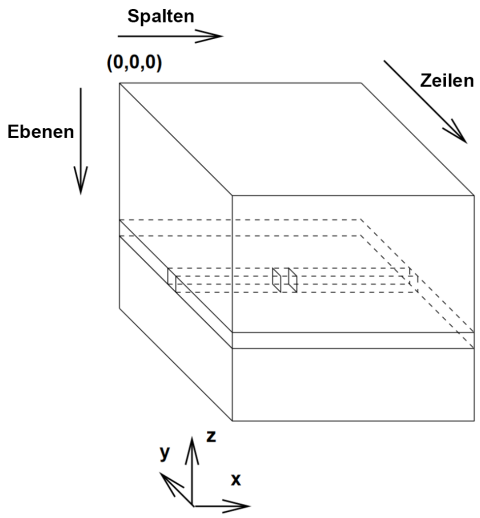

Abbildung 2.2: Ebenen, Zeilen und Spalten eines Voxelrasters (Quelle: Basierend auf [GP04])

Morphologische Operationen bieten die Möglichkeit, die äußere Form der in einem Raster dargestellten Objekte durch Anwendung spezieller Filter, die auf Nachbarschaften basieren, zu manipulieren. Diese Operationen haben ihren Ursprung in der 2D-Bildverarbeitung, wo sie die Veränderung von Formen in Binär- oder Grauwertbildern bewirken [Ser83]. Die in der Methode von Gorte und Pfeifer verwendeten morphologischen Operationen stellen eine 3D-Erweiterung solcher 2D-Operationen dar [GP04].

Zu den klassischen morphologischen Operationen gehören Erosion (*erosion*), Dilatation (*dilation*), Öffnung (*opening*) und Schließung (*closing*) [Sun+22]. Bei der Erosion werden Objekte durch das „Entfernen“ von Voxeln (d. h. deren Wert wird auf „0“ gesetzt) auf ihrer Oberfläche verkleinert, ähnlich einem Minimumfilter (vgl. Abbildung 2.3 (b)). Analog werden bei der Dilatation Objekte durch das „Hinzufügen“ von Voxeln (ihr Wert wird auf „ungleich 0“ gesetzt) vergrößert, vergleichbar mit einem Maximumfilter (c). Die Öffnung stellt eine Kombination einer Erosion mit anschließender Dilatation dar, was eine Glättung der Kontur und Filterung feiner Strukturen zur Folge hat (d). Die Schließung, die eine Dilatation mit anschließender Erosion beschreibt, erlaubt es neben der Glättung Löcher im Objekt zu füllen (e) [Sun+22].

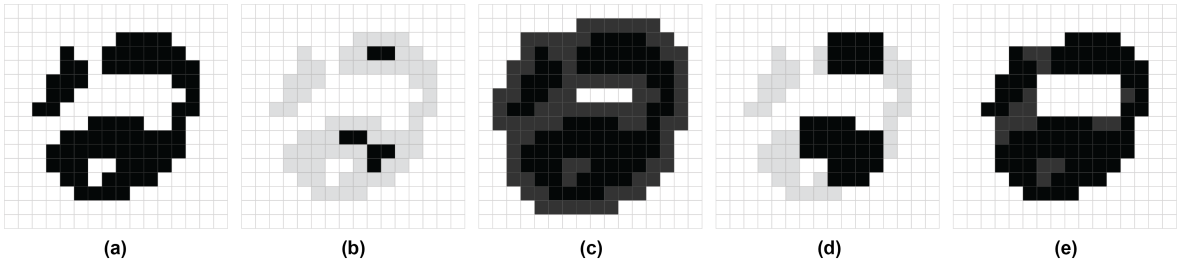

Abbildung 2.3: Klassische morphologische Operationen in einem 2D-Raster. Eingabe (a); Erosion (b); Dilatation (c); Öffnung (d); Schließung (e). Schwarze Pixel haben den Wert 1, weiße Pixel den Wert 0 (Quelle: Basierend auf [Sun+22])

Das strukturierende Element, also der Filterkern (auch *kernel*), legt die Form und Größe der Nachbarschaft eines Voxels fest, die im Rahmen einer Filteroperation berücksichtigt werden soll. In einem dreidimensionalen Voxelraster können drei verschiedene Typen von direkten Nachbarschaften definiert werden, wobei sich jeweils die Art der Konnektivität und die Anzahl der zu einer Nachbarschaft gehörenden Voxel unterscheiden: 6-Adjazenz (nur Voxel mit einer gemeinsamen Fläche sind benachbart), 18-Adjazenz (auch diejenigen mit einer gemeinsamen Kante) und 26-Adjazenz (auch diejenigen mit einer gemeinsamen Ecke) [GP04]. Die drei Nachbarschaftstypen sind in Abbildung 2.4 veranschaulicht.

Eine weitere Operation, die bei der Skelettierung eine wichtige Rolle spielt, ist das Verdünnen (*thinning*) von Voxeldaten. Hierbei werden iterativ solange Schichten von der Oberfläche des Objekts abgetragen und dessen Dicke reduziert, bis eine ein Voxel dicke Struktur entsteht (s. Abbildung 2.5) [GP04]. Damit diese erhalten bleibt, ist es im Gegensatz zur Erosion bei der Bestimmung der zu löschenden Voxel unerlässlich, die Dicke des Objekts zu berücksichtigen. Durch die gleichmäßige Verkleinerung von außen endet die resultierende Struktur im Zentrum des Objekts, womit diese eine Näherung des Skeletts darstellt.

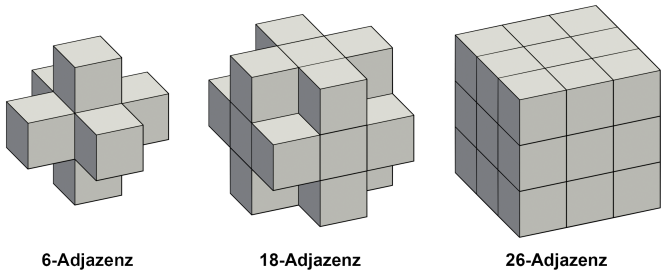

Abbildung 2.4: Voxel-Nachbarschaftstypen im Vergleich (Quelle: Basierend auf [GP04])

Abbildung 2.5: Verdünnen eines Aststückes in mehreren Iterationen

Es existiert eine Vielzahl verschiedener Methoden zur Verdünnung, die zu unterschiedlichen Ergebnissen führen können. Im Rahmen des voxelbasierten Skelettierungsalgorithmus wurde die Methode von Palágyi und Kuba (1999) verwendet, die laut Gorte und Pfeifer eine ausgezeichnete Leistung zeigt [GP04]. Bei der Verdünnungsmethode von Palágyi und Kuba wird das Objekt innerhalb einer Iteration durch 8 Subiterationen konsekutiv von den 8 diagonalen Richtungen im Voxelraster „angegriffen“ [GP04]. In jeder Subiteration wird jedes Voxel mit seinen 26 Nachbarn anhand einer Reihe von „Templates“ der Größe $3 \times 3 \times 3$ auf spezifische Anordnungen der Voxel geprüft. Jedes Template definiert eine bestimmte Voxel-Konfiguration, bei der bestimmte Voxel nicht gegeben sein dürfen (Voxelwert = 0), gegeben sein müssen (Voxelwert $\neq$ 0) oder mindestens einer von einer Gruppe von Voxeln gegeben sein muss. Wenn ein passendes Template an einem Voxel gefunden wird, wird dieses entfernt. Der Vorgang wird wiederholt, bis keine weiteren Voxel in einer Iteration entfernt wurden [PK99].

2.3.2 Funktionsweise

Die Extraktion des Skeletts erfolgt in mehreren Schritten (i–v), die der Reihe nach durchgeführt werden müssen.

i Konvertierung der Punktdaten in Voxeldaten

Da die folgenden Operationen eine Voxel-Repräsentation des Baums voraussetzen, ist es vorab erforderlich, die Punkte der Punktwolke in einen Voxelraum zu übertragen. Bei der Definition des Voxelrasters ist neben der räumlichen Ausdehnung insbesondere die Auflösung des Rasters (d. h. die Anzahl der Voxel, in die das Raster unterteilt wird) wichtig.

Gorte und Pfeifer weisen darauf hin, dass die Wahl einer geeigneten Auflösung hauptsächlich von zwei konkurrierenden Faktoren bestimmt wird. Zunächst ist die *Punktdichte* in der Punktwolke zu berücksichtigen. Punkte, die zu einem Baum gehören, sollten eine zusammenhängende Menge von Voxeln erzeugen. Eine zu feine Auflösung kann zu Löchern führen, die den Baum in mehrere separate Objekte zerteilen. Andererseits sollte die Auflösung nicht zu grob gewählt werden, da sonst nicht zusammengehörige Äste und Objekte verbunden werden könnten. Die inhomogene Punktdichte der Punktwolke erschwert zudem die Wahl einer geeigneten Auflösung. Auf der anderen Seite wird die Auflösung durch *technische Limitierungen* begrenzt. Es ist wichtig zu beachten, dass die Anzahl der Voxel und somit die Speicherkapazität sowie die Laufzeit des Algorithmus kubisch mit Vergrößerung der Auflösung in einer Dimension zunehmen. Eine zu feine Auflösung kann daher schnell unpraktikabel werden [GP04]. Gorte und Pfeifer empfehlen eine Auflösung von zwei bis fünf Zentimetern (bzgl. der Größe eines Voxels), wobei dies an den jeweiligen Anwendungsfall angepasst werden sollte [GP04].

Die Transformation der Punkte in das Voxelraster entspricht einer Quantisierung der Punkt-Positionen. Nach der Rasterung entspricht der Wert eines Voxels der Anzahl der Punkte, die in ihn fallen [GP04]. Abbildung 2.6 (b) zeigt eine beispielhafte Voxelstruktur, die nach Quantisierung der Punktwolke in (a) entsteht.

ii Datenvorverarbeitung

Für die weiteren Schritte ist es erforderlich, die Daten aufzubereiten. Hierbei werden drei bestehende Probleme adressiert, die unter anderem mit Hilfe der zuvor erwähnten morphologischen Operationen behoben oder reduziert werden können.

Das erste Problem entsteht durch Rauschen in der Punktwolke. Nach der Übertragung der Punktdaten in Voxeldaten können einzelne isolierte Voxel oder kleine zusammenhängende Gruppen von Voxeln entstehen, die für die Skelettierung nicht relevant sind. Diese fehlerhaften Voxel können durch die Anwendung von Nachbarschaftsoperationen gefiltert werden. Ein weiteres Problem besteht darin, dass Überdeckungen Löcher oder Lücken in der Baumstruktur verursachen können. Durch eine morphologische Closing-Operation können solche Lücken geschlossen werden. Die Größe des strukturierenden Elements legt die maximale Größe der Lücken fest, die repariert werden können. Zuletzt ist es problematisch, dass die Punkte in der Punktwolke nur die Oberfläche des Baums abbilden.

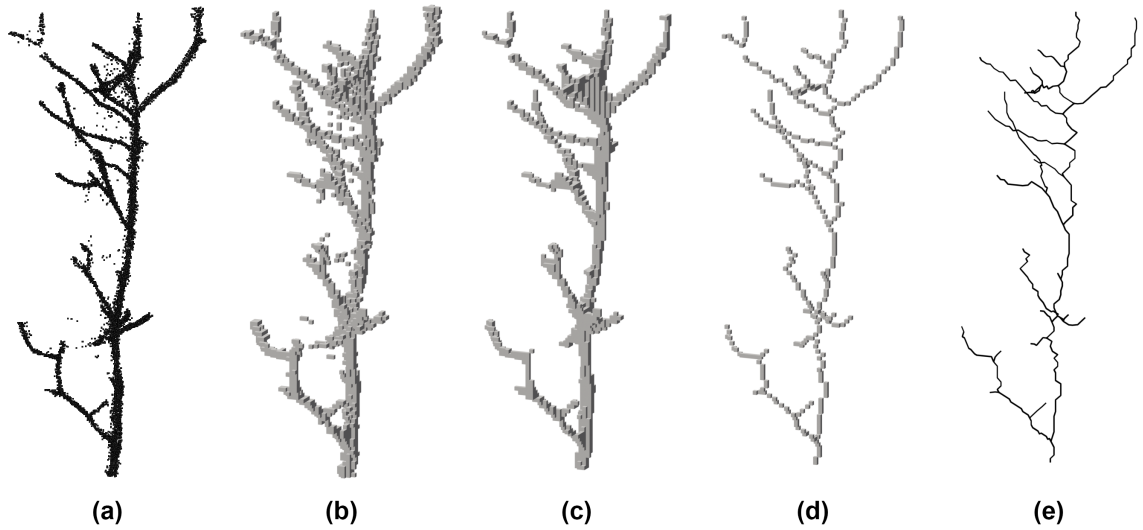

Abbildung 2.6: Zwischenergebnisse der voxelbasierten Skelettextraktion. Punktwolke (a); Voxelstruktur nach Quantisierung (b); Voxelstruktur nach Datenvorverarbeitung (c); Voxelstruktur nach Thinning (d); Extrahiertes Skelett (e)

Dies hat zur Folge, dass nach der Rasterung nur Voxel vorhanden sind (nicht „0“ sind), die die Oberfläche schneiden, wodurch der Stamm sowie besonders breite Äste hohl bleiben. Mit Hilfe einer zweiten Closing-Operation gilt es, diese Hohlräume zu schließen (s. Abbildung 2.7). Das strukturierende Element ist hierbei horizontal ausgerichtet (vergleichbar mit einer runden Scheibe), da die Füllung hauptsächlich in dem vertikalen Stamm erfolgen muss, und dessen Größe muss an die maximale Breite des Stamms angepasst werden [GP04]. Das Ergebnis nach der Datenvorverarbeitung ist in Abbildung 2.6 (c) dargestellt.

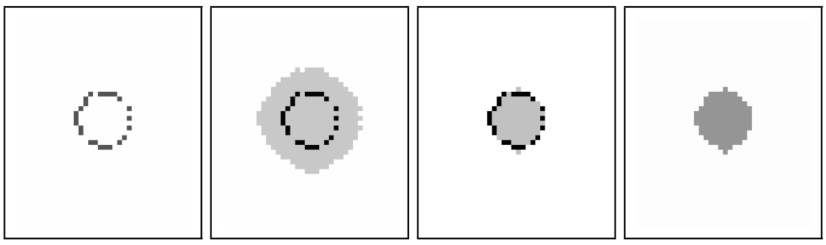

Abbildung 2.7: Closing-Operation auf einem Baum-Querschnitt. V. l. n. r. Eingabe; Dilatation; Erosion; geschlossenes Resultat (Quelle: [GP04])

iii Skelettierung durch Verdünnen

Nach der Aufbereitung der Voxeldaten wird die morphologische Thinning-Operation von Palágyi und Kuba [PK99] angewendet, um die Dicke der Äste auf eine Voxelbreite zu reduzieren [GP04]. Das Resultat ist eine ein Voxel dicke Struktur, die die Form des Baum-Skeletts approximiert (s. Ab-

bildung 2.6 (d)). Gorte und Pfeifer weisen darauf hin, dass die Qualität dieses Zwischenergebnisses im Wesentlichen den Erfolg der nachfolgenden Schritte beeinflusst [GP04].

iv Extraktion des Skeletts

Die im vorherigen Schritt erzeugte Voxelstruktur repräsentiert zwar die Form des Skeletts, es ist jedoch zu beachten, dass sie keine Informationen über die tatsächliche Struktur des Skeletts enthält (hinsichtlich der Äste und Verzweigungen), da keine Verbindungen zwischen den Voxeln bestehen. Werden 26-adjazente Voxelpaare als verbunden betrachtet, so können zudem ungewollte Verbindungen entstehen, die Zyklen in der Skelettstruktur verursachen [GP04]. Dies steht im Widerspruch zum gewünschten Skelett, da die Struktur eines Baums von Natur aus zyklenfrei ist. Ziel dieses Schrittes ist es, aus den erzeugten Voxeldaten das Skelett zu extrahieren, das eindeutig die Struktur des Baums widerspiegelt und frei von Zyklen ist.

Als Ausgangspunkt wird ein ungerichteter Graph erzeugt, der Paare benachbarter Voxel miteinander verbindet. Die Knoten dieses Graphen repräsentieren die Voxel, die Teil des Skeletts sind. Knoten, deren Voxel gemäß der 26-Adjazenz als benachbart gelten, werden jeweils durch eine Kante miteinander verbunden. Die hinzugefügten Kanten werden basierend auf den Regeln von Verwer [Ver91] abhängig von der Konnektivität der beiden Voxel gewichtet. Hierbei erhalten Kanten die Gewichtungen 3, 4 oder 5, je nachdem, ob die beiden Voxel über eine Fläche, Kante oder Ecke benachbart sind [GP04].

Zur Extraktion des finalen Skeletts wird der Algorithmus von Dijkstra herangezogen. Dieser liefert von jedem beliebigen Knoten eines zusammenhängenden Graphen den kürzesten Weg zu einem Zielknoten. Für den folgenden Schritt wird ein Wurzelknoten als jener Knoten definiert, dessen Voxel unter allen zusammenhängenden Voxeln räumlich am tiefsten liegt. Durch Anwendung des Dijkstra-Algorithmus auf den ungerichteten Graphen, mit den zuvor gewählten Kantengewichtungen und dem Wurzelknoten als Zielknoten, erhält man als Ergebnis von jedem Knoten den kürzesten Weg zur Wurzel. Die Menge aller kürzesten Wege bildet einen vom Wurzelknoten minimalen Spannbaum (im Sinne der Graphentheorie), da durch die Eindeutigkeit der kürzesten Wege alle Zyklen aufgelöst werden (vgl. Abbildung 2.8). Somit entspricht der minimale Spannbaum dem gesuchten Skelett (s. Abbildung 2.6 (e)) [GP04].

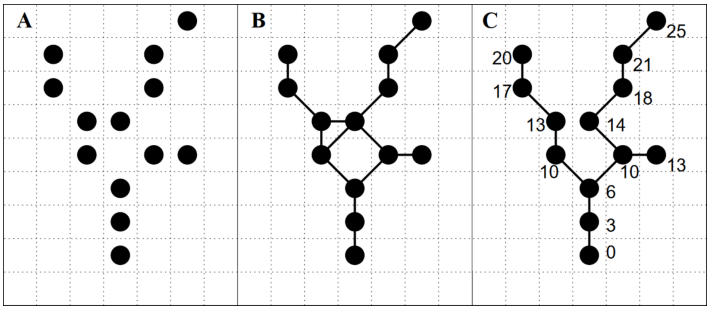

Abbildung 2.8: Skelettextraktion aus der verdünnten Voxelstruktur. Mittelpunkte der Voxel (A); Ungerichteter Graph (B); Minimaler Spannbaum mit kürzesten Wegen zu den Knoten (C) (Quelle: [GP04])

v Segmentierung zusammenhängender Komponenten

Es kann vorkommen, dass nach der Vorverarbeitung der Voxeldaten mehrere nicht zusammenhängende Komponenten im Voxelraster übrig verbleiben. Dies stellt ein Problem für die Extraktion eines Skeletts dar. Gorte und Pfeifer betonen, dass der Algorithmus von Dijkstra nur die Bestimmung eines Zielknotens (ein Wurzelknoten) erlaubt und die kürzesten Wege nur für Voxel gefunden werden können, die mit diesem Zielknoten verbunden sind. Alle weiteren Bäume, die im Voxelraster enthalten sind, werden vernachlässigt [GP04].

Um dieses Problem zu lösen, werden alle Komponenten separat betrachtet und die im vorherigen Schritt beschriebene Extraktion eines Skeletts für jede zusammenhängende Komponente individuell durchgeführt. Gorte und Pfeifer beschreiben in ihrem Paper eine modifizierte Methode zur Kennzeichnung verbundener Komponenten (*connected component labeling*), bei der gleichzeitig alle Skelette extrahiert werden. Dabei werden alle Voxel in der Voxeldomäne schichtweise von unten nach oben durchlaufen. Sobald ein noch nicht besuchtes Voxel gefunden wird, erhält dieses zunächst ein neues Label zur eindeutigen Identifizierung des Skeletts. Ausgehend von diesem Voxel wird ein zusammenhängender Graph gebildet und der Dijkstra-Algorithmus (mit dem aktuellen Voxel als Wurzel) angewendet, um ein Skelett für diese zusammenhängende Komponente zu erzeugen. Anschließend wird das Skelett durchlaufen und jedem Skelett-Voxel das entsprechende Label zugewiesen [GP04].

Nachdem alle Voxel durchlaufen wurden, werden alle gefundenen Komponenten gesammelt und in mehrere Datensätze aufgeteilt (*component separation*). Eine Sortierung der Komponenten nach der Anzahl der Voxel ermöglicht deren Priorisierung. Bei bekannter Anzahl n der Bäume in der Punktwolke ist es so möglich, nur die n dominierenden (bzw. größten) Skelette auszugeben oder solche, die eine Mindestanzahl an Voxeln aufweisen [GP04].

Der voxelbasierte Algorithmus von Gorte und Pfeifer umfasst noch weitere Schritte, die auf eine strukturelle Segmentierung der Punktwolke abzielen [GP04]. Diese Schritte sind jedoch für die Skelettextraktion nicht relevant und werden hier nicht weiter behandelt.

2.4 Graphenbasierter Algorithmus

Im folgenden Abschnitt werden die Grundidee und die Funktionsweise des graphenbasierten Skelettierungsalgorithmus von Livny et al. erläutert. Eine ausführliche Erklärung findet sich in [Liv+10].

2.4.1 Grundlagen

Im Unterschied zum voxelbasierten Algorithmus, bei dem die meisten Berechnungen in einem diskreten Voxelraster durchgeführt werden, operiert der graphenbasierte Algorithmus direkt im kontinuierlichen Vektorraum, in dem die Punktwolke liegt. Ebenso wird das Skelett direkt innerhalb des Vektorraums konstruiert. Livny et al. bezeichnen dieses auch als *Branch-Structure Graph* [Liv+10] (kurz „BSG“), der gemäß der Graphentheorie definiert wird als räumlich eingebetteter, gerichteter Baum (zusammenhängender, gerichteter, azyklischer Graph). Räumlich eingebettet bedeutet, dass jeder Knoten des Skeletts eine Position im Vektorraum zugeordnet bekommt. Zur terminologischen Abgrenzung eines einfachen Graph-Knotens von einem positionierten Skelett-Knoten wird letzterer auch als *Vertex* (Plural: *Vertices*) bezeichnet. Die Wurzel des BSG entspricht dem untersten Punkt eines Baums und die Kanten sollen im Zentrum der Baumäste liegen [Liv+10].

Die Konstruktion des Skeletts erfolgt primär durch eine Reihe von globalen Optimierungsprozessen, die von Annahmen über biologische Wachstumseigenschaften von Bäumen abgeleitet wurden. Basierend auf den Ausarbeitungen von Honda [Hon71] führen Livny et al. drei natürliche und charakteristische Eigenschaften von Bäumen auf, wobei sich diese vorrangig auf einfache, unverzweigte Astsegmente (werden auch als *branch chains* bezeichnet, siehe Abbildung 2.9) beziehen. Die erste und maßgebende Eigenschaft ist, dass sich typische Baumstrukturen durch einen glatten Verlauf der Äste kennzeichnen. Im Skelett der unverzweigten Astsegmente sollten die Winkel zwischen zwei inzidenten (d. h. miteinander verbundenen) Kanten daher tendenziell klein sein. Die zweite Eigenschaft beschreibt, dass diese einfachen Astsegmente in der Nähe des Baumstamms breiter und länger sind, während sie in der Baumkrone eher schmaler und kürzer sind. Dies geht mit der Eigenschaft einher, dass breite Äste auf Höhe des Stammes allgemein glatter verlaufen, wohingegen feine Zweige in der Baumkrone mehr Details in ihrer Struktur aufweisen. Die dritte Eigenschaft besagt, dass sich die Dichte der unverzweigten Astsegmente antiproportional zur Astbreite verhält, sodass insgesamt mehr schmale Zweige in der Baumkrone zu finden sind als breitere Äste in der Nähe des Stamms [Liv+10].

Die obigen Eigenschaften bilden die Grundlage für die Formulierung globaler Optimierungsprobleme, die zur Berechnung des Skeletts gelöst werden müssen. Konkret werden Funktionen definiert, die alle Knoten des Skeletts in Abhängigkeit voneinander setzen und mit Hilfe der Methode der kleinsten Quadrate einen bestimmten Fehler quantifizieren, den es zu minimieren gilt. Im folgenden Abschnitt werden die Funktionen genauer dargelegt. Laut Livny et al. besteht der Vorteil eines globalen Optimierungsprozesses im Vergleich zu lokalen Berechnungen darin, dass er robuster gegenüber inkonsistenter Datenqualität in der Punktwolke ist, wie beispielsweise Rauschen, ungleichmäßige Punktdichte oder unvollständige Daten [Liv+10].

Ein weiterer bedeutender Schritt besteht in der Berechnung von Gewichtungen (sog. *importance weights*) für die einzelnen Vertices des Skeletts. Die Gewichtung eines Vertex entspricht der Größe des Unterbaums, der diesen Vertex als Wurzel besitzt, und liefert somit ein grobes Maß für die „Masse“,

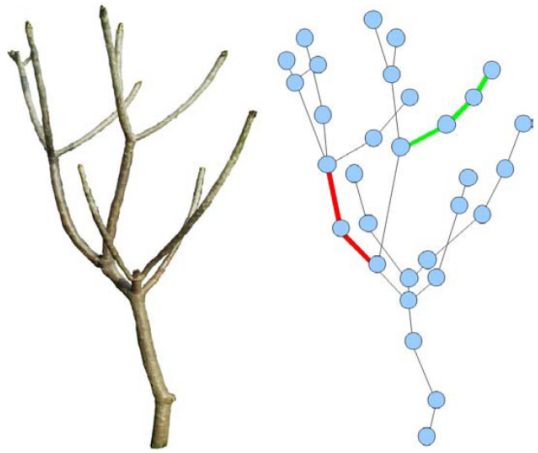

Abbildung 2.9: Aststück (links) und BSG mit zwei Branch Chains in rot und grün (rechts) (Quelle: [Liv+10])

die ein bestimmter Ast trägt. Typischerweise sind Äste, die hohe Gewichtungen der Vertices aufweisen, von Natur aus breiter als solche mit geringen Gewichtungen. Diese Maßzahlen werden bei der globalen Optimierung berücksichtigt: Durch unterschiedliche Gewichtungen der Vertices in den Kostenfunktionen wird unter anderem sichergestellt, dass breite Äste nahe der Wurzel stärker geglättet werden als schmale Zweige in der Baumkrone [Liv+10]. Die Berechnung dieser Gewichtungen wird im folgenden Kapitel beschrieben.

2.4.2 Funktionsweise

Nachdem die Grundlagen des graphenbasierten Algorithmus erläutert wurden, wird im Folgenden die genaue Funktionsweise des Algorithmus erklärt. An dieser Stelle sei angemerkt, dass der durch Livny et al. beschriebene Algorithmus eine automatische Segmentierung und Skelettierung mehrerer Bäume in der Punktwolke zugleich erlaubt. So ist es nicht notwendig, die einzelnen Bäume in der Punktwolke manuell zu segmentieren, was oftmals schwierig und umständlich ist [Liv+10]. Zum besseren Verständnis fokussieren sich die folgenden Erklärungen auf die Extraktion eines einzelnen Skeletts, sofern nicht anders erwähnt.

