Technische Universität München
Fakultät für Bauingenieur- und Vermessungswesen
Lehrstuhl für Wasserbau und Wasserwirtschaft
Prof. Dr. Peter Rutschmann
Lehrstuhl Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

Aufbereitung von Laserscandaten zur Berechnung überregionaler Hochwasserszenarien

Dorothea Pörtge
Bachelorarbeit

Studiengang: Umweltingenieurwesen B. Sc.

Autor: Dorothea Pörtge
Matrikelnummer: [redacted]
1. Betreuer: Prof. Dr.-Ing. André Borrmann
2. Betreuer: M. Sc. Tobias Liepert

Ausgabedatum: 13.06.2012
Abgabedatum: 29.08.2012

Zusammenfassung

Die vorliegende Arbeit behandelt nach einer Einführung in die Grundlagen der Beschaffung von Laserscandaten zunächst den Vergleich von Aufbereitungsalgorithmen und schließlich die Methoden und Ergebnisse einer eigenen Implementierung eines solchen Algorithmus.

Topologische Informationen über ein Überflutungsgebiet in Form von digitalen Geländemodellen sind die Grundlage für hydrodynamisch-numerische Berechnungen, die auch bei Hochwassersimulationen zum Einsatz kommen. Digitale Geländemodelle mit hoher Punktdichte werden heutzutage durch modernes flugzeuggestütztes Laserscanning (Airborne Laserscanning) beschafft. Um Rechenzeiten bei Hochwassersimulationen niedrig zu halten, gibt es Algorithmen, die dafür verwendet werden, digitale Geländemodelle nach bestimmten Kriterien mit möglichst wenig Informationsverlust zu vereinfachen.

Zum einen gibt es Algorithmen, die aus einer Ausgangsdatenmenge wichtige topologische Objekte wie Geländebruchkanten automatisch erkennen und extrahieren. Zum anderen gibt es solche, die Datensätze nach bestimmten Kriterien ausdünnen und so das Netz um einen gewissen Prozentsatz verkleinern.

Der verwendete Algorithmus bei der Implementierung mit Matlab beruht auf dem von J. Wu und L. Kobbelt in „A Stream Algorithm For the Decimation of Massive Meshes“ vorgestellten Algorithmus [23].

Abstract

This thesis examines after a short introduction to the basics of Airborne Laserscanning firstly a comparison from different types of processing algorithms and secondly the methods and results from an own implementation of such an algorithm.

Topological information for flood plains in digital terrain model are the basic data for hydrodynamical-numerical computation, which are for example part of flood simulation. Digital terrain models with a high densitiy of measurement points are derived from Airborne Laserscanning. In order to receive fast computing times, there are algorithms to simplify such models with only a little loss of information. On the one hand there are algorithms which automatically derive and extract important topological objects, such as breaklines; on the other hand there are algorithms which thin unnecessary points away form the data by means of certain criteria. Part of this thesis is an own implementation with Matlab. The programmed algorithm is based on the paper “A Stream Algorithm For the Decimation of Massive Meshes” [23]from J. Wu and L. Kobbelt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Bildnachweis

Abbildung 1: http://www.terraimaging.de/upload/Bilder/Technology/9_tech_laser_2_pop.jpg. Abgerufen: 05.08.2012

Abbildung 2: Lischinski D., Incremental Delaunay-Triangulation, S. 3.

Abbildung 3: Lischinski D., Incremental Delaunay-Triangulation, S. 3. Mit Gimp bearbeitet.

Abbildung 4: Kobbelt L., Campagna S., Seidel H.-P., A General Framework for Mesh Decimation, S. 2.

Abbildung 5: Lund K., Sigmon P., Schlicker S., The Occurrence of Fibonacci and Lucas Numbers in the Geometry of H(Rn), S. 2

Abbildung 6: Kobbelt L., Campagna S., Seidel H.-P., A General Framework for Mesh Decimation, S. 6.

Abbildung 7: Brügelmann R., Automatic Breakline Detection from Airborne Laser Range Data, in International Archives of Photogrammetry and Remote Sensing. Vol. XXXIII, Part B3. S. 113.

Abbildung 8: Dyn A., Iske A., Wendland H., Meshfree Thinning of 3D Point Clouds, S. 17.

Abbildung 9: Erstellt mit Google Earth.

Abbildung 10: Wu J., Kobbelt L., A Stream Algorithm for the Decimation of Massive Meshes, S. 1.

Abbildung 11: Erstellt mit AutoCAD.

Abbildung 12-23: Erstellt mit Matlab.

Abbildung 24-25: Erstellt mit Gnuplot.

1. Einführung in das Thema: Bedeutung von numerischen Aufbereitungsalgorithmen für Hochwassersimulationen

1.1 Problemstellung

In den letzten Jahren gab es einige schwere Hochwasser in Deutschland. Besonders gravierend waren die Folgen des Pfingsthochwassers im Jahre 1999 und des Elbhochwassers 2002. Der dabei verursachte Schaden war vor allem deswegen so groß, weil heutzutage durch Siedlungsdruck oftmals die für den natürlichen Hochwasserschutz wichtigen Auengebiete von Flüssen besiedelt werden. Der Auenzustandsbericht des Bundesministeriums für Umwelt, Naturschutz und Reaktorsicherheit hat ergeben, dass in einigen Bereichen der Donau bis zu 90 % der Auen verloren gegangen sind. Mit der Besiedelung von Flussauen sind außerdem bauliche Maßnahmen wie Flussbegradigungen verbunden – auch dieser Aspekt wirkt einem Hochwasser keinesfalls entgegen [14] [22]. Die Notwendigkeit von wirksamen Schutzvorrichtungen wird immer bedeutender, um in Zukunft die Auswirkungen von Überflutungen zu begrenzen.

Hochwasserschutz ist ein wichtiger Bestandteil der Wirkungsbereiche wasserwirtschaftlicher Maßnahmen. Die ökonomischen Effekte, die daraus erzielt werden, sind die Verhinderung von Schäden durch Hochwasser sowie daraus verminderte Kosten für Schadensbehebung. Durch Maßnahmen wie Hochwasserrückhaltebecken oder Aufforstung soll es ermöglicht werden, Hochwasserabflüsse zu reduzieren und zu kontrollieren. Ein nicht-baulicher Bestandteil des Hochwasserschutz ist die Hochwasservorhersage [19].

Zweck der Hochwasservorhersage ist es beispielsweise mit möglichst geringer Vorlaufzeit nach einem Starkregenereignis die gefährdeten Überschwemmungsgebiete zu bestimmen und die Betroffenen vorzuwarnen.

Grundlagen für Hochwasservorhersagen bilden Ergebnisse aus hydrodynamisch-numerischen Simulationen. Solche Simulationen können zum Beispiel mit der kostenlosen Software BASEMENT durchgeführt werden (siehe dazu 6. Ausblick). Neben vielseitigen Eingabedaten werden vor allem Informationen über die Geländetopografie benötigt. Diese kann man durch modernes flugzeuggestütztes Laserscanning (Airborne Laserscanning) bestimmen. Das Problem dabei ist, dass die Punktdichte des resultierenden digitalen Geländemodells zu groß ist, um die Berechnungsdauern in einem zeitlich sinnvollen Rahmen halten zu können. Die Rechenzeiten würden

schlichtweg zu lange dauern, um dann überhaupt noch von einer Vorhersage sprechen zu können. Um dieses Problem zu umgehen, gibt es Algorithmen, die diese große Datenmenge modifizieren, so die Rechenzeit minimieren und eine Vorhersage von Überflutungsgebieten ermöglichen. Diese Algorithmen besitzen stets ein digitales Geländemodell als Eingabe- und Ausgabedaten, basieren aber auf verschiedenen Grundideen: Einige können wichtige topologische Bereiche, wie zum Beispiel Bruchkanten, automatisch auffinden und extrahieren, andere löschen Punkte nach bestimmten Kriterien und dünnen so das Netz aus.
Ziel dieser Bachelorarbeit ist es, einige dieser Aufbereitungsalgorithmen zu beschreiben, um dann mittels Testdaten eine selbst programmierte Implementierung zu testen.

1.2 Aufbau der Arbeit

Nach dem Einführungskapitel wird im zweiten Kapitel der Arbeit die Datenbeschaffung behandelt. Darin wird das Augenmerk vor allem auf die heutzutage vielseitig eingesetzte Technologie des Airborne Laserscannings gelegt. Zudem werden grundlegende Aspekte der Generierung von Dreiecksnetzen erläutert, da diese oftmals die Datengrundlage von Aufbereitungsalgorithmen darstellen. Im dritten Kapitel dieser Arbeit werden einige schon entwickelte Algorithmen kurz vorgestellt. Danach wird im vierten Kapitel eine eigene Implementierung beschrieben. Dort werden neben dem zugrunde liegenden Algorithmus die verwendeten Methoden bei der Implementierung und die Ergebnisse der Aufbereitung anhand der Testdaten aufgezeigt. Zum Schluss wird ein Ausblick darüber gegeben, inwieweit man diese aufbereiteten Daten für eine Hochwassersimulation verwenden kann.

2. Beschaffung von digitalen Geländemodellen mittels Airborne Laserscanning

Um Punkte im Gelände aufzunehmen und zu digitalisieren, hat sich in den letzten Jahren das Airborne Laserscanning unter den herkömmlichen Methoden wie der terrestrischen Tachymetrie hervorgehoben. Das Laserscanning ist vor allem sehr zeiteffizient, da zum einen in kürzerer Zeit großflächige Gebiete vermessen werden können und zum anderen die Nachbearbeitung der aufgenommenen Daten in einem kleineren Umfang als bei der terrestrischen Tachymetrie nötig

ist. Dies bringt auch einen Kostenvorteil mit sich.
Seit 2001 wird dieses Prinzip auch von der Bayerischen Landesvermessungsgesellschaft vermehrt eingesetzt [4].
Im Folgenden werden die notwendigen Schritte von der Punkterfassung bis hin zum digitalen Geländemodell erklärt.

2.1. Prinzip des LiDAR Airborne Laserscannings

Um die Topografie eines Geländes möglichst genau zu bestimmen, greift man heutzutage oft auf Airborne Laserscanning zurück. Hierbei fliegt ein mit der erforderlichen Messsensorik ausgestattetes Flugzeug (oder Hubschrauber) über das zu erfassende Gebiet und nimmt es dabei durch Aussenden von Laserpulsen auf. Eine weit verbreitete Technologie im Bereich des Airborne Laserscannings ist das LiDAR-Verfahren (Light Detection and Ranging). Der LiDAR Laser sendet Lichtwellen im UV- und nahem IR-Bereich aus [1]. Der Scanner tastet das Gebiet, je nach Flughöhe h und verwendetem Scansystem, in Streifen mit einer Breite von bis zu 0,7*h ab [12]. Dabei werden bis zu 200.000 Laserpulse pro Sekunde ausgesendet, die dann an der Erdoberfläche bzw. von der örtlichen Bebauung und Vegetation reflektiert und von einem Detektor registriert werden [3]. Der Detektor misst die reflektierte Energie des Lichtimpulses. Für jede Reflexion werden durch Laufzeitmessung des Lichtsignals 3D-Koordinaten berechnet. Dabei ist es zwingend erforderlich, die Position des Flugzeugs durch GPS (Global Positioning System) und außerdem dessen räumliche Lage über ein INS (Inertial Navigation System) zu bestimmen. So lassen sich Vektoren im Raum berechnen, welche die Laserpunkte im dreidimensionalen Raum repräsentieren. Ein solcher Vektor ist durch einen Anfangspunkt (Positionsbestimmung durch GPS), Richtung (durch INS) und Länge (durch Laserdistanzmessung) definiert. Die erreichbare Punktdichte kann bis zu 1 Punkt/m² betragen [21] [18].
Messfehler ergeben sich vor allem durch fehlerhafte Justierung der Messtechnik. Eliminiert man solche systematischen Fehler, ist es möglich, eine Genauigkeit von 5 cm zu erreichen [20].
Für ein 100 ha großes Gebiet ergeben sich bei der oben genannten Punktdichte 1.000.000 Messpunkte. Diese große Menge an Punkten verbietet jedoch eine schnelle Berechnung von Hochwasserszenarien, vor allem bei der Betrachtung langer Flussabschnitte. Deshalb sind Algorithmen notwendig, welche die Punkte nach bestimmten Kriterien ausdünnen oder anderweitig modifizieren.

2.2 Die „first pulse – last pulse“-Methode

In der Regel werden die vom Laser ausgesendeten Wellenpakete mehrfach reflektiert. Dann ist es möglich, zwischen dem „first pulse“ und dem „last pulse“ (auch: „second pulse“) zu unterscheiden. Der „first pulse“ wird von der Oberfläche reflektiert, also von der vorhandenen Vegetation und Bebauung, der „last pulse“ hingegen vom Gelände bzw. von der Erdoberfläche. Durch das Prinzip der Charakterisierung von Mehrfachreflexion kann durch Laserscanning sowohl ein digitales Oberflächenmodell (DOM) als auch ein digitales Geländemodell (DGM) abgeleitet werden [12]. Die Unterscheidung in „first pulse – last pulse“ ist aber nicht immer möglich: Bei dicht gewachsenem Gras oder Feldfrüchten erreicht der Laser in der Regel nur die Vegetationsoberfläche. Die Folge sind Zuordnungsfehler und damit das Ableiten eines fehlerhaften Modells.

Dies hat speziell für den Bereich der Hochwassersimulation gravierende Auswirkungen, denn gerade dort ist die Unterscheidung von Oberfläche und Gelände von großer Bedeutung: Die vorhandene Vegetation im Uferbereich eines Flusses dient dem Regenrückhalt bei Niederschlagsereignissen. So kann in der Vegetationsperiode Grünland bis zu zwei und Wald bis zu fünf Liter Niederschlag pro Quadratmeter speichern [5]. Aus einer falschen Punktzuordnung würden also gravierende Berechnungsfehler entstehen.

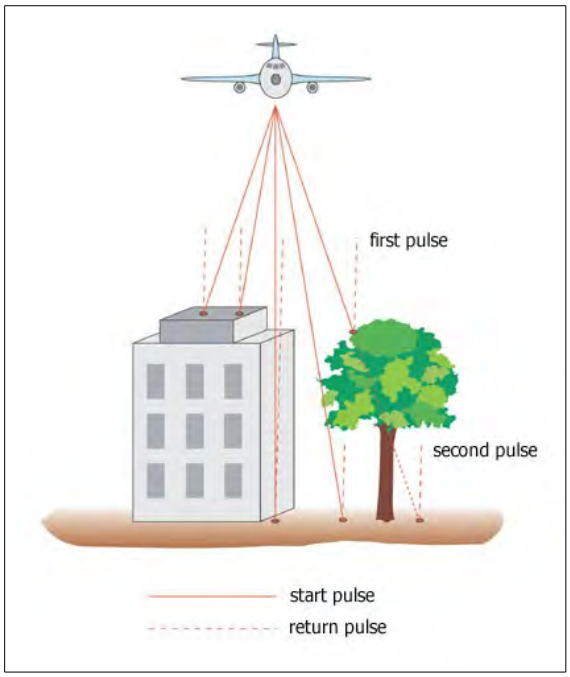

Abbildung 1: Die „first pulse – last pulse“-Methode.

2.3 Kategorisierung von Punkten

Um ein digitales Geländemodell oder ein digitales Oberflächenmodell zu erstellen, ist eine Kategorisierung der aufgenommenen 3D-Punkte in Boden- und Nichtbodenpunkte notwendig. Je weniger Fehler die Ausgangsdatenmenge enthält, umso bessere Ergebnisse werden erzielt.

Es gibt verschiedene Ansätze zur Zuordnung der Punkte, wie zum Beispiel die „morphologische Filterung“. Dabei wird nur eine kleine Teilmenge der Punktmenge betrachtet. Der niedrigste Punkt innerhalb dieser Punktemenge wird als Bodenpunkt angesehen. Wenn nun ein Nachbarpunkt des Bodenpunktes einen vertikalen Abstand zum Bodenpunkt hat, der einen vordefinierten Wert überschreitet, so ist dieser Punkt kein Bodenpunkt [20].

Nach der bayerischen Vermessungsverwaltung lassen sich die Messpunkte in folgende Punktklassen unterteilen:

Punktklasse **1**: Bodenpunkt

Punktklasse **2**: nicht zuzuordnende Punkte in Bodennähe

Punktklasse **3**: Objektpunkt

Punktklasse **4**: Gebäudepunkt

Die „last pulse“-Punkte der Punktklasse 1 ergeben ein DGM, wohingegen ein DOM aus allen „first pulse“-Punkten abgeleitet wird [2].

2.4 Von der Punktwolke zur Dreiecksvermaschung – die Delaunay-Triangulation

Eine kategorisierte 3D-Punktwolke kann bereits als digitales Geländemodell verwendet werden. Oftmals werden zusätzlich zu den Datenpunkten auch Luftbilder in das Modell geladen, um einen Überblick über das vorliegende Gelände und dessen Landnutzung zu erhalten. Durch Berechnung von Schummerungen entsteht ein plastischer Eindruck.

Es gibt allerdings neben dem Plot der Punktwolke noch weitere Möglichkeiten, digitale Geländemodelle zu visualisieren bzw. zu modellieren. Die gängigsten Verfahren basieren auf der Generierung von Netzen, wie zum Beispiel einem regelmäßigem Gitter (Grid) oder einem unregelmäßigen Dreiecksnetz (Triangulated Irregular Network, kurz TIN). Da in der im Kapitel 4 vorgestellten Implementierung auch aus einer Punktwolke ein Dreiecksnetz entworfen wird, soll hier auf die Generierung eines solchen genauer eingegangen werden. Ein bekanntes Verfahren hierzu ist die Delaunay-Triangulation.

Bei der Delaunay-Triangulation wird um jedes Dreieck eines Dreiecksnetzes ein Umkreis (2D, im 3D eine Umkugel) gebildet. Dabei darf jedoch kein weiterer Eckpunkt eines anderen Dreiecks innerhalb dieses Umkreises liegen. Diese „leere Umkreisbedingung“ wird in Abbildung 2 ersichtlich. Der dort konstruierte Umkreis ist nicht leer, die Bedingung nicht erfüllt. Diese Einschränkung hat zur Folge, dass das Netz so lange modifiziert werden muss, bis die Bedingung überall eingehalten wird. Durch die Bedingung des leeren Umkreises wird der kleinste Innenwinkel der so entstehenden Dreiecke maximiert [15].

