Universität Leipzig

Fakultät für Mathematik und Informatik
Institut für Informatik

Implementierung und Anwendung von Algorithmen zum Erzeugen und Manipulieren von Linienmengen

Bachelorarbeit

Leipzig, Januar 2014

vorgelegt von
Simon Vetter
Studiengang Informatik

Betreuer:
Dipl.-Inf. Stefan Koch
Dr. Jens Kasten
Jun.-Prof. Dr. Mario Hlawitschka
Universität Leipzig, Institut für Informatik, Bild- und Signalverarbeitung

Inhaltsverzeichnis

1 Einleitung

FAnToM, ein Akronym für „*F*ield *An*alysis using *To*pological *M*ethods“, ist eine Visualisierungssoftware, die 1999 von Prof. Gerik Scheuermann, dem Leiter der Abteilung für Bild und Signalverarbeitung des Instituts für Informatik an der Universität Leipzig, initiiert wurde und seitdem kontinuierlich weiterentwickelt wird. Nach zehnjähriger Entwicklungszeit wurde *FAnToM* in einem Handout zur „Refactoring Visualization from Experience“-Konferenz (ReVisE) als Visualisierungsplattform für allgemeine Strömungsvisualisierung vorgestellt und Erkenntnisse während der Entwicklung präsentiert.[10]

Derzeit wird *FAnToM* in zweiter Version neu entwickelt. Eine der wichtigsten Neuerungen ist dabei die Umstellung des User-Interface auf *Datenflussnetzwerke*, welche den Benutzern eine verbesserte Steuerung ermöglichen. Dabei können aus einfachen Filterbausteinen komplexe Algorithmen erstellt werden, indem die Daten von einer Eingabe, beispielsweise einer Datei, durch ein Netzwerk von Filtern, den *Datenalgorithmen*, geleitet, verarbeitet und anschließend an eine Ausgabe, einen *Visualisierungsalgorithmus* gelenkt werden. Die Daten werden zwischen den Datenfluss-Bausteinen in verschiedenen Datenstrukturen transportiert, beispielsweise in Feldern oder als Gitter.

Mit dieser Arbeit wurden als eine solche Datenstruktur die *Linienmengen* neu in *FAnToM* eingeführt. Sie sind eine wertvolle Datenstruktur im Bereich der Visualisierungssoftware, die sehr vielseitig eingesetzt werden kann. So eignen sie sich zum Beispiel als Eingabe für verschiedene Algorithmen und für die Visualisierung von Linien. In dieser Arbeit werden grundlegende Eigenschaften von Linienmengen dargestellt und einige Algorithmen zum Erzeugen und Manipulieren derselben vorgestellt, die auch als *Datenalgorithmen* in die *FAnToM*-Software implementiert wurden.

Dabei wurde großer Wert auf die Effizienz der Algorithmen gelegt, da sich *FAnToM* unter anderem auch dadurch von anderen Visualisierungssystemen abhebt, dass es auch große

Mengen an Daten von unstrukturierten Gittern schnell und effizient verarbeiten kann (vgl. [10]).

Im Folgenden sollen zunächst die Grundlagen geschaffen werden, indem elementare Begriffe und Konzepte dieser Arbeit definiert werden. Danach sollen drei Kategorien von *Datenalgorithmen* für *Linienmengen* vorgestellt werden. In jeder dieser Kategorien werden zwei Algorithmen detailliert erörtert und ein Augenmerk auf die effiziente Implementierung für *FAnToM* geworfen. Anschließend folgt eine Diskussion über die erzielten Ergebnisse.

In dieser Arbeit werden drei Beispieldatensätze verwendet, auf welche die Algorithmen angewendet werden. Der erste Datensatz (Abbildung 1.1) beinhaltet ein Vektorfeld, welches eine Luftströmung um einen Nurflüglerkonfiguration simuliert. Er entstand im Rahmen der Forschung von Markus Rütten am Deutschen Zentrum für Luft- und Raumfahrt (DLR) in Göttingen. Während sich die Luft außerhalb des Nurflüglers mit hoher Geschwindigkeit linear fortbewegt, entstehen direkt hinter dem Flügler Wirbel.