Der Algorithmus kann im Wesentlichen in drei Phasen unterteilt werden: In der ersten Phase (Schritte i–ii) erfolgt die Konstruktion eines initialen Skeletts. Dieses wird in der zweiten Phase (Schritte iii–vi) durch globale Optimierung verfeinert. Um ein noch optimaleres Ergebnis zu erzielen, wird diese Phase iterativ durchgeführt. Die letzte Phase (Schritte vii–viii) dient der Bereinigung des Skeletts von fehlerhaften Ästen [Liv+10].

i Identifizierung der Bäume in der Punktwolke

Bevor ein initiales Skelett konstruiert werden kann, ist es erforderlich, die Anzahl sowie die Positionen der Bäume in der Punktwolke zu bestimmen. Für jeden Baum wird ein Punkt in der Punktwolke

ausgewählt, der den Wurzelknoten des Baumskeletts repräsentiert und einen Startpunkt für die Konstruktion des Skeletts bildet [Liv+10].

Zunächst werden die Punkte in der Punktwolke auf die Grundebene (*ground plane*) projiziert. Da bei gescannten Bäumen in der Regel viele Punkte übereinander liegen, entsteht durch die Projektion für jeden Baum ein Cluster mit hoher Punktdichte [Liv+10] in der 2D-Grundebene. Punkte, die in Bereichen mit niedriger Punktdichte liegen, werden entfernt, da diese meist nur den Boden abbilden. Die Aufgabe, die Anzahl der Bäume und ihre Positionen zu bestimmen, entspricht daher einer Clusteranalyse der projizierten Punkte. Livny et al. verdeutlichen, dass der Stamm eines Baums typischerweise im Zentrum eines Baum-Clusters liegt. Daher wird der Punkt, der dem Mittelpunkt des Clusters am nächsten liegt, als Wurzelknoten für diesen Baum ausgewählt [Liv+10].

ii Initiale Konstruktion des Skeletts

Im nächsten Schritt wird aus der Punktwolke für jeden Baum ein initiales Skelett konstruiert. Als Ausgangspunkt dient ein vollständiger Graph P, der alle Punkte der Punktwolke als Knoten bzw. Vertices enthält und diese miteinander verbindet. Die Kanten des Graphen P werden basierend auf der Entfernung der Vertices zueinander gewichtet. Eine Kante, die zwei Vertices u und v (Positionen der Vertices) verbindet, wird gewichtet durch den quadrierten euklidischen Abstand $\|u-v\|^2$ [Liv+10].

Zur Extraktion der Baumskelette ist es nun erforderlich, einen minimalen Spannbaum über alle Vertices von P zu suchen. Der Sinn dieses Ansatzes begründet sich darin, dass die Vertices in den Baumskeletten mit hoher Wahrscheinlichkeit über möglichst kurze Wege miteinander verbunden sind und die gesuchten Skelette damit eine minimale Gesamt-Kantenlänge aufweisen. Ähnlich wie bei dem voxelbasierten Algorithmus wird der minimale Spannbaum mit Hilfe des Algorithmus von Dijkstra gebildet. Ein Problem ist allerdings, dass bei Vorhandensein mehrerer Bäume mehrere Skelette, das heißt mehrere aufspannende Bäume mit unterschiedlichen Wurzelknoten, gebildet werden müssen. Wie bereits erwähnt, erlaubt der Algorithmus von Dijkstra allerdings nur die Suche zu einem Zielknoten. Um dieses Problem zu überwinden, wird ein zusätzlicher temporärer Knoten hinzugefügt, der durch „0“-gewichtete Kanten mit den Wurzelknoten aller Skelette verbunden wird. Durch die Konstruktion des minimalen Spannbaums von diesem Knoten aus werden automatisch alle Bäume berücksichtigt. Der temporäre Knoten wird anschließend wieder entfernt, was eine automatische Segmentierung der Punktwolke in die einzelnen Bäume zur Folge hat [Liv+10].

Die resultierenden Bäume entsprechen den initialen Skeletten, bezeichnet als $\{T_1, \ldots, T_m\}$, die anschließend unabhängig voneinander weiterverarbeitet werden [Liv+10]. Da die Skelette durch einfaches Verbinden der Punkte in der Punktwolke erzeugt wurden, sei darauf hingewiesen, dass diese nicht zentriert sind und viele „überflüssige“ Verzweigungen entstehen (vgl. Abbildung 2.10 (b)). In den folgenden Schritten gilt es, diese Probleme durch Optimierung der initialen Skelette zu beheben.

iii Berechnung der Knotengewichtungen

Wie im vorherigen Kapitel beschrieben, stellt die Berechnung von Knotengewichtungen (*importance weights*) einen wichtigen Schritt dar, da diese den Einfluss der Vertices in den globalen Optimierungs-

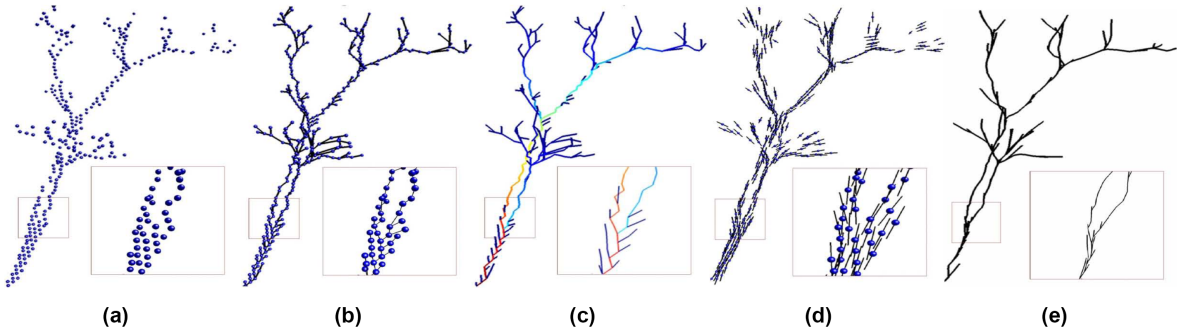

Abbildung 2.10: Zwischenergebnisse der graphenbasierten Skelettextraktion. Punktwolke (a); Initiales Skelett (b); Knotengewichtungen farblich gekennzeichnet (c); Orientierungsfeld (d); Optimiertes Skelett (e) (Quelle: [Liv+10])

prozessen steuern. Die Gewichtung eines Vertex berechnet sich aus der Summe aller Kantenlängen des Unterbaums, der von dem Vertex ausgeht. Die Länge einer Kante, die zwei Vertices u und v verbindet, entspricht dem euklidischen Abstand $\|u - v\|$. Alternativ lässt sich die Gewichtung eines Vertex auch aus der Summe der Gewichtungen seiner direkten Kind-Vertices sowie den Längen der inzidenten Kanten berechnen. Die Gewichtungen werden für jeden Vertex der Skelette aus dem vorherigen Schritt berechnet und zur weiteren Verwendung in den folgenden Schritten zwischengespeichert [Liv+10]. Das Ergebnis ist in Abbildung 2.10 (c) dargestellt.

iv Konstruktion eines globalen Orientierungsfeldes

In den folgenden Schritten erfolgt die globale Optimierung des initialen Skeletts. Ziel ist es, das initiale Skelett durch Verschiebung der Vertices an eine Baumstruktur anzugleichen, die von einem natürlichen Baum zu erwarten wäre. Hierbei werden die Eigenschaften berücksichtigt, die im vorherigen Kapitel diskutiert wurden, wie beispielsweise ein glatter Verlauf der Äste [Liv+10].

Die globale Optimierung erfolgt auf Grundlage eines geglätteten Orientierungsfeldes (*orientation field*), das Informationen über die allgemeine Richtung der Äste enthält. Zur Konstruktion dieses Orientierungsfeldes wird mit Hilfe von globaler Optimierung für jeden Vertex des Skeletts eine bestimmte Richtung in Form eines Vektors berechnet und diesem zugewiesen [Liv+10]. Man betrachte hierfür die folgenden Funktionen [Liv+10, Formeln 1 und 2]:

$$\Delta O(T_i) = \sum_{v \in T_i} \left(\frac{c_{v_p} + c_v}{2} \left\| o_{v_p} - o_v \right\| \right)^2 \tag{2.1}$$

$$\Delta E(T_i) = \sum_{v \in T_i} \left(c_v \left\| o_v - \frac{e(v_p, v)}{\left\| e(v_p, v) \right\|} \right\| \right)^2 \tag{2.2}$$

T_i ist das Skelett, für das das Orientierungsfeld berechnet werden soll. v repräsentiert einen beliebigen Vertex in T_i mit Gewichtung c_v, während v_p dessen Eltern-Vertex mit der Gewichtung c_{v_p}

darstellt. $e(v_p, v)$ ist der Verbindungsvektor von v_p und v. Die gesuchten Orientierungen, die die Richtungen des Orientierungsfeldes in Form von Vektoren repräsentieren, werden als o_v (für v) bzw. o_{v_p} (für v_p) bezeichnet ($o_v, o_{v_p} \in \mathbb{R}^3$) [Liv+10].

Die obigen Formeln beschreiben zwei Abhängigkeiten: Der Ausdruck $\|o_{v_p} - o_v\|$ in (2.1) quantifiziert die Abweichung zwischen den Orientierungen der Vertices o_{v_p} und o_v. Durch Minimierung von $\Delta O(T_i)$ werden alle Abweichungen der Orientierungen zwischen adjazenten Vertices minimiert [Liv+10], was zu einer allgemeinen Glättung des Orientierungsfeldes führt. Demgegenüber wird in (2.2) durch $\|o_v - \frac{e(v_p,v)}{\|e(v_p,v)\|}\|$ die Abweichung zwischen der Orientierung eines Vertex und der Richtung der eingehenden Kante im initialen Skelett quantifiziert. Eine Minimierung von $\Delta E(T_i)$ minimiert diese Abweichungen für alle Vertices [Liv+10] und zwingt damit die Orientierungen, die allgemeine Richtung der Äste beizubehalten. Um sicherzustellen, dass das Orientierungsfeld nicht durch die kleinen, fehlerhaften Äste verfälscht wird, werden die quantifizierten Abweichungen mit Hilfe der berechneten Knotengewichtungen gewichtet [Liv+10].

Durch Minimierung von $\Delta O(T_i) + \Delta E(T_i)$ wird ein Orientierungsfeld gebildet, das geglättet ist und die Hauptrichtungen der Äste repräsentiert [Liv+10]. Ein beispielhaftes Ergebnis ist in Abbildung 2.10 (d) dargestellt.

v Globale Verfeinerung des Skeletts

Mit Hilfe des Orientierungsfeldes wird in diesem Schritt das initiale Skelett durch Verschiebung der Vertices verbessert [Liv+10]. Wie im vorherigen Schritt, erfolgt die Berechnung der neuen Vertex-Positionen durch eine globale Optimierung, bei der bestimmte Fehlerquadrate minimiert werden. Hierfür werden die folgenden Funktionen definiert [Liv+10, Formeln 3 und 4]:

$$\Delta A(T_i) = \sum_{e(u,v)\in T_i} \left(\frac{c_u + c_v}{2} \left\| (u' - v') - \frac{\|u - v\|\,(o_u + o_v)}{\|o_u + o_v\|} \right\| \right)^2 \tag{2.3}$$

$$\Delta F(T_i) = \sum_{e(u,v)\in T_i} \left(c_v \left\| \frac{u' + v'}{2} - \frac{u + v}{2} \right\| \right)^2 \tag{2.4}$$

Hier repräsentiert (u, v) eine beliebige Kante in dem Skelett T_i, die zwei Vertices an den Positionen u, mit Gewichtung c_u und konstanter Orientierung o_u und v, mit Gewichtung c_v und Orientierung o_v, verbindet. Gesucht sind neue Vertex-Positionen u' (bzw. v') für u (bzw. v), die $\Delta A(T_i) + \Delta F(T_i)$ minimieren [Liv+10].

Auch hier wird der Optimierungsprozess durch zwei Abhängigkeiten gesteuert: Der Ausdruck $\|(u' - v') - \frac{\|u-v\|(o_u+o_v)}{\|o_u+o_v\|}\|$ in (2.3) quantisiert die Abweichung zwischen der Richtung einer neuen Kante (u', v') und der gewünschten Orientierung aus dem Orientierungsfeld $\frac{o_u+o_v}{\|o_u+o_v\|}$. Durch Minimierung von $\Delta A(T_i)$ wird das Skelett an die Richtungen des Orientierungsfeldes angeglichen und damit geglättet [Liv+10]. In (2.4) wird durch $\|\frac{u'+v'}{2} - \frac{u+v}{2}\|$ die Abweichung des neuen Kanten-Mittelpunktes von dem der alten Kante quantisiert. Durch die Minimierung von $\Delta F(T_i)$ soll sichergestellt werden, dass die Form des Skeletts nach der Glättung nicht zu stark von der des initialen Skeletts abweicht [Liv+10].

Des Weiteren werden erneut die Knotengewichtungen einbezogen, die unterschiedlich starke Einflüsse der Vertices erlauben [Liv+10].

Durch Minimierung von $\Delta A(T_i)+\Delta F(T_i)$ entsteht ein neues Skelett T_i' mit neuen Vertex-Positionen, das eine glatte Struktur aufweist und räumlich innerhalb der Punktwolke eingebettet ist [Liv+10]. Das verfeinerte Skelett ist in Abbildung 2.10 (e) dargestellt.

vi Iterative Verfeinerung

Die Autoren Livny et al. weisen darauf hin, dass der Algorithmus durch die Unvollständigkeit der Eingabedaten möglicherweise noch kein optimales Ergebnis nach nur einem Durchlauf liefert. Um ein noch genaueres Skelett zu erzielen, besteht die Möglichkeit, den Optimierungsprozess auf Basis eines neuen initialen Skeletts zu wiederholen, das einen besseren Startpunkt für den Optimierungsprozess bietet [Liv+10].

Um Informationen über das aktuelle Skelett an die nachfolgende Iteration zu übergeben, werden die Kantengewichtungen auf dem vollständigen Graphen P angepasst. Hierfür wird zunächst für jede Kante $e = (u', v')$ des optimierten Skeletts die Kante $e_P = (P_{u'}, P_{v'})$ in dem vollständigen Graphen P gesucht, deren Vertices am nächsten liegen. Anschließend wird die Gewichtung dieser Kante um das Skalarprodukt der Kanten-Richtungen, $\frac{e}{\|e\|} \cdot \frac{e_P}{\|e_P\|}$, skaliert. Mit Hilfe des Algorithmus von Dijkstra wird ein neuer aufspannender Baum erzeugt. Falls sich dieser aufspannende Baum strukturell von dem zuvor erzeugten Skelett unterscheidet, wird der Optimierungsprozess mit dem neuen aufspannenden Baum als initiales Skelett wiederholt. Laut Livny et al. konvergiert dieser Prozess nach 2 bis 3 Iterationen [Liv+10].

vii Approximation der Astradien

Es ist zu beachten, dass das Skelett in den vorherigen Schritten lediglich durch Verschiebung der Vertices optimiert wurde. Fehlerhafte Äste, die bereits bei der initialen Konstruktion entstanden sind, bleiben auch im optimierten Skelett erhalten. Das Ziel der letzten beiden Schritte besteht darin, diese fehlerhaften Äste zu entfernen.

Zur Identifizierung der zu entfernenden Äste wird für jeden Vertex eine Information über den Astradius an der jeweiligen Stelle benötigt. Aufgrund der begrenzten Informationsmenge in der Punktwolke gestaltet sich die Berechnung eines exakten Radius schwierig. Daher werden diese basierend auf den von Xu et al. [XGC07] vorgestellten Allometrien[2] natürlicher Bäume geschätzt [Liv+10]. Die Berechnung erfolgt wieder durch eine globale Optimierung der folgenden Funktionen [Liv+10, Formeln 6 und 7]:

[2] „Allometrie“ ist das Messen und Vergleichen von Größen, insbesondere in natürlichen Systemen, um Beziehungen zwischen verschiedenen biologischen Eigenschaften zu analysieren [UL15]

$$\Delta R(T_i') = \sum_{u',v' \in T_i'} \left| r_{u'} - \left(\frac{c_u}{c_v}\right)^{2.5} r_{v'} \right|^2 \tag{2.5}$$

$$\Delta D(T_i') = \left(\sum_{v' \in T_i'} c_v r_{v'} \right)^2 - \left(d_{\text{avg}} \sum_{v' \in T_i'} c_v \right)^2 \tag{2.6}$$

Durch $\Delta R(T_i')$ in (2.5) werden die gesuchten Radien $r_{u'}$ und $r_{v'}$ zweier benachbarter Vertices u' und v' in T_i', abhängig von ihren Gewichtungen c_u und c_v, in ein Verhältnis zueinander gesetzt [Liv+10]. Mit Hilfe von $\Delta D(T_i')$ in (2.6) wird der durchschnittliche Radius des Baums gesteuert. Die Variable d_{avg} stellt dabei die durchschnittliche Distanz zwischen den Punkten und den Kanten des Skeletts dar. Die Radien werden durch Minimierung von $\Delta R(T_i') + \Delta D(T_i')$ ermittelt und den Vertices zugewiesen [Liv+10].

viii Bereinigung des Skeletts durch Entfernen von Kanten

Im letzten Schritt wird das Skelett durch Löschen von überflüssigen Kanten und Ästen bereinigt. Zunächst werden die fehlerhaften Verzweigungen, die bei der Konstruktion des Skeletts am Stamm des Baums entstanden sind, entfernt. Da diese falschen Verzweigungen zu kurzen Ästen in der Nähe des Stamms führen, sind auch ihre Radien klein. Mit Hilfe der großen Differenz der Radien zwischen den fehlerhaften Ästen und dem Stamm werden diese Äste identifiziert und vollständig entfernt [Liv+10].

Zum Zweck der folgenden Maßnahmen wird die geometrische Form des Baums als Aneinanderreihung verallgemeinerter Zylinder entlang der Skelett-Segmente betrachtet, welche die berechneten Astradien aufweisen (s. Abbildung 2.11 (a)). Durch Berechnung der Schnittvolumen benachbarter Zylinder lassen sich Kanten identifizieren, die nur einen geringfügigen Beitrag zum Skelett leisten. Diese werden im letzten Schritt reduziert. Es werden zwei Fälle unterschieden: Im ersten Fall werden die Überschneidungen von Zylindern an Verzweigungen betrachtet. Gegeben sei ein Vertex u, mit zwei Kind-Vertices v und w. Wenn das Schnittvolumen der Zylinder (u, v) und (u, w) mehr als zur Hälfte das Volumen von (u, w) überschreitet (s. Abbildung 2.11 (b), in blau), wird die Kante (u, w) durch Verschmelzen der Vertices w und u oder von w und v entfernt. Im zweiten Fall werden unverzweigte Knoten (mit Grad 2) betrachtet. Gegeben sei ein Vertex v, mit Eltern-Vertex u und genau einem Kind-Vertex w. Wenn das Volumen eines neuen Zylinders $(u, v + (v - u))$ mehr als die Hälfte des Volumens von (v, w) überschneidet (s. Abbildung 2.11 (b), in grün), wird die Kante (u, v) durch Verschmelzen der beiden Vertices entfernt. In beiden Fällen entsteht beim Verschmelzen zweier Vertices u und v ein neuer Vertex, der am gewichteten Mittelpunkt $(c_u \cdot u + c_v \cdot v)/(c_u + c_v)$ positioniert wird (s. Abbildung 2.11 (c,d)). Der Vorgang wird so lange wiederholt, bis keine weiteren Kanten vorhanden sind, die die genannten Bedingungen erfüllen [Liv+10].

Das Ergebnis ist ein Skelett, das die wesentlichen Aststrukturen repräsentiert und die charakteristischen Merkmale eines natürlichen Baums aufweist. Livny et al. schlagen vor, basierend auf L-Systemen zusätzlich feinere Äste prozedural zu erzeugen und Blätter zu generieren [Liv+10]. Da

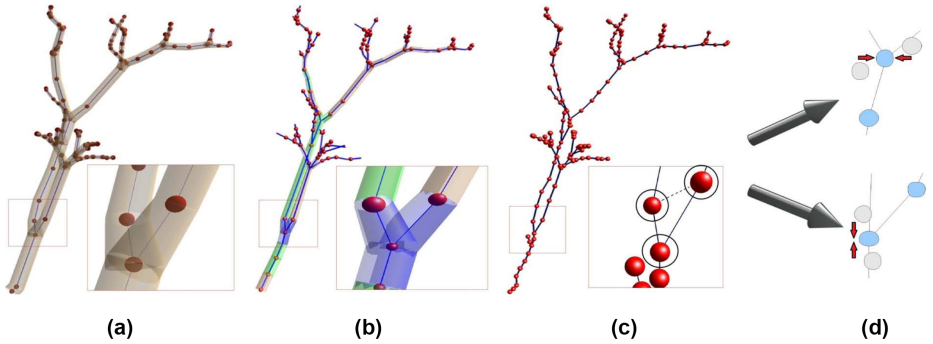

Abbildung 2.11: Schritte der Skelettbereinigung. Verallgemeinerte Zylinder mit berechneten Astradien (a); Identifizierung der zu entfernenden Kanten durch Berechnung der Schnittvolumen (b); Entfernen einer Kante durch Verschmelzen der Vertices (c), wobei es zwei Möglichkeiten gibt (d) (Quelle: [Liv+10])

diese Schritte keinen Einfluss auf die Extraktion des Skeletts haben, werden sie in dieser Arbeit nicht weiter erläutert.

3 Implementierung

Im Folgenden wird die Implementierung der ausgewählten Skelettierungsalgorithmen für Baum-Punktwolken thematisiert. Zunächst werden allgemeine Informationen zur technischen Basis gegeben, einschließlich der verwendeten Bibliotheken, Informationen zur grundlegenden Softwarearchitektur sowie der eingesetzten Entwicklungsumgebung (**Abschnitt 3.1**). Des Weiteren wird ein Untermodul vorgestellt, das dem Import und der Verarbeitung von Punktwolkendaten dient (**Abschnitt 3.2**). Anschließend wird die Umsetzung des voxelbasierten (**Abschnitt 3.3**) sowie des graphenbasierten Skelettierungsalgorithmus (**Abschnitt 3.4**) im Detail erläutert.

3.1 Technische Basis

Die Skelettierungsalgorithmen wurden in Form eines Add-ons für die 3D-Modellierungssoftware Blender implementiert. Blender ist ein äußerst vielseitiges Werkzeug mit einer umfassenden Palette an Funktionen zur Modellierung von 3D-Objekten, Erstellung von Animationen, Simulationen und visuellen Effekten, zum Rendering und mehr [Ble23]. Darüber hinaus bietet Blender eine Programmierschnittstelle (*API*), die auf Basis der Programmiersprache Python eine Erweiterung der Software durch *Scripting* ermöglicht. Diese Schnittstelle erleichtert nicht nur die prozedurale Generierung von 3D-Objekten, sondern ermöglicht auch die Erweiterung der grafischen Benutzeroberfläche (Abbildung 3.1), wodurch die Konfiguration von Eingabeparametern für die implementierten Algorithmen vereinfacht werden kann.

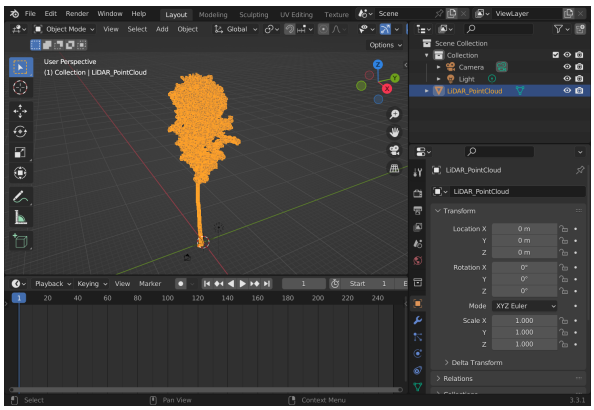

Abbildung 3.1: Grafische Benutzeroberfläche von Blender

Es sei darauf hingewiesen, dass bei der Umsetzung bewusst auf die Verwendung einer existierenden Code-Basis verzichtet wurde. Diese Entscheidung begründet sich einerseits durch die Freiheit der Implementierung, die eine gezielte Abstimmung der Algorithmen an die gegebenen Anforderungen erlaubt. Andererseits ermöglicht die eigenständige Implementierung, ein umfassendes Verständnis der Verfahren zu erlangen, wodurch fundierte Aussagen über die Ergebnisse der Vergleichsanalyse getroffen werden können.

Für die Implementierung der Algorithmen in Python wurden verschiedene Bibliotheken verwendet. Eine zentrale Rolle spielt die Bibliothek *NumPy* (im Code `np`), die numerische Berechnungen

und mathematische Funktionen auf multidimensionalen Arrays ermöglicht. Durch ihre maschinennahe Implementierung und optimierten Code ist eine effiziente Verarbeitung großer Datenmengen möglich [Num23]. Eine weitere verwendete Bibliothek ist *SciPy*, die auf NumPy aufbaut und zusätzliche Funktionen zur Lösung von Problemen aus unterschiedlichen Themenbereichen der Mathematik und Statistik zur Verfügung stellt. Beispiele hierfür sind Algorithmen zur Optimierung, Integration, Clusteranalyse, multidimensionale Bildverarbeitung oder die Einführung räumlicher Datenstrukturen [Sci23]. Zur Modellierung und Analyse von Graphen wurde die Bibliothek *NetworkX* (im Code `nx`) eingesetzt. Sie bietet abstrakte Datenstrukturen, um gerichtete oder ungerichtete Graphen zu modellieren, und ermöglicht eine effiziente Manipulation und Analyse komplexer Netzwerke [Net23]. Um die zur Verfügung gestellten Datensätze zu importieren, was das Lesen spezieller Dateiformate erforderte, wurde die Bibliothek laspy genutzt [BM23]. Eine genauere Erläuterung erfolgt im nachfolgenden Kapitel.

Das Add-on wurde technisch als Python-Modul realisiert und zur Trennung von Verantwortlichkeiten weiter in verschiedene Untermodule unterteilt. Ein Untermodul „dependencies“ übernimmt die automatische Verwaltung und Installation der benötigten Bibliotheken. Um Informationen in der Konsole auszugeben, wie beispielsweise den Ladefortschritt während längerer Prozeduren, wurden Funktionen in einem Untermodul „logger“ implementiert. Die Logik zum Importieren der Punktwolkendaten wurde in ein separates Untermodul „las_importer“ ausgelagert. Schließlich wurden die ausgewählten Skelettierungsalgorithmen zusammen mit allen Hilfsfunktionen in einem Untermodul „skeletonization“ implementiert.

3.2 Datenimport und -verarbeitung

Wie bereits erwähnt, ist die Ausgabe eines LiDAR-Scans eine Punktwolke, die eine Menge von Punkten im dreidimensionalen Raum darstellt. Zur Speicherung dieser Punktwolkendaten wurden spezielle Dateiformate entwickelt, die auf die spezifischen Anforderungen und Eigenschaften von Punktwolken zugeschnitten sind. Ein weit verbreitetes und standardisiertes Format ist LAS (*LASer*), das eine effiziente Speicherung und den Austausch von LiDAR-Punktwolken ermöglicht [ASP19]. Mit Hilfe des Dateiformats LAZ (*LASzip*) lassen sich die Daten zudem verlustfrei komprimieren, um den Speicherplatzbedarf zu reduzieren [Ise13].