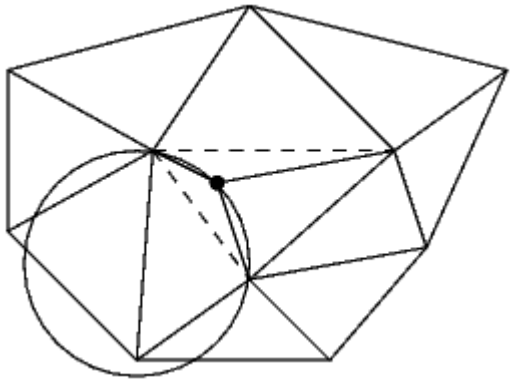

Abbildung 2: Vorgehensweise bei der Delaunay-Triangulation. Die Bedingung des leeren Umkreises ist hier nicht erfüllt.

Es gibt mehrere Methoden, um eine Delaunay-Triangulation durchzuführen (z.B. „Sweep“, „Flip“, „Incremental Construction“ u.v.m.). Bei der Methode der schrittweisen Konstruktion („incremental construction“) werden nach und nach Punkte in ein Netz eingefügt. Sind diese Punkte in einer sinnvollen Reihenfolge angeordnet, werden also benachbarte Punkte nacheinander in das Netz eingefügt, so liegt die Komplexität dieses Algorithmus bei $\mathrm{O}(n \log(n))$ [6].
Die Triangulation beginnt mit einem Dreieck, welches alle anderen Punkte der Datenmenge umfassen kann. Wird nun ein Punkt eingefügt, kommt es zu einer Veränderung des vorhandenen Netzes. Der Algorithmus geht dann wie folgt vor, um das Netz zu aktualisieren (siehe dazu Abbildung 3):

1. Schritt: Dreieck D (gestrichelt dargestellt) finden, in dem der neu eingefügte Punkt P enthalten ist.
2. Schritt: Neue Kanten generieren, die P mit den Eckpunkten des Dreiecks D verbinden.
3. Schritt: Die Kanten des Dreiecks D werden hinsichtlich der „leeren Umkreisbedingung“ überprüft. Ist die Bedingung immer noch erfüllt, so bleiben die neu verbundenen Kanten bestehen. Sollte die Bedingung für eine Kante (hier Kante 1) durch einen im Umkreis lie-

genden Punkt Q verletzt sein, so folgt der vierte Schritt.

4. Schritt: Die Kante 1, die dazu führt, dass die Bedingung des leeren Umkreises verletzt ist, wird durch eine neue Kante ersetzt. Als neue Kante dient eine Diagonale von P zu Q (rote Kante).
5. Schritt: Das/Die so neu generierte/n Dreieck/Dreiecke liefern nun neue Kanten, die zu Kandidaten für die Untersuchung der Umkreisbedingung werden. Schritte drei bis fünf werden für die verbliebenen Kanten des Dreiecks so lange wiederholt, bis die „leere Umkreisbedingung“ erfüllt ist [16].

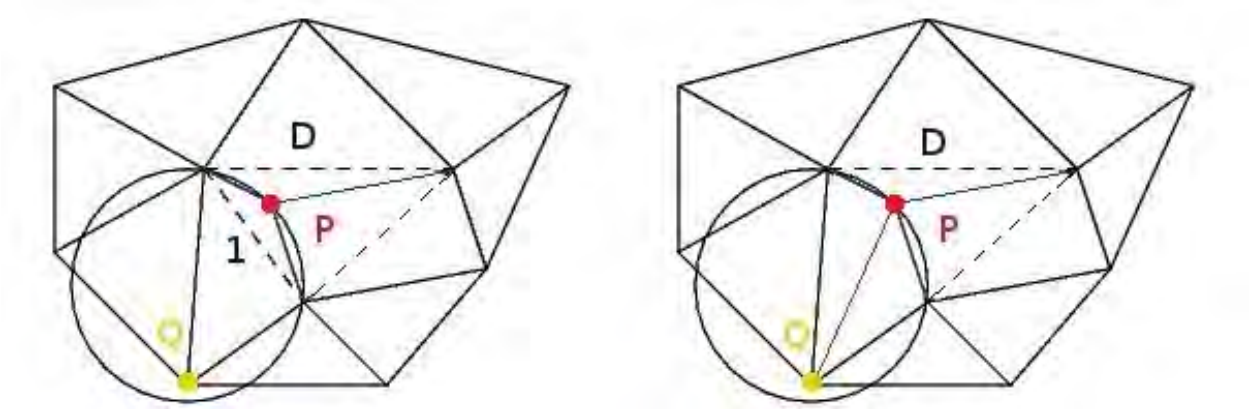

Abbildung 3: Delaunay-Triangulation. Modifikation des Netzes aufgrund der nicht erfüllten „leeren Umkreisbedingung“.

3. Überblick über einige wissenschaftliche Arbeiten zu Aufbereitungsalgorithmen

Dieses Kapitel soll einen Überblick über wissenschaftliche Arbeiten zu Aufbereitungsalgorithmen geben. Es werden drei Algorithmen vorgestellt und ihre grundlegenden Funktionsweisen erläutert.

In der Regel verwenden Aufbereitungsalgorithmen Punktwolken oder Dreiecksnetze als Eingabemengen. Je nach Algorithmus soll der Input nach bestimmten Kriterien und Methoden modifiziert werden und als Ausgabe wieder Punktwolken oder Dreiecksnetze herauskommen.

Zu Beginn eines jeden Abschnittes werden in einem Informationskasten die wichtigsten Merkmale eines jeden Algorithmus genannt. Der meiner Implementierung zugrunde liegende Algorithmus wird im Kapitel 4 behandelt und dort weit umfangreicher erklärt.

3.1 „A general framework for Mesh Decimation“

Infobox

Urheber: L. Kobbelt, S. Campagna, H.-P. Seidel (1998)
Kategorie: Ausdünnungsalgorithmus
Topologischer Operator: half edge collapse
Klasse: Greedy-Algorithmus
Prioritätenliste: ja

In „A General Framework For Mesh Decimation“ [13] wird zunächst ein Überblick über vorhandene Ausdünnungsmethoden gegeben. Diese werden hinsichtlich der verwendeten Strukturen analysiert, um dann daraus einen neuen Algorithmus zu entwickeln, der ein vorhandenes Netz inkrementell dezimiert. Die drei wesentlichen Bestandteile vieler Ausdünnungsalgorithmen sind:

- **Topologischer Operator:** Ein topologischer Operator modifiziert die Topologie eines Netzes. Der Operator „vertex removal“ entfernt einen vorher zu bestimmenden Punkt aus dem Netz (siehe Abb. 4, links). Auf diese Weise werden zwei Dreiecke aus dem Netz entfernt. Es ist danach allerdings eine erneute Triangulation des Netzes erforderlich.
  Der „edge collapse“ (siehe Abb. 4, Mitte) wandelt eine Kante in einen einzigen Punkt um, die Anzahl der Dreiecke reduziert sich also in der Regel ebenfalls um zwei Dreiecke. Sollte ein aus nur einem Dreieck bestehendes Netz vom „edge collapse“ modifiziert werden, so kann auch nur ein Dreieck entfernt werden. Der optimale Ort für den neu einzufügenden Punkt muss bestimmt werden.
  Eine weitere Möglichkeit der Netzmodifikation bietet der „half edge collapse“ (siehe Abb. 4, rechts). Sei $\overline{pq}$ eine Kante vom Punkt p zum Punkt q. Wendet man nun den „half edge collapse“ an, so wird die Kante $\overline{pq}$ zu einem Punkt kontrahiert, indem man den Punkt p in q „zieht“. Diese Operation unterscheidet sich also vom „edge collapse“ darin, dass keine neue Position für einen neuen Punkt bestimmt werden muss.

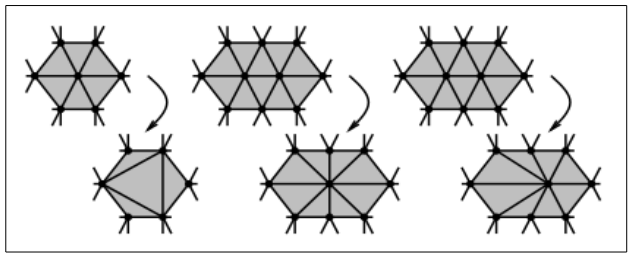

Abbildung 4: Verschiedene topologische Operatoren. Links: „vertex temoval“. Mitte: „edge collapse., Rechts: „half edge collapse“.

- **Distance Measure:** Die meisten Algorithmen verwenden eine Fehlergrenze, um festzulegen, wie sehr das modifizierte Netz vom ursprünglichen abweichen darf. Durch die Berechnung der ersten Ableitung des Netzes ist eine Bewertung der Abweichung möglich.

  Genauere Ergebnisse erhält man durch die Berechnung einer Fehlerquadrik, um so den Fehler zu bestimmen, der bei topologischer Veränderung des Netzes (z.B. durch einen „edge collapse“) entsteht. Eine Fehlerquadrik (Berechnung siehe Abschnitt 4.2) wird jedem Punkt und damit auch jeder Kante in einem Dreiecksnetz zugeordnet. Dadurch kann der Fehler berechnet werden, der beim Löschen einer Kante entsteht. Des Weiteren lässt sich durch die Fehlerquadrik die optimale Position des neu einzufügenden Punktes bestimmen.

  Eine weitere Methode zur Toleranzbestimmung ist die Berechnung der einseitigen oder zweiseitigen Hausdorff-Distanz. Die Hausdorff-Distanz beschreibt, wie sehr sich zwei Mengen überlappen. Um diese zu berechnen, muss das Maximum der minimalen Abstände zwischen zwei Punkten gefunden werden (siehe dazu Abbildung 5).

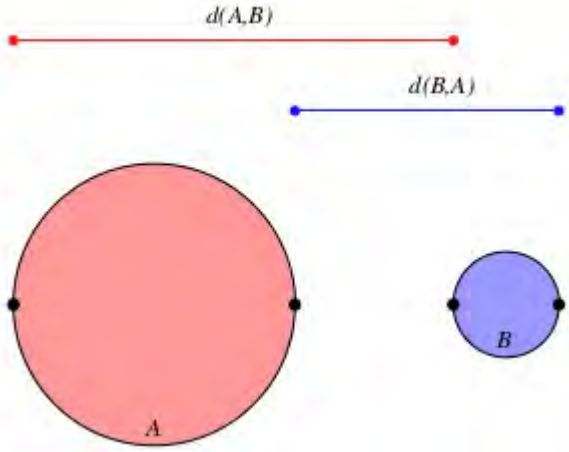

Abbildung 5: Veranschaulichung der Berechnung des Maximums der minimalen Abstände (nicht symmetrisch).

Die einseitige Hausdorff-Distanz wird von der Menge A ausgehend zur Menge B berechnet oder umgekehrt. Sie ist nicht symmetrisch, das heißt, berechnet man die einseitige Hausdorff-Distanz von der Menge B ausgehend zur Menge A, ist das Ergebnis im Allgemeinen ein anderes. Bei der zweiseitigen Hausdorff-Distanz wird sowohl die einseitige Hausdorff-Distanz von A nach B als auch von B nach A berechnet und das Maximum beider Berechnungen gewählt [17].

Eine weitere Möglichkeit zur Bestimmung der Toleranz ist die Berechnung der euklidischen Norm zwischen zwei Punktpaaren.

- **Fairness Criteria:** Das „Fairness Criteria“ bezieht sich auf das gesamte resultierende Netz und bietet eine Möglichkeit, die Qualität des Netzes zu bewerten**.** Durch ein „Fairness Criteria“ kann man die Entscheidung, welcher Schritt als nächstes ausgeführt werden soll, steuern, damit ein Netz entsteht, welches die Fehlergrenze möglichst wenig ausschöpft. Es muss also eine Abfrage implementiert werden, die angibt, ob ein Reduktionsschritt die maximale Toleranzgrenze verletzen würde. Nach diesem Schema wird eine Prioritätenliste eingeführt, sodass in jedem Reduktionsschritt derjenige ausgewählt werden kann, welcher die Qualität am meisten verbessert, oder zumindest am wenigsten verschlechtert.

Der in [13] entwickelte Algorithmus verwendet den topologischen Operator „half edge collapse“. Es wird eine Prioritätenliste erstellt, in der die möglichen „half edge collapses“ berechnet und nach einem „Fairness Criteria“ bewertet werden. Das „Fairness Criteria“ berechnet

sich aus der einseitigen Hausdorff-Distanz. Die Prioritätenliste muss nach jeder Operation aktualisiert werden. Um rechnerischen Aufwand einzusparen, wird die einseitige Hausdorff-Distanz lediglich für einen Bereich berechnet, der in der Umgebung des Ortes liegt, wo der „half edge collapse“ stattgefunden hat.

Randpunkte dürfen nicht in das Innere der Oberfläche rutschen. Deshalb dürfen diese nicht von dem topologischen Operator dezimiert werden. Dieser Algorithmus ist dem der in Kapitel 4 erläuterten eigenständigen Implementierung sehr ähnlich, allerdings verzichtet jener auf die Verwendung einer Prioritätenliste.

Die Funktionalität des Algorithmus ist von den Autoren u.a. an der Datenmenge „Stanford Dragon“ der Universität von Stanford getestet worden (siehe Abbildung 6, Download der Daten auf http://www-graphics.stanford.edu möglich). Es ist deutlich zu erkennen, dass durch die Datenausdünnung zwar Informationen verloren gegangen sind, allerdings sind topologisch interessante Gebiete wie zum Beispiel der Muskel des linken Vorderbeins auch in der ausgedünnten Datenmenge gut zu erkennen. Dies ist wie erwähnt der Sinn von Ausdünnungsalgorithmen: Datenmengen reduzieren und Informationsverlust in der Topologie möglichst gering halten.

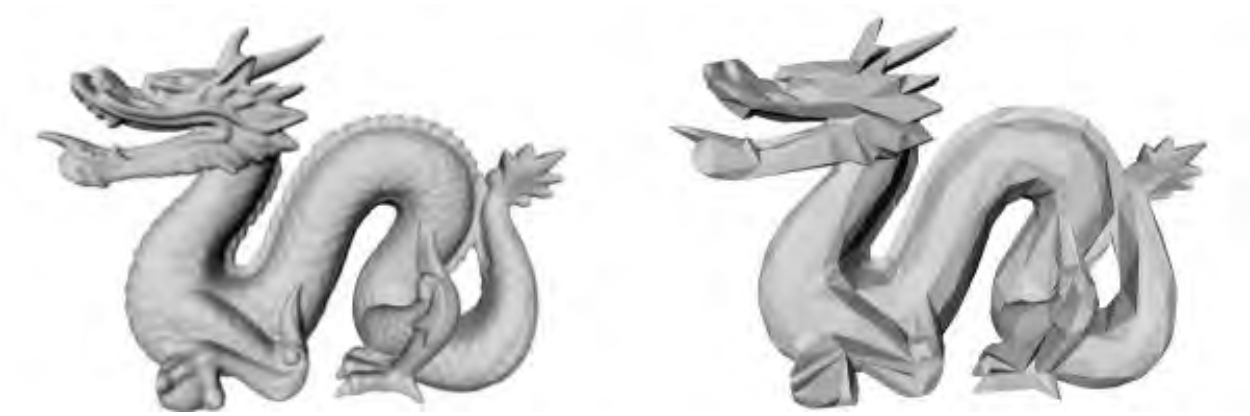

Abbildung 6: Ergebnisse des Testlaufs an der Testdatenmenge „Stanford Dragon“.

3.4 Automatisches Ableiten von Bruchkanten – „Automatic breakline detection from airborne laser range data“

Infobox

Urheber: R. Brügelmann (2000)

Kategorie: Aufbereitung durch automatisches Ableiten von Geländebruckanten

Methode: Berechnung der zweiten Ableitung und Hypothesentest

Dieser Abschnitt beschreibt den in „Automatic breakline detection from airborne laser range data“ von R. Brügelmann vorgestellten Algorithmus [7]. Im Gegensatz zu dem unter Abschnitt 3.3 vorgestellten Algorithmus handelt es sich hierbei nicht um einen Ausdünnungsalgorithmus, sondern vielmehr um einen Aufbereitungsalgorithmus. Dieser ist darauf ausgerichtet, Bruchkanten automatisch aufzuspüren, anstatt diese mit hohem Aufwand photogrammetrisch zu bestimmen. Er basiert auf Hypothesentests. Die verwendeten Input-Daten sind gefilterte Laserdaten, in denen keine Häuser, Bäume o.Ä. zu erkennen sind (siehe dazu Abschnitt 2.3).
Die folgenden Methoden zur Auffindung von Bruchkanten werden vorgestellt:

- **Erste Ableitung:** Die erste Ableitung kann dazu verwendet werden, Grenzen von Steigungsgebieten aufzufinden. Da man Pixel anhand ihrer Steigung in x- bzw. y-Richtung als „flache Pixel“ oder „Steigungspixel“ charakterisieren kann, sind Bruchkanten dann genau die Kanten, die die Grenze zwischen flachen und steilen Bereichen repräsentieren. Dazu werden alle acht Nachbarpixel eines jeden flachen Punktes untersucht. Sobald ein „steiler“ Punkt unter den Nachbarn ist, wird der Punkt selbst als Bruchkanten-Pixel charakterisiert.
- **Laplacian of gaussian operator (LoG)**

  Der Laplace-Operator stellt die zweite Ableitung einer Flächenfunktion dar und kann dazu verwendet werden, Kanten in einer Punktmenge aufzufinden. Die zweite Ableitung beschreibt generell die Krümmung eines Objekts. Allerdings kann es passieren, dass durch reine Anwendung des Laplace-Operators Kanten abgeleitet werden, die durch Rauschen fälschlicherweise als solche charakterisiert werden (Rauschen: Störsignale in den Laserdaten). Um genauere Ergebnisse in der automatischen Suche nach Bruchkanten

zu erzielen, wendet man den Laplace-Operator auf die Gauß-Funktion an. (LoG) Überall dort, wo große Änderungen in der Oberfläche (also Bruchkanten) vorhanden sind, besitzt der LoG einen Vorzeichenwechsel. Der Nulldurchgang wird als Bruchkantenpunkt definiert.

- **Zweite Ableitung und Hypothesentest**

  Ein „range image“ ist ein Bild d (c,r) mit diskreten Koordinaten c (column) und r (row). Je weiter ein Punkt bei der Aufnahme durch einen Laser von diesem entfernt ist, desto dunkler erscheint dieser Punkt auf dem Bild. Kanten sind Grenzen homogener Regionen. Die Homogenität h_2 kann durch Formel 1 bestimmt werden. „d_{cc}“ beschreibt dabei die zweifache Ableitung in c-Richtung, „d_{rr}“ die zweifache Ableitung in r-Richtung usw. Es ist ebenso möglich, die Homogenität als Summe des Produkts aus dem Quadrat der Eigenwerte (λ_1^2, λ_2^2) mal der Hessematrix ($\boldsymbol{H}$) zu bestimmen. Die so berechnete Homogenität h_2 ist an Kanten größer als in homogenen Regionen [10].

  $$h_2 = (d_{cc}^2 + d_{cr}^2) + (d_{rc}^2 + d_{rr}^2) = \lambda_1^2(\boldsymbol{H}) + \lambda_2^2(\boldsymbol{H}) \quad (1)$$

Entlang von Kanten ist die Homogenität lokal maximal. Pixel in flachen Regionen besitzen eine Homogenität mit dem Wert gleich null. Wird die Homogenität nun noch mit der Varianz des Rauschens gewichtet, erhält man eine X_3^2 verteilte Teststatistik. Diese kann man mit dem gewünschten Quantil vergleichen (Hypothesentest, „ist der betrachtete Bereich nicht homogen?“).