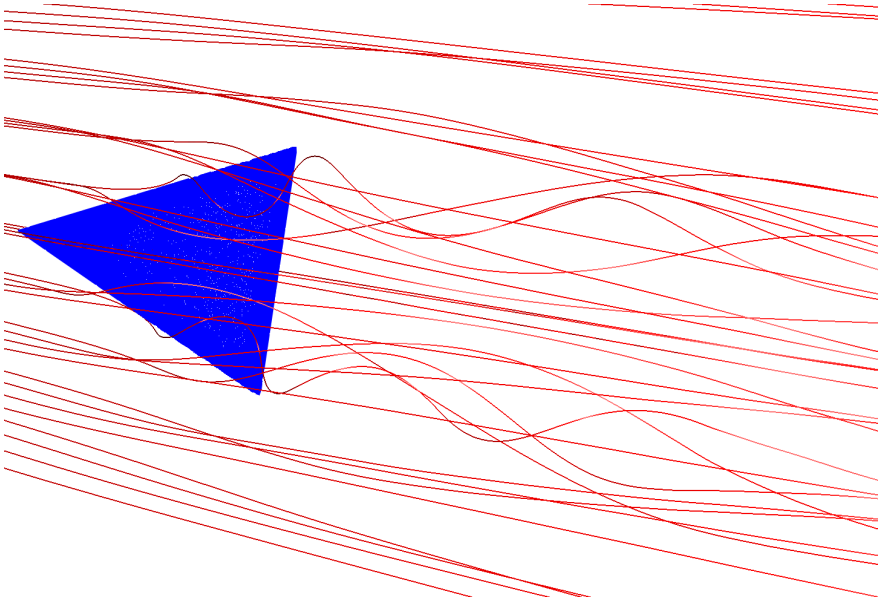


Abbildung 1.1: Stromlinien um einen Delta Wing

Der zweite Datensatz (Abbildung 1.2) beinhaltet das Simulationsergebnis einer Gas- und Luftströmung in einer Gasbrennkammer, wie sie in Heizungen von Einfamilienhäusern verwendet wird. Durch eine Öffnung in der Seite strömt eine große Menge Luft in die Kammer, während von unten und oben Gas zugeführt wird. Für eine effiziente Verbren-

nung müssen Gas und Luft gut durchmischt werden, resultierend in einer turbulenten Strömung.

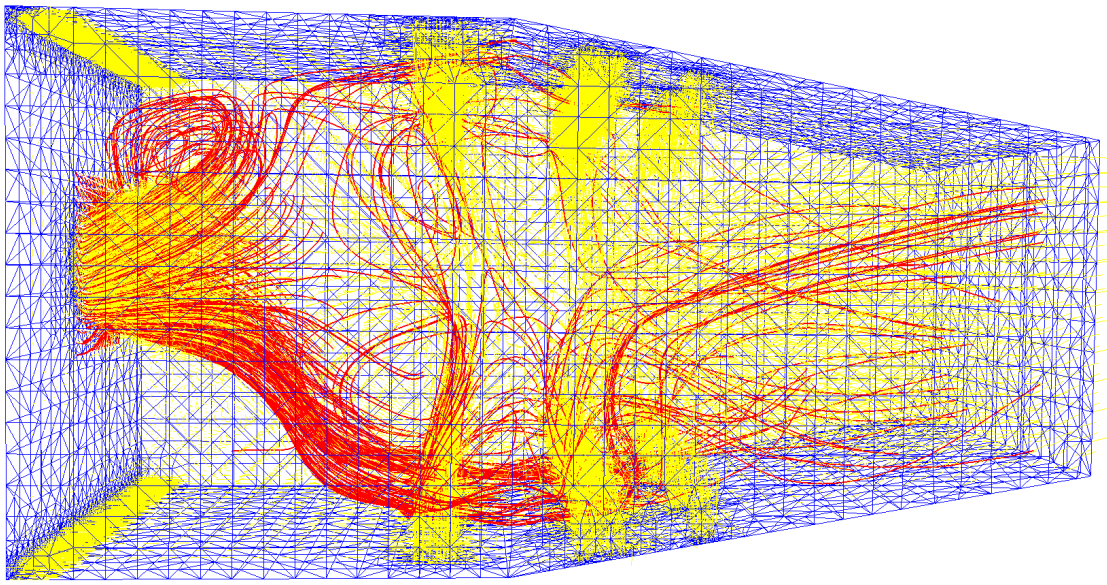


Abbildung 1.2: Rote Stromlinen der einströmenden Luft und gelbe Vektoren, welche den Gaseintritt darstellen

Der dritte Datensatz (Abbildung 1.3) stellt eine Kugel dar, die in der Mitte mit einem Loch versehen wurde. Luft strömt von der Seite teils durch das Loch, teils um die Kugel herum. Hinter dem Loch entstehen drei Hauptwirbel, welche entgegengesetzt rotieren. Die Wirbelkernlinien sind in Abbildung 1.4 sichtbar.