Die zur Verfügung gestellten Datensätze, die zum Testen der Algorithmen verwendet werden sollen, liegen im LAZ-Dateiformat vor. Um die Handhabung der LAZ-Dateien und die Extraktion der Punktwolkendaten zu erleichtern, wurde die Python-Bibliothek *laspy* verwendet. Laspy wurde speziell für die Verarbeitung von LAS- und LAZ-Dateien entwickelt und bietet Funktionen zur einfachen Ein- und Ausgabe von Punktwolkendaten in den genannten Formaten [BM23].

Zum Importieren der Datensätze in Blender wurde eine Import-Funktion implementiert, die LiDAR-Daten im LAS- oder LAZ-Dateiformat unterstützt. Wenn eine entsprechende Datei über diese Funktion ausgewählt wird, werden die Daten mithilfe von laspy eingelesen. Laspy liefert separate NumPy-Arrays für die einzelnen Attributwerte der Punkte, einschließlich der x-, y- und z-Koordinaten ihrer Positionen. Anhand dieser Daten wird ein Mesh-Objekt generiert, das die Punktwolke in der Szene repräsentiert. Für jeden Punkt der Punktwolke wird dabei ein Vertex im Mesh erzeugt. Das

Mesh ermöglicht eine manuelle Vorverarbeitung der Punktwolke vor der Ausführung des Skelettierungsalgorithmus, beispielsweise durch das manuelle Segmentieren von Teilen der Punktwolke oder das Verschieben oder Löschen einzelner Punkte.

Es ist zu beachten, dass die Punktwolke nicht zwangsläufig um den Ursprung des Koordinatensystems zentriert ist. In Fällen, in denen die Punktwolke weit vom Ursprung entfernt liegt, können aufgrund der begrenzten Präzision des Vertex-Speicherformats signifikante Rundungsfehler bei der Positionierung der Punkte auftreten. Um diese Rundungsfehler zu umgehen, wurde eine zusätzliche Option implementiert, die es ermöglicht, die Punktwolke vor der Erzeugung des Meshes zu zentrieren. Hierbei wird die minimale Begrenzungsbox betrachtet, die alle Punkte umfasst. Die Punkte werden so verschoben, dass der Mittelpunkt der Grundfläche der minimalen Begrenzungsbox im Ursprung liegt.

Abschließend wurde eine weitere Funktionalität implementiert, die es mittels einer Schwellwertoperation ermöglicht, die Punkte anhand gespeicherter Abweichungswerte (*deviation*) zu filtern. Es wurde festgestellt, dass feine Details in der Punktwolke, wie beispielsweise Blätter und dünne Zweige, oft eine niedrige Genauigkeit aufweisen und zu erhöhtem Rauschen in der Punktwolke führen können. Durch den Einsatz des Filters besteht die Möglichkeit, dieses Rauschen effektiv zu reduzieren.

3.3 Implementierung des voxelbasierten Algorithmus

In diesem Abschnitt wird die Implementierung des voxelbasierten Skelettierungsalgorithmus erläutert. Die Implementierung basiert im Wesentlichen auf den Beschreibungen von Gorte und Pfeifer [GP04] in ihrer Publikation. Es wurden darüber hinaus jedoch Anpassungen am Algorithmus vorgenommen, um die Effizienz des Algorithmus und die Qualität der Ergebnisse weiter zu verbessern. Im Weiteren wird die Implementierung entsprechend der Abfolge der fünf Algorithmus-Schritte (i–v) erläutert (vgl. Kapitel 2.3.2) und anhand kurzer Codeausschnitte veranschaulicht. Zur besseren Verständlichkeit wurden die Codeausschnitte aus der eigentlichen Implementierung reduziert oder abgeändert. Der Ausgangspunkt ist eine Punktwolke, die in Form eines Meshes gegeben ist.

i Konvertierung der Punktdaten in Voxeldaten

Zunächst wurde eine Funktion implementiert, die ein Mesh mit n Vertices ausliest und die Vertex-Positionen, welche als Punktwolke interpretiert werden, in ein zweidimensionales Array der Form $n \times 3$ lädt. Optional werden die Positionen mit Hilfe der World-Matrix des Mesh-Objekts transformiert, um eine Berechnung des Skeletts im globalen Koordinatensystem zu ermöglichen. In der Implementierung wird das Array mit den Punkt-Positionen als `points` bezeichnet.

Anhand der minimalen Begrenzungsbox der Punktwolke und mit Hilfe der gewünschten Auflösung des Voxelrasters werden die Größe und Positionierung einer Voxeldomäne berechnet. Diese Voxeldomäne ermöglicht eine optimale Abdeckung der Punktwolke und eine gleichmäßige Unterteilung der drei Dimensionen, sodass die Voxel frei von Verzerrungen sind. Dies kann dazu führen, dass die Größe der Voxeldomäne minimal über die Begrenzungsbox hinausgeht. Als Referenz für die Bestim-

mung der Auflösung dient eine Zellengröße `cell_size`, die der Benutzer definiert. Es ist wichtig, dass die Wahl einer geeigneten Zellengröße unter Berücksichtigung der in Kapitel 2.3.2 beschriebenen Faktoren erfolgt. Die genauen Berechnungen zur Bestimmung der Voxeldomäne erfolgen in der Funktion `compute_voxel_domain` (s. Zeilen 6–14) im Codeausschnitt 3.1. Die Voxeldomäne wird durch die drei Werte `domain_origin` (Position eines Eckpunkts der Domäne im Raum, in $\mathbb{R}^3$), `domain_size` (räumliche Größe der Domäne, in $\mathbb{R}^3$) und `domain_resolution` (Anzahl der Voxel in jede Dimension, in $\mathbb{N}^3$) definiert.

Listing 3.1: Bestimmung der Voxeldomäne

```
1  def compute_bounding_box(points):
2      bounding_min = np.amin(points, axis=0)
3      bounding_max = np.amax(points, axis=0)
4      return bounding_min, bounding_max
5
6  def compute_voxel_domain(points, cell_size):
7      # Minimale Begrenzungsbox der Punktwolke bestimmen
8      bounding_min, bounding_max = compute_bounding_box(points)
9
10     # Definition der Voxeldomäne
11     domain_resolution = np.ceil((bounding_max - bounding_min) /
           cell_size).astype(int)
12     domain_size = domain_resolution * cell_size
13     domain_origin = 0.5 * (bounding_max + bounding_min - domain_size)
14     return domain_origin, domain_size, domain_resolution
```

Mit den gegebenen Informationen zur Voxeldomäne können die Positionen der Punkte anschließend durch Quantisierung in diskrete Voxelkoordinaten umgewandelt werden (s. Zeile 17). Mit Hilfe der NumPy-Funktion zur Berechnung eines multidimensionalen Histogramms werden die Häufigkeiten der Voxelkoordinaten unter allen Punkten erfasst, um ein Voxelraster zu erstellen (s. Zeile 20). Das Ergebnis ist ein dreidimensionales Array `voxels`, das das Voxelraster repräsentiert und für jeden Voxel die Anzahl der enthaltenen Punkte speichert. Das Voxelraster ist in der Regel dünn besetzt, d. h. viele Voxel haben den Wert 0. Es existieren Datenstrukturen, die eine effiziente Speicherung dünnbesetzter Matrizen ermöglichen, wie sie beispielsweise in der originalen Implementierung von Gorte und Pfeifer verwendet wurden [GP04]. In Anbetracht des zeitlichen Rahmens dieser Arbeit wurde jedoch eine einfache Implementierung mit NumPy-Array gewählt.

Listing 3.2: Transformieren der Punkte in ein Voxelraster

```
15 def create_voxel_raster_from_points(points, domain_origin, domain_resolution,
       cell_size):
16     # Quantisierung der Punkt-Positionen liefert Voxel-Index-Array
17     voxel_indices = np.floor((points - domain_origin) / cell_size).astype(int)
18
19     # Häufigkeiten der quantisierten Punkt-Positionen sammeln
20     voxels, _ = np.histogramdd(voxel_indices, bins=domain_resolution)
21     return voxels.astype(int)
```

ii Datenvorverarbeitung

Nach der Generierung des Voxelrasters erfolgt die Vorverarbeitung der Daten. Gorte und Winterhalder (2004) schlagen die Anwendung eines Schwellwertfilters basierend auf der Dichte der Punktwolke vor [GW04]. Dieser Filter dient dazu, Rauschen und unzureichend repräsentierte Merkmale herauszufiltern. Wie zuvor erwähnt, kann die Anzahl der Punkte in einem Voxel als Maß für die Dichte der Punktwolke innerhalb des Voxels interpretiert werden. Um den Filter unabhängig von der Auflösung des Voxelrasters zu machen, wird der Schwellwert anhand des Voxelvolumens und einer vom Benutzer festgelegten Mindestdichte `min_density` berechnet (s. Zeile 24). Voxel, die diesen Schwellwert unterschreiten, werden auf den Wert 0 gesetzt (s. Zeile 25).

Listing 3.3: Dichtebasierter Schwellwertfilter

```
22 def density_threshold_filter(voxels, cell_size, min_density):
23     cell_volume = cell_size ** 3
24     min_value = math.ceil(min_density * cell_volume)
25     voxels[voxels < min_value] = 0
26     return voxels
```

Um isolierte Voxel und kleine Gruppen zusammenhängender Voxel zu filtern, wird mit Hilfe von Funktionen aus dem SciPy-Modul für multidimensionale Bildverarbeitung („scipy.ndimage“) ein Labeling, also eine Kennzeichnung, der zusammenhängenden Komponenten durchgeführt (s. Zeilen 29–30). Anschließend werden die Komponenten hinsichtlich ihrer Größe (Anzahl der Voxel, aus denen sie bestehen) analysiert (s. Zeile 33). Komponenten, deren Größe unterhalb einer festgelegten Mindestgröße `min_size` liegt, werden maskiert und entfernt, indem ihre Voxel auf den Wert 0 gesetzt werden (s. Zeilen 36–37).

Listing 3.4: Entfernen von isolierten Voxeln und kleinen Gruppen zusammenhängender Voxel

```
27 def filter_isolated_components(voxels, min_size):
28     # Labeling zusammenhängender Komponenten
29     kernel = ndimage.generate_binary_structure(3,3)
30     label_ids, num_labels = ndimage.label(voxels, structure=kernel)
31
32     # Größen der Komponenten berechnen (Anzahl Voxel)
33     component_sizes = np.array(ndimage.sum_labels(voxels.astype(bool),
           label_ids, range(num_labels+1)), dtype=int)
34
35     # Isolierte Komponenten maskieren und filtern
36     id_mask = (component_sizes < min_size)
37     voxels[id_mask[label_ids]] = 0
38     return voxels
```

Für die nachfolgenden Schritte ist die genaue Anzahl der Punkte in den Voxeln nicht mehr erforderlich. Um die Leistung der folgenden Operationen zu steigern, wird das Integer-Voxelraster in ein Boolean-Voxelraster umgewandelt. Alle Voxel mit einem Wert größer als 0 bekommen den Wert `True` zugewiesen.

Der abschließende Schritt der Datenvorverarbeitung stellt das Schließen von Lücken und hohler Äste dar. Für beide Operationen werden geeignete strukturierende Elemente (*kernel*) erzeugt. Beim Schließen von Lücken wird je nach Benutzerwahl ein strukturierendes Element basierend auf einem Nachbarschaftstyp (6-, 18- oder 26-adjazent) oder einer Kugel mit einem festgelegten Radius verwendet. Zum Schließen der hohlen Äste wird ein horizontal ausgerichtetes strukturierendes Element in Form einer Scheibe eingesetzt. Für die Generierung einer Kugel (oder Scheibe) als strukturierendes Element mit einem Radius von r (in Anzahl der Voxel) wird ein dreidimensionales Array der Größe $s \times s \times s$ (bzw. $s \times s \times 1$), wobei $s = 2r + 1$, erstellt. Dabei werden alle Elemente, die innerhalb einer Kugel (bzw. eines Kreises) mit dem Radius r liegen, auf den Wert `True` gesetzt. Zum Schließen der Lücken wird die binäre Closing-Operation aus dem Modul „scipy.ndimage“ angewendet.

iii Skelettierung durch Verdünnen

Zur Reduktion des Voxelvolumens auf eine Voxelbreite wurde die Verdünnungsoperation von Palágyi und Kuba [PK99] implementiert. Die Implementierung dieser Methode stellt eine besondere Herausforderung dar, da sie eine effiziente Verarbeitung großer Datenmengen erfordert.

Wie bereits in Kapitel 2.3.1 erläutert, erfolgt die Identifizierung der zu löschenden Voxel in einer Subiteration durch das Überprüfen der Voxelwerte anhand einer Reihe von Templates der Größe $3\times3\times3$. Palágyi und Kuba definieren 7 Basis-Templates [PK99], die durch zusätzliche Spiegelung an drei diagonalen Ebenen 20 verschiedene Templates pro Subiteration und durch die unterschiedliche Drehungen der 8 diagonalen Hauptrichtungen insgesamt 160 Templates ($8 \cdot 20$) für alle 8 Subiterationen ergeben [PK99]. Diese werden zur Laufzeit dynamisch durch die beschriebenen Spiegelungen und Drehungen aus den 7 Basis-Templates generiert.

Jedes Template beschreibt die Anordnung der Voxel anhand von drei Regeln [PK99], die in der Implementierung durch numerische Werte codiert werden. Voxel, die leer sein müssen (Voxelwert = `False`), werden im Template mit einer 0 gekennzeichnet (Regel 1), wohingegen solche, die nicht leer sein dürfen (Voxelwert = `True`), mit einer 1 markiert werden (Regel 2). Eine 2 gruppiert mehrere Voxel, unter denen mindestens ein Voxel gegeben sein muss (Regel 3). Voxel an Stellen einer -1 können beliebig sein und werden nicht berücksichtigt.

Eine intuitive Vorgehensweise wäre zunächst eine Schleife über die Voxel, wobei für jedes Voxel die entsprechenden Templates geprüft werden. In der vorliegenden Implementierung wurde dieser Ansatz jedoch umgekehrt, um unter Berücksichtigung der gegebenen Umstände (Voxelraster ist als NumPy-Array gegeben) eine optimale Effizienz durch Anwenden von Minimum- und Maximumfilter aus dem Modul „scipy.ndimage“ zu erzielen. In jeder Subiteration wird jedes der 20 Templates (s. Variable `templates` in Zeile 40) individuell auf das Voxelraster angewendet. Für jedes Template werden nacheinander die drei Regeln aus dem Template mit Hilfe der Variable `rule_footprint` maskiert (s. Zeilen 44, 51 und 57) und als Maske (`footprint`) für die Anwendung der Minimum- und Maximumfilter verwendet (s. Zeilen 46, 53 und 59). Die Variable `rule_mask` maskiert alle Voxel, die eine bestimmte Regel erfüllen. Durch Konjunktion dieser Zwischenergebnisse in der Variable `template_mask` (s. Zeilen 48, 54 und 60) werden alle Voxel maskiert, die zu einem bestimmten Template passen. Durch eine disjunktive Verknüpfung der Ergebnisse mehrerer Templates in der Variable

`mask` werden alle gesuchten Voxel gesammelt (s. Zeile 62) und anschließend gelöscht (bzw. auf den Wert `False` gesetzt, s. Zeile 63).

Listing 3.5: Ausschnitt aus einer Subiteration der Verdünnungsoperation

```
39 mask.fill(False)
40 for template in templates:
41     template_mask.fill(True)
42
43     # Regel 1: False-Werte prüfen
44     np.equal(template, 0, out=rule_footprint)
45     if np.any(rule_footprint):
46         ndimage.maximum_filter(voxels, footprint=rule_footprint,
               mode='constant', output=rule_mask)
47         np.logical_not(rule_mask, out=rule_mask)
48         np.logical_and(template_mask, rule_mask, out=template_mask)
49
50     # Regel 2: True-Werte prüfen
51     np.equal(template, 1, out=rule_footprint)
52     if np.any(rule_footprint):
53         ndimage.minimum_filter(voxels, footprint=rule_footprint,
               mode='constant', output=rule_mask)
54         np.logical_and(template_mask, rule_mask, out=template_mask)
55
56     # Regel 3: True-Werte in Gruppen prüfen
57     np.equal(template, 2, out=rule_footprint)
58     if np.any(rule_footprint):
59         ndimage.maximum_filter(voxels, footprint=rule_footprint,
               mode='constant', output=rule_mask)
60         np.logical_and(template_mask, rule_mask, out=template_mask)
61
62     np.logical_or(mask, template_mask, out=mask)
63 voxels[mask] = False
```

Die obige Prozedur wird nacheinander für jede der 8 Subiterationen ausgeführt und es wird eine Information darüber zurückgegeben, ob Voxel innerhalb einer Subiteration gelöscht wurden. Das Verfahren über alle 8 Subiterationen wird iterativ fortgesetzt und terminiert, wenn keine weiteren Voxel mehr entfernt wurden [PK99].

iv Extraktion des Skeletts

Um das Skelett als Graph zu extrahieren, wird zunächst ein ungerichteter Graph erzeugt, der Paare benachbarter Voxel miteinander verbindet. Dies erfolgt durch Anwendung eines sogenannten Floodfill-Algorithmus von dem Voxel aus, das die Wurzel des Skeletts repräsentiert (wird im nächsten Schritt identifiziert). Durch diese Vorgehensweise wird sichergestellt, dass nur zusammenhängende Voxel, die zum Skelett gehören, in den Graphen aufgenommen werden. Der Graph wird als NetworkX-Graph repräsentiert, wobei der Knoten mit dem Index 0 als Wurzel für das Skelett verwendet wird. Jeder Knoten erhält die zugehörigen Voxelkoordinaten als Attribut. Wie in Kapitel 2.3.2 beschrie-

ben, werden die Kanten des Graphen entsprechend der Nachbarschaftsbeziehungen der Voxel mit den Werten 3, 4 oder 5 gewichtet [GP04].

Die Extraktion des Skeletts erfolgt basierend auf dem Algorithmus von Dijkstra [GP04] in der Funktion `dijkstra_spanning_tree` (s. Codeausschnitt 3.6). Durch Anwendung des Dijkstra-Algorithmus mit Hilfe von NetworkX auf dem vollständigen Graphen `graph` und mit dem Wurzelknoten `root` als Zielknoten wird eine Liste `predecessors` generiert (s. Zeile 70), die die direktesten Wege von jedem Knoten zum Wurzelknoten beschreibt. Daraus wird anschließend ein aufspannender Baum `spanning_tree` generiert (s. Zeilen 71–74), der das Skelett repräsentiert.

Listing 3.6: Konstruktion eines minimalen Spannbaums mittels Dijkstra

```
64 def dijkstra_spanning_tree(graph, root)
65     # Graph mit Knoten erzeugen
66     spanning_tree = nx.DiGraph()
67     spanning_tree.add_nodes_from(graph.nodes(data=True))
68 
69     # Minimalen Spannbaum mit Hilfe von Dijkstra extrahieren
70     predecessors, _ = nx.dijkstra_predecessor_and_distance(graph, root)
71     for u, vs in predecessors.items():
72         if u == root or len(vs) == 0: continue
73         edge_data = graph.get_edge_data(u, vs[0], {})
74         out.add_edge(vs[0], u, **edge_data)
75     return spanning_tree
```

v Segmentierung zusammenhängender Komponenten

Abschließend wird das Problem der nicht zusammenhängenden Komponenten im Voxelraster behandelt. Gemäß der Algorithmusbeschreibung von Gorte und Pfeifer werden zunächst die Skelette für alle zusammenhängenden Komponenten erzeugt und anschließend die „nicht relevanten“ Skelette verworfen [GP04]. In der vorliegenden Implementierung erfolgt jedoch bereits vor der Skelettierung die Segmentierung und Auswahl der zusammenhängenden Komponenten. Dadurch muss das Skelett lediglich für die relevanten Komponenten extrahiert werden.

Zuerst werden die zusammenhängenden Komponenten ähnlich wie beim Labeling-Verfahren, das zur Filterung isolierter Komponenten während der Datenvorverarbeitung verwendet wurde (vgl. 3.4), gekennzeichnet. Die Komponenten werden nach ihrer Größe absteigend sortiert und es werden nur diejenigen ausgewählt, die für die Skelettierung relevant sind. Der Benutzer kann eine maximale Anzahl von Komponenten entsprechend der Anzahl der gesuchten Bäume festlegen, sowie eine Mindestgröße der Komponenten. Für jede ausgewählte Komponente wird anschließend das Skelett extrahiert (wie im vorherigen Schritt beschrieben), wobei das unterste Voxel als Wurzel des Skeletts ausgewählt wird.

Der Skelettierungsalgorithmus kann mit Hilfe eines Operators in Blender ausgeführt werden. Über eine Dialogbox können Parameterwerte für den Algorithmus konfiguriert werden (s. Abbildung A.1

(a) im Anhang A.1). Das Skelett kann wahlweise als Mesh oder Kurve generiert werden, wobei die Vertices des Skeletts durch gerade Kanten verbunden werden.

3.4 Implementierung des graphenbasierten Algorithmus

Nach den Ausführungen zum voxelbasierten Skelettierungsalgorithmus folgt nun eine Beschreibung der Implementierung des graphenbasierten Algorithmus für die Skelettierung von Baum-Punktwolken. Die Implementierung basiert auf den Beschreibungen des Papers von Livny et al. [Liv+10], wurde jedoch an einigen Stellen an die spezifischen Anforderungen dieser Arbeit angepasst oder erweitert. Die Struktur der Erläuterung orientiert sich an den acht Schritten (i–viii) des Algorithmus (vgl. Kapitel 2.4.2) und es werden vereinfachte Codeausschnitte präsentiert, um ein besseres Verständnis zu ermöglichen. Die Eingabe des Algorithmus ist erneut eine Punktwolke, die in Form eines Meshes vorliegt.

i Identifizierung der Bäume in der Punktwolke

Die Vertex-Positionen des Meshes, das die Punktwolke repräsentiert, werden zunächst mit Hilfe derselben Funktion, die beim voxelbasierten Algorithmus verwendet wurde, geladen. Es ist zu beachten, dass die Anzahl der Punkte in der Punktwolke einen erheblichen Einfluss auf die Laufzeit des Algorithmus hat (dies wird im Kapitel 4 genauer untersucht). Im Entwicklungsprozess hat es sich als praktikabel erwiesen, die Anzahl der Punkte zu reduzieren, da dies sowohl zu einer verkürzten Laufzeit als auch zu verbesserten Ergebnissen führen kann.

Aus diesem Grund wurden Funktionen implementiert, die ein automatisches Downsampling der Punktwolke ermöglichen. Dem Benutzer stehen zwei Downsampling-Methoden zur Auswahl: Eine einfache Zufallsauswahl, bei der eine Zufallsstichprobe aus allen Punkten ausgewählt wird, und eine voxelbasierte Zufallsauswahl, bei der die Punktwolke mit Hilfe eines Voxelrasters räumlich unterteilt wird und pro Voxel eine Zufallsstichprobe ausgewählt wird. Die zweite Methode ermöglicht eine gleichmäßigere räumliche Verteilung der ausgewählten Punkte. Für beide Methoden kann der Benutzer den Downsampling-Faktor (*ratio*) sowie einen Seed zur Steuerung der Pseudo-Zufallsauswahl festlegen und für die voxelbasierte Methode eine Voxelgröße definieren. Die Punkt-Positionen der reduzierten Punktwolke werden in einem zweidimensionalen Array `points` gespeichert.

Um eine effiziente Verarbeitung der Punkte zu ermöglichen, werden sie in einem k-d-Baum organisiert. Ein k-d-Baum ist ein multidimensionaler binärer Suchbaum, der verschiedene Arten von Abfragen effizient bearbeiten kann, beispielsweise die Suche nach Punkten innerhalb eines bestimmten Bereichs oder die Suche nach den k nächsten Nachbarn eines Punktes [Ben75]. Der k-d-Baum wird unter Zuhilfenahme der gleichnamigen Klasse im Modul „scipy.spatial“ erzeugt (s. Zeile 2 im Codeausschnitt 3.7).

Zur Identifizierung der Bäume in der Punktwolke werden die Punkte zunächst auf die Grundebene projiziert, indem die dritte Koordinate der Punkt-Positionen, welche sich in diesem Fall auf die vertikale Achse bezieht, verworfen wird (s. Zeile 5). Dadurch wird das Array `points` mit den Positionen im dreidimensionalen Raum ($\mathbb{R}^3$) in das Array `points_projected` umgewandelt, das

die Positionen in der zweidimensionalen Grundebene ($\mathbb{R}^2$) enthält. Anschließend wird der k-means-Algorithmus aus dem Modul „scipy.cluster.vq“ auf den projizierten Punkten angewendet, wobei die Anzahl der Bäume `num_trees` durch den Benutzer spezifiziert wird. Der k-means-Algorithmus ist ein Verfahren zur Clusteranalyse, bei dem eine gegebene Datenmenge (hier die projizierte Punktwolke) in k Gruppen („Cluster“) unterteilt wird [Mac67]. Die Ausgabe sind die Cluster-Schwerpunkte (bzw. Cluster-Mittelpunkte), die in der Variable `tree_centroids` gespeichert werden (s. Zeile 8). Nach Rücktransformation der Mittelpunkte in den dreidimensionalen Raum durch Anhängen einer 0 als dritte Koordinate (s. Zeile 9) wird mit Hilfe des k-d-Baums für jeden Cluster-Mittelpunkt der nächstgelegene Punkt in der Punktwolke gesucht und als Wurzel-Vertex des jeweiligen Baums ausgewählt (s. Zeile 12). Durch eine abschließende `unique`-Operation werden potenzielle Duplikate entfernt und somit sichergestellt, dass jeder Wurzel-Vertex eindeutig einem Baum zugewiesen ist (s. Zeile 13).

Listing 3.7: Identifizierung der Wurzel-Vertices

```
1  # KD-Baum für Punktwolke generieren
2  points_kdtree = spatial.KDTree(points)
3
4  # Projektion der Punkte auf die Grundebene
5  points_projected = points[:,:2]
6
7  # Cluster-Mittelpunkte bestimmen
8  tree_centroids, _ = vq.kmeans(points_projected, num_trees)
9  tree_centroids = np.hstack(( tree_centroids,
       np.zeros((tree_centroids.shape[0],1)) ))
10
11 # Wurzel-Vertices der Skelette ermitteln
12 _, bsg_root_nodes = points_kdtree.query(tree_centroids)
13 bsg_root_nodes = np.unique(bsg_root_nodes)
```

Die automatische Segmentierung der Baum-Punkte von dem Boden, wie im Verfahren beschrieben, wird hier nicht durchgeführt, da Überlappungen der Objekte nach der Projektion zusätzliche Herausforderungen darstellen, deren Lösung über den zeitlichen Rahmen dieser Arbeit hinausgeht. Das Auslassen dieses Schrittes stellt jedoch kein Problem dar, da die verwendeten Punktwolken bereits segmentiert sind.

ii Initiale Konstruktion des Skeletts

Der beschriebene Algorithmus sieht vor, dass das Skelett anhand eines vollständigen Graphen konstruiert wird, der alle Punkte der Punktwolke miteinander verbindet. Es ist zu beachten, dass die Anzahl der Kanten in einem vollständigen Graphen quadratisch mit der Anzahl der Knoten zunimmt ($k \cdot (k-1)/2$ Kanten bei k Knoten). Während beispielsweise bei 100 Knoten 4.950 Kanten erforderlich sind, um alle Knoten miteinander zu verbinden, steigt diese Zahl bei 100.000 Knoten auf knapp unter 5 Milliarden an. Da dies die Laufzeit des Algorithmus signifikant beeinträchtigen würde, ist die Konstruktion eines vollständigen Graphen nicht praktikabel. Stattdessen werden nur Punkte miteinander

verbunden, die in der Punktwolke eine enge räumliche Nachbarschaft aufweisen, da diese, wie bereits von Livny et al. angedeutet, mit erhöhter Wahrscheinlichkeit zu demselben Ast gehören [Liv+10].