Um nun diese Methoden auf eine Datenmenge anzuwenden, muss diese in einem vorbereitenden Schritt zunächst geglättet werden, um das sich mit jeder Ableitung verstärkende Rauschen zu reduzieren. Starkes Rauschen hätte zur Folge, dass Kanten extrahiert werden, die nur fälschlicherweise als solche charakterisiert wurden.

Bruchkanten sind in der Regel mehrere Pixel breit. Durch Anwendung der „non-maxima suppression“, welche alle Pixel unterdrückt, die kein lokales Maximum besitzen, erhält man Bruchkanten mit der Breite von einem Pixel. Die Autorin hat unter Verwendung des auf der Teststatistik beruhenden Hypothesentests folgende Ergebnisse erhalten, dabei gilt: Je heller der Bereich, desto weniger homogen ist dieser. In sehr hellen Bereichen existieren also Bruchkanten (siehe Abbildung 7).

Abbildung 7: Ergebnisse des Testlaufs. Links: Teststatistik. Mitte: Ergebnis des Hypothesentests. Rechts: Non-Maxima Suppression.

Das linke Bild zeigt das Ergebnis der Teststatistik, das mittlere Bild zeigt die automatisch abgeleiteten Bruchkanten – dort wo der Hypothesentest positiv ausgefallen ist. Das rechte Bild zeigt die Bruchkanten mit der Breite von einem Pixel nach Anwendung der „non-maxima suppression“ (dünne weiße Linien).

3.6 Netzfreies Ausdünnen: „Meshfree Thinning of 3D point clouds“

Infobox

Urheber: N. Dyn, A. Iske, H. Wendland (2007)
Kategorie: Ausdünnungsalgorithmus
Topologischer Operator: vertex removal
Klasse: Greedy-Algorithmus
Prioritätenliste: nein
Besonderheit: kein topologisches Netz (meshfree)

Die Autoren stellen in „Meshfree Thinning of 3D Point Clouds“ [9] einen rekursiven Algorithmus vor, der kein zugrunde liegendes topologisches Netz benötigt, sondern als Input-Daten eine 3D-Punktwolke verwendet. Durch lokale Oberflächenannäherung kann die Signifikanz eines Punktes berechnet werden. Dies muss möglichst genau geschehen, da das fälschliche Entfernen von Punkten nicht mehr rückgängig gemacht werden kann (Greedy-Aspekt).

Der Algorithmus kann, anders als andere, auch für Oberflächen, die nicht eben sind, ein Signifikanzniveau berechnen. Er weist einem Punkt eine hohe Signifikanz zu, wenn dessen Nachbarn nur schlecht durch eine Fläche approximiert werden können.
Um eine hohe Performance zu gewährleisten, sollten die Rekonstruktion und das Erstellen eines Signifikanzniveaus **lokal** erfolgen. Aus diesem Grund werden Nachbarschaftsmengen eingeführt, welche nur Punkte aus der direkten Umgebung enthalten. Eine Besonderheit ist, dass in diesem Algorithmus auch bereits gelöschte Punkte in die Signifikanzberechnung mit eingehen.
Um die lokale Oberflächenannäherung zu erhalten, wird zunächst eine tangentiale Ebene durch die Oberfläche an einem Punkt angelegt. Dann werden die Nachbarpunkte von dem betrachteten Punkt auf die tangentiale Ebene projiziert. Sollten zwei verschiedene Punkte aus der Oberfläche auf ein und denselben Punkt in der tangentialen Ebene projiziert werden, so wird dem betrachteten Punkt eine hohe Signifikanz zugeordnet. Dies ist ein Indiz dafür, dass sich der betrachtete Punkt in einem nicht ebenen Bereich befindet. Die Berechnung der Signifikanz eines Punktes x erfolgt ansonsten (also für Punkte x, die in einem ebenen Bereich liegen) nur in dessen Nachbarschaft. Die Punkte, die in der Nachbarschaftsmenge liegen, werden dazu verwendet, eine lokale Approximation der Oberfläche zu bestimmen. Die Signifikanz eines Punktes ist der Betrag des Maximums der Differenz aus dem Abstand der Oberflächenapproximation und des Punktes sowie dem Abstand des Punktes zur tangentialen Ebene. Je größer der Wert der Signifikanz, desto wichtiger ist der betrachtete Punkt.

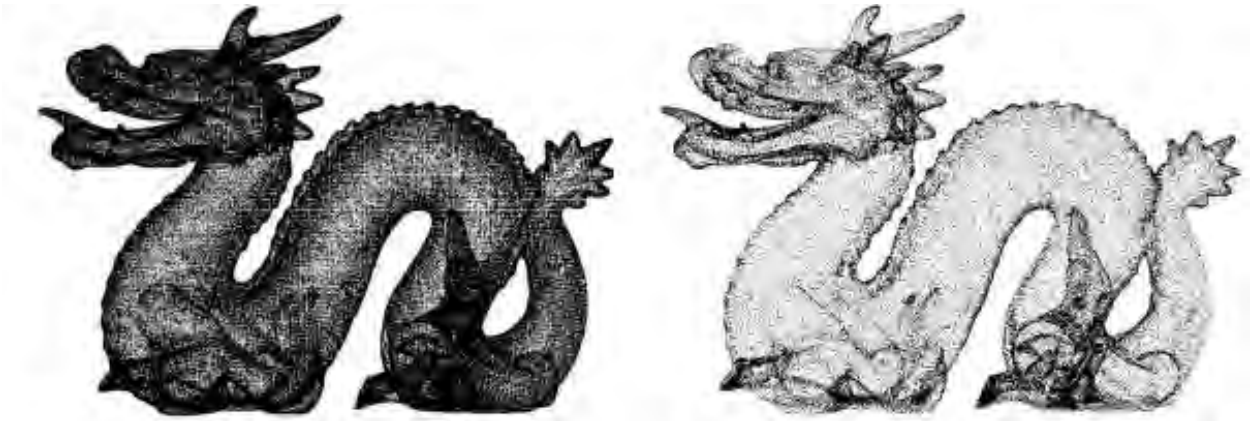

Abbildung 8: Ergebnisse des Testlaufs an der Datenmenge Stanford Dragon. Links: Ausgangsdatenmenge. Rechts: Auf 25 % der Ausgangsdaten dezimiert.

Auch dieser Algorithmus wurde anhand der Testdatenmenge „Stanford Dragon“ auf Funktionali-

tät geprüft (siehe Abbildung 8). Hier ist deutlich zu erkennen, dass der Algorithmus ohne die Konstruktion eines Netzes auskommt. Die Bilder zeigen, dass annähernd ebene Bereiche wie der Körper des Drachen um einiges mehr ausgedünnt sind als Bereiche mit höherer Informationsdichte wie zum Beispiel der Kopf des Drachens.

4. Implementierung mit Matlab

Matlab ist eine Software des Unternehmens „The MathWorks“, die es erlaubt, numerische Algorithmen aufzustellen, Datensätze zu analysieren und auszuwerten sowie Visualisierungen zu erzeugen. Der Vorteil von Matlab ist, dass viele erforderliche Formeln und Funktionen bereits vorinstalliert sind. Dadurch kann eine schnelle Bearbeitung erfolgen. Matlab ist darauf ausgerichtet, Matrizen für Berechnungen zu verwenden. Dies ist in vielen Fällen ein Vorteil, da Messdaten, wie zum Beispiel Windgeschwindigkeiten, Temperaturverläufe oder, wie in dieser Arbeit, Geländepunkte, oftmals in Matrizen gespeichert werden. Matrixmultiplikationen können mit Matlab einfach und schnell durchgeführt werden.
Im Folgenden werden zunächst die vom Lehrstuhl Wasserbau zur Verfügung gestellten Testdaten beschrieben und anschließend die eigene Implementierung mit Matlab vorgestellt.

4.1 Testgebiet und Testdaten

Die vom Lehrstuhl Wasserbau zur Verfügung gestellten Laserscandaten bilden die Donau bei Donauwörth ab. Die Aufnahmen, durchgeführt vom Bayerischen Landesamt für Vermessung, stammen aus dem Jahr 2008 und zeigen ein Punktraster mit einer Gitterweite von 1 x 1 m.
Das gesamte Gebiet ist in 105 Teilgebiete gegliedert. Jedes dieser Teilgebiete besteht aus 1.000.000 Datenpunkten, die durch x-, y- und z-Koordinaten beschrieben werden. Der Testlauf beschränkt sich auf eines dieser Teilgebiete mit den folgenden Koordinaten (Tabelle 1):

	Gauß-Krüger-Koordinatensystem		Gradmaß	
	Rechtswert	Hochwert	Breite	Länge
Eckpunkt 1	4410000	5398999	48° 43' 22.96"	10° 46' 35.84"
Eckpunkt 2	4410999	5398999	48° 43' 23.47"	10° 47' 24.72"
Eckpunkt 3	4410999	5398000	48° 42' 51.14"	10° 47' 25.49"
Eckpunkt 4	4410000	5398000	48° 42' 50.62"	10° 46' 36.63"

Tabelle 1: Koordinaten des Testausschnitts.

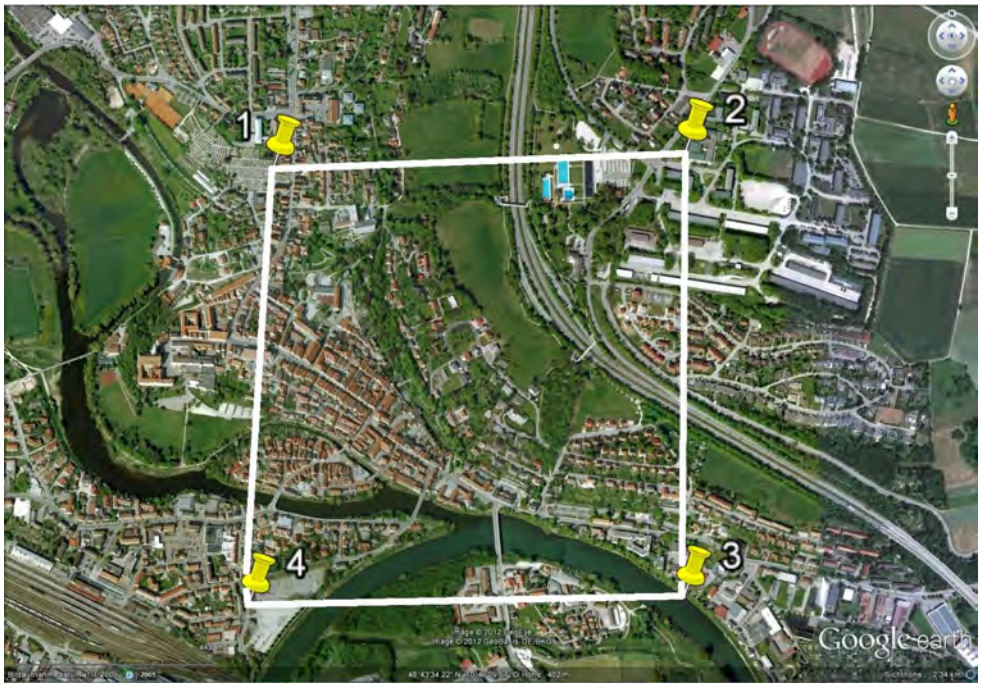

Abbildung 9: Testgebiet an der Donau bei Donauwörth.

Das in Abbildung 9 gekennzeichnete Gebiet bildet die Mündung der Wörnitz in die Donau ab. Im Bild sind die beiden Flüsse deutlich zu erkennen. Der gebogene Flusslauf in der unteren Hälfte stellt die Donau dar, die Wörnitzmündung befindet sich etwa auf dem Scheitel des Bogens.
Dieses Teilgebiet bildet die Grundlage für den Testlauf der Implementierung. Im nächsten Abschnitt werden die dafür verwendeten Methoden beschrieben.

4.2 Implementierung zugrunde liegender Algorithmus – „A Stream Algorithm for the Decimation of Massive Meshes“

Infobox

Urheber: J. Wu, L. Kobbelt (2003)
Kategorie: Ausdünnungsalgorithmus
Topologischer Operator: edge collapse
Klasse: randomized multiple choice optimization
Prioritätenliste: nein

Der der eigenen Implementierung zugrunde liegende Algorithmus wird in „A Stream Algorithm for the Decimation of Massive Meshes“ [23] vorgestellt.

Die Geometriedaten sollten möglichst im STL-Format (Surface Tesslation Language) vorliegen; das bedeutet, dass Topologieinformationen im Gegensatz zu dem in Abschnitt 3.6 vorgestellten Algorithmus vorhanden sein müssen. In diesem Format ist jedes Dreieck durch seine drei Eckpunkte mit je drei Koordinaten definiert. Das hat eine hohe Redundanz zur Folge, da jeder Netzpunkt mehrfach gespeichert ist. Der Vorteil dieses Datenformats besteht darin, dass man keine Indizierung benötigt.

Ein Unterschied im Gegensatz zu anderen Ausdünnungsalgorithmen ist, dass nicht der komplette Datensatz auf einmal dezimiert wird, sondern lediglich eine kleinere Untermenge, hier als Incore-Speicher bezeichnet. Auf diese Weise ist eine schnellere Berechnung möglich. Diese Neuerung hat sich vor allem daraus ergeben, dass durch die heutigen Möglichkeiten der digitalen Geländeaufnahme (wie im 2. Kapitel beschrieben) das Input-Geländemodell einen enormen Informationsgehalt besitzt. Würde man eine so große Input-Datenmenge auf einmal durch eine Dezimierung bearbeiten lassen, hätte das einen hohen Performanceverlust zur Folge. Die Größe des Incore-Speichers ist unabhängig von der Größe des Input-Netzes.

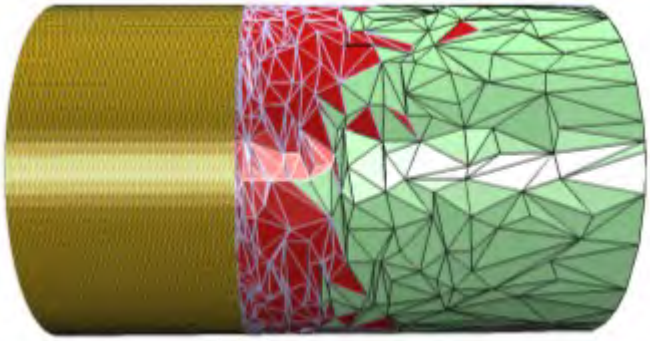

Abbildung 10: Der Aufbau des Algorithmus. Linker Teil: Ausgangsdatenmenge. Mittlerer Teil: Incore-Speicher. Rechter Teil: Dezimierter Output.

Der Algorithmus lässt sich, wie in Abbildung 10 dargestellt, logisch in drei Teile gliedern.
Der linke Abschnitt stellt den noch nicht dezimierten Input dar. Der mittlere Bereich repräsentiert die laufende Incore-Menge und der rechte Bereich ist der dezimierte Output. Des Weiteren soll der Algorithmus das Input-Netz auf einen gewissen Prozentsatz verkleinern und das Füllen des Incore-Speichers zufällig geschehen.
Ein Greedy-Algorithmus führt eine Folge kleiner Schritte aus und wählt stets den nächsten Schritt aus, der, bezogen auf den aktuellen Stand, der beste ist. Dafür ist eine Prioritätenliste erforderlich.
Der Ansatz der „multiple choice optimization“ hingegen, auf dem auch dieser Algorithmus basiert, umgeht eine solche Liste, indem er innerhalb des Incore-Speichers eine kleine Untermenge aus 5-15 Dreiecken zufällig auswählt und den nächstbesten Schritt in dieser Untermenge sucht. Der Rechenaufwand wird dabei reduziert, das Ergebnis erleidet jedoch – verglichen mit einem Greedy-Ansatz – keinen Qualitätsverlust.
Der Algorithmus basiert auf drei grundlegenden Funktionen:

- **Read(k):** Die nächsten k Dreiecke des Input-Streams werden in die laufende Incore-Menge $N_{curren\ t}$eingefügt: $N_{current} \leftarrow N_{current} + k$
- **Decimate(k):** Führt k „edge collapse“-Operationen innerhalb der Incore-Menge des Netzes aus. Jeder „edge collapse“ entfernt in der Regel zwei Dreiecke aus dem Netz.: $N_{current} \leftarrow N_{current} - 2k$.

  Sollte nur ein Dreieck aus dem Netz entfernt werden, wie es bei der Dezimierung von unzusammenhängenden Dreiecken (siehe Abbildung 11) der Fall ist, so gilt: $N_{current} \leftarrow N_{current} - k$

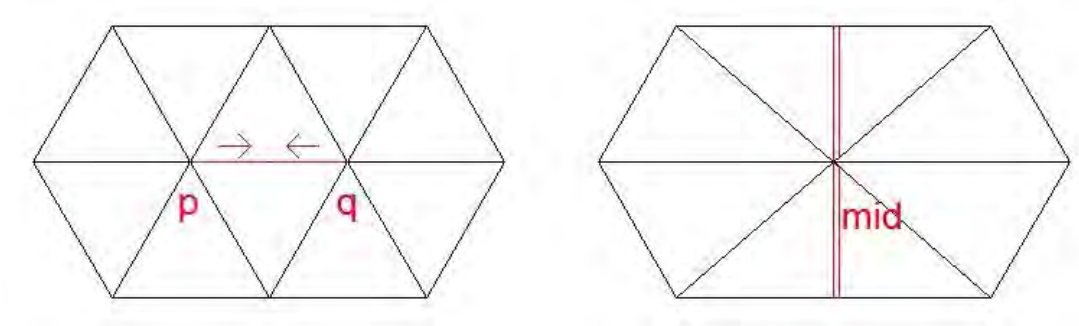

Abbildung 11: Falsche Dreiecke durch „edge collapse“.

Die optimale Position des Punktes, in den p und q nach Ausführen des „edge collapse“ zusammengefügt werden, kann durch die Berechnung der Fehlerquadrik (zur Berechnung siehe Formel (2)) bestimmt werden. In dem Implementierungsteil dieser Arbeit wird diese Position als Mittelpunkt der Kante $\overline{pq}$ angenommen.