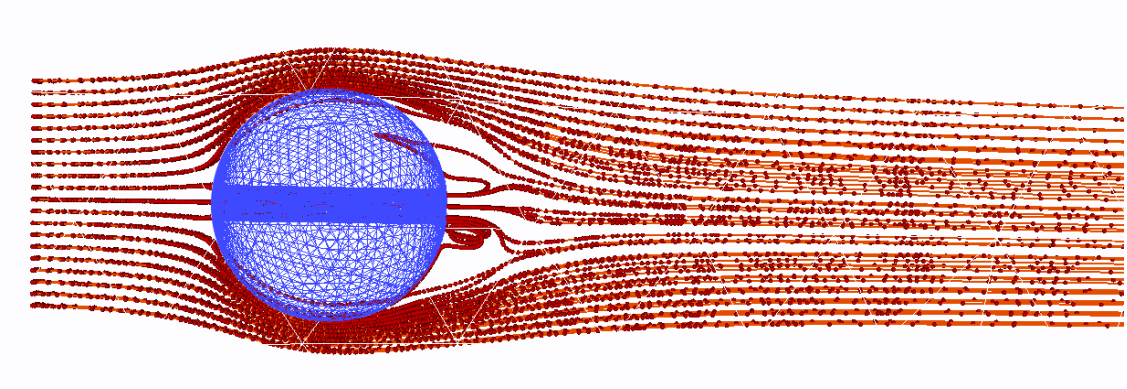


Abbildung 1.3: Rote Stromlinien um eine Lochkugel

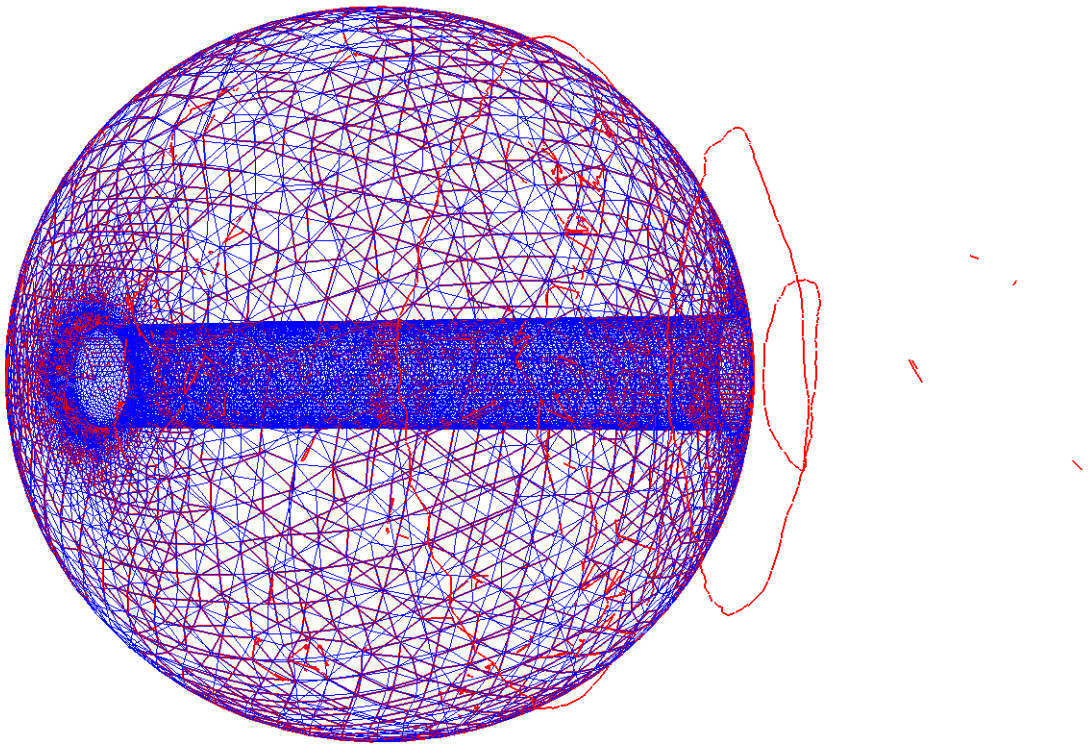


Abbildung 1.4: Rote Wirbelkernlinien um eine Lochkugel

2 Grundlagen

2.1 Begriffsklärung

Bevor mit dem inhaltlichen Teil dieser Arbeit begonnen werden kann, müssen zunächst die verwendeten Begriffe geklärt und die benötigten Grundlagen geschaffen werden.

Der Begriff *Linienmenge* bezeichnet eine Menge von *Linien* im euklidischen Raum. Der Begriff *Linie* wurde dabei vom englischen Begriff *line* in *FAnToM* übersetzt. Eine alternative Bezeichnung wäre „offener Polygonzug“. Die *Linie* einer *Linienmenge* hat genau zwei Endpunkte und ist nicht unterbrochen, vergleiche Abbildung 2.1.

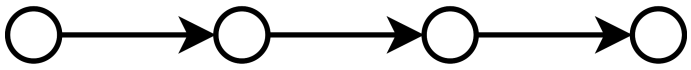


Abbildung 2.1: Eine Linie

Obwohl eine Linie keine spezifizierte Richtung hat, besitzt sie in *FAnToM* aus technischen Gründen einen Anfang und ein Ende. Abbildung 2.2 zeigt, wie die Linien innerhalb von *FAnToM* gespeichert werden. Diese Speicherung ist später in Kapitel 3.4.1 von Bedeutung. Eine aus dem Bereich der Algorithmen und Datenstrukturen bekannte, äquivalente Datenstruktur ist die einfach verkettete Liste von Punkten (vgl. [8]). In *FAnToM* sollen diese *Linienmengen* unter Zuhilfenahme der später vorgestellten *Algorithmen* manipuliert werden.