Mit Hilfe des k-d-Baums werden Paare von Punkten gesucht, deren Abstand eine festgelegte Suchdistanz `max_search_radius` nicht überschreitet (s. Zeile 20 im Codeausschnitt 3.8). Der initiale Graph `initial_graph` wird mit allen Punkten als Knoten initialisiert, wobei unter Anwendung einer implementierten Hilfsfunktion die Positionen der Punkte als Knotenattribute zugewiesen werden (s. Zeilen 15–17). Die gefundenen Punkt-Paare werden als Kanten hinzugefügt, wobei diese mit dem quadrierten euklidischen Abstand der Punkte gewichtet werden und die Kantenlängen als zusätzliche Kantenattribute zugewiesen werden (s. Zeilen 21–24). Darüber hinaus wird ein temporärer Knoten hinzugefügt, der über 0-gewichtete Kanten mit den Wurzelknoten der Skelette verbunden wird (s. Zeilen 27–30).

Listing 3.8: Konstruktion des initialen Graphen

```
14 # Initialen Graph erzeugen und Knoten hinzufügen
15 initial_graph = nx.Graph()
16 initial_graph.add_nodes_from(range(len(points)))
17 set_node_attributes_from_list(initial_graph, points, 'position')
18
19 # Kanten basierend auf nächsten Nachbarn der Punkte erzeugen
20 edges = points_kdtree.query_pairs(max_search_radius, output_type='ndarray')
21 edge_weights = np.sum((points[edges[:,1]] - points[edges[:,0]])**2, axis=1)
22 edge_lengths = np.sqrt(edge_weights)
23 for edge, edge_weight, edge_length in zip(edges, edge_weights, edge_lengths):
24     initial_graph.add_edge(edge[0], edge[1], weight=edge_weight,
          length=edge_length)
25
26 # Temporären Knoten hinzufügen
27 tmp_node = len(points)
28 tmp_edges = ((tmp_node, u) for u in bsg_root_nodes)
29 initial_graph.add_node(tmp_node, position=np.zeros(3))
30 initial_graph.add_edges_from(tmp_edges, weight=0.0, length=0.0)
```

Mit Hilfe der in Abschnitt 3.3 definierten Funktion `dijkstra_spanning_tree` wird ausgehend vom temporären Knoten ein minimaler Spannbaum `mst` konstruiert (s. Zeile 32). Durch Auswahl aller Unterbäume, die von den zuvor identifizierten Wurzelknoten ausgehen, werden die Skelette (BSGs) segmentiert und der Liste `bsgs` zugewiesen (s. Zeilen 37–41). Des Weiteren wird für jedes Skelett aus dem initialen Graphen ein Teilgraph mit den zugehörigen Knoten extrahiert und für die spätere Verwendung in der Liste `bsg_initial_graphs` gespeichert (s. Zeilen 44–45).

Listing 3.9: Initialen Konstruktion der Skelette

```
31 # Minimalen Spannbaum erzeugen
32 mst = dijkstra_spanning_tree(initial_graph, tmp_node)
33
34 bsgs = []
35 bsg_initial_graphs = []
36 for root_node in bsg_root_nodes:
```

```
37      bsg_nodes = {root_node} | nx.descendants(mst, root_node)
38
39      # Skelett extrahieren
40      bsg = mst.subgraph(bsg_nodes).copy()
41      bsgs.append(bsg)
42
43      # Initialen Graph des Skeletts extrahieren
44      bsg_initial_graph = initial_graph.subgraph(bsg_nodes).copy()
45      bsg_initial_graphs.append(bsg_initial_graph)
```

iii Berechnung der Knotengewichtungen

Bei der Berechnung der Knotengewichtungen wird die Eigenschaft ausgenutzt, dass die Gewichtung eines Knotens aus der Summe der Gewichtungen seiner Kinder und der entsprechenden Kantenlängen resultiert. Es wurde eine Funktion implementiert, die das Skelett basierend auf einer Tiefensuche iterativ durchläuft. Dabei wird der zurückgelegte Weg mit Hilfe eines Stacks gespeichert, um ihn beim Aufsummieren der Gewichtungen zurückzuverfolgen. Aufgrund der begrenzten Rekursionstiefe und der zusätzlichen Kosten, die mit Funktionsaufrufen einhergehen, wurde bewusst von einem rekursiven Ansatz abgesehen.

iv Konstruktion eines globalen Orientierungsfeldes

Im nächsten Schritt wird das globale Orientierungsfeld durch Optimierung der Funktion $\Delta O(T_i) + \Delta E(T_i)$ (vgl. Formeln 2.1 und 2.2 in Abschnitt 2.4.2) berechnet. Livny et al. weisen darauf hin, dass eine exakte Lösung effizient durch Lösen eines linearen Gleichungssystems erzielt werden kann [Liv+10]. Aufgrund der großen Anzahl der Funktionsparameter stellte dies jedoch eine Herausforderung dar, die innerhalb des zeitlichen Rahmens, der für die Implementierung dieses Algorithmus vorgesehen war, nicht bewältigt werden konnte. Stattdessen wurde zur Minimierung der Funktion das Gradientenabstiegsverfahren (*gradient descent*) gewählt, das in einem Test mit anderen Optimierungsverfahren aus dem Modul „scipy.optimize“ (einschließlich lokaler Verfahren wie „L-BFGS-B“ oder „Powell“ sowie globalen Verfahren wie „Dual Annealing“ [The23]) als das effizienteste Verfahren identifiziert wurde. Die Umsetzung war problemlos möglich, da die beschriebenen Funktionen differenzierbar sind.

Zur Berechnung des Gradienten $\nabla(\Delta O(T_i)+\Delta E(T_i))$, der für das Gradientenabstiegsverfahren benötigt wird, erfolgt eine partielle Ableitung nach den gesuchten Orientierungen (bzw. den skalaren Komponenten der Orientierungen). Dabei werden die Knotengewichtungen c_v sowie die Vertex-Positionen v als konstant betrachtet. In der Implementierung kann aufgrund der geltenden Summenregel für Ableitungen gemäß den Funktionen in den Formeln (2.1) und (2.2) eine Schleife über alle Vertices erstellt werden, wobei in jedem Schleifendurchlauf nach den vorkommenden Orientierungen partiell abgeleitet wird.

Auf Basis dieser theoretischen Ansätze wurde zunächst eine Funktion `dode_func` implementiert, die die zu minimierende Funktion $\Delta O(T_i) + \Delta E(T_i)$ repräsentiert (s. Zeilen 47–52 im Codeausschnitt

3.10), sowie eine Funktion `dode_func_grad`, die anhand eines gegebenen Orientierungsfeldes den Gradienten von `dode_func` liefert (s. Zeilen 55–68). Um eine vereinfachte und effiziente Datenverarbeitung während des Optimierungsprozesses zu gewährleisten, werden die relevanten Informationen des Skeletts in linearen Datenstrukturen, speziell in Form von NumPy-Arrays, gespeichert. Jedem Vertex wird fortlaufend ein Index zugewiesen, der zur Adressierung der zugehörenden Vertex-Informationen in den Daten-Arrays dient. Die Variable `x` repräsentiert ein bestimmtes Orientierungsfeld, wobei die Orientierungen zeilenweise in einem zweidimensionalen Array organisiert sind. Das eindimensionale Array `parents` verweist für jeden Vertex auf seinen Eltern-Vertex. `node_weights` speichert die Gewichtungen der Knoten, c_v, und `edge_weights` enthält die vorab berechneten gemittelten Gewichtungen $(c_{v_p} + c_v)/2$. Die Kanten-Richtungen $e(v_p, v)/\|e(v_p, v)\|$ werden ebenfalls im Voraus berechnet und in `edge_dirs` gespeichert.

Listing 3.10: Implementierung der Funktion $\Delta O(T_i) + \Delta E(T_i)$ und Berechnung ihres Gradienten

```
46 # Optimierte Funktion
47 def dode_func(x, parents, node_weights, edge_weights, edge_dirs):
48     do = np.sum(
49         (edge_weights * np.linalg.norm(x[parents] - x, axis=1))**2 )
50     de = np.sum(
51         (node_weights * np.linalg.norm(x - edge_dirs, axis=1))**2 )
52     return do + de
53
54 # Gradient der optimierten Funktion
55 def dode_func_grad(x, parents, node_weights, edge_weights, edge_dirs):
56     x_grad = np.zeros(x.shape)
57     for v in range(x.shape[0]):
58         p = parents[v]
59         ew2 = 2 * edge_weights[v]**2
60         nw2 = 2 * node_weights[v]**2
61         dodp = ew2 * (x[p] - x[v])
62         dedv = nw2 * (x[v] - edge_dirs[v])
63
64         # Partielle Ableitungen
65         x_grad[v] += -dodp # dO/do_v
66         x_grad[p] += dodp  # dO/do_v_p
67         x_grad[v] += dedv  # dE/do_v
68     return x_grad
```

Das Gradientenabstiegsverfahren wurde in einer Funktion `gradient_descent` implementiert, wie im vereinfachten Codeausschnitt unten dargestellt. Ausgehend von einem Startvektor `x0` (der auch eine Matrix sein kann) wird iterativ mit Hilfe des Gradienten aus `func_grad` (s. Zeile 79) die Funktion `func` minimiert (s. Zeile 80). `args` erlaubt die Übergabe zusätzlicher Parameterwerte an die Funktionen zur Berechnung des Fehlers und des Gradienten. Die Anzahl der Iterationen wird durch `max_iter` limitiert und `rate` steuert die Schrittweite. Mit Hilfe von `atol` und `rtol` lassen sich Abbruchbedingungen als Schwellwerte für die absolute und relative Änderungsrate spezifizieren.

Listing 3.11: Implementierung des Gradientenabstiegsverfahrens

```
69 def gradient_descent(func, func_grad, x0, args, rate, max_iter, atol, rtol):
70     n = x0.size
71     x = x0.copy()
72
73     y = y0 = func(x, *args)
74     previous_y = 0
75     for i in range(max_iter):
76         previous_y = y
77
78         # Absteigen
79         x_grad = func_grad(x, *args)
80         x -= (rate / n) * x_grad
81
82         # Abbruchbedingungen prüfen
83         y = func(x, *args)
84         y_change = y - previous_y
85         if atol is not None and np.abs(y_change) < atol: break
86         if rtol is not None and y0 != 0.0 and np.abs(y_change/y0) < rtol: break
87     return x
```

Zur Konstruktion des Orientierungsfeldes werden zunächst die oben beschriebenen Datenstrukturen aufgebaut und ein initialer Startwert `x0` anhand der Kantenrichtungen ausgewählt und geglättet. Anschließend erfolgt die Optimierung des Orientierungsfeldes durch Anwendung des Gradientenabstiegsverfahren `gradient_descent` auf Basis der definierten Funktionen `dode_func` und `dode_func_grad`. Die Optimierung erfolgt hierbei in zwei Phasen: In der ersten Phase wird das Orientierungsfeld für Teilbäume der dominantesten Äste des Skeletts lokal optimiert. Die niedrigen Knotengewichtungen bei kurzen Ästen führen zu geringeren partiellen Ableitungen ihrer Orientierungen, verglichen mit dem Stamm des Baums. Durch Vergrößern der Schrittweite im Rahmen einer lokalen Optimierung wird die Konstruktion des Orientierungsfelds hier beschleunigt. In der zweiten Phase wird das Orientierungsfeld des gesamten Skeletts durch globale Optimierung verfeinert. Eine optimale Schrittweite für die Optimierungsprozesse, die sowohl eine schnelle Minimierung als auch ein stabiles System gewährleistet, wurde experimentell ermittelt. Die Ausgabe ist ein geglättetes Orientierungsfeld, dessen Orientierungen den Knoten des Skelett (NetworkX-Graph) als Attribute zugewiesen werden.

v Globale Verfeinerung des Skeletts

Die Berechnung der neuen Vertex-Positionen auf Basis des Orientierungsfeldes erfolgt wie im vorherigen Schritt beschrieben durch Anwendung des Gradientenabstiegsverfahren. Hierfür werden zwei neue Funktionen `dadf_func` und `dadf_func_grad` definiert, die die Funktion $\Delta A(T_i) + \Delta F(T_i)$ (vgl. Formeln 2.3 und 2.4 in Abschnitt 2.4.2) sowie ihren Gradienten bezüglich der Orientierungen beschreiben. Die Implementierung der Funktionen ist vereinfacht im Codeausschnitt 3.12 dargestellt. Um Rechenzeit einzusparen, werden einige konstante Informationen vorab berechnet und als Parameter übergeben. Die eindimensionalen Arrays `parents`, `node_weights` und `edge_weights` entsprechen in ihrer Bedeutung den gleichnamigen Parametern bei der Optimierung des Orientierungsfeldes.

Die Mittelpunkte der Kanten $(u + v)/2$ (entsprechend der Formel 2.4) werden zeilenweise im zweidimensionalen Array `edge_ctrs` gespeichert. Das Array `opt_edges` enthält Vektoren, die durch den Ausdruck $\|u - v\| \cdot (o_u + o_v)/\|o_u + o_v\|$ in der Formel 2.3 berechnet werden.

Listing 3.12: Implementierung der Funktion $\Delta A(T_i) + \Delta F(T_i)$ und Berechnung ihres Gradienten

```
88  # Optimierte Funktion
89  def dadf_func(x, parents, node_weights, edge_weights, edge_ctrs, opt_edges):
90      da = np.sum(
91          (edge_weights * np.linalg.norm((x - x[parents]) - opt_edges, axis=1))**2
                )
92      df = np.sum(
93          (node_weights * np.linalg.norm(((x + x[parents]) / 2) - edge_ctrs,
                axis=1))**2 )
94      return da + df
95
96  # Gradient der optimierten Funktion
97  def dadf_func_grad(x, parents, node_weights, edge_weights, edge_ctrs,
        opt_edges):
98      x_grad = np.zeros(x.shape)
99      for u in range(x.shape[0]):
100         v = parents[u]
101         ew2 = 2 * edge_weights[v]**2
102         nw2 = node_weights[v]**2
103         dadu = ew2 * (x[u] - x[v] - opt_edges[u])
104         dfdu = nw2 * ((x[u] + x[v]) / 2 - edge_ctrs[u])
105
106         # Partielle Ableitungen
107         x_grad[u] +=  dadu # dA/du
108         x_grad[v] += -dadu # dA/dv
109         x_grad[u] += dfdu  # dF/du
110         x_grad[v] += dfdu  # dF/dv
111     return x_grad
```

Durch Anwendung des Gradientenabstiegsverfahrens mittels der Funktion `gradient_descent` wird das Skelett verfeinert, wobei die Positionen der Vertex-Knoten des initialen Skeletts als Startvektor `x0` dienen. Die Optimierung erfolgt gemäß der zuvor beschriebenen Vorgehensweise in zwei Phasen, zuerst lokal auf einzelnen Ästen und anschließend global. Die Knotenattribute, die die alten Vertex-Positionen enthalten, werden mit den neuen Positionen des optimierten Skeletts überschrieben.

vi Iterative Verfeinerung

Im nächsten Schritt wird überprüft, ob eine weitere Iteration zur Verbesserung des Skeletts erforderlich ist. Hierfür wurde eine Funktion implementiert, die die Kantengewichtungen der initialen Graphen `bsg_initial_graphs`, die im Codeausschnitt (3.9) definiert wurden, entsprechend der Algorithmusbeschreibung basierend auf dem optimierten Skeletts aus dem letzten Schritt anpasst. Die

Suche nach nächstgelegenen Kanten erfolgt auf Basis eines k-d-Baums, der mit den Vertices des initialen Graphen konstruiert wird.

Nach Anpassung der Kantengewichtungen wird durch Aufruf der Funktion `dijkstra_spanning_tree` aus Abschnitt 3.3 ein neuer minimaler Spannbaum gebildet. Falls dieser Baum und das optimierte Skelett nicht isomorph sind, wird der Spannbaum als neues Skelett verwendet und der Optimierungsprozess ab Schritt (iii) wiederholt. Die Überprüfung auf Isomorphie erfolgt mit Hilfe von NetworkX.

Es wurde festgestellt, dass das iterative Verfahren nicht, wie in [Liv+10] beschrieben, nach zwei bis drei Iterationen konvergiert. Ein möglicher Grund dafür könnte die Wahl des Optimierungsverfahrens sein, da dieses keine exakten Lösungen liefert. Um zu verhindern, dass der Skelettierungsprozess die maximale Rechenzeit von 30 Minuten überschreitet, wurde die Anzahl der Iterationen auf 5 begrenzt.

vii Approximation der Astradien

Die Approximation der Astradien erfolgt in der Algorithmusbeschreibung von Livny et al. durch einen dritten globalen Optimierungsprozess. Im Zuge der Entwicklung wurde jedoch festgestellt, dass die Optimierung durch das Gradientenabstiegsverfahren aufgrund des geringen Einflusses des Gradienten in Bereichen mit niedrigen Knotengewichtungen zu unzureichenden Ergebnissen führt. Daher werden die Radien stattdessen direkt basierend auf den Allometrien von Xu et al. [XGC07] berechnet.

Zur Berechnung der Astradien im Verfahren von Xu et al. werden adjazente Knoten des Skeletts bezüglich ihrer tragenden Gesamt-Astlängen in ein Verhältnis zueinander gesetzt. Die zuvor berechneten Knotengewichtungen stellen bereits ein Maß dafür dar und werden im Folgenden wiederverwendet. Bei der Berechnung der Astradien werden basierend auf zwei verschiedenen Modellen zwei Fälle unterschieden: Wenn ein Elternknoten nur einen Kindknoten r_v hat, kann der Radius des Kindknotens mit Hilfe der Formel 3.1 aus dem Radius des Elternknotens r_u geschätzt werden [XGC07]. Dabei werden die Gewichtungen als c_v bzw. c_u für den Kind- bzw. Elternknoten bezeichnet. Im zweiten Fall, wenn ein Elternknoten mehr als einen Kindknoten hat, können die Radien der einzelnen Kindknoten durch die Formel 3.2 berechnet werden, wobei Livny et al. den ersten Teil der Formel beschreiben [XGC07]. r_{v_i} ist der Radius des i-ten Kindknotens und c_{v_i} bezeichnet die Gewichtung des i-ten Kindknotens. Das Modell von Xu et al. wurde hier weiter vereinfacht, indem die Gewichtung des Elternknotens als Näherung für die Summe der Gewichtungen der Kindknoten angenommen wurde, $\sum_j c_{v_j} \approx c_u$.

$$r_v = r_u(\frac{c_v}{c_u})^{1.5} \tag{3.1}$$

$$r_{v_i} = r_u(\frac{c_{v_i}}{\sum_j c_{v_j}})^{\frac{1}{2.49}} \approx r_u(\frac{c_{v_i}}{c_u})^{0.4} \tag{3.2}$$

Die Berechnung der Astradien erfolgt basierend auf den oben genannten Regeln innerhalb einer Funktion `estimate_branch_radii` (vgl. Codeausschnitt 3.13). Der Benutzer spezifiziert einen maximalen Stamm-Radius `trunk_radius`, der dem Wurzelknoten `root` zugewiesen wird (s. Zeile 113).

Anschließend wird eine Tiefensuche auf dem Skelett `bsg` durchgeführt, bei der für jeden Knoten `v` der Radius anhand des entsprechenden Verhältnisses zum Elternknoten `u` berechnet und als Knotenattribut zugewiesen wird (s. Zeilen 119–121).

Listing 3.13: Berechnung der Astradien basierend auf den Allometrien von [XGC07]

```
112 def estimate_branch_radii(bsg, root, trunk_radius):
113     bsg.nodes[root]['radius'] = trunk_radius
114     for u, v in nx.dfs_edges(bsg, root):
115         v_siblings = sum(1 for _ in bsg.neighbors(u)) - 1
116         v_weight = bsg.nodes[v]['weight']
117         u_weight = bsg.nodes[u]['weight']
118 
119         ratio = v_weight / u_weight if u_weight > 0.0 else 0.0
120         ratio **= 1.5 if v_siblings == 0 else 0.4
121         bsg.nodes[v]['radius'] = ratio * bsg.nodes[u]['radius']
122     return bsg
```

viii Bereinigung des Skeletts durch Entfernen von Kanten

Im abschließenden Schritt wird das Skelett durch Entfernen von Kanten bereinigt. In der Implementierung erfolgt die Ausführung der beiden beschriebenen Prozesse in umgekehrter Reihenfolge, da dies zu besseren Ergebnissen führt.

Zunächst werden alle Kanten, die gemäß der Algorithmusbeschreibung in Kapitel 2.4.2 eine signifikante Überlappung mit einem anderen Zylinder aufweisen, als zu löschende Kandidaten gesammelt. Die Berechnung der Schnittvolumen wurde durch die Schnittpunktberechnung eines Strahls, der die Mittelachse des einen Zylinders repräsentiert, mit der Oberfläche des anderen Zylinders vereinfacht. Die Entfernung des Schnittpunkts im Verhältnis zur Länge des Zylinders dient als Maß für die Überschneidung der beiden Zylinder. Da Livny et al. keine spezifische Reihenfolge für das Entfernen der Kanten vorgeben, wurde eine Methode gewählt, bei der die Vertices mit dem geringsten Abstand zueinander zuerst bearbeitet werden. Diese Vorgehensweise lieferte optimale Ergebnisse in durchgeführten Tests. Da sich die Vertices durch das Verschmelzen verschieben, werden die Kanten-Kandidaten kontinuierlich um die veränderten Bereiche aktualisiert. Der Prozess endet, sobald keine weiteren Kandidaten zum Entfernen mehr vorhanden sind.

Fehlerhafte Verzweigung, die übrig geblieben sind, werden abschließend anhand ihrer Gewichtungen, die als Maß für die Radien interpretiert werden können, identifiziert und entfernt. Hierfür wird eine Tiefensuche durchgeführt, bei der jeder Knoten anhand von zwei Kriterien überprüft wird. Zum einen wird geprüft, ob die Gewichtung eines Knotens einen bestimmten Schwellwert unterschreitet, um die absolute Länge des Astes zu begrenzen. Zum anderen wird das Verhältnis zwischen der Knotengewichtung und der des Elternknotens mittels eines weiteren Schwellwertes überprüft, um Verzweigungen mit erheblicher Differenz der Astradien auszuwählen. Wenn beide Kriterien erfüllt sind, wird der Ast, der von diesem Knoten ausgeht, als nicht relevant betrachtet und vollständig entfernt.

Der graphenbasierte Skelettierungsalgorithmus wird in Blender über denselben Operator wie der voxelbasierte Algorithmus ausgeführt. Es besteht auch hier die Möglichkeit, die Parameterwerte über eine Dialogbox zu konfigurieren (s. Abbildung A.1 (b) im Anhang A.1), und das Skelett kann als Mesh oder Kurve generiert werden.

4 Auswertung und Vergleich

In diesem Kapitel werden die beiden implementierten Skelettierungsalgorithmen anhand einer Reihe unterschiedlicher Baum-Punktwolken ausgewertet und hinsichtlich ihrer Laufzeit sowie der Genauigkeit der resultierenden Skelette miteinander verglichen.

Zunächst werden Kriterien aufgestellt, die der Bewertung der Ergebnisse sowie dem Vergleich der beiden Algorithmen dienen (**Abschnitt 4.1**). Anschließend wird die Datengrundlage erläutert, indem die Punktwolken, die zur Auswertung der Algorithmen verwendet werden, dargelegt werden (**Abschnitt 4.2**). Die Auswertung der Algorithmen und der Vergleich erfolgt zunächst qualitativ (**Abschnitt 4.3**) und anschließend quantitativ anhand definierter Messgrößen (**Abschnitt 4.4**).

4.1 Vergleichskriterien

Um die beiden implementierten Skelettierungsalgorithmen miteinander vergleichen zu können, werden in diesem Abschnitt Kriterien definiert, anhand derer die Skelette und die Algorithmen bewertet und verglichen werden können. Die Auswahl der Kriterien basiert auf der Zielsetzung dieser Arbeit sowie auf relevanten Merkmalen und Anforderungen für die Skelettierung von Baum-Punktwolken. Nach Definition dieser Kriterien können die Skelettierungsalgorithmen bzw. ihre resultierenden Skelette entweder qualitativ oder quantitativ bewertet und verglichen werden.

Das übergeordnete Ziel dieser Arbeit besteht darin, Skelettierungsalgorithmen zu implementieren, deren erzeugten Skelette den Originaldatensatz, also die relevanten Strukturen (Stamm, Äste und Zweige) des in einer Punktwolke abgebildeten Baums, *hinreichend genau* repräsentieren sollen. Dabei war sicherzustellen, dass die Berechnungsdauer der Algorithmen pro Skelettierungsvorgang einen Zeitraum von 30 Minuten ($T_{\max}$) nicht überschreitet. Die Zielsetzung bildet den Rahmen für die Auswahl der Vergleichskriterien, die als Leitfaden für die Bewertung und den Vergleich der Skelettierungsalgorithmen dienen.

Ausgehend von der Zielsetzung stellt sich die Frage, anhand welcher Aspekte die Genauigkeit und die Qualität von Skeletten im Allgemeinen gemessen werden kann. Hierfür bieten Cornea et al. (2007) einen detaillierten Überblick über zwölf wünschenswerte Eigenschaften von Skeletten, die sich zunächst auf allgemeine Skelette beliebiger Strukturen beziehen [CSM07]. Einige der aufgeführten Eigenschaften erfordern in diesem Kontext keine besondere Untersuchung, da sie durch die Funktionsweise der Skelettierungsalgorithmen automatisch gegeben sind. Eine wünschenswerte Eigenschaft bezieht sich beispielsweise auf die Dicke: Skelette stellen eindimensionale Strukturen dar und sollten daher dünn sein bzw. nach Cornea et al. maximal ein Voxel breit [CSM07]. Da bei beiden Skelettierungsalgorithmen die erzeugten Skelette als Graphen mit eindimensionalen Kanten repräsentiert sind, haben die Skelette keine Breite. Eine weitere wünschenswerte Eigenschaft ist die Kon-

nektivität des Skeletts. Ein Skelett, das die Struktur eines zusammenhängenden Objekts beschreibt, darf selbst nur aus einer Komponente bestehen, um die Topolgie des Objektes zu bewahren [CSM07]. Diese Eigenschaft ist ebenfalls bei beiden Skelettierungsalgorithmen erfüllt. Die Algorithmen liefern für jeden gesuchten Baum ein einzelnes, zusammenhängendes Skelett.

Von den von Cornea et al. aufgelisteten Kriterien werden drei Eigenschaften von Skeletten und eine weitere Eigenschaft von Skelettierungsalgorithmen zur Festlegung der Vergleichskriterien betrachtet. Die erste Eigenschaft ist die *Homotopie* [CSM07] hinsichtlich der Topologie des Skeletts. Durch die Skelettierung eines Objektes sollte die Topologie, also die Gestalt des Objekts unter Berücksichtigung von Verformungen, erhalten bleiben [CSM07]. Bucksch et al. (2009) weisen darauf hin, dass damit die Navigierbarkeit durch den Baum sichergestellt wird [BLM09]. Auf Basis dieser Darstellung wird vereinfacht festgelegt, dass jeder Ast des Baums durch das Skelett repräsentiert sein soll. Diese Eigenschaft wird als erstes Kriterium (K1) festgelegt.