- **Write(k):** Entfernt k Dreiecke aus dem aktuellen Incore-Speicher und schreibt sie in den Output-Stream: $N_{current} \leftarrow N_{current} - k$

Diese drei Funktionen werden zufällig ausgeführt, jedoch gibt es einige Bedingungen an die Reihenfolge, die eingehalten werden müssen:

- Die Größe des Incore-Speichers darf eine Größe von N_{max} Einträgen nicht übersteigen.
- Der Grad der Füllung des Incore-Speichers sollte so hoch wie möglich gehalten werden. So wird garantiert, dass die Anzahl an Kandidaten für die „multiple choice optimization“ groß genug ist.
- Das Output-Netz hat nur dann überall die gleiche Auflösung, wenn „Decimate“ etwa so oft aufgerufen wird wie „Write“.

Um den letzten Punkt zu garantieren, kann man sich durch einfache mathematische Überlegungen den Anteil der Decimate-Operationen an der Gesamtanzahl von Write- und Decimate-Operationen berechnen.

Durch das Einführen der „multiple choice optimization“ wird dem Algorithmus ein Bewertungssystem hinzugefügt. Es werden von einer kleinen Untermenge des Incore-Speichers (5-15 Dreiecke) alle zu den Kanten zugehörigen „quadric errors“ berechnet. Dafür muss jeder Kante eine Fehlerquadrik zugewiesen werden. Die Berechnung der Fehlerquadrik geschieht nach dem „Garland-Heckbert-Algorithmus“ [11].

Ziel ist es, die Kante zu finden, bei der nach Ausführung des „edge collapse“ der durch diese Operation entstandene Fehler (siehe Formel (2)) möglichst klein ist.

$$dist(mid, F)^2 = (mid_x\ mid_y\ mid_z\ 1)\ \ Q \begin{pmatrix} mid_x \\ mid_y \\ mid_z \\ 1 \end{pmatrix} = (mid_x\ mid_y\ mid_z\ 1) \begin{pmatrix} nn^T & -nn^T b \\ -b^T nn^T & b^T nn^T b \end{pmatrix} \begin{pmatrix} mid_x \\ mid_y \\ mid_z \\ 1 \end{pmatrix} \quad (2)$$

$$n = Normalvektor\ der\ Fläche\ F\ ,\ \ b = beliebiger\ Punkt\ der\ Fläche\ F$$

Der Fehler ist das Quadrat der Abstände, die der aus den Kantenpunkten generierte Punkt „mid“ zu allen an die Kantenpunkte angrenzenden Dreiecken hat. Q ist die Quadrik einer Dreiecksfläche F. Die Quadrik Q eines Punktes erhält man, indem man die Summe der Quadriken seiner angrenzenden Dreiecke bildet. Die Quadrik einer Kante berechnet sich wiederum als Summe der Quadriken des Anfangs- und Endpunktes. Zur Berechnung einer Quadrik benötigt man den Normalvektor n der betrachteten Fläche und einen in dieser Fläche liegenden Punkt b (z.B. einen Eckpunkt eines Dreiecks). Die Kante, die den kleinsten Fehler bei der Berechnung besitzt, wird dann tatsächlich vom „half edge collapse“ modifiziert.

Im nächsten Abschnitt wird erklärt, wie die in diesem Abschnitt genannten Strukturen des Algorithmus bei der Implementierung umgesetzt wurden.

4.3 Methoden

Dieser Abschnitt soll besonders ausführlich dargestellt werden, da er zum einen eine zentrale Bedeutung in dieser Arbeit einnimmt und zum anderen die der Implementierung zugrunde liegende Logik so verständlich wie möglich dargestellt werden soll. Des Weiteren muss erwähnt werden, dass der in „A Stream Algorithm for the Decimation of Massive Meshes“ beschriebene Algorithmus lediglich eine Vorlage für den hier implementierten Algorithmus darstellt. Es bestehen einige Unterschiede zwischen den Algorithmen.

4.3.1 Vorbereitende Schritte

Als ersten Schritt müssen zunächst die Testdaten, die im Datenformat .g01dgm vorliegen, in das Datenformat .txt konvertiert werden, um diese mit Matlab öffnen zu können. Ein einfaches Umbenennen der Dateiendung ist dazu ausreichend. Die Rohdaten werden mit der Funktion „load“ in das Programm geladen. Nun ist die Matrix, welche aus x-, y- und z-Koordinaten besteht, für Berechnungen verfügbar. Die Variable „m“ wird im Programmcode vom Benutzer eingegeben. Durch folgende Methode wird nur jede m-te Zeile der ursprünglichen Datei verwendet.

```
x = Gebiet(1:m:end,1);
```

Nach dem gleichen Schema werden auch die Vektoren y und z erstellt. Neben der Variablen „m“ werden auch noch der Prozentsatz „pro“, auf den das Netz verkleinert werden soll, und die Größe des Incore-Speichers „sizeincore“ vom Benutzer festgelegt. Durch diese Parameter lässt sich die Dezimierung steuern.
Da der Input des Ausdünnungsalgorithmus ein Dreiecksnetz ist, muss nun mithilfe der in Matlab vorinstallierten Funktion „delaunay“ eine Delaunay-Triangulation der Punktwolke erzeugt werden. Dabei entsteht die Eingabematrix „eingabe“:

```
eingabe = delaunay(x,y);
trisurf(eingabe,x,y,z);
```

Die Delaunay-Triangulation wird auf die Vektoren x und y angewendet. Die Funktion erstellt eine kx3-Matrix (k: Anzahl Dreiecke), in der die Dreiecke durch eine automatisch erstellte Indizierung gespeichert werden. Danach wird mit der Funktion „trisurf“ ein Plot der Triangulation erstellt, wobei den Dreiecken die x-, y- und z-Koordinaten (damit also auch die Höhe) zugeordnet werden. Dieser Plot dient der Visualisierung des Ausgangsnetzes.
Die vorbereitenden Schritte sind somit abgeschlossen.
Wichtig ist, dass die Dezimierung von Randkanten nicht zulässig ist, da sonst das gesamte Gebiet zusammenschrumpfen würde. Deshalb wird im Folgendem bestimmt, welche Punkte zu den gesperrten Randdreiecken gehören.

4.3.2 Bestimmung der Randdreiecke

Quellcode 1 – Bestimmung der Randdreiecke:

```
1  points = size(x,1);
2  adjacencyMatrix = sparse(...
3      [ eingabe(:,1); eingabe(:,2); eingabe(:,3) ], ...
4      [ eingabe(:,2); eingabe(:,3); eingabe(:,1) ], ...
5      1, points, points);
6  adjacencyMatrix = adjacencyMatrix + adjacencyMatrix';

7  isRandDreieck = zeros(size(eingabe,1), 1);
8  isGesperrteEcke = zeros(points,1);


9  for l = 1 : size(eingabe,1)
10     isRandDreieck(l) = ...
11         (adjacencyMatrix(eingabe(l,1), eingabe(l,2)) == 1)  || ...
12         (adjacencyMatrix(eingabe(l,2), eingabe(l,3)) == 1)  || ...
13         (adjacencyMatrix(eingabe(l,3), eingabe(l,1)) == 1);
14     if(isRandDreieck(l))
15         isGesperrteEcke(eingabe(l,1)) = 1;
16         isGesperrteEcke(eingabe(l,2)) = 1;
17         isGesperrteEcke(eingabe(l,3)) = 1;
18     end
19 end
```

In Zeile 1 wird ein Vektor „points“ erstellt, der so viele Zeilen wie es Datenpunkte gibt besitzt. Nun kann man mithilfe einer arbeitsspeichersparenden Sparse-Matrix eine Adjazenzmatrix „adjacencyMatrix“ erstellen. Diese Matrix ist quadratisch, wobei sich die Anzahl der Zeilen und Spalten aus der Anzahl der Einträge im Vektor „points“ ergibt. Es werden zu jedem Dreieck alle gerichteten Kanten erstellt. Der Adjazenzmatrix wird eine 1 an der Stelle Adjazenzmatrix(x, y) zugewiesen, wenn von einem Punkt x zu einem Punkt y eine Kante existiert. Die Adjazenzmatrix wird in Zeile 6 symmetrisiert. Durch die Symmetrisierung werden gerichtete Kanten in ungerichtete umgewandelt: Sollte es sowohl von x nach y als auch von y nach x eine Kante geben, so erhöht sich der Eintrag der Adjazenzmatrix an diesen beiden Stellen auf 2. Eine Randkante ist überall dort vorhanden, wo die Einträge in der Adjazenzmatrix weiterhin 1 betragen.

Um die Randdreiecke zu bestimmen, erstellt man die Matrizen „isRandDreieck“ und „isGesperrteEcke“ (Zeile 7 und 8). Diese enthalten nur Nullen („Nein“) und Einsen („Ja“). Damit machen sie eine Aussage darüber, ob ein betrachtetes Dreieck oder eine betrachtete Ecke ein Randdreieck oder eine gesperrte Ecke ist. Nun kann man erneut alle Dreiecke in einer *for*-

Schleife (Zeile 9-19) durchgehen und alle Kanten eines Dreiecks auf ihre Häufigkeit in der Adjazenzmatrix überprüfen. Sollte entweder die Kante vom ersten zum zweiten oder vom zweiten zum dritten oder vom dritten zum ersten Punkt des Dreiecks nur mit einer 1 in der Adjazenzmatrix beziffert sein, so ist das gesamte Dreieck ein Randdreieck. In diesem Fall muss in der folgenden *if*-Abfrage (Zeile 14-18) jede Ecke dieses betroffenen Dreiecks gesperrt werden, indem jedem Punkt eine 1, also ein „Ja“ zugeordnet wird.
Nun müssen die drei grundlegenden Funktionen „read“, „decimate“ und „write“ implementiert werden, um die Modifikation des Dreiecksnetzes durchführen zu können.

4.3.3 Implementierung der drei grundlegenden Funktionen

Quellcode 2 – Funktion „read“:

```
1 function [incore, j] = read(eingabe, i, k, incore)
2 if(i+k < size(eingabe,1))
3     j = i + k;
4 else
5     j = size(eingabe, 1);
6 end
7 incore = [incore; eingabe(i+1:j,:)];
```

Die Funktion „read“ besitzt folgende Übergabeparameter: die Eingabematrix, die Laufvariable i, die Variable k und den Incore-Speicher. Als Output der Funktion wird der modifizierte Incore-Speicher und die Laufvariable j zurückgegeben. Die Variable k entspricht der Anzahl der Dreiecke, die von der Eingabematrix in den Incore-Speicher verschoben werden soll. Die Variable i stellt dabei die Position in der Eingabematrix dar und ist am Anfang des Algorithmus auf i = 1 gesetzt. Nachdem die ersten k Dreiecke in den Incore-Speicher verschoben wurden, erhöht sich i auf i = i + k. Durch eine *if*-Abfrage (Zeilen 2-6) wird sichergestellt, dass die Laufvariable j nicht größer werden kann als es Zeilen in der Eingabematrix gibt. Wird nämlich die Anzahl Zeilen der Eingabematrix (size(eingabe,1)) überschritten, so wird j auf genau diesen Wert zurückgesetzt.
In Zeile 7 des Quellcodes 1 wird der Incore-Speicher aufgefüllt. Durch den Matlaboperator „ ; “ werden der bestehenden Incore-Matrix die Zeilen i + 1 bis j der Eingabematrix angehängt.

Quellcode 3 – Funktion „write“:

```
1 function [output, incore] = write (output, incore, k)
2 if length(incore) < k

3     output = [output; incore(1:k,:)];
4     incore = incore(k+1:end, :);

5 end
```

Um die Output-Matrix zu füllen, müssen der Funktion „write“ der zu Beginn noch leere Output, der Incore-Speicher und die Anzahl der Dreiecke, die aus dem Incore-Speicher in die Output-Matrix verschoben werden sollen, übergeben werden. Dies geschieht durch eine *if*-Abfrage (Zeile 2-5). In Zeile 2 des Quellcodes 3 werden dem Output die Zeilen 1 bis k der Incore-Menge angehängt. Zudem muss der Incore-Speicher um die ersten k Zeilen verkleinert werden (Zeile 4).

Quellcode 4 – Funktion „decimate“:

```
1  function [trineu, pneu, q1,q2] = decimate(eingabe,eingabeGesamt, p,
      isGesperrteEcke)

2     %Variante 1: Beste Kante finden
3  [k,i,j] = findBesteKante(eingabe, eingabeGesamt, p, isGesperrteEcke);
4     %Variante 2: Zufällige Kante finden
5  [k, i, j] = findZufaelligeKante (eingabe, p, isGesperrteEcke)

6  mid = (p(eingabe(k,i)) + p(eingabe(k,j)))/2;
7  pneu = p;
8  pneu(eingabe(k,i)) = mid;
9  pneu(eingabe(k,j)) = mid;
10 trineu = deleteFalscheDreiecke(eingabe, p);
11 q1 = eingabe(k,i);
12 q2 = eingabe(k,j);
```

Die Funktion „decimate“ besitzt die vier Übergabeparameter: „eingabe“, welcher innerhalb der Funktion der Incore-Speicher ist, „eingabeGesamt“, welcher die komplette Dreiecksmatrix enthält, die Punktkoordinatenmatrix „p“ und „isGesperrteEcke“, die Menge der Punkte, die nicht dezimiert werden dürfen.

Man bestimmt entweder durch die Funktion „findBesteKante“ (Zeile 3, siehe Quellcode 6) oder durch die Funktion „findZufaelligeKante“ (Zeile 5, siehe Quellcode 7) das k-te Dreieck und die Kante, die durch den i-ten und j-ten Punkt des Dreiecks k definiert ist (jeweils Zeile 3 oder Zeile 5 im Quellcode auskommentieren). Diese Kante wird dann von dem „edge collapse“ modifiziert.

Dies geschieht durch eine Verschiebung der ausgewählten Punkte i und j des Dreiecks k auf die Mitte „mid“ der Kante (Zeile 6), deren drei Punktkoordinaten in einer 1 x 3-Matrix gespeichert werden. In den Zeilen 8 und 9 wird nun sowohl dem i-ten Eckpunkt des k-ten Dreiecks als auch dem j-ten Eckpunkt des k-ten Dreiecks dieser Mittelpunkt „mid“ zugeordnet und dem Rückgabeparameter „pneu“ zugewiesen. Dort existieren nun also zwei Punkte übereinander. Dies kann allerdings zur Folge haben, dass durch eine spätere Dezimierung einer Kante, die einen solchen doppelten Punkt als Ecke besitzt, Löcher im Netz entstehen. Dies geschieht, weil durch den „edge collapse“ nur einer der übereinanderliegenden Punkte verschoben wird. Die Folge ist ein Loch im Netz.
Um das Entstehen von Löchern im Output zu vermeiden, werden der Funktion die Rückgabeparameter q1 und q2 hinzugefügt. Diese Variablen beschreiben die Punktindizes aus der Menge der Dreiecke, die nach einem „edge collapse“ übereinanderliegen. In Zeile 11 und 12 werden diesen Variablen die Punktindizes der Punkte übergeben, die übereinanderliegen. Somit ist diese Information auch außerhalb der Funktion bekannt. Im Programm selbst muss dann noch die Funktion „canonizePointIndices“ aufgerufen werden (siehe Quellcode 5). Dort wird dem Programm beigebracht, dass zwei übereinanderliegende Punkte als einer zu behandeln sind.
Wie bereits unter 4.2 beschrieben, entstehen durch den „edge collapse“ „Dreiecke“ mit nur zwei Eckpunkten. Um diese zu eliminieren, muss die Funktion „deleteFalscheDreiecke“ aufgerufen werden (Zeile 10). Die so modifizierte Eingabemenge stellt den Output der Funktion „decimate“ „trineu“ dar. Die Funktion „deleteFalscheDreiecke“ wird in Quellcode 8 ausführlich beschrieben.

Quellcode 5 – Funktion „canonizePointIndices“:

```
1 function [t] = canonizePointIndices(t, q1,q2)

2 for d=1:3
3     inds = find(t(:,d) == q1);
4     t(inds,d) = q2;
5 end
```

Die Funktion „canonizePointIndices“ wird dazu verwendet, um zwei übereinanderliegende Punkte zu einem einzigen zu machen. Der Funktion wird die Dreiecksmatrix (innerhalb der Funktion mit „t“ bezeichnet) und die Punktindizes der übereinanderliegenden Punkte q1 und q2 übergeben. In einer *for*-Schleife (Zeilen 2-5) wird das Dreieck gesucht, welches q1 als Index enthält. Dieser Index wird in Zeile 4 mit q2 überschrieben. Der Rückgabeparameter der Funktion

ist die Dreiecksmatrix „t“.

4.3.4 Variante 1: Wahl einer geeigneten Kante durch Quadrik Error Metric

Die Funktion „findBesteKante“ stellt die Implementierung der „multiple choice optimization“ nach dem Garland-Heckberg-Algorithmus [11] dar. Die Methodik ist im Abschnitt 4.2 erläutert.
Da dieser Teil der Implementierung sehr lang ist, wird der Quellcode der Funktion „findBesteKante“ der Übersicht halber in sechs Teile aufgeteilt. Teil 1 des Quellcodes 6 beschreibt die Berechnung des Normalvektors.

Quellcode 6, Teil 1 – Funktion „findBesteKante“: Berechnung des Normalvektors

```
1   function   [k,i,j]   =   findBesteKante(eingabe,   eingabeGesamt,   p,
      isGesperrteEcke)
2 NormV = zeros(size(eingabeGesamt,1), 3);

3  for i = 1:size(eingabeGesamt,1)
4     Richtungsvektor1 = p(eingabeGesamt(i,2),:)-p(eingabeGesamt(i,1),:);
5     Richtungsvektor2 = p(eingabeGesamt(i,3),:)-p(eingabeGesamt(i,1),:);

6     Vneu = cross(Richtungsvektor1, Richtungsvektor2);

7     if(norm(Vneu) < 1e-10)
      %Gefahr der Division durch 0.
8     else
9           NormV(i,:) = (1/norm(Vneu))*Vneu;
10    end
11 end
```

Der Funktion „findBesteKante“ wird neben der Eingabematrix aus der „decimate“-Funktion (also der Incore-Speicher) noch die gesamte Dreiecksmatrix, die Punktematrix und die Matrix der gesperrten Eckpunkte „isGesperrteEcke“ übergeben. Die Rückgabeparameter sind die Variablen k, i und j, wobei k das zu dezimierende k-te Dreieck und i und j der Anfangs- und Endpunkt der zu dezimierenden Kante ist.
Der Normalvektor einer Fläche berechnet sich aus dem Kreuzprodukt zweier Richtungsvektoren eines Dreiecks, geteilt durch die Länge des Kreuzprodukts. In der *for*-Schleife (Zeile 3-13) werden **alle** Dreiecke durchgegangen und zu jedem der jeweilige Normalvektor bestimmt. Die Normalvektoren werden in der Matrix „NormV“ gespeichert. Durch die Matlabfunktion „cross“ kann das Kreuzprodukt zweier Vektoren berechnet werden (Zeile 6), die Funktion „norm“ bestimmt die Länge eines Vektors (Zeile 9).