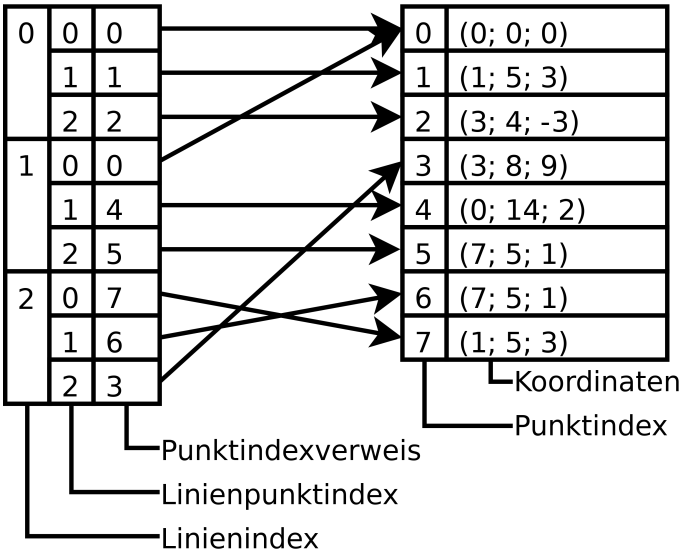


Abbildung 2.2: Interne Speicherung von Linienmengen in *FAnToM*

2.2 Algorithmische Grundlagen

2.2.1 Linienmenge

Punkt. Die Punkte einer Linie repräsentieren ein Objekt ohne Ausdehnung mit bestimmten Koordinaten im Raum oder in der Ebene. Ein Punkt lässt sich durch seinen Ortsvektor darstellen. In den folgenden Algorithmen werden Punkte mit Kleinbuchstaben bezeichnet, beispielsweise p. Der Ortsvektor dieses Punktes wird als $\vec{p}$ deklariert. Die Indizes p_x, p_y und p_z bezeichnen die einzelnen Koordinaten des Punktes.

Linie. In dieser Arbeit besteht eine Linie aus mehreren, der Reihe nach verbundenen, Punkten. Jeder dieser Punkte wird durch seinen Ortsvektor beschrieben, der Vektor zwischen zwei verbundenen Punkten p und q kann also aus der Differenz zwischen den Ortsvektoren $\vec{p}$ und $\vec{q}$ berechnet werden und wird in dieser Arbeit als $\overrightarrow{pq}$ dargestellt. Eine Linie wird als Menge von Punkten angesehen und durch einen Großbuchstaben repräsentiert, in dieser Arbeit meist L. Die einzelnen Punkte der Linie werden durch Indizes mit der Notation L_n bezeichnet. Dabei beginnt die Nummerierung bei 0, sodass L_0 das erste und L_{n-1} das letzte Element einer Linie mit n Elementen ist. Möglich sind neben den Indizes auch Schlüsselwörter wie *Erstes* und *Letztes*, um jeweils das erste beziehungsweise letzte Element dieser Linie zu wählen, also L_{erstes} und $L_{letztes}$. Die Anzahl der Punkte in einer Linie lässt sich mit L_{laenge} ermitteln.

Linienmenge. Eine Linienmenge ist eine Menge von Linien. Auch sie wird mit einem Großbuchstaben bezeichnet, meist M. Auch die Linien einer Linienmenge lassen sich durch Indizes anwählen (M_n).

2.2.2 Bézierkurven

Bézierkurven. Bézierkurven sind Kurven, deren Form durch wenige Punkte beschrieben wird. Sie finden zum Beispiel Anwendung in Vektorgrafiken und beim Computer Aided Design (CAD). Eine Béziekurve n-ten Grades besteht aus $n+1$ Kontrollpunkten. Sie ist durch die Formel

$$C(t) = \sum_{i=0}^{n} B_{i,n}(t) p_i \tag{2.1}$$

definiert, wobei p_i die Kontrollpunkte sind und $t \in [0,1]$. $B_{i,n}(t)$ bezeichnet das i-te Bernsteinpolynom n-ten Grades, welches wie folgt definiert ist:

$$B_{i,n}(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i} \tag{2.2}$$

Das Bernsteinpolynom kann auch rekursiv als

$$B_{i,n}(t) = (1-t) \cdot B_{i,n-1}(t) + t \cdot B_{i-1,n-1}(t) \tag{2.3}$$

dargestellt werden. Für $i < 0$ oder $i > n$ gilt $B_{i,n} = 0$, außerdem ist $B_{0,0} = 1$ [4]. Die Bézierkurve verläuft durch ihre Endpunkte und innerhalb der konvexen Hülle des beschreibenden Polygons aus den Kontrollpunkten.

In dieser Arbeit werden in Kapitel 3.5.2 quadratische Bézierkurven mit $n = 2$ verwendet.