Die zweite ausgewählte Eigenschaft bezieht sich auf die *Zentriertheit* [CSM07] des Skeletts in dem Baum. Das Skelett sollte auf der medialen Achse (bzw. Fläche) liegen, wobei die Wichtigkeit dieser Eigenschaft abhängig vom Anwendungsfall ist [CSM07]. Im Kontext der Skelettierung von Baumstrukturen ist diese Eigenschaft wünschenswert, da dies eine korrekte Messung [BLM09] und Modellierung der Astbreiten ermöglicht. Daher wird für das zweite Kriterium (K2) diese Zentriertheit festgelegt. Bucksch et al. betonen, dass im Kontext der Skelettierung von Bäumen unter anderem die Kriterien (K1) und (K2) von besonderer Relevanz sind [BLM09].

Eine dritte wünschenswerte Eigenschaft ist die *Glattheit* [CSM07] des Skeletts. Cornea et al. definieren die Glattheit eines Kurvensegments als Variation der tangentialen Richtung, die bei einer Traversierung des Segments entsteht [CSM07]. Ein glattes Skelett weist auf unverzweigten Astsegmenten geringe Richtungsänderungen auf. Dies führt zu einem insgesamt ästhetisch ansprechenden Resultat, was insbesondere bei der Skelettierung eines Baums zum Zweck der Visualisierung von Bedeutung ist. Das Vergleichskriterium (K3) beschreibt daher die Eigenschaft der Glattheit.

Im Zusammenhang mit dem Skelettierungsprozess und dem Skelettierungsalgorithmus ist die *Recheneffizienz* eine weitere zu berücksichtigende Eigenschaft [CSM07]. Im Kontext der Skelettierung von Bäumen auf Basis von Punktwolken ist dies besonders wichtig, da die durch Laserscanning erzeugten Punktwolken typischerweise aus einer großen Anzahl an Punkten bestehen [BLM09]. Die Effizienz des Algorithmus im Sinne der Laufzeit wird als viertes Vergleichskriterium (K4) definiert.

Die drei Kriterien (K1), (K2) und (K3) spezifizieren die Eigenschaft der Genauigkeit eines Skeletts. Mit dem vierten Vergleichskriterium (K4) wird die zweite Voraussetzung der Zielsetzung, die Laufzeit des Algorithmus, berücksichtigt. Damit erfüllen die ausgewählten Vergleichskriterien die Anforderungen der Aufgabenstellung und ermöglichen eine umfassende Bewertung und einen Vergleich der implementierten Skelettierungsalgorithmen.

4.2 Datengrundlage

Für die Auswertung der implementierten Skelettierungsalgorithmen werden Punktwolken von verschiedenen Bäumen verwendet. Diese umfassen einerseits Punktwolken realer Bäume, die mit Hilfe unterschiedlicher Laserscanning-Verfahren aus der Natur erfasst wurden, sowie künstliche Punktwolken, die anhand von prozedural generierten Baum-Modellen erzeugt wurden. Bei der Auswahl der Punktwolken wurde darauf geachtet, eine breite Variation an Szenarien abzudecken, die unterschiedliche Baumarten in Bezug auf Größe, Struktur und Komplexität umfassen. Dies soll es ermöglichen, die Skelettierungsalgorithmen auf verschiedene Gegebenheiten zu testen und ihre Effizienz in vielfältigen Kontexten zu beurteilen.

Zur Evaluierung der Skelettierungsalgorithmen anhand realer Bäume stellte die Firma Graswald Datensätze von Weiser et al. [Wei+22b] zur Verfügung, die LiDAR-Punktwolken von Bäumen verschiedener Arten umfassen. Aus diesen Datensätzen wurden Punktwolken von drei verschiedenen Bäumen ausgewählt. Die Aufnahmen der Baum-Punktwolken erfolgten im Jahr 2019 in einem Waldstück in Bretten, Baden-Württemberg [Wei+22a]. Die ausgewählten Datensätze umfassen Punktwolken einer Trauben-Eiche (Quercus petraea), einer Rotbuche (Fagus sylvatica) sowie einer Douglasie (Pseudotsuga menziesii). Durch den Einsatz unterschiedlicher Messtechniken (TLS sowie ULS [Wei+22a]) variieren die Punktdichten und Genauigkeiten der Punktwolken. Darüber hinaus wurden die Bäume zu verschiedenen Jahreszeiten erfasst, was ebenfalls zu unterschiedlichen Bedingungen führt [Wei+22a]. Eine Evaluation der Algorithmen anhand dieser realen Daten ermöglicht es, die Skelettierungsalgorithmen auf die in Kapitel 1.2 beschriebenen Herausforderungen wie Rauschen oder Überdeckungen in der Punktwolke zu testen und somit in Bezug auf ihre Praxistauglichkeit zu validieren.

Die Datensätze von Weiser et al. bieten den Vorteil, dass sie bereits segmentiert und eine Registrierung sowie ein Post-Processing durchgeführt wurden [Wei+22a]. Dadurch sind die Datensätze optimal für die Skelettierung vorbereitet und erforderten keine zusätzliche Vorverarbeitung. Die Punktwolken stehen im LAZ-Dateiformat bereit und können mithilfe der entwickelten Import-Funktion problemlos eingelesen werden.

Neben den realen Bäumen wurden künstliche Punktwolken mit Hilfe prozedural generierter Baum-Modelle erstellt. Dieser Ansatz wurde gewählt, da er die Erfassung bestimmter Daten (Anzahl der Äste und Gesamtlänge der Äste) für die Auswertung der Algorithmen erleichtert. Für die Erzeugung der Baum-Modelle wurde das Add-on *Sapling Tree Gen* in Blender verwendet, welches basierend auf der Methode von Weber und Penn [WP95] natürliche Baumstrukturen generiert [HBC23]. Die Baum-Modelle wurden ohne Blätter erzeugt und mit Hilfe von Partikelsystemen in Punktwolken umgewandelt, die die Oberflächen der Objekte abbilden. Bei der Generierung der Punktwolken wurden die Punkte zudem zufällig versetzt, um die Ungenauigkeiten, die bei Laserscanning-Verfahren auftreten, zu simulieren. Hierbei wurde jedoch darauf geachtet, dass keine Merkmale der Aststrukturen verloren gingen.

Die ausgewählten Testdatensätze sind in Tabelle 4.1 mit zusätzlichen Informationen zu den Baumarten und ihren Größen sowie der Anzahl der Punkte in den Punktwolken aufgeführt. In den folgenden Ausarbeitungen werden die Baum-Punktwolken anhand der IDs (B1–B6) referenziert. Die ersten drei Punktwolken (B1–B3) repräsentieren die realen Aufnahmen und (B4–B6) stellen die pro-

zedural erzeugten dar. Die Anzahl der Äste in einem Baum wird als „NOB“ (*number of branches*) und die Gesamtlänge der Äste als „TBL“ (*total branch length*) bezeichnet. Die Punktwolken der sechs Bäume sind in Abbildung 4.1 dargestellt.

ID	Quelle	Baumart	Dimension ($L \times B \times H$)	NOB	TBL	Punktanzahl
B1	[Wei+22b, Baum „QuePet_BR01_02“]	Quercus petraea (Trauben-Eiche)	$7,23 \times 7,38 \times 16,9$	k. A.	k. A.	727.797
B2	[Wei+22b, Baum „FagSyl_BR01_05“]	Fagus sylvatica (Rotbuche)	$10 \times 9,3 \times 30,9$	k. A.	k. A.	4.028.454
B3	[Wei+22b, Baum „PseMen_BR01_P9T5“]	Pseudotsuga menziesii (Douglasie)	$12,7 \times 12,3 \times 41,3$	k. A.	k. A.	169.410
B4	*Prozedural erstellt* (Preset „japanese maple“)	Acer palmatum (Fächer-Ahorn)	$9,62 \times 10,1 \times 8,65$	286	460,4	100.000
B5	*Prozedural erstellt* (Preset „white birch“)	Betula papyrifera (Papier-Birke)	$12,6 \times 12,9 \times 16,1$	291	599,07	100.000
B6	*Prozedural erstellt* (Preset „quaking aspen“)	Populus tremula (Zitter-Pappel)	$7,53 \times 7,87 \times 17,3$	30	116,83	50.000

Tabelle 4.1: Übersicht der verwendeten Baum-Punktwolken mit Informationen zu den Baumarten, Baumeigenschaften und Anzahl der Punkte in der Punktwolke

Abbildung 4.1: Punktwolken B1–B6 im Größenvergleich. B1–B3 sind LiDAR-Punktwolken von [Wei+22b]; B4–B6 sind künstlich erzeugte Punktwolken

4.3 Qualitative Auswertung und Vergleich

In diesem Abschnitt werden die resultierenden Skelette der beiden Skelettierungsalgorithmen visuell verglichen. Dabei wird die allgemeine Form und Struktur der Skelette betrachtet und mit der Baumstruktur in den ursprünglichen Punktwolken verglichen. Die Auswertung der beiden Algorithmen wurde anhand der drei Datensätze B1–B3 durchgeführt. Die resultierenden Skelette sind in Abbildung 4.2 dargestellt und im Folgenden wird speziell auf die markierten Bereiche (a)–(i) eingegangen. Die zugrundeliegenden Messwerte sind in den Tabellen 4.2 und 4.3 für die Baumpunktwolken B1–B3 aufgeführt und werden im Abschnitt 4.4 betrachtet.

Zunächst kann gesagt werden, dass beide Skelettierungsalgorithmen in der Lage sind, aus einer gegebenen Punktwolke ein Skelett zu konstruieren, das der groben Form des in der Punktwolke abgebildeten Baums ähnelt. Die Skelette liegen im Wesentlichen innerhalb der Punktwolken und stellen die Grundstruktur bzw. die relevantesten Äste der Bäume dar. Bei näherer Betrachtung fallen jedoch einige qualitative Unterschiede zwischen den Skeletten auf. Zum einen weist das Skelett des graphenbasierten Algorithmus wesentlich mehr Details in Form von feinen Ästen auf, während der Detailgrad bei dem voxelbasierten Skelettierungsalgorithmus auf die Auflösung des Voxelrasters beschränkt ist. Xu et al. betonen, dass der voxelbasierte Skelettierungsalgorithmus von Gorte und Pfeifer darauf abzielt, nur die dominantesten Strukturen und keine feinen Details zu rekonstruieren [XGC07].

Das Skelett des graphenbasierten Skelettierungsalgorithmus deckt die Punktwolken insgesamt besser ab, während das Skelett des voxelbasierten Algorithmus einige Bereiche nicht oder nur geringfügig repräsentiert. Dies ist beispielsweise an der Spitze des Baumes B1 zu beobachten, wo die Punktwolke unvollständig ist und niedrige Punktdichte sowie signifikantes Rauschen aufweist. Während der voxelbasierte Skelettierungsalgorithmus nicht in der Lage ist, diesen Bereich adäquat zu rekonstruieren (a), liefert der graphenbasierte Skelettierungsalgorithmus hier eine überzeugende Approximation der Aststruktur.

Bei der Betrachtung der Skelettstrukturen fällt auf, dass die resultierenden Skelettstrukturen des voxelbasierten Algorithmus im Allgemeinen eher unrealistisch sind und nicht die korrekten Baumstrukturen darstellen. Es treten vermehrt falsche Verbindungen in den Skeletten auf, die zu unplausiblen Verzweigungen und Astrichtungen führen (e). Besonders im Bereich der Baumkrone, wo viel Rauschen in der Punktwolke vorhanden ist, verstärkt sich dieses Problem, wodurch die Skelette noch chaotischer und unstrukturierter wirken. Im Gegensatz dazu weisen die Skelette des graphenbasierten Skelettierungsalgorithmus logischere Verbindungen auf und es ist eine klare Struktur erkennbar. Im Gegensatz zu den voxelbasierten Skeletten sind hier auch die Stämme innerhalb der Baumkronen zu erkennen, die relativ geradlinig durch die Baumkronen verlaufen. Die plausibleren Skelettstrukturen des graphenbasierten Algorithmus sind unter anderem darauf zurückzuführen, dass bei der Konstruktion der Skelette durch die globalen Optimierungsprozesse der gesamte Baum berücksichtigt wird.

Bei der Untersuchung der Skelette bezüglich des dritten Kriteriums (K3) zeigt sich, dass die mit dem graphenbasierten Skelettierungsalgorithmus erzeugten Skelette im Vergleich zu den mit dem voxelbasierten Algorithmus erzeugten Skeletten einen insgesamt glatteren Verlauf aufweisen. Die Arbeitsweise des voxelbasierten Skelettierungsalgorithmus führt bei schrägen Ästen zu stufenförmigen Kantenverläufen im Skelett. Hingegen fällt bei der Betrachtung des zweiten Kriteriums (K2) auf, dass die Skelette des graphenbasierten Algorithmus nicht besonders gut zentriert sind (h). Da das

Skelett aus Punkten konstruiert wird, die auf der Oberfläche des Stammes liegen, ist zu erwarten, dass das Skelett auch entlang der Oberfläche verläuft. Im Inneren des Stammes, mit Ausnahme des treppenförmigen Verlaufs der Kanten, liefert der voxelbasierte Skelettierungsalgorithmus eine bessere Zentrierung (d). Es ist jedoch zu beachten, dass dies vom jeweiligen Baum abhängt. Bucksch und Lindenbergh weisen darauf hin, dass Skelettierungsalgorithmen, die auf morphologischen Verdünnungsoperationen basieren, empfindlich auf Änderungen der Objektform an der Oberfläche reagieren [BL08]. Eine grobe Oberflächenstruktur führt zu einer stärkeren Abweichung des Skeletts von der medialen Achse (b) als eine glatte Oberfläche (d). An Stellen, an denen Verzweigungen auftreten, führt dies dazu, dass das Skelett „hin und her springt“ (f), was von Chaudhury und Godin (2020) auch als „Zickzack“-Problem beschrieben wird [CG20].

Die Skelette des voxelbasierten Skelettierungsalgorithmus weisen Bereiche mit besonders vielen Kantenzügen auf kleinem Raum auf (c). Dieses Problem tritt auf, wenn durch Rauschen in der Punktwolke oder durch Closing-Operationen während der Datenvorverarbeitung geschlossene Hohlräume im Objekt entstehen. Die verwendete Verdünnungsoperation ist nicht in der Lage, diese Hohlräume zu öffnen, sodass sie nach der Verdünnung als Blasen zurückbleiben. Bei der Skelettextraktion werden die Kanten entlang der verbleibenden Voxel gezogen.

Weiterhin ist zu beachten, dass die implementierten Algorithmen keine automatische Segmentierung der Bäume von anderen Objekten durchführen. Dies zeigt der Datensatz B3, in dem ein Teil des Bodens in der Punktwolke enthalten ist. Die Algorithmen unterscheiden nicht zwischen verschiedenen Objekten und skelettieren daher auch den Boden in beiden Algorithmen (g, i). Daher dürfen in der Punktwolke keine anderen Objekte oder Teile von Objekten als die zu skelettierenden Bäume enthalten sein.

Zusammenfassend kann gesagt werden, dass der graphenbasierte Skelettierungsalgorithmus visuell glaubwürdigere Ergebnisse liefert als der voxelbasierte Algorithmus. Das Skelett ist im Allgemeinen glatter, während das Skelett des voxelbasierten Algorithmus eine bessere Zentrierung in den klaren Baumstrukturen aufweist, mit Ausnahme der beschriebenen Probleme. Einige der in Kapitel 1.2 beschriebenen Probleme wie inhomogene Punktdichte, signifikantes Rauschen und Unvollständigkeit der Daten werden durch den graphenbasierten Skelettierungsalgorithmus überwunden. Die Genauigkeit, mit der das Skelett die tatsächliche Baumstruktur repräsentiert, erfordert jedoch weitere Untersuchungen.

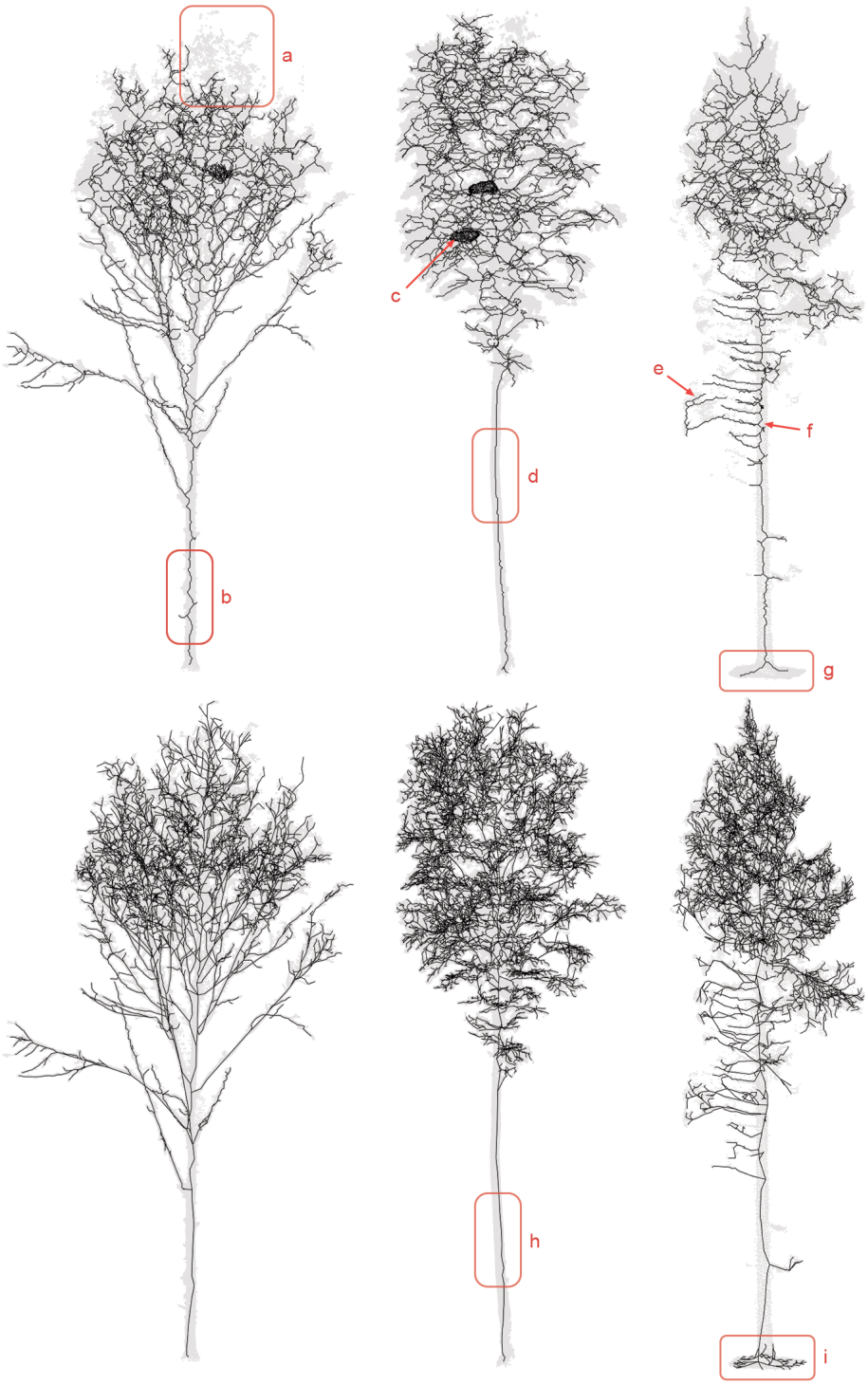

Abbildung 4.2: Skelettierungsergebnisse der Baumpunktwolken B1 (links), B2 (Mitte) und B3 (rechts). Oben die Skelette des voxelbasierten Algorithmus, unten die Skelette des graphenbasierten Algorithmus

4.4 Quantitative Auswertung und Vergleich

In diesem Abschnitt werden die Genauigkeit der Skelette und die Effizienz der Skelettierungsalgorithmen anhand verschiedener Metriken quantifiziert und verglichen. Das Ziel ist, mit Hilfe dieser Daten fundierte Aussagen über die Leistung und Effizienz der implementierten Skelettierungsalgorithmen zu treffen und diese miteinander zu vergleichen.

4.4.1 Messgrößen

Basierend auf den zuvor definierten Vergleichskriterien werden vier Messgrößen definiert, die wichtige Informationen über die Eigenschaften der resultierenden Skelette liefern und Aufschluss über die Leistung der Algorithmen geben.

Die erste Messgröße ist die Anzahl der Äste (NOB, *number of branches*). Da jeder Ast eines Skeletts einen eigenen Endknoten haben muss, kann hierfür die Anzahl der Blattknoten im Skelett gezählt werden. Gleichzeitig kann diese Größe als Maß für die Anzahl der Verzweigungen im Skelett betrachtet werden, da durch jede Verzweigung ein neuer Ast entsteht. Die TBL (*total branch length*) gibt die Gesamtlänge aller Äste im Skelett an und liefert Informationen über die Ausdehnung und die Komplexität des Skeletts. Zur Berechnung dieser Größe werden die einzelnen Kantenlängen im Skelett aufsummiert. Die mittlere quadratische Abweichung (MSE, *mean squared error*) wird verwendet, um die Entfernung der Punktwolken-Punkte vom Skelett zu quantifizieren und ermöglicht somit eine Einschätzung der Genauigkeit des Skeletts hinsichtlich der originalen Punktwolke. Dazu werden für alle Punkte die kürzesten Entfernungen zum Skelett ermittelt und ein Durchschnitt der quadrierten Entfernungen berechnet. Als letzte wichtige Messgröße wird die Laufzeit der Algorithmen gemessen. Es werden die Teillaufzeiten t_i ($i \in \{1, 2, 3, 4\}$) der für die Skelettierung relevanten Prozeduren erfasst. Die Gesamtlaufzeit eines Skelettierungsvorgangs $\sum t_i$ ergibt sich aus der Summe der einzelnen Teillaufzeiten.

Die festgelegten Messgrößen dienen dem Vergleich von Skeletten im Hinblick auf die definierten Kriterien. Das Kriterium der Homotopie (K1) wird vereinfacht geprüft, indem die Anzahl der Äste und deren Länge mit den tatsächlichen Werten abgeglichen werden. Dadurch lässt sich überprüfen, ob das Skelett eine entsprechende Anzahl und Länge von Ästen aufweist, die mit den Originaldaten übereinstimmen. Es ist zu beachten, dass dies nicht die Homotopie des Skeletts nachweist, da damit nicht die tatsächliche Struktur der Äste geprüft werden kann. Die mittlere quadratische Abweichung (MSE) wird verwendet, um die allgemeine Genauigkeit des Skeletts zu überprüfen. Diese hängt jedoch sowohl von der Anzahl der tatsächlich gefundenen Äste als auch von der Gesamtlänge aller Äste im Skelett (TBL) ab. Eine größere Anzahl von Ästen im Skelett kann die durchschnittliche Abweichung der Punkte vom Skelett verringern. Betrachtet man beispielsweise ein Skelett, das alle Punkte der Punktwolke miteinander verbindet, kann die MSE minimal sein, obwohl dieses Skelett kaum die tatsächliche Struktur des Baums abbildet. Daher sollte bei der Beurteilung der MSE auch die Homotopie des Skeletts berücksichtigt werden. Abschließend ermöglicht die Messung der Laufzeit eine Bewertung der Recheneffizienz der Algorithmen (K4).

Neben den Messgrößen werden auch die für die Skelettierung relevanten Parameter dokumentiert. Der Downsampling-Faktor (DSF) gibt die relative Anzahl der verwendeten Punkte der ursprünglichen

Punktwolke an. Die neue, reduzierte Punktanzahl wird mit n gekennzeichnet. Für den voxelbasierten Skelettierungsalgorithmus gibt s_x die Größe eines Voxels in Metern an (*cell size*) und n_x ist die daraus resultierende Auflösung in x-Richtung (Anzahl der Voxel). Die Erfassung dieser Informationen ist wichtig, um die Ergebnisse richtig einzuordnen und etwaige Unterschiede aufgrund unterschiedlicher Parametereinstellungen zu berücksichtigen.

4.4.2 Genauigkeit

Im folgenden Abschnitt wird die Genauigkeit der Skelettierungsalgorithmen anhand der festgelegten Metriken untersucht. Dazu wurden die Algorithmen mit Hilfe der in Abschnitt 4.2 beschriebenen Datensätze ausgewertet. Um die resultierenden Skelette unabhängig von den individuellen Laufzeiten der Algorithmen vergleichen zu können, wurden in den einzelnen Testläufen Parameter für die Algorithmen bezüglich der Auflösung des Voxelrasters bzw. der Punktanzahl gewählt, die zu vergleichbaren Laufzeiten führen. Es wurde ein Zeitfenster von 19–21 Minuten festgelegt, in dem die Algorithmen jeweils das Skelett erzeugen sollten. Die Messwerte sind in den Tabellen 4.2 (für den voxelbasierten Algorithmus) und 4.3 (für den graphenbasierten Algorithmus) aufgeführt. Die resultierenden Skelette der ersten drei Datensätze B1–B3 sind in der Abbildung 4.2 dargestellt.

ID	s_x (n_x)	NOB (Anteil)	TBL (Anteil)	MSE	t_1	t_2	t_3	t_4	$\sum t_i$
B1	0,045 (161)	1441 (k. A.)	692,872 (k. A.)	0,021	0,245	0,996	1215,576	2,1	1219
B2	0,061 (164)	1671 (k. A.)	1213,849 (k. A.)	0,068	1,105	2,041	1228,993	2,692	1235
B3	0,089 (143)	561 (k. A.)	883,301 (k. A.)	0,264	0,107	1,266	1191,822	1,305	1194
B4	0,029 (332)	379 (**133%**)	344,823 (**75%**)	0,01173	0,243	5,18	1229,491	2,046	1237
B5	0,043 (294)	334 (**115%**)	476,086 (**79%**)	0,02794	0,227	7,269	1189,417	1,764	1199
B6	0,027 (279)	42 (**140%**)	118,697 (**102%**)	0,01331	0,307	11,75	1142,389	1,759	1156

Tabelle 4.2: Auswertung des voxelbasierten Skelettierungsalgorithmus anhand unterschiedlicher Baumarten; t_1: Konvertierung der Punktdaten in Voxeldaten; t_2: Datenvorverarbeitung; t_3: Thinning-Operation; t_4: Extraktion des Skeletts

ID	DSF (n)	NOB (Anteil)	TBL (Anteil)	MSE	t_1	t_2	t_3	t_4	$\sum t_i$
B1	0,017 (14.104)	1498 (k. A.)	892,176 (k. A.)	0,007	1,678	92,143	1099,131	6,493	1199
B2	0,0033 (14.722)	1671 (k. A.)	1492,84 (k. A.)	0,02433	8,185	131,876	1086,444	8,364	1235
B3	0,07 (14.038)	1967 (k. A.)	2193,858 (k. A.)	0,03739	0,631	92,901	1060,258	4,203	1158
B4	0,12 (12.624)	260 (**91%**)	408,268 (**89%**)	0,00609	0,392	87,694	1115,426	9,682	1213
B5	0,12 (13.005)	299 (**103%**)	566,37 (**95%**)	0,01802	0,391	52,656	1123,356	18,502	1195
B6	0,26 (13.078)	51 (**170%**)	118,292 (**101%**)	0,04169	0,175	168,93	968,93	23,3	1161

Tabelle 4.3: Auswertung des graphenbasierten Skelettierungsalgorithmus anhand unterschiedlicher Baumarten; t_1: Identifizierung der Bäume in der Punktwolke; t_2: Initiale Konstruktion des Skeletts; t_3: Globale Optimierung; t_4: Bereinigung des Skeletts

Bei einem Vergleich der Messwerte der beiden Skelettierungsalgorithmen treten einige Unterschiede auf. Insbesondere bei den Datensätzen der realen Bäume (B1–B3) weist das Skelett des graphenbasierten Algorithmus tendenziell mehr bzw. längere Äste auf. Dieser Unterschied zeigt sich besonders deutlich im dritten Datensatz. Durch eine visuelle Inspektion der Skelette und der Punktwolke konnte der Grund dafür ermittelt werden. Die Punktwolke der Douglasie (B3) enthält signifikantes Rauschen und unvollständige Bereiche aufgrund von Überdeckungen in der Baumkrone. Die relevanten Äste werden in der Punktwolke nur geringfügig repräsentiert. Da der graphenbasierte Algorithmus das Skelett direkt durch Verbinden der Punkte erzeugt, ohne die Plausibilität der Verbindungen zu prüfen, entstehen bei Vorliegen von erheblichem Rauschen in der Punktwolke vermehrt zusätzliche Äste und Details. Diese verringern zwar die mittlere quadratische Abweichung (MSE), repräsentieren jedoch nicht die tatsächliche Struktur des Baums. Bei dem voxelbasierten Algorithmus hingegen hat das Rauschen weniger starken Einfluss auf die Anzahl der Äste im resultierenden Skelett. Es gehen tendenziell Details verloren, da diese durch die beschränkte Auflösung des Voxelrasters nicht repräsentiert werden können.