Quellcode 6, Teil 2 – Funktion „findBesteKante“: Kanten erstellen

```
1 kante = zeros(3*length(eingabe),2);

2 for l = 1 : length(eingabe)
3    kante(3*(l-1)+1,:) = [eingabe(l,1), eingabe(l,2)];
4    kante(3*(l-1)+2,:) = [eingabe(l,2), eingabe(l,3)];
5    kante(3*(l-1)+3,:) = [eingabe(l,3), eingabe(l,1)];
6 end

7 for w=1:length(kante)
8     kantesort(w,:) = sort(kante(w,:));
9 end

10 alleNurEinmal = unique(kantesort, 'rows');
```

In Zeile 1 wird eine Matrix „kante“ erstellt, in der für jedes Dreieck des Incore-Speichers die zugehörigen drei Kanten gespeichert werden sollen. Die *for*-Schleife (Zeile 2-6) geht alle Dreiecke durch und erstellt für jedes Dreieck drei Kanten, die durch Anfangs- und Endpunkt definiert sind. Die jeweils erste Kante, die erstellt wird, ist die Kante vom ersten Punkt des l-ten Dreiecks zum zweiten Punkt des l-ten Dreiecks (Zeile 3) usw..

Da man auf diese Weise gerichtete Kanten erstellt, d.h. eine Kante von Punkt 1 zu Punkt 2 ist nicht die gleiche wie die von Punkt 2 nach Punkt 1, müssen diese jetzt zu ungerichteten Kanten zusammengefasst werden. Dafür werden in einer weiteren *for*-Schleife (Zeile 7-9) alle Zeilen der Kantenmatrix mit der Funktion „sort“ aufsteigend sortiert. Somit wird aus der Kante von 2 nach 1 die Kante 1 nach 2. Durch die Funktion „unique“ kann man jetzt noch die in der Matrix doppelt vorkommenden Kanten löschen (Zeile 10); dabei entsteht die Matrix „alleNurEinmal“.

Quellcode 6, Teil 3 – Funktion „findBesteKante“: Nachbardreiecke finden und Quadrik der Punkte berechnen

```
1 qalle = zeros(size(p,1)*4,4);

2 pIncore = [eingabe(:,1); eingabe(:,2); eingabe(:,3)];
3 pIncore = unique(pIncore, 'rows');
4 inIncoreEnthalten = zeros(size(p,1),1);
5 inIncoreEnthalten(pIncore) = 1;

6 for i=1:size(p,1)
7     if(inIncoreEnthalten(i) == 0)
8           continue;
9     end

10  Nachbardreiecke = find(eingabeGesamt(:,1) == i |
11                    eingabeGesamt(:,2) ==i | eingabeGesamt(:,3)==i);

12   q = zeros(4,4);
13   for l=1:size(Nachbardreiecke,1)
14       nnT = ((NormV(Nachbardreiecke(l,1),:))')
15            *(NormV(Nachbardreiecke(l,1),:));
16      minusnnTb = (-((NormV(Nachbardreiecke(l,1),:))')
17                  *(NormV(Nachbardreiecke(l,1),:))*((p(i, :))'));
18      minusbTnnT = ((((p(i, :))))*(NormV(Nachbardreiecke(l,1),:))')
19                   *(NormV(Nachbardreiecke(l,1),:));
20      bTnnTb = ((p(i, :))*((NormV(Nachbardreiecke(l,1),:))')
21                *(NormV(Nachbardreiecke(l,1),:))*(p(15 i, :))');

22      qneu = [nnT minusnnTb; minusbTnnT bTnnTb];
23      q = q + qneu;
24   end

25   qalle((i-1)*4+1:(i-1)*4+4,:) = q;
26 end
```

In der Zeile 1 wird die leere Matrix „qalle“ angelegt. In ihr wird in der nachfolgenden *for*-Schleife (Zeile 6-26) jedem Punkt eine Quadrik zugewiesen. Dazu wird zunächst (Zeilen 2 und 3) eine Punktematrix erstellt, die nur die Punktindizes der Punkte abspeichert, die sich im Incore-Speicher befinden. Die Variable „inIncoreEnthalten“, die in Zeile 4 erstellt wird, ist zunächst ein leerer Vektor mit „size(p,1)“ Zeilen. Diesem soll in Zeile 5 an den Stellen eine „1“ zugewiesen werden, wo sich ein Punkt aus dem Incore-Speicher befindet. Die „1“ steht hierbei für ein logisches „Ja“. In einer *if*-Abfrage (Zeile 7-9) wird getestet, ob ein Punkt i ein Punkt aus dem Incore-Speicher ist, oder ob sich dieser im Eingabe- oder Ausgabespeicher befindet. Wenn sich dieser Punkt i nicht im Incore-Speicher befindet, ihm also eine „0“ im Vektor „inIncoreEnthalten“ zugewiesen ist, so wird der nächste Durchlauf der *for*-Schleife begonnen. Es

wird für diesen Punkt somit keine Fehlerquadrik berechnet.
In der Matrix „Nachbardreiecke“ werden alle Dreiecke gespeichert, die den gemeinsamen Eckpunkt i besitzen. Die Matrix „qalle“ soll die Summe der Quadriken dieser an den Punkt i angrenzenden Dreiecke enthalten. Da eine Quadrik immer eine 4 x 4-Matrix ist, sind in „qalle“ die Zeilen 1 bis 4 dem ersten Punkt zugeordnet, die Zeilen 5 bis 8 dem zweiten usw..
In den Zeilen 10 und 11 werden für einen Punkt i seine angrenzenden Dreiecke gesucht, indem alle Spalten der Matrix aller Dreiecke zeilenweise nach dem Punkt i durchsucht werden. Dazu wird die Funktion „find“ verwendet. Ist ein Eckpunkt eines Dreiecks der Punkt i, so ist dieses Dreieck ein Nachbardreieck.
Die zweite *for*-Schleife (Zeile 13-24) geht alle gefundenen Nachbardreiecke durch und berechnet dann die Quadrik des Punktes i, indem die Summe der Quadriken der angrenzenden Dreiecke bestimmt wird. Dafür wird unter Verwendung des in Quellcode 6, Teil 1 bestimmten Normalvektors die Quadrik Q (aus Formel (2) im Abschnitt 4.2) berechnet. In Zeile 22 wird ein temporäres „qneu“ bestimmt, das jeweils die Quadrik des l-ten Nachbardreiecks berechnet und schließlich in Zeile 23 auf das laufende q aufaddiert wird. Dieses q wird nun in der leeren Matrix „qalle“ überschrieben (Zeile 25).

Quellcode 6, Teil 4 – Funktion „findBesteKante“: Berechnung des quadratischen Abstandes „dist“

```
1 stelleDesMins = -1;
2 minimalesDist = inf;
3 minimalesA = -1;
4 minimalesB = -1;


5 for l = 1:size(alleNurEinmal,1)

6     a = alleNurEinmal(l, 1);
7     b = alleNurEinmal(l, 2);

8     qa = qalle((4*(a-1)+1):(4*(a-1)+4),:);
9     qb = qalle((4*(b-1)+1):(4*(b-1)+4),:);

10    qkanteneu = qa + qb;

11    mid = (p(a,:) + p(b,:))/2;
12    vneu = [mid 1];
13    distneu = (vneu * qkanteneu )* (vneu');

14    if(distneu < minimalesDist && isGesperrteEcke(a) == 0 &&
15    isGesperrteEcke(b)==0)
16        minimalesDist = distneu;
17        stelleDesMins = l;
18        minimalesA = a;
19        minimalesB = b;
20    end
21 end
```

Dieser Teil der Funktion bestimmt den minimalen Abstand dist(mid, F)² (vgl. Formel (2)). Die Stelle, die am Ende den minimalen Abstand in der Menge der Kanten bezeichnet („StelleDesMins“), wird auf den Wert -1 gesetzt (Zeile 1). Ebenso wird mit dem Anfangspunkt („minimalesA“) und dem Endpunkt („minimalesB“) der Kante, die zu dem minimalen Abstand gehört, verfahren. Der minimale Abstand selbst wird mit ∞ vorbelegt (Zeile 2).

Nun werden alle Kanten in einer *for*-Schleife (Zeile 5-21) nach einer solchen Kante durchsucht. In Zeile 6 und 7 des Quellcodes 6 Teil 4 wird die l-te Kante der Kantenmatrix „alleNurEinmal“ in den Anfangspunkt a und Endpunkt b aufgesplittet. Mit dieser Information kann man nun auf die Quadriken dieser beiden Punkte zugreifen (Zeile 8-9). Dies geschieht, indem auf die Matrix, in der alle Quadriken gespeichert sind, „qalle“, zugegriffen wird und die zum Anfangspunkt a bzw. Endpunkt b gehörigen Quadriken extrahiert werden. Da eine Quadrik eine 4 x 4-Matrix ist, müssen also die richtigen **vier** Zeilen aus „qalle“ gefunden und ausgewählt werden. In Zeile 10 werden die Quadriken des Anfangs- und Endpunktes addiert, um somit die Quadrik der Kante zu

erhalten. Außerdem wird der Mittelpunkt der Kante berechnet, da dieser für Formel (2) benötigt wird (Zeile 11). Nun sind alle Parameter bekannt, um den Abstand „distneu“ zu berechnen. (Zeile 13).

Durch eine *if*-Abfrage (Zeile 14-20) wird getestet, ob dieser Abstand kleiner als der bisher kleinste ist. Zudem muss garantiert werden, dass die Kante keine gesperrten Ecken enthält. Dafür wertet man innerhalb der *if*-Abfrage die logische Abfrage „isGesperrteEcke(a) == 0“ aus. Die „0“ steht hier für „Nein“. Das heißt, es wird nur dann ein neuer minimaler Abstand „minimalesDist“ gesetzt, wenn die Abfrage, ob a eine gesperrte Ecke ist, mit „Nein“ beantwortet werden kann. Selbiges muss auch für den Endpunkt b erfüllt sein.

Quellcode 6, Teil 5 – Funktion „findBesteKante“:

```
1 k = -1;
2 for w =1:size(eingabe,1)
3     if isequal(sort([eingabe(w,1) eingabe(w,2)]),
4        alleNurEinmal(stelleDesMins,:)) || isequal(sort([eingabe(w,2)
5        eingabe(w,3)]), alleNurEinmal(stelleDesMins,:)) ||
6        isequal(sort([eingabe(w,3) eingabe(w,1)]),
7        alleNurEinmal(stelleDesMins,:))
8        k = w;
9        break;
10     end
11 end
```

In der *for*-Schleife (Zeile 2-11) wird ein zu der besten Kante gehörendes Dreieck gesucht, um diesem den Rückgabeparameter k zuzuordnen (Zeile 8). Dafür werden alle „eingabe“ Dreiecke durchsucht. Ein passendes Dreieck ist genau dann gefunden, wenn entweder die Kante von Punkt eins zum Punkt zwei, von zwei nach drei oder von drei nach eins des w-ten Dreiecks genau der Kante entspricht, die der Kante aus der Matrix „alleNurEinmal“ an der „stelleDesMins“ zugeordnet ist. Ist ein solches gefunden, werden die Schleifen mit dem Befehl „break“ verlassen.

Quellecode 6, Teil 6– Funktion „findBesteKante“:

```
1  if eingabe(k, 1) == minimalesA
2   i=1;
3    if eingabe(k,2) == minimalesB
4        j=2;
5    else
6        j=3;
7    end
8  elseif eingabe(k,2)== minimalesA
9    i=2;
10   if eingabe(k,3) == minimalesB
11       j=3;
12   else
13       j=1;
14   end
15 elseif eingabe(k,3)==minimalesA
16       i=3;
17   if eingabe(k,1)==minimalesB
18       j=1;
19   else
20       j=2;
21   end
22 end
```

Nun ist zwar der Parameter k bekannt, es fehlt allerdings noch die Zuordnung der Parameter i und j. Dazu wird das gefundene k-te Dreieck der Eingabematrix auf die Position vom Kantenanfangspunkt „minimalesA“ und Kantenendpunkt „minimalesB“ überprüft. Die umfassende *if*-Abfrage (Zeile 1-22) ist durch zwei weitere *elseif* -Abfragen untergliedert (Zeile 8 und 15). Dadurch sind die drei Fälle abgedeckt, in denen überprüft wird, ob der erste, der zweite oder der dritte Punkt des k-ten Dreiecks der Anfangspunkt „minimalesA“ ist. Somit wäre i = 1, i = 2 oder i = 3.

Um j korrekt zu bestimmen, wird je eine weitere *if*-Abfrage, die dann wiederum durch ein „else“ untergliedert ist, verwendet. Sei i = 1, so wird in der folgenden *if*-Abfrage (Zeile 3-7) getestet, ob der zweite oder der dritte Punkt dem Endpunkt „minimalesB“ entspricht. Ebenso wird mit den Fällen i = 2 und i = 3 verfahren.

4.3.5 Variante 2: Zufällige Wahl einer Kante

Durch Anwendung der Funktion „findZufaelligeKante“ geschieht die Wahl der zu dezimierenden Kante zufällig.

Quellcode 7 – Funktion „findZufaelligeKante“:

```
1 function [k, i, j] = findZufaelligeKante (eingabe, p, isGesperrteEcke)

2  k = 0;
3  while(...
4         k == 0 || ...
5         isGesperrteEcke(eingabe(k,1)) == 1 || ...
6         isGesperrteEcke(eingabe(k,2)) == 1 || ...
7         isGesperrteEcke(eingabe(k,3)) == 1)

8         k = ceil(rand(1) * size(eingabe,1));
9  end

10   i = ceil(rand(1) * 3);
11   j = i;

12   if(rand(1) > 0.5)
13        j = j + 1;
14   else
15        j = j - 1;
16   end

17   if(j == 4)
18       j = 1;
19   elseif(j == 0)
20       j = 3;
21   end
```

Der Funktion „findZufaelligeKante“ sind die Übergabeparameter „eingabe“, „p“ und „isGesperrteEcke“ zugeordnet. Die Rückgabeparameter sind dieselben wie bei der Funktion „findBesteKante“. Anfänglich ist der Parameter k, der das k-te Dreieck der übergebenen „eingabe“-Menge repräsentiert, auf den Wert 0 gesetzt.

Damit nicht aus Versehen ein Dreieck mit gesperrten Eckpunkten ausgewählt wird, stellt die *while*-Schleife (Zeile 3-9) sicher, dass so lange ein neues k bestimmt wird (Zeile 8), bis ein Dreieck k gefunden wird, welches dezimierbar ist.

Um k zufällig auszuwählen, wird mit dem Befehl „rand(1)“ eine zufällige Zahl zwischen 0 und 1 generiert und mit der Anzahl Zeilen der Eingabematrix multipliziert. Durch „ceil( )“ wird dieser Wert noch auf eine ganze Zahl aufgerundet. Auch der Variablen i wird eine zufällige Zahl zwischen 1 und 3 zugewiesen (Zeile 10). Diese repräsentiert somit den ersten, zweiten oder dritten Punkt des k-ten Dreiecks. Die Variable j soll ebenfalls eine Zahl zwischen 1 und 3 sein, allerdings muss $i \neq j$ gelten. Dies wird durch zwei *if*-Abfragen realisiert. Zu Beginn wird i gleich j gesetzt (Zeile 11). Dann wird in der ersten *if*-Abfrage (Zeilen 12-16) j zufällig entweder um 1

reduziert oder um 1 erhöht. Die Fälle j = 4 und j = 0 dürfen dabei nicht auftreten. Deshalb wird in der zweiten *if*-Abfrage (Zeilen 17-21) dem Fall j = 4 der neue Wert j = 1 und j = 0 der neue Wert j = 3 zugeordnet. Nun besitzt j also den Wert 1, 2 oder 3 und ist von i verschieden.

4.3.6 Durchführen der Dezimierung

Quellcode 8 – Auffüllen des Incore-Speichers:

```
1 while size(incore,1) < sizeincore
2     [incore, i] = read(eingabe, i, 1, incore);
3 end
```

Das erste Auffüllen des Incore-Speichers wird durch eine *while*-Schleife vorgenommen. In Zeile 1 wird festgelegt, dass das Füllen so lange ausgeführt wird, bis die Länge des Incore-Speichers die zu Anfang definierte Länge „sizeincore“ übersteigt. Nun wird die Funktion „read“ angewendet (Zeile 2). Dadurch werden Dreiecke aus der Eingabemenge in den Incore-Speicher, wie in Quellcode 2 beschrieben, verschoben.
Durch das zufällige Befüllen des Incore-Speichers sind Löcher im Netz des Incore-Speichers vorhanden und außerdem existieren dort unzusammenhängende Dreiecke (siehe Abbildung 12).

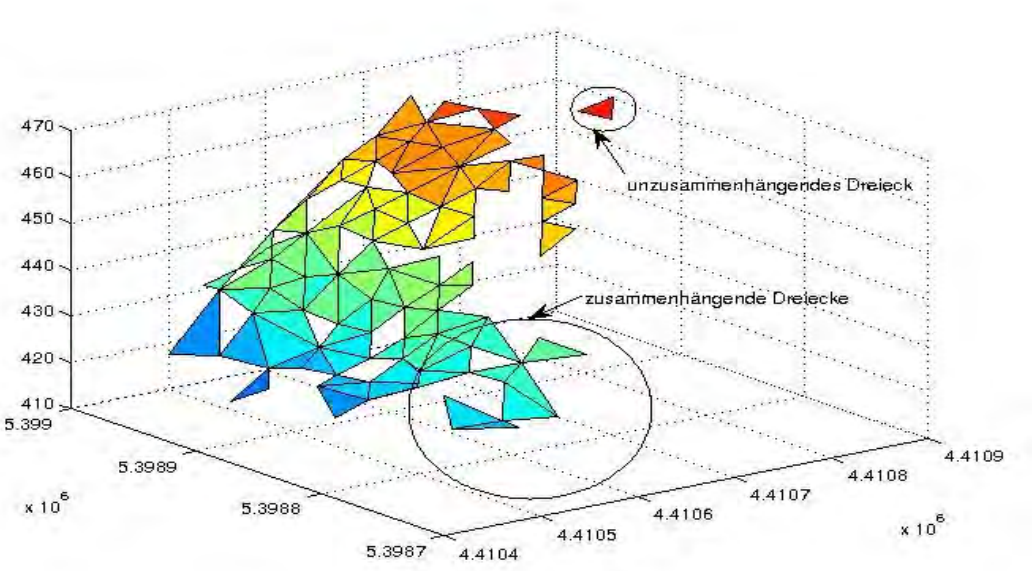

Abbildung 12: Visualisierung des Incore-Speichers zur Erklärung der zusammenhängenden und unzusammenhängenden Dreiecke.