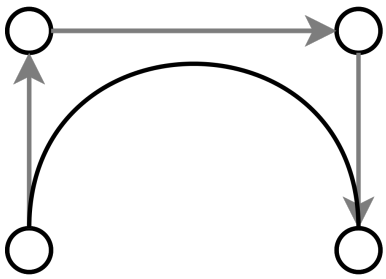


Abbildung 2.3: Kubische Bézierkurve

De-Casteljau-Algorithmus. Der De-Casteljau-Algorithmus ist eine Möglichkeit, eine Bézierkurve numerisch zu berechnen. Dabei wird die rekursive Form des Bernsteinpolynoms (Gleichung 2.3) benutzt.[4]

Sei L eine Linie mit n Punkten L_0 bis L_{n-1}. Dann ist die Gleichung der Bézierkurve nach Gleichung 2.1:

$$C(t) = \sum_{i=0}^{n} L_i B_{i,n}(t) \tag{2.4}$$

Diese Gleichung kann nun rekursiv gelöst werden: Für $i = 0$ gilt $B_{i,0} = 1$, für alle übrigen Werte für i gilt $B_{i,0} = 0$. Anschließend werden solange die berechneten Werte für $B_{i,k}$ in $B_{i,k+1}$ eingesetzt, bis $k = n$ erreicht ist.

Es entsteht eine Gleichung $C(t)$ mit $t \in [0, 1]$, die die Bézierkurve beschreibt. [5]

2.2.3 Verwandte Algorithmen

Sweep-Plane-Algorithmus. Ein Sweep-Line- oder Sweep-Plane-Algorithmus ist ein Verfahren in der Algorithmischen Geometrie, welches Anwendung in verschiedenen Algorithmen findet. Dabei wird eine gedachte Linie oder Ebene entlang einer Koordinatenachse durch den gesamten Datensatz geschoben und beim Treffen auf Datensatzpunkte eine entsprechende Aktion ausgeführt.

Beispielsweise kann ein Schnittkantentest von Liniensegmenten durch einen Sweep-Line-Algorithmus realisiert werden, wie er von Bentley und Ottmann 1979 entwickelt wurde [1]. Trifft die Sweep-Linie auf ein Ende eines Liniensegmentes, so wird dieses in einen binären Suchbaum eingefügt oder aus diesem wieder entfernt. Es müssen nur die Liniensegmente auf Schnitte getestet werden, die im binären Suchbaum nebeneinander liegen. Trifft die Sweep-Line auf einen solchen Schnittpunkt, so müssen die schneidenden Linien im Suchbaum getauscht werden.

Ein Schnitttest alle Liniensegmente hätte einen Aufwand von $O(n^2)$, was in den meisten Fällen zu hoch ist. Der Bentley-Ottmann-Algorithmus benötigt nur $O(n \log n)$. Das Einfügen in und Entfernen aus dem binären Suchbaum hat einen Aufwand von $O(\log n)$, die Sweep-Line läuft linear mit $O(n)$ durch den Datensatz. Einzig das Sortieren des Datensatzes mit einem Aufwand von $(O)(n \log n)$ gibt dem Algorithmus diesen Aufwand.

3 Algorithmen

3.1 Einleitung

Nach H. Cormen ist ein *Algorithmus* „eine Folge von Rechenschritten, die die Eingabe in die Ausgabe umwandeln“[2]. In dieser Arbeit besteht die Eingabe immer aus einer Linienmenge und, wenn nötig, zusätzlich aus Parametern für den Algorithmus. Die Ausgabe enthält immer die manipulierte Linienmenge. Ziel ist es, effiziente Algorithmen zur Linienmengenmanipulation zu entwickeln. Darunter zählt unter anderem das Ausfiltern derjenigen Linien, die eine bestimmte Länge über- oder unterschreiten (Kapitel 3.3.1), das Trennen von Linien an spitzen Winkeln (Kapitel 3.3.2), das Zusammenfügen mehrerer Linien an gemeinsamen Punkten (Kapitel 3.4.2) sowie die Glättung der Linien zur besseren Darstellung (Kapitel 3.5.2). Alle vorgestellten Algorithmen wurden in die *FAnToM* mit der Programmiersprache C++ implementiert. Der Quellcode ist im Anhang angegeben. Die Nutzung von C++ als Programmiersprache bietet spezielle Möglichkeiten, die Effizienz der Algorithmen weiter zu steigern, welche sich nicht im Pseudocode darstellen lassen.

3.2 Generatoren

Linienmengen entstehen als Datenstruktur in der Ausgabe bestimmter Algorithmen und können als Eingabe weiterer Visualisierungsalgorithmen dienen. Erzeugt werden können Linienmenge beispielsweise durch Streamline- oder Pathline-Integration eines Vektorfeldes oder durch Anwenden des Sujudi-Haimes-Algorithmus [9], welcher Wirbelkernlinien in einem Geschwindigkeitsfeld berechnet.

Sie wurden bereits in die *FAnToM*-Software implementiert und für die Erstellung der Bilder dieser Arbeit verwendet. Die generierten Linienmengen können nun als Eingabe für die hier vorgestellten *Datenalgorithmen* dienen.

3.3 Filter

3.3.1 Längenfilter

Der Längenfilter ist einer der einfachsten Algorithmen dieser Arbeit. Als Eingabe erhält er neben der Linienmenge einen Schwellwert für die Länge der Linie. Die Ausgabe besteht aus zwei Linienmengen. Der Algorithmus hat das Ziel, die Linien der Eingabe-Linienmenge je nach Länge einer der beiden Ausgabe-Linienmengen zuzuordnen.