Die Datensätze B4–B6 dienen dem Vergleich der Anzahl der Äste und ihrer Gesamtlänge im Skelett mit den tatsächlichen Werten. Für die Anzahl und die Gesamtlänge der Äste wird jeweils ein Prozentsatz angegeben, in welchem Maß die Skelette die ursprünglichen Werte erreichen. Es fällt auf, dass beide Algorithmen nicht die korrekte Astanzahl liefern. Ein Vergleich zeigt jedoch, dass bei den ersten beiden Datensätzen, B4 und B5, der graphenbasierte Skelettierungsalgorithmus Ergebnisse liefert, die näher an den tatsächlichen Werten liegen (Abweichung von ±3% bis ±11%) als der voxelbasierte Algorithmus (Abweichung von ±15% bis ±33%). Die Punktwolke B6 stellt eine simple Baumstruktur dar, die nur aus wenigen, signifikanten und klar voneinander abgetrennten Ästen besteht. Der Datensatz zeigt, dass beide Algorithmen trotz einfacher Strukturen in der originalen Punktwolke in manchen Fällen mehr Äste finden als tatsächlich im Baum vorhanden sind. Besonders deutlich wird dies bei dem graphenbasierten Algorithmus. Zuletzt kann festgestellt werden, dass der graphenbasierte Algorithmus in Bezug auf den MSE tendenziell genauere Ergebnisse liefert, wobei dies, wie bereits erwähnt, auch auf die unterschiedlichen Gesamtlängen der Äste in den Skeletten zurückzuführen ist.

Um den Zusammenhang zwischen der Genauigkeit und der Laufzeit der Algorithmen bzw. der eingestellten Auflösung genauer zu untersuchen, werden zuletzt die Messreihen aus den Tabellen 4.4 und 4.5 herangezogen, in denen der Datensatz B1 mit beiden Skelettierungsalgorithmen in unterschiedlichen Auflösungen (Auflösung des Voxelrasters bzw. Punktanzahlen) skelettiert wurde. Betrachtet man den Zusammenhang zwischen der Gesamtastlänge (TBL) und der Abweichung (MSE) zeigt sich, dass der graphenbasierte Algorithmus bei gleicher Gesamtastlänge eine geringere Abweichung und damit ein genaueres Ergebnis liefert als der voxelbasierte Algorithmus. Dieser Zusammenhang ist in der Abbildung 4.3 veranschaulicht.

In den Abbildungen 4.4 und 4.5 ist der Zusammenhang zwischen der Laufzeit und der TBL sowie des MSE der resultierenden Skelette dargestellt. Die Ergebnisse zeigen, dass der graphenbasierte Skelettierungsalgorithmus im Vergleich zum voxelbasierten Algorithmus ab einer bestimmten Auflösung trotz gleicher Laufzeit mehr Äste sowie insgesamt eine höhere Genauigkeit (geringere MSE) liefert und somit bei gleicher Laufzeit bessere Ergebnisse liefern kann.

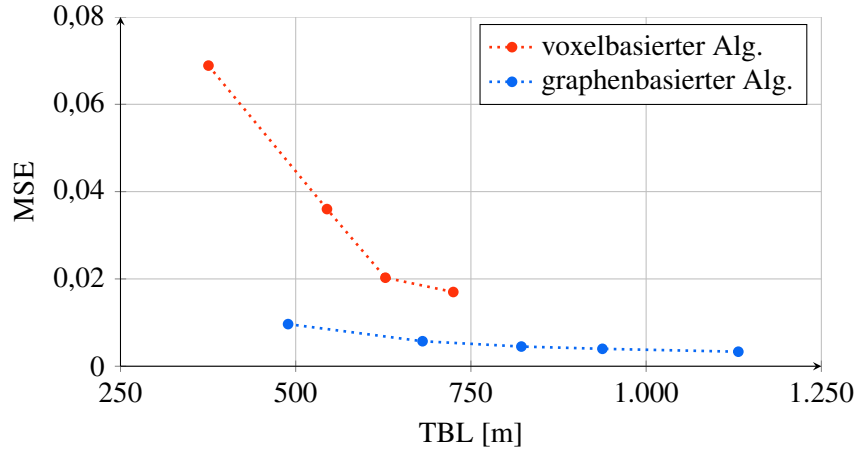

Abbildung 4.3: Abhängigkeit des MSE von der TBL

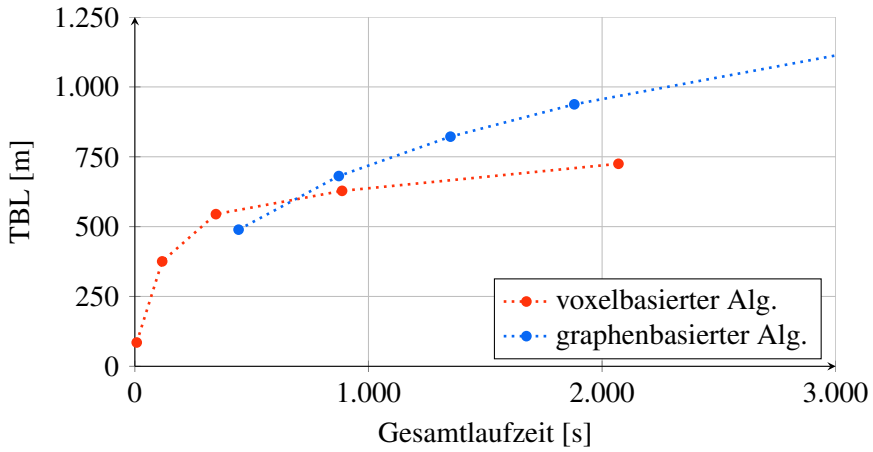

Abbildung 4.4: Abhängigkeit der TBL von der Gesamtlaufzeit

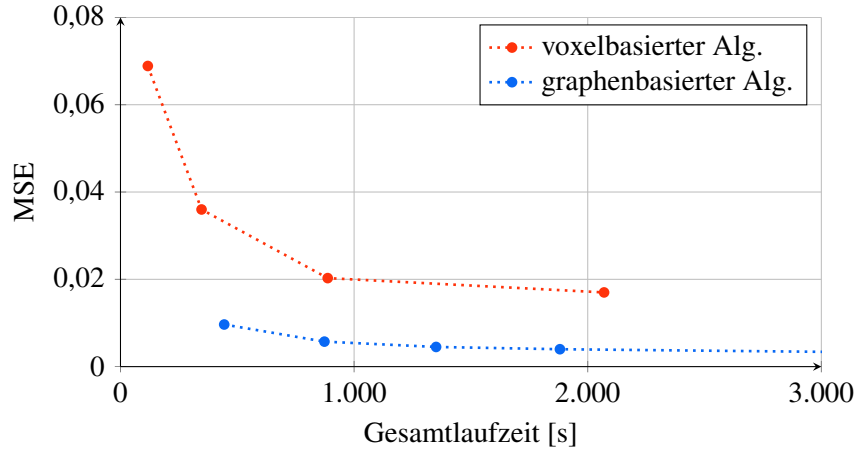

Abbildung 4.5: Abhängigkeit des MSE von der Gesamtlaufzeit

4.4.3 Laufzeit

Um das Laufzeitverhalten der Algorithmen zu untersuchen, wurden die beiden Skelettierungsalgorithmen für einen Baum (B1) auf unterschiedliche Datenmengen getestet. Dabei wurden die Laufzeiten in Abhängigkeit von Faktoren gemessen, die einen maßgeblichen Einfluss auf die Ausführungszeit der Algorithmen haben. Da der voxelbasierte Skelettierungsalgorithmus auf Voxeloperationen basiert, wird seine Laufzeit im Wesentlichen durch die Anzahl der Voxel bzw. die Auflösung des Voxelrasters bestimmt. Im Gegensatz dazu operiert der graphenbasierte Algorithmus direkt auf den Punkten der Punktwolke, wodurch die Anzahl der Punkte den entscheidenden Faktor darstellt.

Um den Zusammenhang zwischen den genannten Einflussfaktoren und der Laufzeit der Algorithmen zu untersuchen, wurden verschiedene Maßnahmen ergriffen. Beim graphenbasierten Algorithmus wurde die Punktwolke durch einfaches Downsampling auf unterschiedliche Punktzahlen reduziert, während beim voxelbasierte Algorithmus verschiedene Zellengrößen verwendet wurden, was zu unterschiedlichen Auflösungen des Voxelrasters führte. Zur genaueren Analyse der Laufzeiten wurden die relevantesten Prozeduren der Algorithmen identifiziert und die Laufzeit jeder einzelnen Prozedur aufgenommen. Beim voxelbasierten Algorithmus umfassen diese Prozeduren die Konvertierung der Punktdaten in Voxeldaten (t_1), die Datenvorverarbeitung (t_2), die Skelettierung der Voxeldaten durch Verdünnen (t_3) und die Extraktion des Skeletts (t_4). Beim graphenbasierten Skelettierungsalgorithmus wurden die Laufzeiten für die Identifizierung des Baums in der Punktwolke (t_1), die initiale Konstruktion des Skeletts (t_2), den globalen Optimierungsprozess (t_3) sowie die Bereinigung des Skeletts von fehlerhaften Ästen (t_4) erfasst. Um eine robuste und zuverlässige Messung der Laufzeit zu gewährleisten, wurden die Laufzeiten für jede Prozedur aus drei unabhängigen Durchläufen gemittelt.

Die Messwerte sind in den Tabellen 4.4 und 4.5 dargestellt. Aus den Messwerten lässt sich zunächst festhalten, dass beide Skelettierungsalgorithmen in der Lage sind, Skelette innerhalb der vorgegebenen Zeit von 30 Minuten zu erzeugen, sofern die gewählte Auflösung nicht zu hoch ist (s. Einstellungen zur Zellenbreite bzw. Punktanzahl in den Tabellen).

Die mittleren Laufzeiten in Abhängigkeit von den jeweils bestimmenden Faktoren (Auflösung des Voxelrasters beim voxelbasierten Algorithmus und Anzahl der Punkte beim graphenbasierten Algorithmus) werden in den Abbildungen 4.6 und 4.7 veranschaulicht. Es ist zu erkennen, dass beim voxelbasierten Algorithmus der Großteil der Laufzeit auf den Verdünnungsprozess (t_3) entfällt, während die anderen Prozesse nur einen geringen Anteil ausmachen. Beim graphenbasierten Algorithmus dominieren die globalen Optimierungsprozesse (t_3) die Laufzeit, wobei auch die initiale Konstruktion des Skeletts (t_2) einen signifikanten Beitrag leistet. Der Verlauf der Laufzeit des voxelbasierten Algorithmus zeigt eine ausgeprägte Krümmung in Abhängigkeit von der Auflösung des Voxelrasters, während die Laufzeit des graphenbasierten Algorithmus nur eine leichte Krümmung mit steigender Punktzahl aufweist.

Es wurde festgestellt, dass die absolute Laufzeit des implementierten graphenbasierten Algorithmus deutlich höher ausfällt als in der Beschreibung von Livny et al. Hier lieferte der Algorithmus bei etwa 14.000 Punkten bereits nach nur 7 Sekunden ein Ergebnis [Liv+10], in der eigenen Implementierung ist die Laufzeit jedoch deutlich höher. Dies könnte auf die verwendete Optimierungsmethode zurückzuführen sein, da das Gradientenabstiegsverfahren anstelle einer exakten Lösung durch ein lineares Gleichungssystem eingesetzt wurde.

ID	s_x (n_x)	NOB	TBL	MSE	n_{Th}	t_1	⊘	t_2	⊘	t_3	⊘	t_4	⊘	$\sum t_i$	⊘
B1	0,2 (37)	33	85,221	0,90944	12	0,19	0,18	0,013	0,01	7,488	7	0,051	0,05	7,742	**8**
						0,178		0,015		7,421		0,052		7,666	
						0,17		0,016		7,508		0,054		7,748	
B1	0,1 (73)	427	375,534	0,06887	27	0,187	0,19	0,078	0,08	117,441	116	0,529	0,56	118,234	**117**
						0,19		0,078		115,431		0,505		116,204	
						0,185		0,075		115,947		0,639		116,846	
B1	0.07 (104)	890	544,622	0,036	28	0,192	0,2	0,239	0,25	344,025	345	1,167	1,29	345,625	**347**
						0,195		0,248		345,378		1,474		347,295	
						0,2		0,268		345,985		1,225		347,678	
B1	0.05 (145)	1143	627,992	0,02028	27	0,24	0,23	0,635	0,64	879,406	885	1,819	1,7	882,1	**888**
						0,229		0,655		880,701		1,544		883,13	
						0,23		0,615		893,919		1,643		896,407	
B1	0.04 (181)	1612	724,857	0,017	32	0,266	0,27	1,347	1,35	2088,583	2066	2,87	2,577	2093,066	**2070**
						0,266		1,327		2066,475		2,385		2070,454	
						0,273		1,362		2042,767		2,458		2046,86	

Tabelle 4.4: Auswertung des voxelbasierten Skelettierungsalgorithmus bei unterschiedlichen Rasterauflösungen. t_1: Konvertierung der Punktdaten in Voxeldaten; t_2: Datenvorverarbeitung; t_3: Thinning-Operation; t_4: Extraktion des Skeletts

ID	n	NOB	TBL	MSE	t_1	⊘	t_2	⊘	t_3	⊘	t_4	⊘	$\sum t_i$	⊘
B1	5.000	559	489,143	0,00965	0,02	0,02	12,646	13	425,283	429	1,49	1,5	439,439	**444**
					0,023		13,819		433,745		1,583		449,17	
					0,022		12,878		427,769		1,482		442,151	
B1	10.000	1054	680,982	0,00573	0,03	0,03	52,217	51	829,135	818	4,169	4,2	885,551	**873**
					0,031		49,257		803,838		4,206		857,332	
					0,03		51,024		822,051		4,271		877,376	
B1	15.000	1556	821,982	0,00451	0,04	0,04	114,708	111	1222,701	1232	7,91	7,8	1345,359	**1351**
					0,039		109,095		1244,662		7,489		1361,286	
					0,041		110,481		1227,296		8,119		1345,936	
B1	20.000	2035	937,648	0,00399	0,047	0,05	208,287	206	1663,684	1663	13,09	12,2	1885,108	**1881**
					0,049		199,021		1672,344		10,754		1882,167	
					0,048		209,542		1653,073		12,693		1875,357	
B1	30.000	2974	1131,542	0,00333	0,065	0,12	477,659	472	2574,49	2625	24,768	24	3076,982	**3121**
					0,072		469,488		2652,231		22,376		3144,168	
					0,223		468,191		2647,921		24,855		3141,191	

Tabelle 4.5: Auswertung des graphenbasierten Skelettierungsalgorithmus bei unterschiedlichen Punktanzahlen. t_1: Identifizierung der Bäume in der Punktwolke; t_2: Initiale Konstruktion des Skeletts; t_3: Globale Optimierung; t_4: Bereinigung des Skeletts

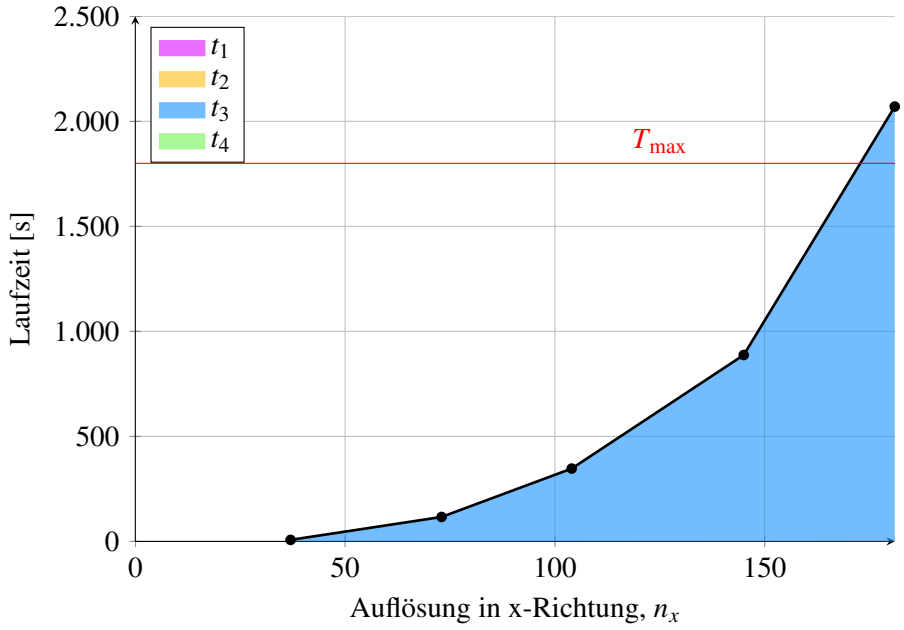

Abbildung 4.6: Abhängigkeit der Laufzeit des voxelbasierten Algorithmus von der Auflösung. t_1: Konvertierung der Punktdaten in Voxeldaten; t_2: Datenvorverarbeitung; t_3: Thinning-Operation; t_4: Extraktion des Skeletts

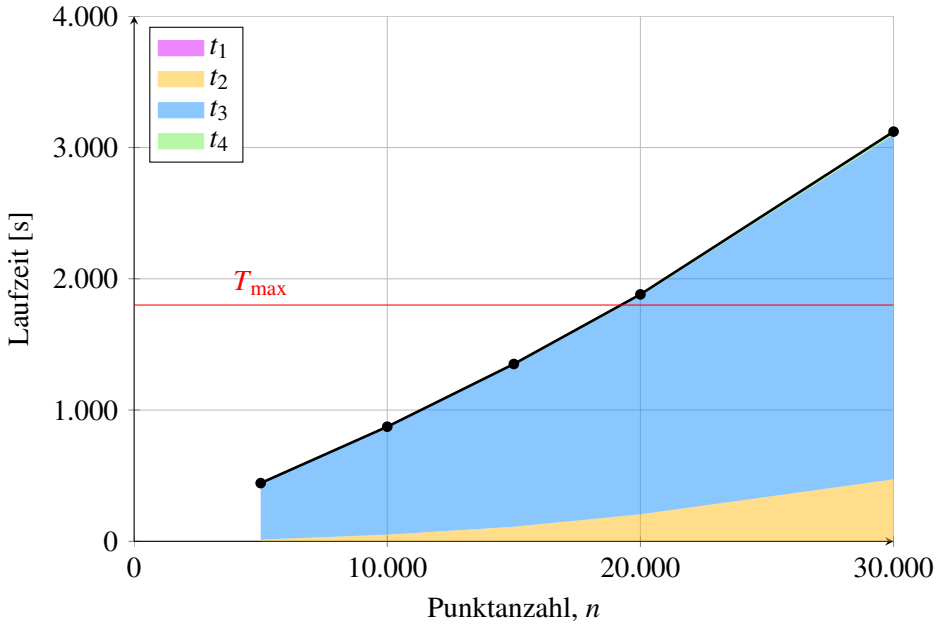

Abbildung 4.7: Abhängigkeit der Laufzeit des graphenbasierten Algorithmus von der Punktanzahl. t_1: Identifizierung der Bäume in der Punktwolke; t_2: Initiale Konstruktion des Skeletts; t_3: Globale Optimierung; t_4: Bereinigung des Skeletts

Die Ergebnisse lassen sich durch die Laufzeitkomplexität der Algorithmen erklären. Laut Bucksch et al. (2012) gehören Skelettierungsalgorithmen, die auf morphologischen Operationen basieren, zur Laufzeitklasse $O(nw)$, wobei n die Anzahl der Voxel und w die Größe des für die morphologischen Operationen verwendeten strukturierenden Elements ist [BLM12]. Die Größe des strukturierenden Elements kann hier als konstant betrachtet werden, da die Verdünnungsoperation von Palágyi und Kuba [PK99] grundsätzlich einen $3 \times 3 \times 3$ Voxel großen Filter erfordert. Vereinfacht wächst die Laufzeit somit linear mit der Anzahl der Voxel, $O(n)$. Die Gesamtanzahl der Voxel n ergibt sich aus den Teilauflösungen des Voxelrasters in jeder Dimension, $n = n_x \cdot n_y \cdot n_z$ mit $n_x, n_y, n_z \in \mathbb{N}$. Bei einer festgelegten räumlichen Größe $l \cdot b \cdot h$ ($l, b, h \in \mathbb{R}^+$) des Voxelrasters, was für die Untersuchung eines einzelnen Objekts gilt, können die Voxelanzahlen der einzelnen Dimensionen ebenfalls in ein festes Verhältnis zueinander gesetzt werden (ohne Berücksichtigung von Rundungen), sodass $n_y = (b/l) \cdot n_x$ und $n_z = (h/l) \cdot n_x$ gelten. Somit berechnet sich die Gesamtanzahl der Voxel als $n = n_x \cdot ((b/l) \cdot n_x) \cdot ((h/l) \cdot n_x) = n_x^3 \cdot (b \cdot h)/l$, wobei $(b \cdot h)/l$ konstant ist. In Bezug auf die Auflösung des Voxelrasters in einer Dimension, wie in Abbildung 4.6 dargestellt, steigt die Laufzeit des Algorithmus somit kubisch mit zunehmender Auflösung an, d. h. der Algorithmus hat eine Laufzeitklasse von $O(n_x^3)$ (Worst Case), wobei n_x die Anzahl der Voxel in x-Richtung ist.

Ein weiterer relevanter Faktor, der in den obigen Ausführungen nicht berücksichtigt wurde, ist der Anstieg der notwendigen Iterationen der Verdünnungsoperation mit zunehmender Anzahl von Voxeln. Bei höherer Auflösung werden die Äste bezüglich der Anzahl der Voxel dicker und erfordern mehrere Iterationen, um diese auf eine ein Voxel dicke Struktur zu reduzieren. Dies wird in den Messergebnissen in Tabelle 4.4 deutlich, wo 12 Iterationen bei einer Voxelbreite von 0.2 und 32 Iterationen bei 0.04 festgestellt wurden (s. Spalte n_{Th}). Die genaue Auswirkung auf die Laufzeit des Algorithmus erfordert jedoch weitere Untersuchungen und wird in dieser Arbeit nicht weiter behandelt.

Der Verlauf der Laufzeit des graphenbasierten Algorithmus lässt sich ebenfalls durch die Laufzeitkomplexität erklären. Bei einer Analyse der Programmlogik zeigt sich, dass der Rechenaufwand eines Optimierungsprozesses durch das Gradientenabsteigsverfahren im Wesentlichen linear mit der Anzahl der zu optimierenden Parameter k ansteigt. Da die Anzahl der Iterationen in der Implementierung nach oben begrenzt ist, kann diese bei der Untersuchung der Worst-Case-Laufzeit als konstant betrachtet werden. Die Anzahl der Parameter k wächst proportional zur Anzahl der Punkte n, da bei jeder Optimierung für jeden Punkt die gleiche Menge an Informationen zu ermitteln ist. Da die Anzahl der Iterationen und somit die Anzahl der einzelnen Optimierungsprozesse begrenzt ist, liegt das Laufzeitverhalten des gesamten Optimierungsprozesses bei $O(n)$ (für t_3). Der Rechenaufwand für die initiale Konstruktion des Skeletts (für t_2) wird maßgeblich durch den Aufwand der Suche nach Paaren benachbarten Punkten im k-d-Baum beeinflusst, sowie die Anzahl der dadurch resultierenden Verbindungen, die anschließend weiterverarbeitet werden müssen. Die Komplexität für die Suche nach einem nächsten Nachbarn, basierend auf der von SciPy verwendeten Methode, ist logarithmisch $O(\log(n))$ [MM99]. Die Suche nach Paaren, die letztendlich eine Suche nach den nächsten Nachbarn für alle n Punkte darstellt, hat damit eine Komplexität von $O(n \cdot \log(n))$. Aus den beiden relevanten Teilaufwänden ergibt sich für den graphenbasierten Skelettierungsalgorithmus eine Laufzeitkomplexität von $O(n \cdot \log(n))$.

Zusammenfassend kann festgehalten werden, dass beide Skelettierungsalgorithmen bei geeigneter Parametrisierung in der Lage sind, ein Skelett in einem Zeitfenster von 30 Minuten zu generieren. Der graphenbasierte Algorithmus zeichnet sich durch eine skalierbare Laufzeitklasse von $O(n \cdot \log(n))$ aus, wobei n die Anzahl der Punkte darstellt, die für die Skelettierung verwendet werden. Im Gegensatz dazu hängt die Laufzeit des voxelbasierten Algorithmus von der Auflösung des Voxelrasters ab. Der voxelbasierte Skelettierungsalgorithmus hat eine Laufzeitkomplexität von $O(n_x^3)$, wobei n_x die Auflösung des Voxelrasters in einer Dimension ist. Es kann als Vorteil angesehen werden, dass die Laufzeit des voxelbasierten Skelettierungsalgorithmus unabhängig von der Anzahl der Punkte in der Punktwolke ist. Jedoch geht die Bemühung um eine erhöhte Genauigkeit mit einem stärkeren Anstieg der Laufzeit des voxelbasierten Algorithmus einher.

5 Erweiterung des graphenbasierten Skelettierungsalgorithmus

In diesem Kapitel werden zwei Erweiterungen des graphenbasierten Skelettierungsalgorithmus vorgestellt. Das Ziel besteht darin, für jeden Knotenpunkt des Skeletts zusätzliche Informationen über einen Baum oder seine Aststruktur zu ermitteln und diese im Skelett zu speichern. Bei der Implementierung wurde der graphenbasierte Algorithmus weiterentwickelt, um bereits berechnete Informationen wiederzuverwenden. Es sei jedoch darauf hingewiesen, dass diese Erweiterungen grundsätzlich auf ein gegebenes Skelett angewendet werden können, unabhängig davon, mit welchem Algorithmus es konstruiert wurde. Für die Anwendung ist lediglich die vorherige Berechnung der dafür benötigten Werte erforderlich, wie in Abschnitt 3.4 beschrieben.