Dezimiert der Algorithmus eines dieser einzelnen Dreiecke, so wird die Verschiebungsinformation global auch auf Nachbardreiecke, die sich im Incore-Speicher oder im Output befinden, übertragen.
Die Dezimierung wird durch zwei voneinander unabhängige *while*-Schleifen ausgeführt. Die erste *while*-Schleife wird in Quellcode 9 beschrieben.

Quellcode 9 – Aufrufen der Dezimierung:

```
1  while i < size(eingabe,1)

2     while(size(incore,1) < sizeincore)
3            [incore, i] = read(eingabe, i, 1, incore);
4            incore = deleteFalscheDreiecke(incore, p);
5     end

6      if rand(1) < (((1-pro)/(2*pro))/(((1-pro)/(2*pro))+1)) &&
7           haveFreeTriangle(incore,isGesperrteEcke)

8           groessevorher = size(incore,1);
9           [incore, p]=decimate(incore, eingabe, p, isGesperrteEcke);
10          incore  = canonizePointIndices(incore,  q1,q2);
11          eingabe = canonizePointIndices(eingabe, q1,q2);
12          output  = canonizePointIndices(output,  q1,q2);
13          groessenachher = size(incore,1);

14    [incore, i]=read(eingabe, i,groessevorher-groessenachher,incore);
15    else
16          [output, incore] = write(output, incore, 1);
17          [incore, i] = read(eingabe, i, 1, incore);
18     end
19 end
```

Die *while*-Schleife (Zeile 1-19) wird durchgeführt, solange die Laufvariable i nicht den Wert der Länge der Eingabematrix besitzt. Das bedeutet: Erst wenn alle Dreiecke von der Eingabematrix in den Incore-Speicher verschoben wurden, ist die erste *while*-Schleife der Dezimierung beendet. Eine Bedingung des Algorithmus ist, dass der Incore-Speicher möglichst immer voll sein sollte (siehe 4.2). Dies wird mit der *while*-Schleife (Zeile 2-5) erreicht. Diese füllt in jedem Durchlauf den Incore-Speicher bis zur maximalen Größe „sizeincore“ auf. Die *while*-Schleife muss zusätzlich die Funktion „deleteFalscheDreiecke“ aufrufen, um sicherzugehen, dass sich keine falschen Dreiecke mehr unter den Dreiecken befinden (siehe dazu Quellcode 12).
Um die Bedingung einzuhalten, dass die Anzahl der „decimate“- und der „write“-Operationen gleich groß ist, muss der Anteil der „decimate“-Operationen an der Gesamtzahl der „decimate“- und der „write“-Operationen bestimmt werden.

Wenn nun eine zufällige Zahl zwischen 0 und 1 kleiner als dieser Anteil ist, wird durch Aufruf der Funktion „decimate“ (Zeile 9) eine Anzahl an Dreiecken gelöscht und wiederum die gleiche Anzahl aus der Eingabematrix in den Incore-Speicher eingelesen. Dies geschieht mithilfe der Funktion „read“ (Zeile 11). Um diese Anzahl zu bestimmen, kann man sich durch Einführung der Variablen „groessevorher“ und „groessenachher“ jeweils die Einträge des Incore-Speichers vor und nach Stattfinden der Dezimierung bestimmen. Die Differenz ist die Anzahl der mit „read“ einzulesenden Dreiecke.
Des Weiteren müssen alle Dreiecksmengen durch die im Quellcode 5 vorgestellte Funktion „canonizePointIndices“ modifiziert werden, um das Entstehen von Löchern zu verhindern. Deshalb wird in den Zeilen 10, 11 und 12 diese Funktion aufgerufen.
„Decimate“ kann nur ausgeführt werden, wenn es noch Dreiecke gibt, die dezimiert werden dürfen (Randdreiecke sind ja für die Dezimierung gesperrt). Deshalb wird als Zusatz in der *if*-Abfrage getestet, ob noch verfügbare Dreiecke vorhanden sind. Dies geschieht, indem durch ein logisches „und“ die Funktion „haveFreeTriangle“ (siehe Quellcode 11) an die erste Bedingung (Prozentsatz) angehängt wird. Ist die erste *while*-Schleife beendet und sind damit alle Dreiecke aus der Eingabematrix in den Incore-Speicher verschoben worden, muss der Incore-Speicher geleert werden. Dies wird im Quellcode 10 beschrieben.

Quellcode 10 – Leeren des Incore-Speichers:

```
1 while size(incore,1) > 0

2     if rand(1) < (((1-pro)/(2*pro))/(((1-pro)/(2*pro))+1)) &&
3                 haveFreeTriangle(incore, isGesperrteEcke)
4          [incore, p] = decimate(incore, eingabe, p, isGesperrteEcke);
5           incore  = canonizePointIndices(incore,  q1,q2);
6           eingabe = canonizePointIndices(eingabe, q1,q2);
7           output  = canonizePointIndices(output,  q1,q2);
8     else
9         [output, incore] = write(output, incore, 1);
10     end
11 end
```

Die *while*-Schleife (Zeile 1-11) endet, sobald der Incore-Speicher leer ist. Wieder entscheidet in einer *if*-Abfrage (Zeile 2-10) eine zufällig generierte Zahl und die Verfügbarkeit von dezimierbaren Dreiecken, ob entweder eine Dezimierung durchgeführt (Zeile 4) oder ob ein Dreieck mithilfe der Funktion „write“ in den Output geschrieben werden soll (Zeile 9). Nach

Aufrufen von „decimate“ müssen die Dreiecksmengen wieder durch die Funktion „canonizePointIndices“ angepasst werden, um Löcher zu vermeiden.

Quellcode 11 – Funktion „haveFreeTriangle“:

```
1 function [result] = haveFreeTriangle(t, isGesperrteEcke)

2 result = 0;

3 for k = 1:size(t,1)
4    if(isGesperrteEcke(t(k,1)) == 0 && ...
5        isGesperrteEcke(t(k,2)) == 0 && ...
6        isGesperrteEcke(t(k,3)) == 0)
7        result = 1;
8        break;
9    end
10 end
```

Gegen Ende eines Programmdurchlaufs trat beim Leeren des Incore-Speichers das Problem auf, dass im Incore-Speicher ab einem gewissen Zeitpunkt nur noch Dreiecke mit mindestens einem gesperrten Eckpunkt vorhanden waren, sodass es nicht möglich war, diese durch Dezimierung zu entfernen. Abhilfe schafft die Funktion „haveFreeTriangle“. Dieser Funktion werden der Incore-Speicher als Matrix t und die gesperrten Eckpunkte übergeben. Die Funktion gibt ein Ergebnis „result“ zurück, das entweder 0 („Nein“) oder 1 („Ja“) beträgt. Der Rückgabewert „result“ wird mit 0 vorbelegt. Findet sich nun innerhalb der noch bestehenden Incore-Menge ein Dreieck, das keine gesperrte Ecke als Punkt hat, so ist dieses Dreieck ein weiterer Kandidat für die Dezimierung und „result“ wird auf 1 gesetzt (Zeile 7).

Quellcode 12 – Funktion „deleteFalscheDreiecke“:

```
1 function tneu = deleteFalscheDreiecke(t, p)

2 epsilon = 1e-5;

3  falscheDreiecke = find(...
4     (abs(p(t(:,1),1) - p(t(:,2),1)) < epsilon & abs(p(t(:,1),2) –
5 p(t(:,2),2)) < epsilon & abs(p(t(:,1),3) - p(t(:,2),3)) < epsilon) | ...
6      (abs(p(t(:,1),1) - p(t(:,3),1)) < epsilon & abs(p(t(:,1),2) –
7 p(t(:,3),2)) < epsilon & abs(p(t(:,1),3) - p(t(:,3),3)) < epsilon) | ...
8      (abs(p(t(:,2),1) - p(t(:,3),1)) < epsilon & abs(p(t(:,2),2) –
9 p(t(:,3),2)) < epsilon & abs(p(t(:,2),3) - p(t(:,3),3)) < epsilon));
10 richtigeDreiecke = setdiff(1:size(t,1), falscheDreiecke);

11 tneu = t(richtigeDreiecke,:);
```

Die Übergabeparameter der Funktion „deleteFalscheDreiecke“ sind zum einen die Eingabematrix, innerhalb des Funktionscodes mit der Variable „t“ bezeichnet, und zum anderen die Matrix aller Punktkoordinaten „p“. Die Funktion hat die Aufgabe, die Dreiecke nach „falschen Dreiecken“ zu durchsuchen, diese zu entfernen und die modifizierte Dreiecksmatrix „tneu“ zurückzugeben.

„Falsche Dreiecke“ entstehen bei der Durchführung des „edge collapse“. Wie bereits in der Beschreibung des Quellcodes 4 erklärt, werden dabei „Dreiecke“ mit nur zwei Eckpunkten geschaffen. Die Variable „falscheDreiecke“ wird in Zeile 3 generiert. Durch Aufruf der Funktion „find“ werden alle Dreiecke in dieser Variable gespeichert, deren erster und zweiter, erster und dritter oder zweiter und dritter Punkt gleich sind. Dazu wird der Betrag der Differenz der jeweiligen x-, y-, oder z-Koordinaten mit einem in Zeile 2 definierten Wert „epsilon“ ($epsilon \approx 0$) verglichen. Ist also dieser Betrag kleiner als der Wert „epsilon“, so kann davon ausgegangen werden, dass es sich bei den untersuchten Koordinatenwerten um die gleichen handelt. Die Funktion „find“ gibt die Indizes der Dreiecke zurück, die zu den falschen Dreiecken gehören. Die Indizes der „richtigeDreiecke“ entstehen, indem die Differenz der Zeilen der Eingabematrix und der Indizes der falschen Dreiecke gebildet wird (Zeile 10). Nur die Indizes, die in dem Vektor „richtigeDreiecke“ gespeichert sind, werden in Zeile 11 dazu verwendet, die Dreiecksmenge „tneu“ zu erstellen. Diese enthält dann nur richtige Dreiecke.

4.4 Ergebnisse

In den folgenden Abschnitten werden die Ergebnisse der verschiedenen Varianten miteinander verglichen und zudem hinsichtlich benötigter Rechenzeiten ausgewertet.

Zunächst muss eine Ausgangsdatenmenge erstellt werden, um die Dezimierung durch die in 4.3.4 bzw. 4.3.5 beschriebenen Varianten 1 bzw. 2 durchzuführen.

4.4.1 Erstellen der Ausgangsdatenmenge

Wie unter 4.1 erklärt, besitzt jedes Teilgebiet der Testdaten 1.000.000 Datenpunkte. Allerdings wurde der Testlauf auf einem Rechner mit niedriger Leistung durchgeführt. Daher war es nicht möglich, in Matlab eine Datei dieser Größe zu öffnen. Da es sich zudem nur um einen Testlauf handelt und eine detailgenaue Beschreibung der Topografie daher keine Bedeutung hat, wurde nur jeder fünfte Datenpunkt eingelesen, um die Bearbeitung mit Matlab zu ermöglichen. Der durch Heranzoomen vergrößerte Ausschnitt der Punktwolke in x-y-Ebene ist in Abbildung 13 zu sehen.

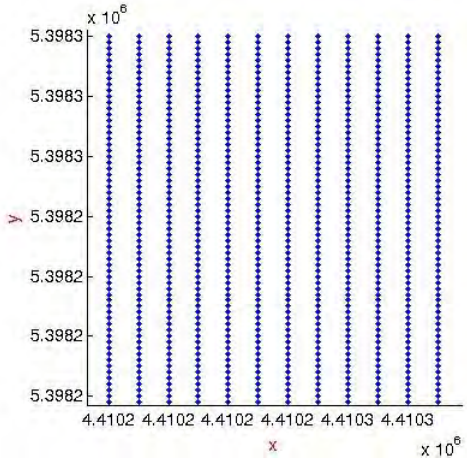

Abbildung 13: Herangezoomter Ausschnitt der Punktwolke bei m = 1.

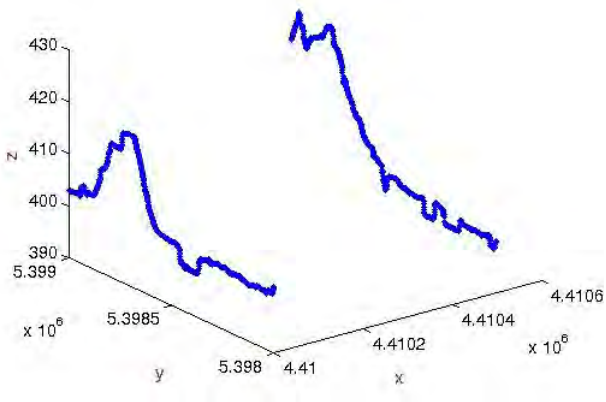

Abbildung 14: Punktwolke bei m = 100.

Es sind deutlich Streifen zu erkennen. Für das Erstellen der Ausgangsdatenmenge wird durch die Vergabe der Variablen „m“ die Datenmenge nochmals verkleinert.

Setzt man die Variable „m“ beispielsweise auf 100, so erhält man die in Abbildung 14 gezeigte Punktverteilung. Diese hat wenig Aussagekraft. Die streifenförmige Verteilung entsteht, weil das Ausgangsgebiet ein quadratisches Gebiet mit 1000 x 1000 Punkteinträgen ist und nun nur noch jeder fünfhunderte Datenpunkt verwendet wird, so bleiben nur (1000 : 500 = 2) zwei verschiedene Streifen mit den jeweils selben zwei x-Werten bestehen.

Variiert man die Variable „m“, so ergeben sich von Grund auf verschiedene Abbildungen, da nicht immer dieselben x-Werte angewählt werden. Die Punkte sind damit im Raum verteilt. Eine gute Punktverteilung erhält man in der Regel, indem man „m“ mit einer Primzahl vorbelegt, da so die Punkte stets versetzt ausgewählt werden. In dem Testdurchlauf wurde m = 107 gesetzt. Die zugehörige Punktwolke ist in Abbildung 15 dargestellt.

Neben der Variablen „m“ müssen auch noch „pro“ und „sizeincore“ vom Benutzer eingegeben werden. Hier wurden „pro“ = 0.5 und „sizeincore“ = 100 gesetzt.

Durch die Wahl der Variable m = 107 erfolgte eine weitere Datenreduktion. Nun haben die x-, y- und z-Vektoren nur noch 1870 Einträge.

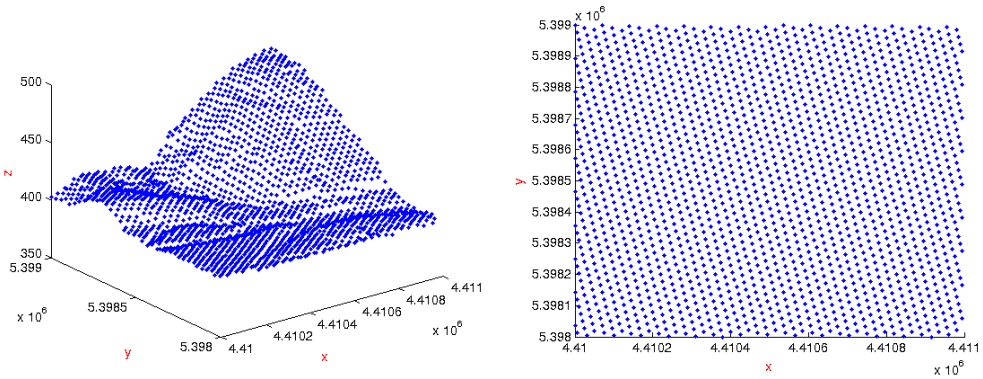

Abbildung 15: Punktwolke der Ausgangsdatenmenge. Links: Schrägansicht. Rechts: Draufsicht.

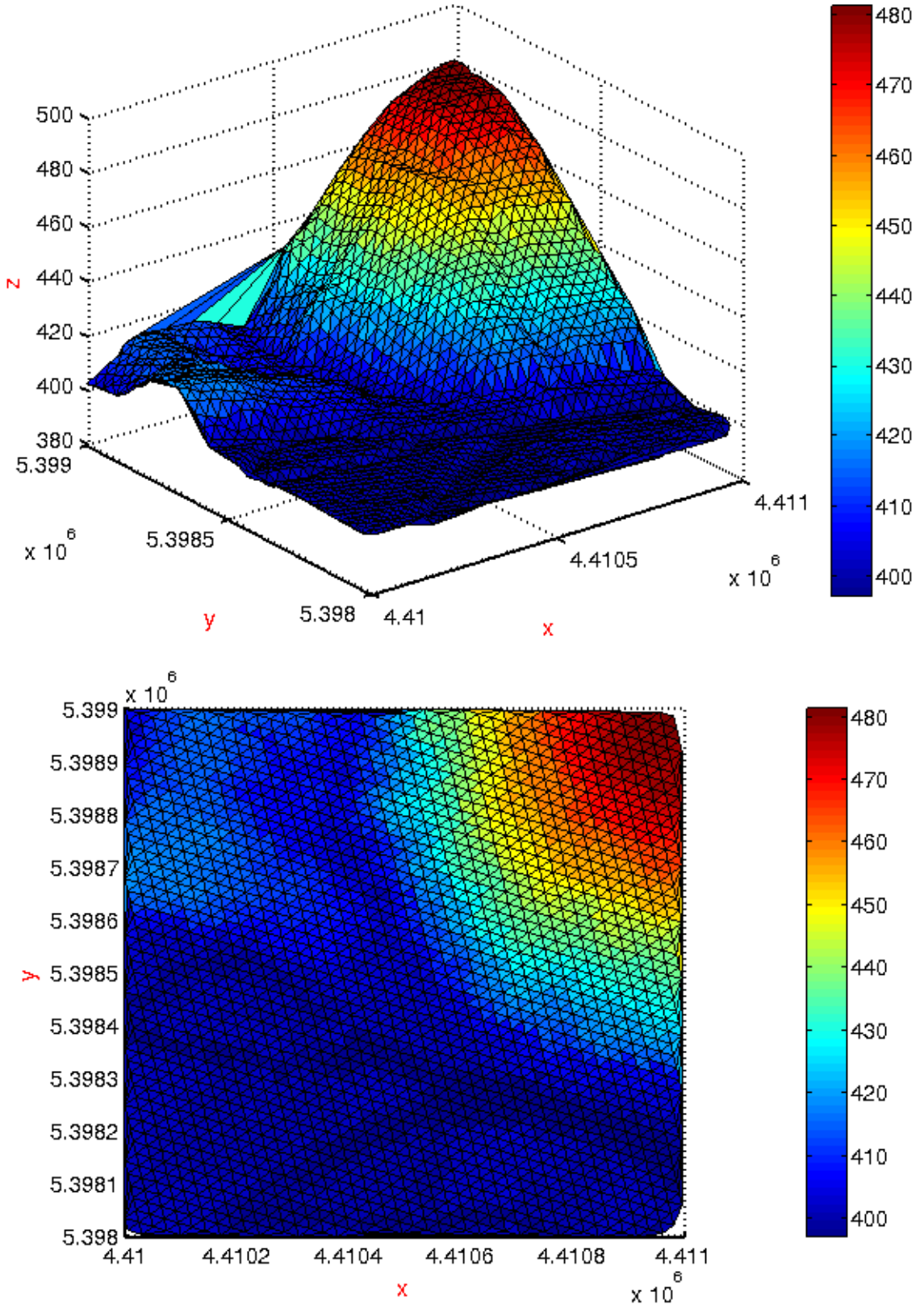

Abbildung 16: Delaunay-Triangulation der Ausgangsdatenmenge. Oben: Schrägansicht. Unten: Draufsicht.