Bei gleicher Integrationszeit sind die Stromlinien mit einer höheren Geschwindigkeit länger als langsamere Stromlinien. Der Längenfilter könnte so schnelle Strömungen von langsamen Strömungen trennen. Beim Sujudi-Haimes-Algorithmus hingegen können kleine Artefakte entstehen. Diese lassen sich durch den Längenfilter von den Wirbelkernlinien trennen.

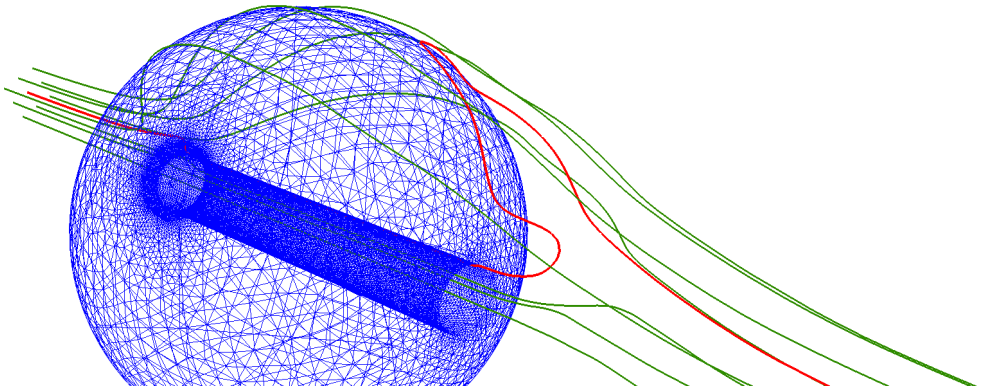


Abbildung 3.1: Nach ihrer Länge gefilterte Linien im Lochkugeldatensatz. Kürzere Linien sind grün dargestellt, längere rot

Die mathematische Grundlage für die Berechnung der Länge einer Linie ist die Summe der Länge der Liniensegmente, welche sich durch die euklidische Norm ermitteln lässt.

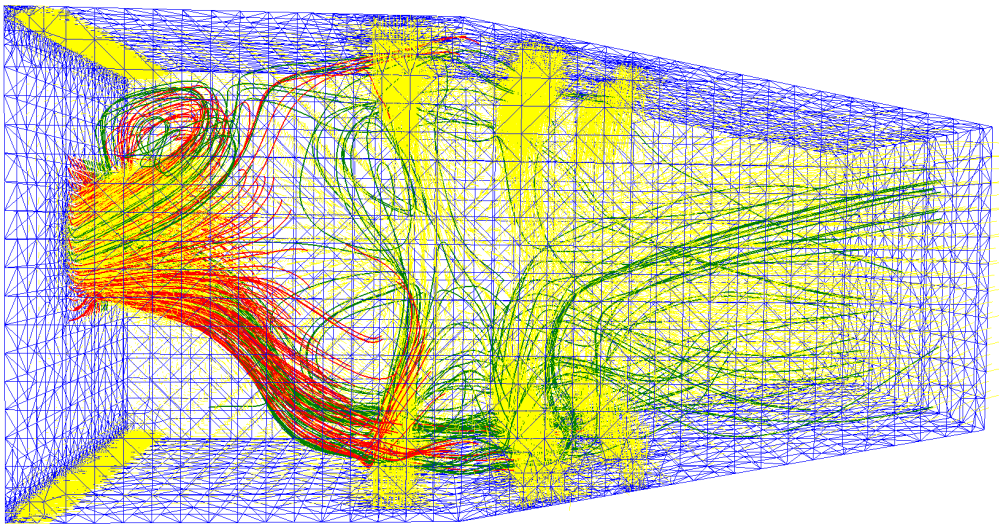


Abbildung 3.2: Lange und kurze Linien des Gasbrennkammer-Datensatzen verschieden eingefärbt

Gegeben sei ein Liniensegment einer Linie mit den beiden Endpunkten a und b. Weiterhin sei $\vec{v} = \overrightarrow{ab}$ der Vektor, der dieses Liniensegment repräsentiert:

$$\vec{v} = \begin{pmatrix} b_x - a_x \\ b_y - a_y \\ b_z - a_z \end{pmatrix} \tag{3.1}$$

Dann ist die Länge l des Liniensegmentes gleich dem Betrag $|\vec{v}|$ dieses Vektors.