Es wurden zwei Maßzahlen ausgewählt, die zur weiteren Analyse eines Baums hilfreich sein können: Zum einen der Astradius (**Abschnitt 5.1**), der eine genauere Rekonstruktion der Oberfläche oder des Volumens eines Baums aus dem Skelett ermöglicht, und zum anderen die Asttiefe (**Abschnitt 5.2**), die eine Interpretation des Typs und der hierarchischen Position eines Astes liefert. Die entwickelten Algorithmen zur Berechnung dieser Werte werden im Folgenden erläutert.

5.1 Berechnung des Astradius

Es sei anzumerken, dass die im Verlauf des graphenbasierten Algorithmus berechneten Astradien lediglich Approximationen der tatsächlichen Astradien darstellen, die auf einer Reihe von verallgemeinerten Allometrien von Bäumen basieren. Obwohl diese Berechnungen gute Näherungen liefern können, spiegeln die Ergebnisse keine exakte Darstellung des realen Baums wider. Solche genauen Werte sind jedoch erforderlich, um die Oberfläche oder das Volumen des originalen Baums repräsentativ zu modellieren. Da die Punktwolke die einzige Informationsquelle darstellt, besteht die Herausforderung darin, die Radien der Äste aus den Punkten der Punktwolke zu gewinnen. Es wurde ein Verfahren entwickelt, das mit Hilfe der ursprünglichen Punktwolke für jeden Knoten eines Skeletts den Radius des entsprechenden Astes berechnet. Hierbei wurde besonderes Augenmerk auf eine robuste und zuverlässige Berechnung gelegt, um potenzielle Probleme durch Rauschen und Fehler in der Punktwolke zu kompensieren.

Die Grundidee besteht darin, für jeden Knoten des Skeletts Punkte in der Punktwolke auszuwählen, die einen Querschnitt des Baums bzw. Astes an der entsprechenden Stelle darstellen, und anhand dieser den Radius des Astes abzuschätzen. Hierfür wird zunächst eine Querschnittsebene E bestimmt, die auf dem Knoten positioniert ist (der Stützvektor entspricht der Position des Knotens) und deren Normale parallel zum Ast ist. Sei v ein bestimmter Skelett-Knoten mit Elternknoten u und den Kindknoten w_i (s. Abbildung 5.1 (a)), wobei $i \in \{1, 2, \ldots, n\}$ den Index eines Kindknotens darstellt. Die

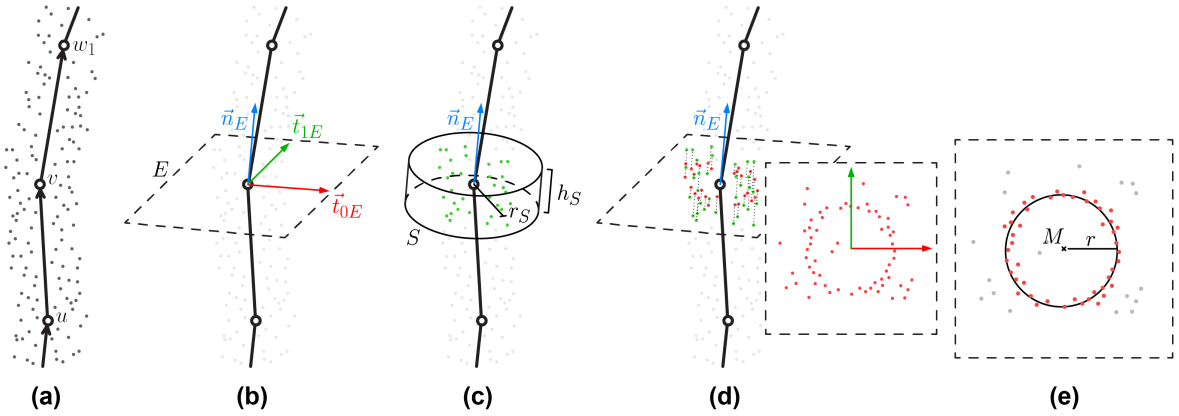

Abbildung 5.1: Berechnung des Astradius. Skelett und Punktwolke (a); Querschnittsebene (b); Zylinder-Suchbereich und ausgewählte Punkte in grün (c); Projektion der Punkte auf die Querschnittsebene (d); Kreisregression mittels RANSAC und Methode der kleinsten Quadrate (e)

Normale der Querschnittsebene $\vec{n}_E$ ergibt sich aus den Richtungen der inzidenten Kanten des Knotens im Skelett entsprechend der Formel in (5.1), wobei $e_{\text{norm}}(u, v)$ die normalisierte Richtung der Kante von u nach v repräsentiert. Hier ist auch zu beachten, dass spezielle Maßnahmen für den Umgang mit Nullvektoren erforderlich sind. Aus der Normalen der Ebene lassen sich gemäß den Formeln in (5.2) zwei Tangentenvektoren $\vec{t}_{0E}$ und $\vec{t}_{1E}$ ableiten (s. Abbildung 5.1 (b)), vorausgesetzt, dass $\vec{n}_E$ nicht parallel zu $\vec{e}_z$ ist.

$$\vec{n}_E = \frac{\vec{e}_{\text{in}} + \vec{e}_{\text{out}}}{\|\vec{e}_{\text{in}} + \vec{e}_{\text{out}}\|}, \vec{e}_{\text{in}} = e_{\text{norm}}(u, v), \vec{e}_{\text{out}} = \frac{\sum_j e_{\text{norm}}(v, w_j)}{\left\|\sum_j e_{\text{norm}}(v, w_j)\right\|} \tag{5.1}$$

$$\vec{t}_{0E} = \frac{\vec{n}_E \times \vec{e}_z}{\|\vec{n}_E \times \vec{e}_z\|}, \vec{t}_{1E} = \frac{\vec{n}_E \times \vec{t}_{0E}}{\|\vec{n}_E \times \vec{t}_{0E}\|}, \vec{e}_z = (0, 0, 1)^\intercal \tag{5.2}$$

Im nächsten Schritt wird in der Punktwolke nach Punkten gesucht, die im Bereich des Querschnitts auf der Oberfläche des Baums liegen. Als Suchbereich wird ein flacher Zylinder S betrachtet (s. Abbildung 5.1 (c)), der auf dem Knoten v zentriert und in Richtung der Normalen der Querschnittsebene $\vec{n}_E$ orientiert ist. Die Höhe des Zylinders h_S wird auf einen vordefinierten Wert festgelegt, während der Radius r_S einem bestimmten Vielfachen des geschätzten Astradius entspricht. Alle Punkte, die sich innerhalb dieses Zylinders befinden, werden für die weiteren Betrachtungen ausgewählt. In der Implementierung werden mit Hilfe eines k-d-Baum zunächst alle Punkte gesucht, die näher als eine halbe Zylinder-Diagonale am Knoten v liegen. Anschließend werden die Punkte, die sich außerhalb des Zylinders befinden, herausgefiltert.

Die Punkte, die innerhalb des Zylinders liegen, werden in das Querschnitts-Koordinatensystem mit den drei Achsen $\vec{t}_{0E}$, $\vec{t}_{1E}$ und $\vec{n}_E$ transformiert. Anschließend werden die Punkte entlang der Achse $\vec{n}_E$ auf die Querschnittsebene E projiziert. Dazu genügt es, die Punktkoordinaten für $\vec{n}_E$ auf 0 zu setzen oder zu ignorieren. Die zweidimensionalen Positionen in Bezug auf die Achsen $\vec{t}_{0E}$ und $\vec{t}_{1E}$ geben die Lage der projizierten Punkte innerhalb der Querschnittsebene an (s. Abbildung 5.1 (d)).

Durch die Projektion entlang der Astrichtung bilden die Punkte auf der Oberfläche des Astes einen Kreis in der Querschnittsebene. Die Aufgabe besteht nun darin, diesen Kreis zu identifizieren und seinen Radius zur Bestimmung des Astradius zu ermitteln. Es ist zu beachten, dass bei der groben Auswahl der Punkte neben den gewünschten Punkten auch solche enthalten sein können, die nicht zu der betrachteten Astoberfläche gehören und somit für die Bestimmung des Astradius irrelevant sind. Dies kann beispielsweise auf Überlappungen des Suchbereichs mit anderen nahe gelegenen Ästen oder auf Rauschen in der Punktwolke zurückzuführen sein. Um die relevanten Punkte von den irrelevanten Punkten zu trennen, wird der RANSAC-Algorithmus verwendet. RANSAC (*Random Sample Consensus*) ist ein Verfahren zur robusten Schätzung eines mathematischen Modells innerhalb einer Datenmenge, die von Rauschen und Ausreißern beeinflusst ist, durch Klassifizierung der Fehlerpunkte [FB81]. Die Anwendung dieses Verfahrens auf den projizierten Punkten bietet eine robuste Möglichkeit, den Kreis trotz des Vorhandenseins von Ausreißern zu identifizieren. Der Algorithmus hierfür wurde in einer Funktion `circle_regression_ransac` implementiert, die die Position M und den Radius r des gefundenen Kreises in der Querschnittsfläche zurückgibt (s. Abbildung 5.1 (e)). Um sicherzustellen, dass der ermittelte Kreis nicht signifikant von der Schätzung abweicht, wird bei der Suche sein Radius auf den Radius des zuvor definierten Zylinders begrenzt.

Nach Anwendung des RANSAC-Algorithmus werden zunächst die Punkte, die sich auf dem ermittelten Kreis befinden (bzw. in der Nähe des Kreises unter Berücksichtigung einer Toleranz), gefiltert und somit von den Ausreißern getrennt. Anschließend wird auf diesen segmentierten Punkten eine zusätzliche Kreisregression unter Verwendung der Methode der kleinsten Quadrate durchgeführt, wobei das Gradientenabstiegsverfahren als Optimierungsmethode angewendet wird, wie bereits in Abschnitt 3.4 beschrieben. Das Ergebnis dieser Regression ist eine optimierte Kreisposition sowie ein Radius des Kreises in der Querschnittsebene.

Um die Radien aller Punkte zu berechnen, wird eine Breitensuche durchgeführt, bei der für jeden Skelett-Knoten der Radius nach dem oben beschriebenen Verfahren ermittelt wird. Aufgrund von Ungenauigkeiten oder unvollständigen Daten kann es zu Rauschen in den berechneten Radien kommen. Um dieses Problem zu mildern, wird eine zusätzliche Glättung über die berechneten Radien durchgeführt, bei der die Radien benachbarter Knoten angeglichen werden. Die berechneten Radien werden dem NetworkX-Graphen, der das Skelett darstellt, als Knotenattribute zugewiesen und mit Hilfe der Radien wird ein Baummodell erzeugt, das die Oberfläche des Baumes darstellt (s. Abbildung 5.2 rechts).

Darüber hinaus wurde das Programm um eine Funktion erweitert, die eine nachträgliche Zentrierung des Skeletts ermöglicht, wodurch das Problem des nicht zentrierten Skeletts kompensiert werden kann. Die Zentrierung erfolgt durch Verschieben der Skelett-Knoten auf die berechneten Kreismittelpunkte. Das Ergebnis im Vergleich zum nicht zentrierten Skelett ist in Abbildung 5.2 dargestellt.

Ein Auszug aus der Implementierung, der die genaue Umsetzung der beschriebenen Verfahren verdeutlicht, ist im Anhang (Codeausschnitt A.1) dargestellt.

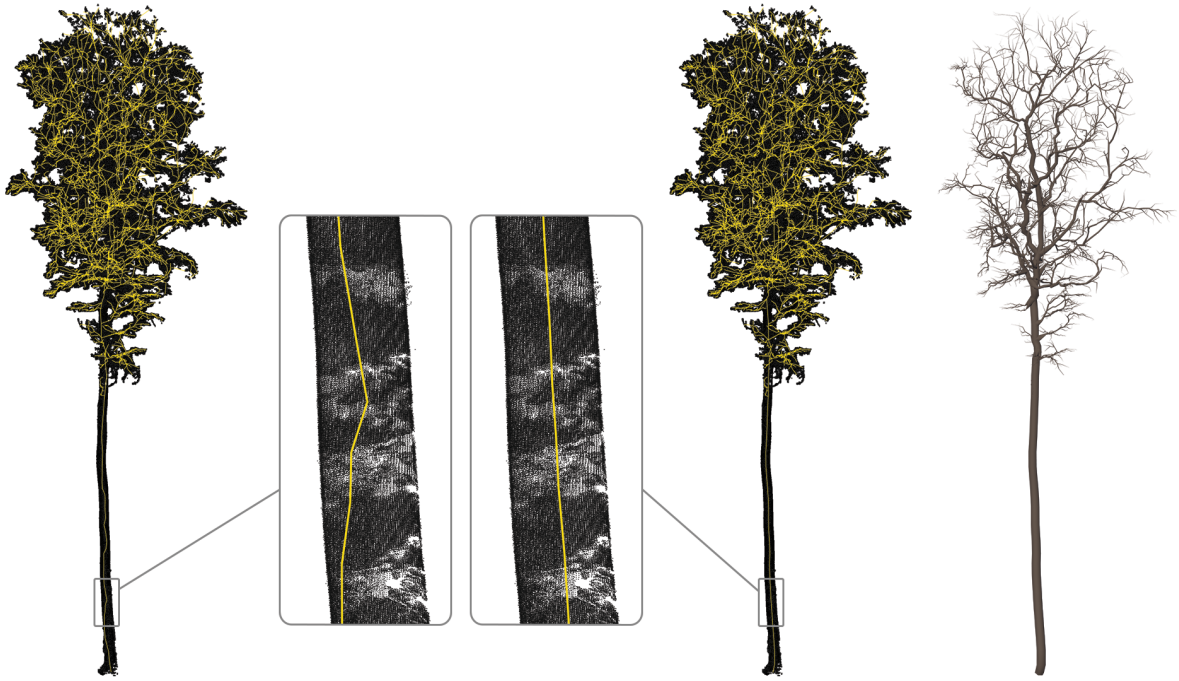

Abbildung 5.2: Skelette des graphenbasierten Algorithmus ohne (links) und mit (Mitte) Zentrierung im Vergleich. Rekonstruiertes Baummodell mit berechneten Astradien (rechts)

5.2 Ermittlung der Asttiefe

Für die nachfolgenden Darlegungen wird die Struktur des Baums als hierarchische Anordnung von Ästen auf mehreren Ebenen betrachtet. Der Stamm des Baums bildet die oberste Ebene der Baumstruktur, mit der Tiefe 0. Von beliebigen Stellen des Stammes aus können Äste abzweigen, die eine Ebene tiefer (Tiefe 1) bilden, und von diesen wiederum weitere Äste oder Zweige mit der Tiefe 2. Dieses Muster wird ähnlich einem Fraktal fortgesetzt. Verallgemeinert befinden sich Äste, die von einem Ast mit der Tiefe k entspringen, auf der Tiefe $k + 1$. Folglich stellt die Asttiefe ein Maß dafür dar, nach wie vielen Verzweigungen ein bestimmter Ast entstanden ist. Ziel des im Folgenden erklärten Verfahrens ist es, für jeden Knoten die hierarchische Tiefe des Astes zu bestimmen, auf dem der Knoten liegt. Es ist jedoch zu beachten, dass die obige Betrachtungsweise sowie die anschließenden Ausführungen ein vereinfachtes Modell von natürlichen Baumstrukturen darstellen, die ihrerseits auf komplexe biologische Wachstumsprozesse zurückzuführen sind.

Die grundlegende Herausforderung besteht darin, Abzweigungen neuer Äste, die folglich zu einer neuen Ebene führen, in der Baumstruktur zu identifizieren. Es sei darauf hingewiesen, dass es aus graphentheoretischer Sicht nicht eindeutig ist, ob eine Verzweigung in der Skelettstruktur (d. h. ein Knoten, dessen Elternknoten noch weitere Kindknoten besitzt) eine Abzweigung zu einem neuen Ast auf einer tieferen hierarchischen Ebene darstellt (Verzweigungstyp T1) oder ob es sich um die Fortsetzung desselben Astes handelt, wobei die Asttiefe unverändert bleibt (Verzweigungstyp T2). Dies ist auf die Tatsache zurückzuführen, dass Äste auch seitlich (auch bekannt als „lateral“ [Her21]) von einem Ast abzweigen können, wodurch in diesem Fall den beiden Abzweigungen unterschiedliche Bedeutungen zukommen.

Zur Unterscheidung der genannten Verzweigungstypen werden zwei Kriterien berücksichtigt. Das erste Kriterium basiert auf einem Gewichtungsverhältnis, das die Gesamtkantenlängen der betreffenden Unterbäume vergleicht. Wenn die Gewichtung durch eine Verzweigung deutlich abnimmt (d. h. der Ast im Verhältnis zum entspringenden Ast klein ist), handelt es sich mit hoher Wahrscheinlichkeit um die Abzweigung eines neuen Astes (T1). Das zweite Kriterium betrachtet den Winkel der Verzweigung. Ist dieser Winkel tendenziell groß, deutet dies ebenfalls mit hoher Wahrscheinlichkeit auf die Abzweigung eines neuen Astes hin (T1). Um den Verzweigungstyp anhand dieser Kriterien eindeutig zu klassifizieren, werden hierfür Schwellwerte festgelegt. Für das erste Kriterium wurde experimentell ein Gewichtungs-Schwellwert von 0.5 und für das zweite Kriterium ein Winkel-Schwellwert von 45° als angemessene Werte ermittelt. Es besteht jedoch die Möglichkeit, diese an spezifische Anforderungen und Eigenschaften bestimmter Pflanzen anzupassen.

Nachdem die grundlegenden Konzepte der Baumstruktur erläutert und ein Verfahren zur Klassifizierung der Verzweigungstypen präsentiert wurden, wird nun der Algorithmus zur Bestimmung der hierarchischen Tiefe der Skelett-Knoten vorgestellt.

Zunächst wird die Tiefe des Wurzelknotens auf den Wert 0 gesetzt. Anschließend wird eine Breitensuche über die Knoten des Skeletts, ausgehend vom Wurzelknoten, durchgeführt. Sei v ein aktuell besuchter Knoten mit Elternknoten u und den Kindknoten $w_i, i \in \{1, 2, \ldots, n\}$. c_v bezeichnet die Gewichtung eines Knotens v und d_v seine Tiefe. Es erfolgt eine Fallunterscheidung basierend auf der Anzahl der Kinder n. Falls v keine Kindknoten hat, wird nichts unternommen. Falls v genau einen Kindknoten w_1 hat, wird die Tiefe des aktuellen Knotens dem Kindknoten zugewiesen, d. h. $d_{w_1} = d_v$.

Falls v mehrere Kindknoten hat ($n > 1$), werden zunächst Kandidaten $w_k, k \in K$ unter den Kindknoten ausgewählt, die den oben genannten Kriterien zur Fortsetzung des Astes (T2) entsprechen. Die Kandidaten müssen den Gewichtungs-Schwellwert des ersten Kriteriums θ_c überschreiten, d. h. $\frac{c_{w_k}}{\sum_j c_{w_j}} > \theta_c$, und den Winkel-Schwellwert des zweiten Kriteriums θ_a unterschreiten, d. h. $\angle(e(u,v), e(v,w_k)) < \theta_a$, wobei $e(u,v)$ der Vektor (die Kante) von u nach v repräsentiert. Wenn Kandidaten ausgewählt wurden ($|K| > 0$), wird, da es nur einen durchgehenden Ast entsprechend (T2) geben kann, aus K ein Kindknoten w_z ausgewählt, der die größte Gewichtung aufweist, also $c_{w_z} = \max\{c_{w_k} | k \in K\}$. Die Tiefe des aktuellen Knotens v wird dann auf diesen übertragen, $d_{w_z} = d_v$. Alle anderen Kindknoten (oder falls keine Kandidaten ausgewählt wurden) stellen Abzweigungen neuer Äste dar, wodurch ihre Tiefe um eins erhöht wird, d. h. $\forall i \in \{1, 2, \ldots, n\}, i \neq z : d_{w_i} = d_v + 1$.

Die Implementierung des Algorithmus ist in vereinfachter Form im Codeausschnitt A.2 im Anhang A.2 abgebildet. Darüber hinaus wurde eine Funktion hinzugefügt, die eine Segmentierung des Skeletts entsprechend der Asttiefen ermöglicht und separate Objekte für die einzelnen Tiefen generiert. Das Ergebnis ist in Abbildung 5.3 dargestellt.

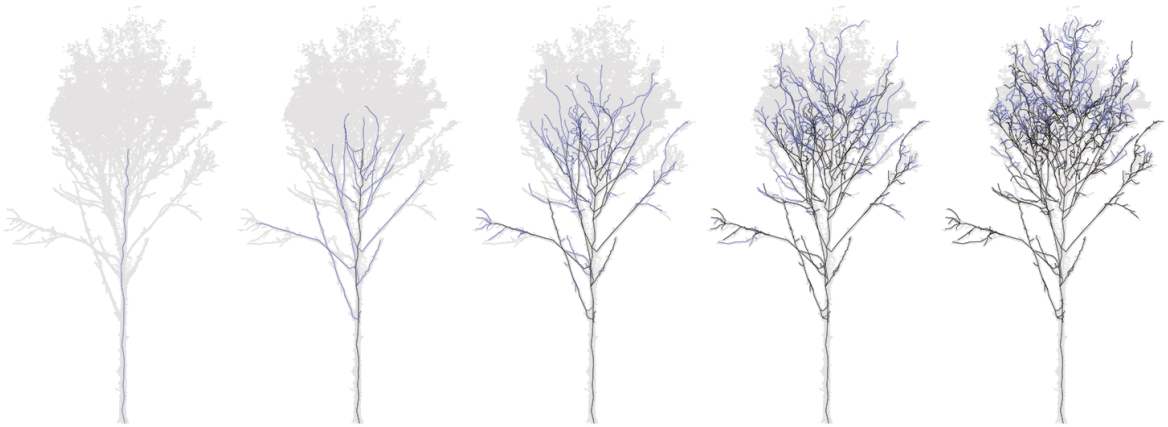

Abbildung 5.3: Segmentierung eines Skeletts nach Asttiefen. V. l. n. r. aufsteigende Asttiefe

6 Fazit

6.1 Zusammenfassung

Die Modellierung botanischer Bäume in virtuellen Welten, die in verschiedenen Bereichen Anwendung finden, ist ein wichtiger Aspekt, um eine realistische und lebendige Umgebung zu schaffen. Neben prozeduralen Verfahren, bei denen Bäume vollständig virtuell erzeugt werden, ermöglicht die Rekonstruktion realer Pflanzen aus der Natur authentischere Modelle. Laserscanning-Verfahren, wie beispielsweise LiDAR, bieten den Vorteil, dass sie direkt 3D-Informationen der erfassten Objekte in Form von Punktwolken liefern, die eine Menge von Punkten im dreidimensionalen Raum repräsentieren. Ein entscheidender Schritt bei der Rekonstruktion von Bäumen aus Punktwolken ist die Skelettierung, bei der ein Skelett extrahiert wird, das die Grundstruktur des Objekts als eindimensionale Struktur beschreibt. Punktwolken stellen die Skelettierung jedoch vor einige Herausforderungen. Sie enthalten keine Informationen über die Konnektivität und können aufgrund von Laserscanning-Verfahren zudem inhomogene Punktdichten, Rauschen oder unvollständige Daten enthalten, was die Aufgabe der Skelettierung erschwert.

Für die Skelettierung von Baum-Punktwolken existieren verschiedene Algorithmen aus unterschiedlichen Verfahrensklassen. Im Rahmen dieser Arbeit wurden zwei Skelettierungsalgorithmen ausgewählt und mit Hilfe der Programmiersprache Python als Add-on für das 3D-Programm Blender implementiert. Der erste Algorithmus ist ein voxelbasierter Skelettierungsalgorithmus (von Gorte und Pfeifer [GP04]), bei dem die Punktdaten in ein dreidimensionales Voxelraster transformiert werden, um daraus mittels morphologischer Operationen ein Skelett zu konstruieren. Der zweite Algorithmus ist ein graphenbasierter Skelettierungsalgorithmus (von Livny et al. [Liv+10]), bei dem das Skelett direkt aus dem kontinuierlichen Vektorraum mit Hilfe der Punktwolke konstruiert und durch einen globalen Optimierungsprozess verfeinert wird. Die Skelette beider Algorithmen sind als Graphen repräsentiert, deren Knoten eine festgelegte Position im Raum haben. Neben den beiden Skelettierungsalgorithmen wurde eine Funktion implementiert, um LAS-Dateien mit Punktwolken zu laden.

Die Auswertung der implementierten Skelettierungsalgorithmen hat gezeigt, dass beide Algorithmen in der Lage sind, Skelette aus gegebenen Punktwolken beliebiger Bäume mit einer maximalen Rechenzeit von 30 Minuten zu konstruieren. Ein genauerer qualitativer und quantitativer Vergleich der beiden Skelettierungsalgorithmen anhand verschiedener Kriterien zeigte jedoch, dass der graphenbasierte Skelettierungsalgorithmus genauer ist und visuell überzeugendere Ergebnisse liefert. Insbesondere gegenüber Rauschen oder fehlerhaften Daten in der Punktwolke ist der graphenbasierte Algorithmus robuster als der voxelbasierte Skelettierungsalgorithmus. Allerdings sind die Skelette des voxelbasierten Algorithmus in bestimmten Fällen besser zentriert als die des graphenbasierten Skelettierungsalgorithmus. Die Laufzeit des graphenbasierten Skelettierungsalgorithmus hängt von der Anzahl der für die Skelettierung berücksichtigten Punkte ab, während die Laufzeit des voxel-

basierten Algorithmus von der Auflösung des Voxelrasters abhängt. Obwohl die Laufzeit des voxelbasierten Algorithmus unabhängig von der Anzahl der Punkte ist, skaliert sie mit zunehmendem Detailgrad deutlich schneller als die des graphenbasierten Skelettierungsalgorithmus.

Zuletzt wurden zwei Erweiterungen für den graphenbasierten Skelettierungsalgorithmus vorgestellt, die für jeden Knoten eines Skeletts sowohl den Radius als auch die Tiefe der Äste berechnen. Mit der Berechnung der Radien ist es außerdem möglich, die Punkte des Skeletts nachträglich zu zentrieren, wodurch das Problem der nicht zentrierten Äste des graphenbasierten Skelettierungsalgorithmus gelöst wird.

6.2 Ausblick

Die in dieser Arbeit implementierten Skelettierungsalgorithmen bieten eine solide Grundlage und Potenzial für Weiterentwicklungen. Der voxelbasierte Skelettierungsalgorithmus benötigt zur effizienten Speicherung und Verarbeitung von Voxeldaten eine geeignete Datenstruktur, die keine "leeren" Voxel speichert. Dadurch kann die Skalierbarkeit des Algorithmus in Bezug auf seine Laufzeit verbessert werden. Zur Verbesserung des graphenbasierten Skelettierungsalgorithmus sollten weitere Optimierungsverfahren in Betracht gezogen werden. Eine effiziente und exakte Lösung der globalen Optimierungsprobleme kann die Laufzeit des Algorithmus erheblich verkürzen und die Qualität der resultierenden Skelette verbessern. Der Vorschlag von Livny et al., ein lineares Gleichungssystem zur Lösung zu verwenden [Liv+10], stellt einen vielversprechenden Ansatz dar. Eine weitere Möglichkeit zur Verbesserung der beiden Algorithmen besteht darin, vor der Skelettierung eine automatische Segmentierung von Objekten durchzuführen, die nicht zum Baum gehören. Dies würde dem Benutzer die zeitaufwändige manuelle Segmentierung ersparen. Darüber hinaus wäre es bei beiden Algorithmen sinnvoll, verschiedene Prozesse zu automatisieren, um die Anzahl der einzustellenden Parameter zu reduzieren. Im Falle des graphenbasierten Skelettierungsalgorithmus ist es beispielsweise erforderlich, die Dicke des Stammes anzugeben, da diese Information zur Bereinigung des Skeletts von fehlerhaften Kanten benötigt wird. Diese Information könnte jedoch mit Hilfe der beschriebenen Erweiterung aus der Punktwolke berechnet werden. Zur weiteren Optimierung der Skelettierung könnte eine Kombination verschiedener Skelettierungsalgorithmen in Betracht gezogen werden, um die Vorteile der einzelnen Verfahren zu kombinieren.