Vor allem in Abbildung 15 im rechten Bild ist die Regelmäßigkeit der Punktwolke, die durch Airborne Laserscanning aufgenommen wurde, gut erkennbar.

Durch Anwendung der Delaunay-Triangulation wird eine 3699 x 3-Matrix erstellt. Das so generierte Netz (siehe Abbildung 16) besitzt also 3699 Dreiecke. Wieder zeigt sich – vor allem in der Draufsicht (x-,y-Ebene) – die Regelmäßigkeit der Daten. Vergleicht man das untere Bild der Abbildung 16 mit der in Kapitel 4.1 in Abbildung 7 gezeigten Luftbildaufnahme, bemerkt man die Übereinstimmungen der beiden Abbildungen. Die Flussläufe der Donau und Wörnitz sind deutlich zu sehen (dunkelste Blauwerte).

4.4.2 Ergebnisse der Dezimierung der Datenmenge unter Anwendung der Variante 1 „findBesteKante“

Die Berechnung der besten zu dezimierenden Kante besitzt einen höheren Rechenaufwand als die Berechnung einer zufälligen Kante (siehe dazu 4.4.4 Bewertung der Ergebnisse). Die Ergebnisse sind in Abbildung 17 dargestellt. Die Umstrukturierung im Output ist deutlich zu erkennen. Abbildung 18 zeigt die Ausgangspunktwolke (blaue Punkte) zusammen mit der ausgedünnten Punktwolke (rote Punkte). Die Regelmäßigkeit der Ausgangspunktwolke ist nach der Dezimierung nicht länger gegeben.

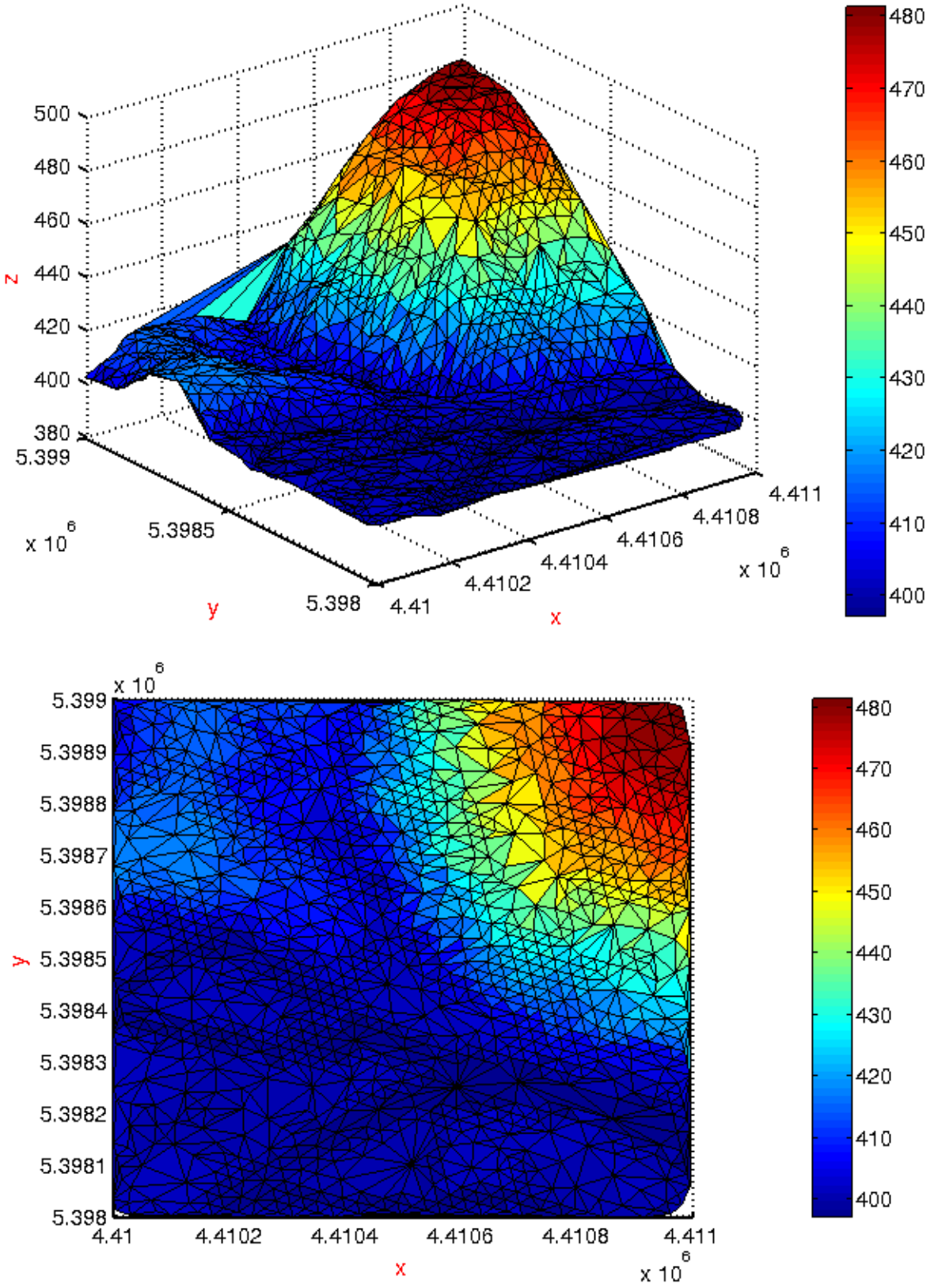

Abbildung 17: Ergebnisse der Dezimierung nach Variante 1. Wahl der besten Kante. Oben: Schrägansicht. Unten: Draufsicht.

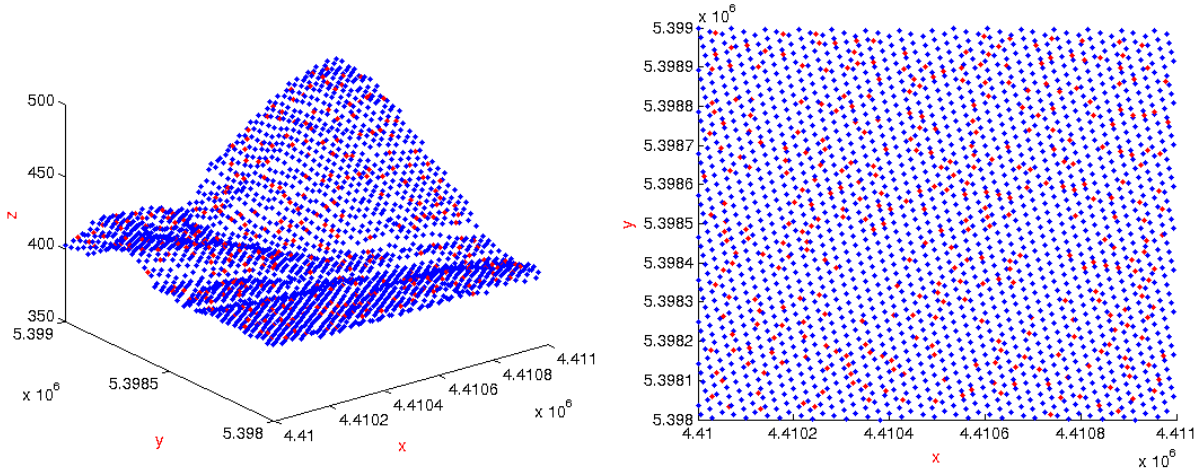

Abbildung 18: Punktwolke vor (blau) und nach (rot) der Dezimierung nach Variante 2. Links: Schrägansicht. Rechts: Draufsicht.

4.4.3 Ergebnisse der Dezimierung unter Anwendung der Variante 2 „findZufaelligeKante“

Im Folgenden werden die Ergebnisse der Dezimierung vorgestellt, die unter Anwendung der Funktion „findZufaelligeKante“ zustande gekommen sind. Grundlage der Dezimierung bildet wieder die unter 4.3.1 präsentierte Punktwolke. Die Abbildung 19 zeigt das dezimierte Dreiecksnetz, die Abbildung 20 die Punktwolke vor (blaue Punkte) und nach (rote Punkte) der Dezimierung.

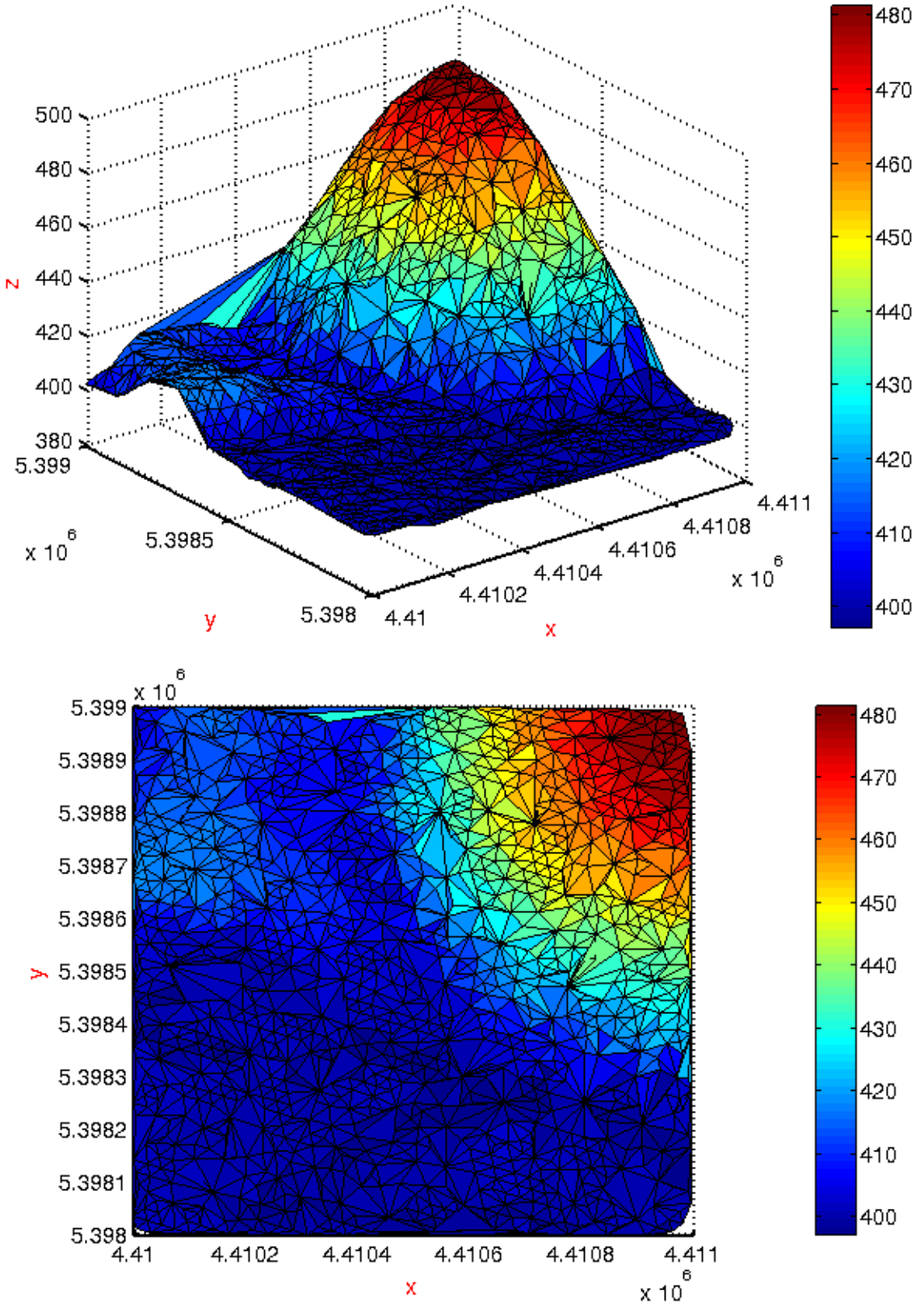

Abbildung 19: Ergebnisse der Dezimierung nach Variante 2: Wahl einer zufälligen Kante. Oben: Schrägansicht. Unten: Draufsicht.

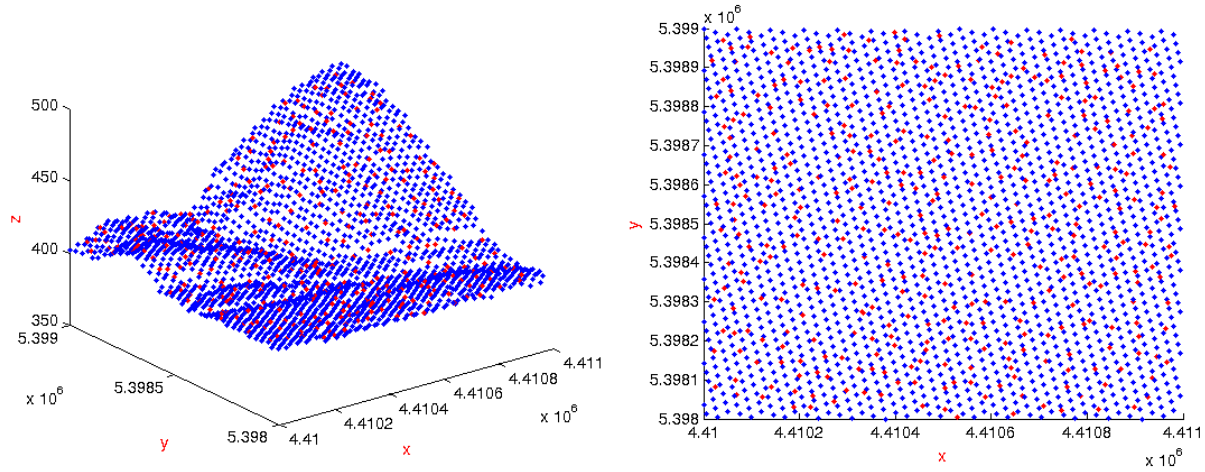

Abbildung 20: Punktwolke vor (blau) und nach (rot) der Dezimierung nach Variante 2. Links: Schrägansicht. Rechts: Draufsicht.

4.4.4 Bewertung der Ergebnisse aus den zwei Varianten

Vergleicht man die Ergebnisse aus 4.4.2 und 4.4.3 miteinander, so fällt auf, dass die Dezimierung bei der Wahl der zu dezimierenden Kante nach Variante 2 – wie gewünscht – weitaus zufälliger auftritt als bei der Wahl nach Variante 1. Betrachtet man beispielsweise den rechteckigen Ausschnitt (roter Kasten) in der Abbildung 21, so erkennt man, dass bei der Dezimierung nach Variante 1 (Wahl der besten Kante, rechts) dieser Bereich deutlich feiner ist als nach Variante 2 (Wahl einer zufälligen Kante, links). Der charakteristische Bereich des Flusslaufs der Donau ist im rechten Bild noch deutlich als solcher zu erkennen. Im linken Bild erscheint dieser Bereich sehr willkürlich und verzerrt.

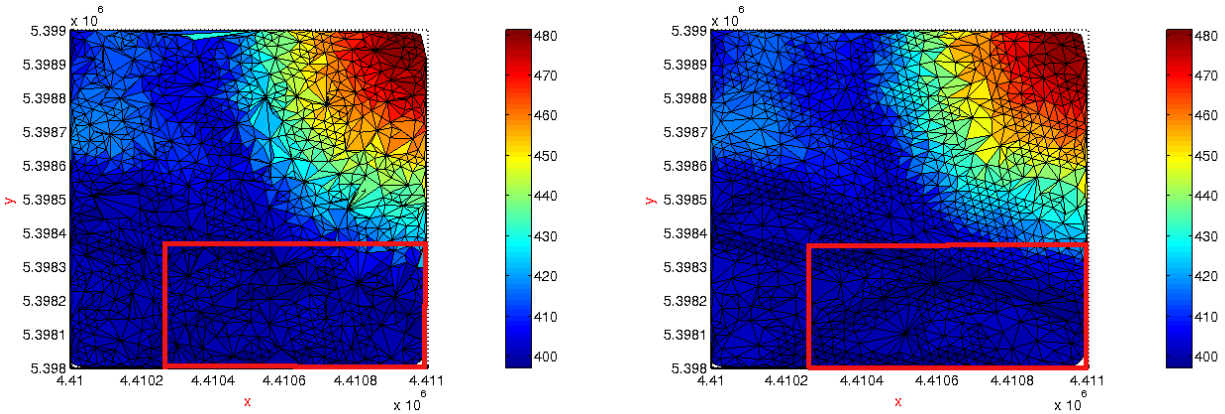

Abbildung 21: Funktionalität des Programms. Links: Ergebnis der Dezimierung nach Wahl einer zufälligen Kante. Rechts: Ergebnis der Dezimierung nach Wahl der besten Kante.

Auch bei Wiederholen des Durchlaufs war der Output in diesem Ausschnitt bei der Wahl der

besten Kante stets ziemlich detailliert zu erkennen.

Die folgenden Plots zeigen die Dezimierung auf 60 % der Ausgangsdatenmenge (pro = 0,6)mit m = 107 und sizeincore = 100 nach Variante 1 (Abbildung 22) und Variante 2 (Abbildung 23).