$$l = |\vec{v}| = \sqrt{(v_x)^2 + (v_y)^2 + (v_z)^2} \tag{3.2}$$

Die Länge l_{line} der gesamten Linie wird durch die Summe der Länge aller Liniensegmente dieser Linie ermittelt. Für eine Linie mit n Liniensegmenten gilt also:

$$l_{line} = \sum_{i=1}^{n} l_n = \sum_{i=1}^{n} |\vec{v_n}| \tag{3.3}$$

Da großer Wert auf Effizienz gelegt wird, wird ein genauer Blick auf den Pseudocode in Algorithmus 1 geworfen. In Zeile 3 lautet die Anweisung:

$$v \leftarrow L_{i+1} - L_i \tag{3.4}$$

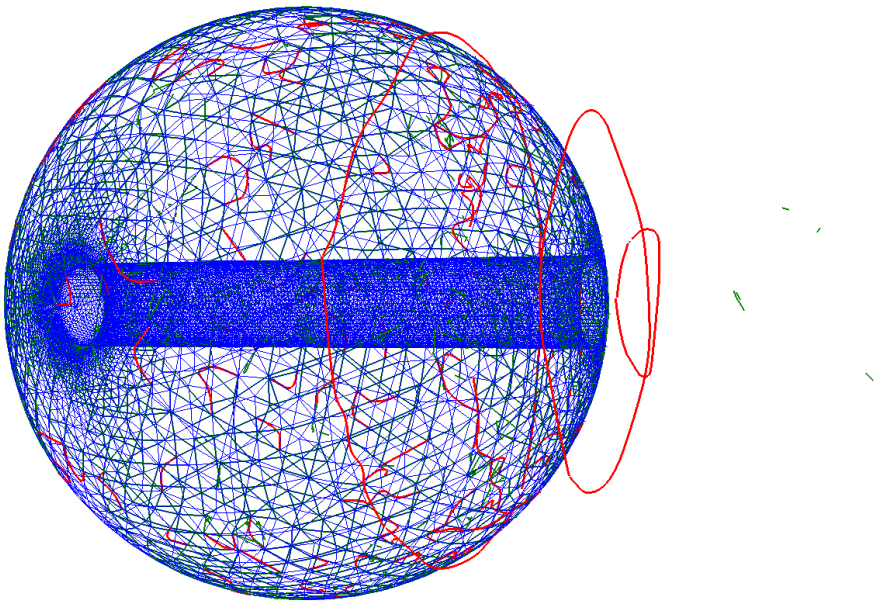


Abbildung 3.3: Lange rote Wirbelkernlinien und kurze grüne Artefakte nach Anwendung des Sujudi-Haimes-Algorithmus auf den Lochkugeldatensatz

Da i als Laufvariable inkrementiert wird, ergibt sich für den darauf folgenden Schleifendurchlauf folgende Formel:

$$v \leftarrow L_{i+2} - L_{i+1} \tag{3.5}$$

Um die Variable L_{i+1} nicht doppelt in den Speicher laden zu müssen, wurde die C++-Implementierung in Quellcode 1 so angepasst, dass der Wert dieser Variable im Speicher gehalten wird. Dabei wurde eine Variable `lastUsedPoint` eingeführt. Aufga-

Algorithmus 1 : Länge einer Linie

Eingabe : Linie L
Ergebnis : Länge l der Linie

1 $l \leftarrow 0$
2 **Für** $i \leftarrow 0$ **bis** $L_{laenge} - 1$
3 $\quad v \leftarrow L_{i+1} - L_i$
4 $\quad l \leftarrow l + \sqrt{v_x + v_y + v_z}$
5 **Ende**
6 **gib zurück** l

be von `lastUsedPoint` ist es, den Wert des Minuenden L_{i+1} aus Formel 3.4 zu speichern und ihn im darauf folgenden Schleifendurchlauf 3.5 als Subtrahend L_{i+1} einzusetzen.

In diesem Algorithmus wird jeder Punkt genau einmal gelesen und der Abstand zum vorherigen Punkt berechnet, daher ist der zeitliche Aufwand der Funktion $O(n)$, wobei n die Anzahl der Punkte der Linie ist.

Nun kann mithilfe der Funktion aus Algorithmus 1 zur Berechnung der Linienlänge ein Algorithmus geschrieben werden, der alle Linien einer Eingabe-Linienmenge je nach Länge in verschiedene Ausgabe-Linienmengen sortiert.

Algorithmus 2 : Längenfilter

Eingabe : Linienmenge M, Schwellwert t
Ausgabe : Linienmenge A, Linienmenge B

```
1 Für jedes L ∈ M
2 |  l ← getLineLength(L)
3 |  Wenn l < t dann
4 |  |  Füge M zu B hinzu
5 |  sonst
6 |  |  Füge M zu A hinzu
7 |  Ende
8 Ende
9 gib zurück A, B
```

Um in der Implementierung in C++ die Laufzeit möglichst effizient zu halten, wurde sofern möglich Speicher in passender Größe reserviert, Variablen außerhalb von Schleifen deklariert und Werte zwischengespeichert, siehe Quelltext 2.

Für jede Linie kann die Länge linear berechnet werden und über alle Linien wird linear iteriert. Der zeitliche Aufwand ist daher $O(n)$ mit n als Anzahl der Linienpunkte.

Durch diesen Algorithmus können nun Linien bestimmter Länge gefiltert werden, so wird beispielsweise in Abbildung 3.1 die längste Linie rot dargestellt, während die restlichen Linien grün eingefärbt sind.