Neben den genannten Verbesserungspotenzialen der Skelettierungsalgorithmen gibt es weiteren Forschungsbedarf im Bereich der Baummodellierung. Nach der Skelettierung stellt sich die Frage, wie aus einem Skelett ein 3D-Modell erzeugt werden kann, welches die Oberfläche des Baums realistisch modelliert. Weiterer Forschungsbedarf besteht daher in der Entwicklung von Verfahren zur Generierung der Baumoberfläche aus dem Skelett. Ein möglicher Ansatz wäre die Rekonstruktion der Baumoberfläche mit Hilfe von künstlicher Intelligenz. Darüber hinaus können prozedurale Verfahren integriert werden, um Details, die durch die Skelettierungsalgorithmen nicht rekonstruiert werden können (wie dünne Zweige oder Bätter), nachträglich synthetisch zu generieren. Des Weiteren wäre es vorteilhaft, aus der Skelettstruktur des Baumes Parameter zu extrahieren, die es ermöglichen, mehrere Varianten derselben Baumart zu generieren, was die Erzeugung großer virtueller Welten erleichtern würde. Hier sind jedoch weitere Untersuchungen erforderlich.

Der Bereich der Baummodellierung ist ein weites Forschungsfeld, das ein erhebliches Potenzial für zukünftige Forschungsarbeiten bietet. Die genannten Entwicklungsmöglichkeiten sind nur einige vielversprechende Richtungen für weiterführende Untersuchungen. Die Erforschung dieser Potenziale ermöglicht es, diesen Bereich weiter voranzutreiben und neue Möglichkeiten für die Generierung virtueller Welten zu erschließen.

A Anhang

A.1 Abbildungen

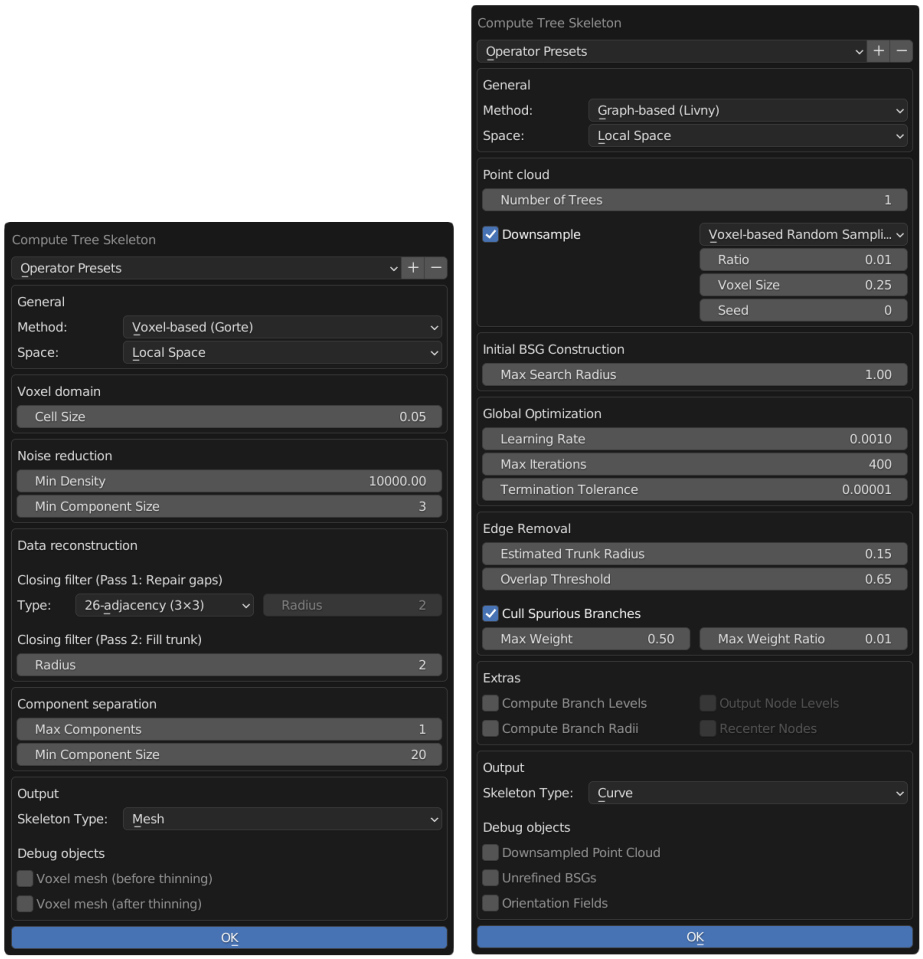

Abbildung A.1: Grafische Benutzeroberfläche des Operators mit Einstellungen zum voxelbasierten Algorithmus (a) und zum graphenbasierten Algorithmus (b)

A.2 Codeausschnitte

Listing A.1: Erweiterung 1 – Berechnung der Astradien

```
def compute_branch_radii(bsg, root, points, search_radius_factor=1.5,
    search_normal_depth=0.04, recenter=False, smoothing_iterations=10,
    smoothing_strength=0.5):
    points_kdtree = spatial.KDTree(points)

    def get_node_direction(current):
        current_pos = bsg.nodes[current]['position']

        # Richtung der eingehenden Kante berechnen
        parent = next(bsg.predecessors(current), current)
        parent_pos = bsg.nodes[parent]['position']
        in_dir = current_pos - parent_pos
        l = np.linalg.norm(in_dir)
        if l > 0.0: in_dir /= l

        # Richtung der ausgehenden Kante berechnen
        out_dir = np.zeros((3,), dtype=float)
        for child in bsg.neighbors(current):
            child_pos = bsg.nodes[child]['position']
            edge_dir = child_pos - current_pos
            l = np.linalg.norm(edge_dir)
            if l > 0.0: edge_dir /= l
            out_dir += edge_dir

        # Richtung des Astes in dem Knoten berechnen
        direction = in_dir + out_dir
        l = np.linalg.norm(direction)
        if l > 0.0: direction /= l
        else: direction = np.array((0,0,1), dtype=float)
        return direction

    def compute_tangent_vectors(normal):
        i3 = np.identity(3)

        if normal[0] == 0.0 and normal[1] == 0.0:
            return (i3[0], i3[1])

        i0 = np.cross(normal, i3[2])
        i0 /= np.linalg.norm(i0)
        i1 = np.cross(normal, i0)
        i1 /= np.linalg.norm(i1)
        return (i0, i1)

    def estimate_node_circle(current):
        current_node = bsg.nodes[current]

        # Position, Normale und Tangenten der Querschnittsebene ermitteln
        origin = current_node['position']
```

```
47          plane_normal = get_node_direction(current)
48          plane_t0, plane_t1 = compute_tangent_vectors(plane_normal)
49
50          # Relevante Punkte suchen
51          search_cylinder_radius = current_node['radius'] * search_radius_factor
52          search_sphere_radius = math.sqrt(search_cylinder_radius**2 +
                search_normal_depth**2)
53          found_point_indices = points_kdtree.query_ball_point(origin,
                search_sphere_radius)
54          found_points = points[found_point_indices]
55
56          # Transformation der Punkte in Ebenen-Koordinatensystem
57          found_points_offset = found_points - origin
58          found_points_n = np.dot(found_points_offset, plane_normal)
59          found_points_t0 = np.dot(found_points_offset, plane_t0)
60          found_points_t1 = np.dot(found_points_offset, plane_t1)
61
62          # Punkte filtern, die nicht im Zylinder liegen
63          found_points_mask = np.logical_and(np.abs(found_points_n) <=
                search_normal_depth, np.sqrt(found_points_t0**2 +
                found_points_t1**2) <= search_cylinder_radius)
64          found_points_t0 = found_points_t0[found_points_mask]
65          found_points_t1 = found_points_t1[found_points_mask]
66
67          # Projektion der Punkte auf die Ebene
68          projected_points = np.column_stack((found_points_t0, found_points_t1))
69
70          # Regression eines Kreises auf Punkten durchführen
71          circle_pos, circle_radius = circle_regression_ransac(projected_points,
                tolerance=0.02, rmax=search_cylinder_radius)
72          filtered_points = filter_points_on_circle(projected_points, circle_pos,
                circle_radius, tolerance=0.04)
73          circle_pos, circle_radius = circle_regression_lsq(filtered_points,
                max_iter=100, x0_pos=circle_pos, x0_r=circle_radius,
                rmax=search_cylinder_radius)
74
75          # Neue Position berechnen
76          circle_position = origin + circle_pos[0] * plane_t0 + circle_pos[1] *
                plane_t1
77          return (circle_position, circle_radius)
78
79      # Phase 1: Radien individuell berechnen
80      circle_position, circle_radius = estimate_node_circle(root)
81      if recenter:
82          bsg.nodes[root]['position'] = circle_position
83      bsg.nodes[root]['radius'] = circle_radius
84
85      for u, v in nx.dfs_edges(bsg, root):
86          circle_position, circle_radius = estimate_node_circle(v)
87          if recenter:
88              bsg.nodes[v]['position'] = circle_position
89          bsg.nodes[v]['radius'] = circle_radius
90
```

```
    # Phase 2: Glättung durchführen
    for _ in range(smoothing_iterations):
        for u, v in nx.dfs_edges(bsg, root):
            u_weight = bsg.nodes[u]['weight']
            u_radius = bsg.nodes[u]['radius']
            v_weight = bsg.nodes[v]['weight']

            # Aktueller Radius
            current_radius = bsg.nodes[v]['radius']

            # Abschätzung des gewünschten Radius (durch Allometrie)
            v_siblings = sum(1 for _ in bsg.neighbors(u)) - 1
            radius_ratio = v_weight / u_weight if u_weight > 0.0 else 0.0
            radius_ratio **= 1.5 if v_siblings == 0 else 0.4
            estimated_radius = u_radius * radius_ratio

            # Gewichtetes Mittel des aktuellen und des gewünschten Radius
            bsg.nodes[v]['radius'] = smoothing_strength * estimated_radius + (1
                - smoothing_strength) * current_radius
    return bsg
```

Listing A.2: Erweiterung 2 – Berechnung der Asttiefen

```
def compute_branch_levels(bsg, root, weight_threshold=0.5,
    angle_threshold=0.7854):
    cos_angle_threshold = np.cos(angle_threshold)

    def process_node_children(u, v):
        u_node = bsg.nodes[u]
        v_node = bsg.nodes[v]
        children = list(bsg.neighbors(v))
        if len(children) == 0: return
        if len(children) == 1:
            bsg.nodes[children[0]]['depth'] = v_node['depth']
            return

        children_weight = sum(bsg.nodes[child]['weight'] for child in children)
        uv_vec = v_node['position'] - u_node['position']
        uv_length = np.linalg.norm(uv_vec)

        # Kandidaten für durchgehende Äste sammeln
        candidates = []
        for w in children:
            w_node = bsg.nodes[w]

            # Erste Bedingung: Gewichtungs-Schwellwert
            if children_weight != 0:
                weight_ratio = w_node['weight'] / children_weight
                if weight_ratio < weight_threshold:
                    continue

            # Zweite bedingung: Winkel-Schwellwert
```

```
29              vw_vec = w_node['position'] - v_node['position']
30              vw_length = np.linalg.norm(vw_vec)
31              if uv_length > 0.0 and vw_length > 0.0:
32                  if np.dot(uv_vec,vw_vec)/(uv_length*vw_length) <
                        cos_angle_threshold:
33                      continue
34              candidates.append(w)
35
36          # Wähle den Kandidaten mit der größten Gewichtung
37          chosen = None
38          if len(candidates) > 0:
39              chosen = max(candidates, key=lambda node: bsg.nodes[node]['weight'])
40              bsg.nodes[chosen]['depth'] = v_node['depth']
41
42          # Inkrementiere die Asttiefen alle anderen Kindknoten
43          next_level = v_node['depth'] + 1
44          for child in children:
45              if child != chosen:
46                  bsg.nodes[child]['depth'] = next_level
47
48      bsg.nodes[root]['depth'] = 0
49      process_node_children(root, root)
50      for u, v in nx.bfs_edges(bsg, root):
51          process_node_children(u, v)
52      return bsg
```

Literatur

[AFM15] A. Araujo, C. França und H. Moura. *COMPLEXITY WITHIN SOFTWARE DEVELOPMENT PROJECTS: AN EXPLORATORY OVERVIEW*. Okt. 2015.

[AK14] F. Aiteanu und R. Klein. Hybrid tree reconstruction from inhomogeneous point clouds. en. In: *The Visual Computer* 30.6-8 (Juni 2014), S. 763–771. ISSN: 0178-2789, 1432-2315. DOI: 10.1007/s00371-014-0977-7. URL: http://link.springer.com/10.1007/s00371-014-0977-7 (besucht am 23. 02. 2023).

[ASP19] The American Society for Photogrammetry & Remote Sensing. *LAS Specification 1.4 - R15*. Juli 2019.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. In: *Communications of the ACM* 18.9 (Sep. 1975), S. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://dl.acm.org/doi/10.1145/361002.361007 (besucht am 29. 06. 2023).

[BL08] A. Bucksch und R. Lindenbergh. CAMPINO — A skeletonization method for point cloud processing. en. In: *ISPRS Journal of Photogrammetry and Remote Sensing*. Theme Issue: Terrestrial Laser Scanning 63.1 (Jan. 2008), S. 115–127. ISSN: 0924-2716. DOI: 10.1016/j.isprsjprs.2007.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0924271607001281 (besucht am 06. 01. 2023).

[Ble23] Blender Foundation. *Blender*. 2023. URL: https://www.blender.org/ (besucht am 11. 07. 2023).

[BLM09] A. Bucksch, R. C. Lindenbergh und M. Menenti. *SkelTre - Fast Skeletonisation for Imperfect Point Cloud Data of Botanic Trees*. en. Accepted: 2013-10-21T18:00:31Z ISSN: 1997-0463. The Eurographics Association, 2009. ISBN: 978-3-905674-16-3. DOI: 10.2312/3DOR/3DOR09/013-020. URL: https://diglib.eg.org:443/xmlui/handle/10.2312/3DOR.3DOR09.013-020 (besucht am 06. 01. 2023).

[BLM12] A. Bucksch, R. Lindenbergh und M. Menenti. SkelTre: Robust skeleton extraction from imperfect point clouds. In: *The Visual Computer* 26 (Apr. 2012), S. 1283–1300. DOI: 10.1007/s00371-010-0520-4.

[BM23] G. Brown und T. Montaigu. *laspy: Native Python ASPRS LAS read/write library*. 2023. URL: https://github.com/laspy/laspy (besucht am 11. 07. 2023).

[CG20] A. Chaudhury und C. Godin. Skeletonization of Plant Point Cloud Data Using Stochastic Optimization Framework. In: *Frontiers in Plant Science* 11 (2020). ISSN: 1664-462X. URL: https://www.frontiersin.org/articles/10.3389/fpls.2020.00773 (besucht am 19. 02. 2023).

[Che+08] X. Chen, B. Neubert, Y.-Q. Xu, O. Deussen und S. B. Kang. Sketch-based tree modeling using Markov random field. en. In: *ACM Transactions on Graphics* 27.5 (Dez. 2008), S. 1–9. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/1409060.1409062. URL: https://dl.acm.org/doi/10.1145/1409060.1409062 (besucht am 14. 06. 2023).

[Col23] Collins Dictionary. *Voxel Definition und Bedeutung*. de. Juni 2023. URL: https://www.collinsdictionary.com/de/worterbuch/englisch/voxel (besucht am 30. 06. 2023).

[CSM07] N. D. Cornea, D. Silver und P. Min. Curve-Skeleton Properties, Applications, and Algorithms. en. In: *IEEE Transactions on Visualization and Computer Graphics* 13.3 (Mai 2007), S. 530–548. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.1002. URL: https://ieeexplore.ieee.org/document/4135658 (besucht am 18. 02. 2023).

[Eas23] Easy Render. *Most Important 3D Architectural Visualization Podcasts*. en. Mai 2023. URL: https://www.easyrender.com/a/most-important-3d-architectural-visualization-podcasts,%20https://www.easyrender.com/a/most-important-3d-architectural-visualization-podcasts (besucht am 12. 07. 2023).

[FB81] M. A. Fischler und R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. In: *Communications of the ACM* 24.6 (Juni 1981), S. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: https://dl.acm.org/doi/10.1145/358669.358692 (besucht am 03. 07. 2023).

[GP04] B. Gorte und N. Pfeifer. Structuring laser-scanned trees using 3D mathematical morphology. In: Bd. 35. Jan. 2004, S. 929–933.

[Gra23a] Graswald GmbH. *Graswald – State of the art 3D nature*. en. 2023. URL: https://www.graswald3d.com/ (besucht am 11. 07. 2023).

[Gra23b] Graswald GmbH. *Gscatter - powerful layer-based scattering for free*. en. 2023. URL: https://www.graswald3d.com/gscatter (besucht am 11. 07. 2023).

[GW04] B. Gorte und D. Winterhalder. Reconstruction of Laser-Scanned Trees Using Filter Operations in the 3D-Raster Domain. In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 36 (Jan. 2004), S. 39–44.

[HBC23] A. Hale, A. Butcher und CansecoGPC. *Sapling Tree Gen — Blender Manual*. 2023. URL: https://docs.blender.org/manual/en/latest/addons/add_curve/sapling.html (besucht am 11. 07. 2023).

[Her21] E. J. Hermsen. *Branching*. en-US. Aug. 2021. URL: https://www.digitalatlasofancientlife.org/learn/embryophytes/tracheophytes/branching/ (besucht am 02. 07. 2023).

[Hon71] H. Honda. Description of the form of trees by the parameters of the tree-like body: effects of the branching angle and the branch length on the sample of the tree-like body. eng. In: *Journal of Theoretical Biology* 31.2 (Mai 1971), S. 331–338. ISSN: 0022-5193. DOI: 10.1016/0022-5193(71)90191-3.

[Hua+13] H. Huang, S. Wu, D. Cohen-Or, M. Gong, H. Zhang, G. Li und B. Chen. L1-medial skeleton of point cloud. In: *ACM Transactions on Graphics* 32.4 (Juli 2013), 65:1–65:8. ISSN: 0730-0301. DOI: 10.1145/2461912.2461913. URL: https://doi.org/10.1145/2461912.2461913 (besucht am 06. 01. 2023).

[Ise13] M. Isenburg. LASzip: lossless compression of LiDAR data. In: *Photogrammetric Engineering & Remote Sensing* 79 (Feb. 2013). DOI: 10.14358/PERS.79.2.209.

[LBW17] R. Li, G. Bu und P. Wang. An Automatic Tree Skeleton Extracting Method Based on Point Cloud of Terrestrial Laser Scanner. en. In: *International Journal of Optics* 2017 (Okt. 2017). Publisher: Hindawi, e5408503. ISSN: 1687-9384. DOI: 10.1155/2017/5408503. URL: https://www.hindawi.com/journals/ijo/2017/5408503/ (besucht am 06. 01. 2023).

[Lin68] A. Lindenmayer. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. en. In: *Journal of Theoretical Biology* 18.3 (März 1968), S. 300–315. ISSN: 0022-5193. DOI: 10.1016/0022-5193(68)90080-5. URL: https://www.sciencedirect.com/science/article/pii/0022519368900805 (besucht am 14. 06. 2023).

[Liu+21] Y. Liu, J. Guo, B. Benes, O. Deussen, X. Zhang und H. Huang. TreePartNet: neural decomposition of point clouds for 3D tree reconstruction. In: *ACM Transactions on Graphics* 40 (Dez. 2021), S. 1–16. DOI: 10.1145/3478513.3480486.

[Liv+10] Y. Livny, F. Yan, M. Olson, B. Chen, H. Zhang und J. El-Sana. Automatic reconstruction of tree skeletal structures from point clouds. In: *ACM Transactions on Graphics* 29.6 (Dez. 2010), 151:1–151:8. ISSN: 0730-0301. DOI: 10.1145/1882261.1866177. URL: https://doi.org/10.1145/1882261.1866177 (besucht am 06. 01. 2023).

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In: 1967. URL: https : / / www . semanticscholar . org / paper / Some - methods - for - classification - and - analysis - of - MacQueen / ac8ab51a86f1a9ae74dd0e4576d1a019f5e654ed (besucht am 30. 06. 2023).

[Mei+16] J. Mei, L. Zhang, S. Wu, W. Zhen und L. Zhang. 3D tree modeling from incomplete point clouds via optimization and L1-MST. In: *International Journal of Geographical Information Science* 31 (Nov. 2016), S. 1–23. DOI: 10.1080/13658816.2016.1264075.

[MM99] S. Maneewongvatana und D. M. Mount. *Analysis of approximate nearest neighbor searching with clustered point sets*. en. Jan. 1999. URL: https://arxiv.org/abs/cs/9901013v1 (besucht am 12. 07. 2023).

[MMD04] A. Martinez, I. Martín und G. Drettakis. Volumetric Reconstruction and Interactive Rendering of Trees from Photographs. In: *ACM Trans. Graph.* 23 (Aug. 2004), S. 720–727. DOI: 10.1145/1015706.1015785.

[Net23] NetworkX developers. *NetworkX — Network Analysis in Python*. 2023. URL: https://networkx.org/ (besucht am 11. 07. 2023).

[NFD07] B. Neubert, T. Franken und O. Deussen. Approximate image-based tree-modeling using particle flows. en. In: *ACM Transactions on Graphics* 26.3 (Juli 2007), S. 88. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/1276377.1276487. URL: https://dl.acm.org/doi/10.1145/1276377.1276487 (besucht am 09. 04. 2023).

[NOA23] N. O. a. A. A. US Department of Commerce. *What is LIDAR*. EN-US. Jan. 2023. URL: https://oceanservice.noaa.gov/facts/lidar.html (besucht am 15. 06. 2023).

[Num23] NumPy developers. *NumPy*. 2023. URL: https://numpy.org/ (besucht am 11. 07. 2023).

[PK99] K. Palágyi und A. Kuba. "Directional 3D Thinning Using 8 Subiterations". en. In: *Discrete Geometry for Computer Imagery*. Hrsg. von G. Goos, J. Hartmanis, J. van Leeuwen, G. Bertrand, M. Couprie und L. Perroton. Bd. 1568. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, S. 325–336. ISBN: 978-3-540-65685-2 978-3-540-49126-2. DOI: 10.1007/3-540-49126-0_25. URL: http://link.springer.com/10.1007/3-540-49126-0_25 (besucht am 26. 02. 2023).

[PL90] P. Prusinkiewicz und A. Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer Verlag, New York, 1990. ISBN: 978-1-4613-8476-2.

[Sci23] SciPy developers. *SciPy*. 2023. URL: https://scipy.org/ (besucht am 11. 07. 2023).

[Ser83] J. Serra. *Image Analysis and Mathematical Morphology*. USA: Academic Press, Inc., 1983. ISBN: 978-0-12-637240-3.

[Shl+01] I. Shlyakhter, M. Rozenoer, J. Dorsey und S. Teller. Reconstructing 3D tree model from instrumented photograph. In: *Computer Graphics and Applications, IEEE* 21 (Juni 2001), S. 53–61. DOI: 10.1109/38.920627.

[Spe23] Spektrum der Wissenschaft Verlag. *Morphologie*. de. 2023. URL: https://www.spektrum.de/lexikon/biologie/morphologie/44060 (besucht am 12. 07. 2023).

[Sun+22] Sunil Bhutada, Nakerakanti Yashwanth, Puppala Dheeraj und Kethavath Shekar. Opening and closing in morphological image processing. en. In: *World Journal of Advanced Research and Reviews* 14.3 (Juni 2022), S. 687–695. ISSN: 25819615. DOI: 10.30574/wjarr.2022.14.3.0576. URL: https://wjarr.com/content/opening-and-closing-morphological-image-processing (besucht am 17. 06. 2023).

[The23] The SciPy community. *Optimization and root finding (scipy.optimize) — SciPy v1.11.1 Manual*. 2023. URL: https://docs.scipy.org/doc/scipy/reference/optimize.html (besucht am 11. 07. 2023).

[UL15] J. Unger und S. Leyer. "Allometrie". de. In: *Dimensionshomogenität: Erkenntnis ohne Wissen?* Hrsg. von J. Unger und S. Leyer. Wiesbaden: Springer Fachmedien, 2015, S. 122–127. ISBN: 978-3-658-05412-0. DOI: 10.1007/978-3-658-05412-0_7. URL: https://doi.org/10.1007/978-3-658-05412-0_7 (besucht am 12. 07. 2023).

[Ver91] B. J. H. Verwer. Local distances for distance transformations in two and three dimensions. In: *Pattern Recognition Letters* 12 (Jan. 1991). ADS Bibcode: 1991PaReL..12..671V, S. 671–682. DOI: 10.1016/0167-8655(91)90004-6. URL: https://ui.adsabs.harvard.edu/abs/1991PaReL..12..671V (besucht am 18. 06. 2023).

[Wei+22a] H. Weiser, J. Schäfer, L. Winiwarter, N. Krašovec, F. E. Fassnacht und B. Höfle. Individual tree point clouds and tree measurements from multi-platform laser scanning in German forests. en. In: *Open Access* (2022).

[Wei+22b] H. Weiser u. a. *Terrestrial, UAV-borne, and airborne laser scanning point clouds of central European forest plots, Germany, with extracted individual trees and manual forest inventory measurements*. en. Publisher: PANGAEA. März 2022. DOI: 10.1594/PANGAEA.942856. URL: https://doi.pangaea.de/10.1594/PANGAEA.942856?format=html#download (besucht am 06. 01. 2023).

[Wit+09] J. Wither, F. Boudon, M.-P. Cani und C. Godin. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. en. In: *Computer Graphics Forum* 28.2 (Apr. 2009), S. 541–550. ISSN: 01677055, 14678659. DOI: 10.1111/j.1467-8659.2009.01394.x. URL: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01394.x (besucht am 14. 06. 2023).

[WP95] J. Weber und J. Penn. Creation and rendering of realistic trees. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. SIGGRAPH '95. New York, NY, USA: Association for Computing Machinery, Sep. 1995, S. 119–128. ISBN: 978-0-89791-701-8. DOI: 10.1145/218380.218427. URL: https://dl.acm.org/doi/10.1145/218380.218427 (besucht am 05. 07. 2023).

[XGC07] H. Xu, N. Gossett und B. Chen. Knowledge and heuristic-based modeling of laser-scanned trees. In: *ACM Transactions on Graphics* 26.4 (Okt. 2007), 19–es. ISSN: 0730-0301. DOI: 10.1145/1289603.1289610. URL: https://doi.org/10.1145/1289603.1289610 (besucht am 18. 02. 2023).

[Zen+07] G. Zeng, J. Wang, S. B. Kang und L. Quan. Image-based tree modeling. In: *ACM Trans. Graph.* 26 (Juli 2007), S. 87. DOI: 10.1145/1275808.1276486.