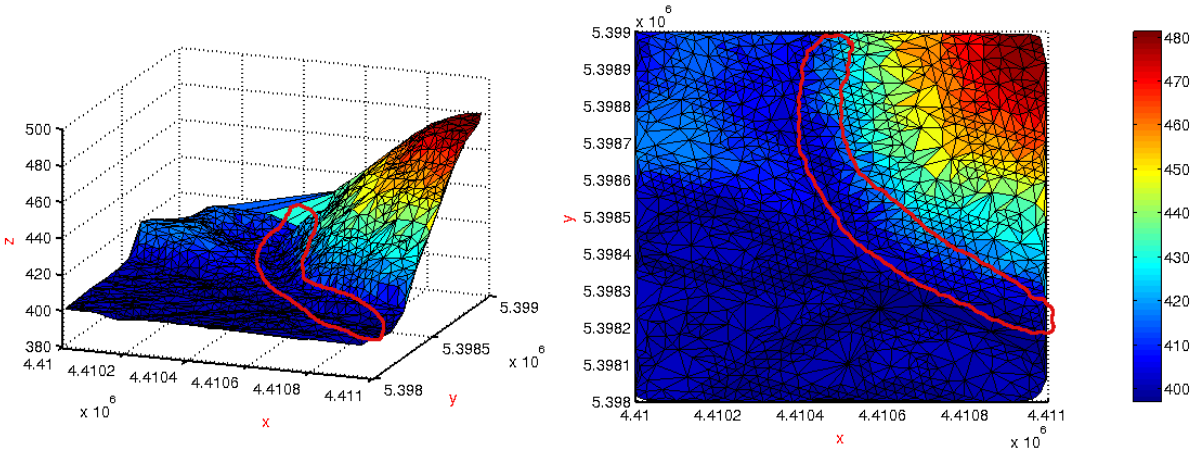

Abbildung 22: Auswertung der Ergebnisse. Dezimierung nach Variante 1. Charakteristische Merkmale wie der Steigungswechsel von flach auf steil bleiben erhalten.

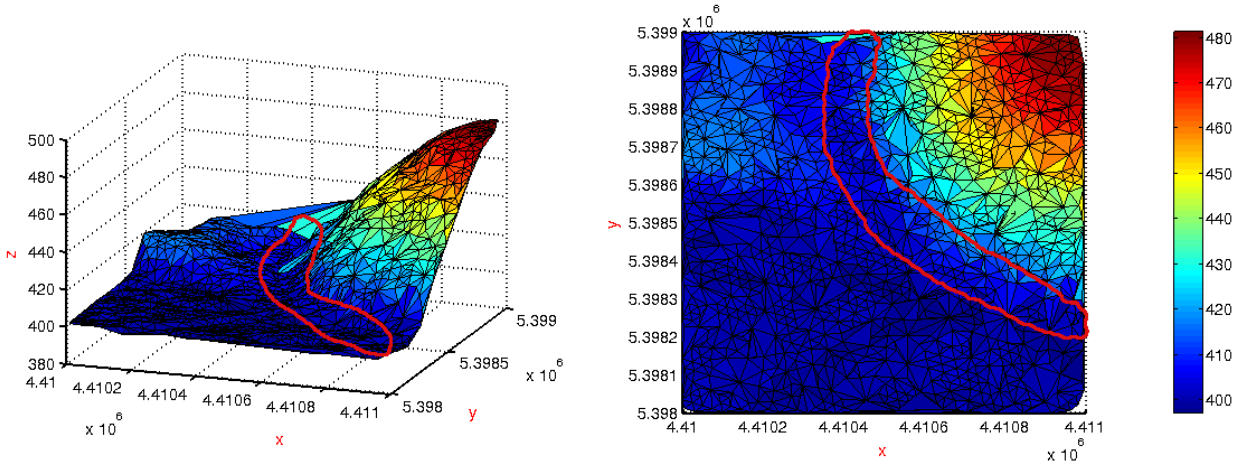

Abbildung 23: Auswertung der Ergebnisse. Dezimierung nach Variante 2. Die Ausdünnung erfolgt auch in wichtigen charakteristischen Bereichen wie dem Steigungswechsel.

In diesen Bildern wird der rot markierte Bereich verglichen. Dieser Bereich stellt den Steigungswechsel aus dem flachen hin zum steilen Gebiet dar. Dieser charakteristische Teil des Geländes ist im Output der Dezimierung nach Variante 1 sehr detailliert erhalten geblieben. Im Gegenteil dazu zeigt Abbildung 23, dass der Bereich durch die Dezimierung nach Variante 2 einen hohen Informationsverlust erfahren hat.

Durch die Matlabfunktion „tic-toc“ kann die für eine Rechenoperation benötigte Zeit gestoppt

werden. In folgender Tabelle sind die benötigten Rechenzeiten für Variante 1 und 2 für verschiedene Variablen m zusammengefasst. Die Parameter pro = 0.6 und sizeincore = 100 bleiben wie in den vorherigen Testläufen bestehen

	Variante 1 Rechenzeiten [s]			**Variante 2 Rechenzeiten [s]**		
	m = 107	m = 413	m = 713	m = 107	m = 413	m = 713
Messung 1	1780,81	148,11	59,87	14,48	3,81	2,44
Messung 2	1834,54	146,01	48,98	14,65	3,97	2,39
Messung 3	1724,66	153,24	60,93	14,72	3,94	2,56
Messung 4	1798,32	149,62	58,76	14,46	3,82	2,41
Messung 5	1659,98	148,79	53,41	14,55	3,91	2,48
Durchschnitt:	8798,31/5 = 1759,66	745,77/5 = 149,15	281,95/5 = 56,39	72,86/5 = 14,57	19,45/5 = 3,89	12,28/5 = 2,46

Tabelle 2: Rechenzeiten der verschiedenen Varianten.

Der Vergleich der Varianten zeigt deutlich, wie sehr sich die Berechnungsdauern der einzelnen Varianten voneinander unterscheiden.

Bei der Verwendung jedes 107-ten Datenpunktes hat die Punktwolke 1870 Punkte. Bei m = 413 entsteht eine Punktwolke mit 485 Punkten, wohingegen m = 713 eine Punktwolke mit 281 Punkten erzeugt.

Bei Variante 1 führt die Verkleinerung der Eingangspunktwolke von 1870 auf 485 Punkten (also auf 26 %) zu einer Verringerung der Rechenzeit von 1759,66 auf 149,15 s, also eine Verringerung auf etwa 8 %. Dies entspricht in etwa der Quadrierung des Verhältnisses (485/1870)^2. Darum lässt sich die Komplexität der Variante 1 etwa mit $O(n^2)$ angeben. Zur Verdeutlichung ist in Abbildung 24 dieser Zusammenhang grafisch dargestellt.

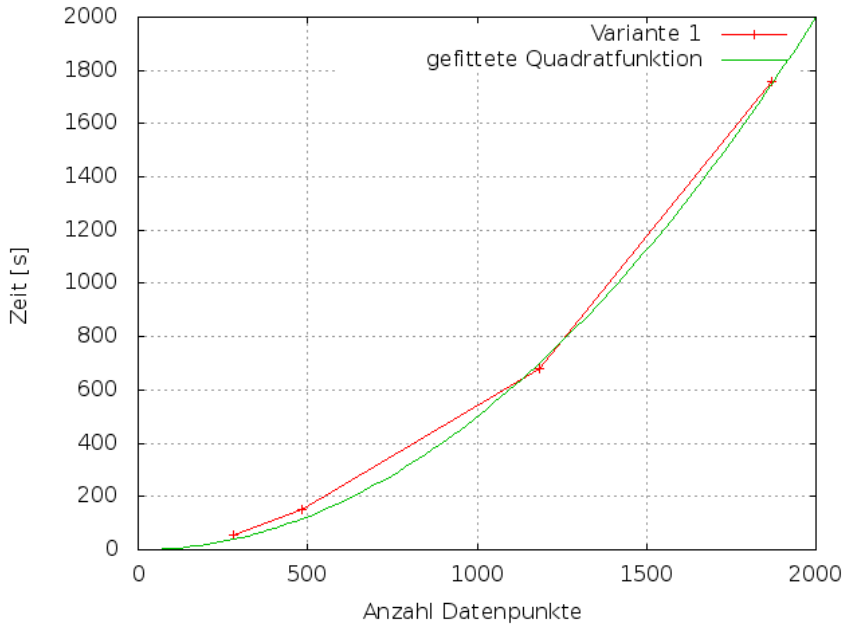

Abbildung 24: Laufzeitverhalten bei Variante 1.

Betrachtet man Variante 2, so stellt man folgende Unterschiede fest: Eine Verkleinerung der Punktwolke auf 26 % der Punkte führt zu einer Abnahme der Rechenzeit von ursprünglich 14,57 s auf 3,89 s, also auf 27 %. Als grobe Abschätzung kann man dafür eine Komplexität von O(n) angeben. Abbildung 25 verdeutlicht das Laufzeitverhalten anschaulich.

Abbildung 25: Laufzeitverhalten bei Variante 2.

Auch die Größe des Incore-Speichers geht in die Berechnungsdauer ein. Je größer der Incore-Speicher zu Anfang des Programms gewählt wird, desto länger dauert die Berechnung.
Es bleibt zu erwähnen, dass Variante 1 die bessere Methode ist, um Netze zu verkleinern, da dabei eine Kante nur dann dezimiert wird, wenn diese das vorhandene Netz möglichst wenig verändert.

5. Fazit

Der implementierte Algorithmus verwendet den „edge collapse“, um die Datenmenge auszudünnen. Die Position des neu einzufügenden Punktes ist der Mittelpunkt der zu dezimierenden Kante. Die Wahl der zu dezimierenden Kante geschieht in Variante 1 durch Berechnung einer Fehlerquadrik durch „quadric error metric“. Bei Variante 2 geschieht die Wahl der Kante zufällig.
Die Fehlerquadrik wird für den laufenden Incore-Speicher berechnet. Die Kante mit dem minimalen Fehler wird vom „edge collapse“ modifiziert.
Der Algorithmus kann insbesondere bei der Positionsbestimmung insofern modifiziert werden, dass auch die Position durch die Fehlerquadrik bestimmt wird.
Da die Fehlerquadrik lediglich für den Incore-Speicher berechnet wird, wird auch nur die beste Kante innerhalb dieser Menge gefunden. Würde man die Suche auf die gesamte Datenmenge ausweiten, würde man bessere Ergebnisse erzielen. Allerdings hätte dies einen enormen Anstieg der Rechenzeit zur Folge. Dies gilt es möglichst zu vermeiden.
Die Variante 2 ist zwar auch ein Ausdünnungsalgorithmus, allerdings wird durch die zufällige Wahl der zu dezimierenden Kante keine gewählt, die das Netz möglichst wenig verändert. Somit ist die Qualität der Ausgabe bei Variante 2 schlechter als bei Variante 1. Dies ist vor allem mit Blick auf die Anwendung für Hochwassersimulationen von großem Nachteil. Gerade dort ist es wichtig, Eingabedaten von guter Qualität zu verwenden.

6. Ausblick

Das Ergebnis einer Datenaufbereitung kann beispielsweise für eine Hochwassersimulation verwendet werden. Hierfür eignen sich Simulationsprogramme wie BASEMENT (Basic Simulation Environment). BASEMENT wurde an der Versuchsanstalt für Wasserbau, Hydrologie und Glaziologie an der ETH Zürich entwickelt. Neben einem aufbereiteten digitalen

Geländemodell benötigt die Software detaillierte Informationen über Abflussganglinien, Geschwindigkeitsprofile, Informationen über den Wasserstand und weitere Randbedingungen wie z.B. die Rauigkeit der Sohle und Druckverteilungen [6].

Durch Hochwassersimulationen lassen sich u.a. Gefahrenkarten erstellen, die einen direkten Bezug zur vorhandenen Landnutzung und Wohnsituation aufweisen. Darin werden mögliche Überschwemmungsgebiete gekennzeichnet. Hochwassergefahrenkarten kommen beispielsweise in Österreich zur Anwendung. Der Onlinedienst HORA – Natural Hazard Overview & Risk Assesment Austria – berechnet das Überschwemmungsrisiko für ein HQ_{30}, HQ_{100}, und HQ_{200}. Dabei wird von einem „Worst-Case-Szenario“ ausgegangen, das heißt, bestehende Hochwasserschutzeinrichtungen wie z.B. Dämme werden in der Berechnung außer Acht gelassen. Das Projekt wurde 2002 vom Lebensministerium und dem Verband der Versicherungsunternehmen Österreich (VVO) ins Leben gerufen, um es allen Bürgerinnen und Bürgern zu ermöglichen, sich über ihr Risiko von Überschwemmungen zu informieren [8].

Auch im Wasserhaushaltsgesetz der Bundesrepublik Deutschland ist im Abschnitt 6 § 74 verankert, dass Gefahrenkarten und Risikokarten für Risikogebiete zu erstellen sind.

Abschließend lässt sich feststellen, dass diese Konzepte, basierend auf hydrodynamisch-numerischen Simulationen, Maßnahmen beinhalten, um effektiv Hochwasserschutz betreiben zu können. Gravierende Folgen, wie die des Pfingsthochwassers im Jahre 1999 oder des Elbhochwassers 2002, können so in Zukunft verhindert oder zumindest vermindert werden.

Anhang

CD

Auf der beigefügten CD sind folgende Daten enthalten:

- Der schriftliche Teil der Bachelorarbeit als PDF-Dokument
- Der gesamte Quellcode des implementierten Programms mit allen erstellten Funktionen
- verwendete Literatur
- Bilder

Literaturverzeichnis

1: Baldenhofer K G. Lexikon der Fernerkundung – LiDAR. K. G. Baldenhofer. 2012. Internetseite. URL: http://www.fe-lexikon.info/lexikon-l.htm#lidar. Abgerufen: 07.08.2012.

2: Bayerische Vermessungsverwaltung. Laserpunkte. Landesamt für Vermessung und Geoinformation. Keine Jahresangabe. Internetseite. URL: http://vermessung.bayern.de/geobasis_lvg/gelaendemodell/laserpunkt.html. Abgerufen: 07.08.12.

3: Bayerische Vermessungsverwaltung. Digitale Geländemodelle. Landesamt für Vermessung und Geoinformation Bayern. 2009. PDF-Dokument. URL: http://www.vermessung.bayern.de/file/pdf/1614/download_faltblatt-dgm09.pdf. Abgerufen: 07.08.12.

4: Bayerische Vermessungsverwaltung. Digitales Geländemodell (DGM). Landesamt für Vermessung und Geoinformation. Keine Jahresangabe. Internetseite. URL: http://vermessung.bayern.de/geobasis_lvg/gelaendemodell.html. Abgerufen: 07.08.2012.

5: Biosphärenreservat Mittelelbe. Hochwassereinflussfaktoren allgemein. Ministerium für Landwirtschaft und Umwelt Sachsen-Anhalt. Keine Jahresangabe. Internetseite. URL: http://www.mittelelbe.com/mittelelbe/hochwasser_einflussfaktoren_73_1.html. Abgerufen: 07.08.2012.

6: Boes R, Fäh R, Müller R, Rousselot P, Volz C, Vonwiller L, Vetsch D. System Manuals of Basement. ETH Zürich. 2011. PDF-Dokument. URL: http://www.basement.ethz.ch/docs/BASEMENT_v2.2_User.pdf. Abgerufen: 07.08.2012.

7: Brügelmann R. Automatic Breakeline Detection from Airborne Laserrange Data. In: International Archives of thePhotogrammetry, Remote Sensing and Spatial Information Sciences, 33. 109-116. 2000.

8: Bundesministerium für Land- und Forstwirtschaft, Umwelt und Wasserwirtschaft. Hochwasserrisikozonierung Austria – HORA. Lebensministerium Österreich. 2011. PDF-Dokument. URL: http://www.hora.gv.at/assets/eHORA/pdf/HORA_Hochwasser_Weiterfuehrende-Informationen_v3.pdf. Abgerufen: 08.08.2012.

9: Dyn N, Iske A, Wendland H. Meshfree Thinning of 3D Point Clouds. In: Foundations of Computational Mathematics. 409-425. 2007.

10: Förstner W. Image Preprocessing for Feature Extraction in Digital Intensity, Color and Range Images. Institut fur Photogrammetrie, Universität Bonn. 1998. PDF-Dokument. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.949&rep=rep1&type=pdf. Abgerufen: 10.08.2012.

11: Garland M, Heckbert P S. Surface Simplification Using Quadric Error Metrics. In: Proceedings of ACM SIGGRAPH 1997. 209-216. 1997.

12: geoinformation.net. Laserscanning. Projektpartner: Universität Karlsruhe – Institut für Photogrammetrie und Fernerkundung. 2003. PDF-Dokument. URL: http://www.geoinformation.net/lernmodule/lm06/pdf/pdf_kap5_laserscanning.pdf. Abgerufen: 08.08.2012.

13: Kobbelt L, Campagna S, Seidel H-P. A General Framework for Mesh Decimation. In: Graphics Interface

'98 Proceedings. 43-50. 1998.
14: Koenzen U, Günther-Diringer D. Auenzustandsbericht – Flussauen in Deutschland. Bundesministerium für Umwelt, Naturschutz und Reaktorsicherheit (BMU). 2009. PDF-Dokument. URL: http://www.bfn.de/fileadmin/MDB/documents/themen/wasser/Auenzustandsbericht.pdf. Abgerufen: 07.08.2012.
15: Lenga M. Poseminar Algorithmen: Delaunay Triangulation. Universität Ulm, Institut für theoretische Informatik. 2010. PDF-Dokument. URL: http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Lehre/WS0910/ProseminarAlgo/ausarbeitung_Delaunay_Triangulationen.pdf. Abgerufen: 07.08.2012.
16: Lischinski D. Incremental Delaunay Triangulation. Cornell University. 1993. PDF-Dokument. URL: http://www.karlchenofhell.org/cppswp/lischinski.pdf. Abgerufen: 07.08.2012.
17: Lund K, Sigmon P, Schlicker S. The Occurrence of Fibonacci and Lucas Numbers in the Geometry of H(Rn). Grand Valley State University, Wake Forest University. 2005. PDF-Dokument. URL: http://www.math.wfu.edu/publications/Student/fibonaccipape1_24_05.pdf. Abgerufen: 10.08.2012.
18: Mandlburger G, Hauer C, Höfle B, Habersack H, Pfeifer N. Optimisation of LiDAR derived terrain models for river flowmodelling. In: Hydrology and Earth System Sciences. 1453-1466. 2009.
19: Maniak U. Wasserwirtschaft – Einführung in die Bewertung wasserwirtschaftlicher Vorhaben. Springer Verlag, Berlin. 21-22. 2001.
20: Pfeifer N. Oberflächenmodelle aus Laserdaten. Delft University of Technology. 2003. PDF-Dokument. URL: http://www.tudelft.nl/live/binaries/3b390ae2-9b55-4cbc-8255-da55928dd925/doc/oberfl.pdf. Abgerufen: 07.08.2012.
21: Professur für Geodäsie und Geoinformatik (GG). Airborne Laserscanning (ALS). Universität Rostock. 2008. Internetseite. URL: http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=-1616705597. Abgerufen: 07.08.2012.
22: Wasserwirtschaftsamt Ingolstadt. Dynamisierung der Donauauen zwischen Neuburg und Ingolstadt. Richard Hofmann. Keine Jahresangabe. Internetseite. URL: http://www.wwa-in.bayern.de/projekte_und_programme/donauauen/index.htm. Abgerufen: 20.8.2012.
22: Wu J, Kobbelt L. A Stream Algorithm for the Decimation of Massive Meshes. In: Proceedings of the Graphics Interface. 185-192. 2003.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

München, Datum

Dorothea Pörtge

Dorothea Pörtge
[redacted]
[redacted]