3.3.2 Winkelfilter

Die Winkel der Liniensegmente untereinander sollten nicht außer Acht gelassen werden. Gerade bei der Anwendung des Algorithmus von Sujudi und Haimes [9] können viele Artefakte bei der Berechnung der Wirbelkernlinien entstehen, die durch einen Winkelfilter entfernt werden können. Daher ist es sinnvoll, einen Filter zu entwickeln, der jede Linie einer Linienmenge durchläuft und an den Punkten auftrennt, an denen der eingeschlossene Winkel zwischen zwei Segmenten einen gegebenen Schwellwert übersteigt.

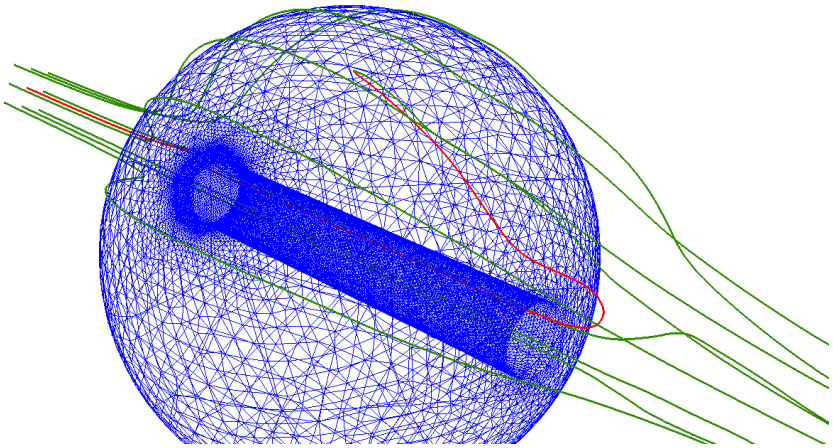


Abbildung 3.4: Eine am spitzen Winkel getrennte Linie im Lochkugeldatensatz

Um den Winkel zwischen zwei Liniensegmenten bestimmen zu können, müssen zunächst die mathematischen Grundlagen erläutert werden. Wichtigste Grundlage für die Winkelberechnung ist die Formel des Skalarproduktes:

$$\vec{pq} \cdot \vec{qr} = |\vec{pq}| \cdot |\vec{qr}| \cdot \cos \sphericalangle(\vec{pq}, \vec{qr}) = (\vec{pq})_x(\vec{qr})_x + (\vec{pq})_y(\vec{qr})_y + (\vec{pq})_z(\vec{qr})_z \tag{3.6}$$

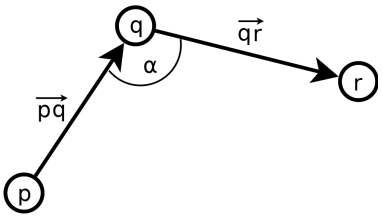


Abbildung 3.5: Der Winkel α zwischen $\vec{pq}$ und $\vec{qr}$

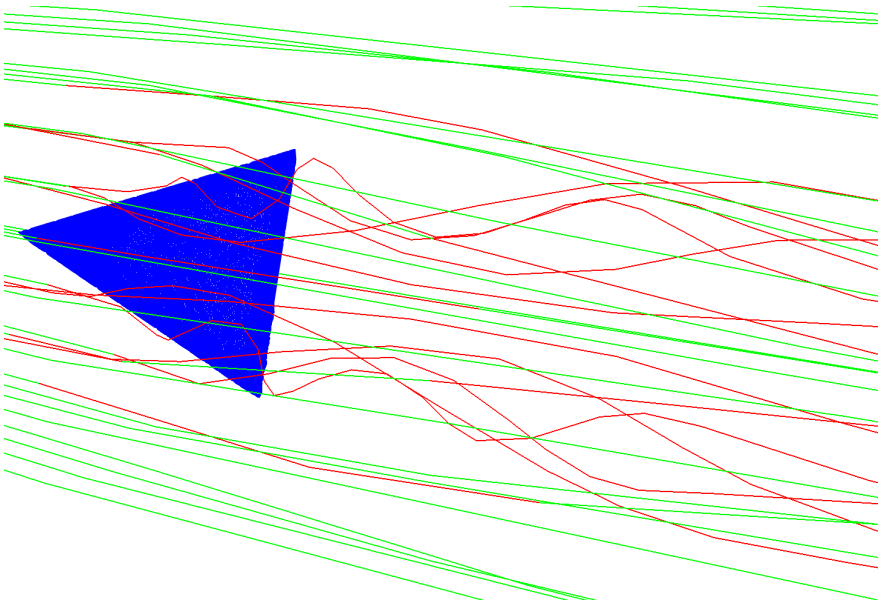


Abbildung 3.6: Linien des Dreiecks-Datensatzes an spitzen Winkeln aufgetrennt und rot eingefärbt

Die Vektoren $\overrightarrow{pq}$ und $\overrightarrow{qr}$ sind die angrenzenden Liniensegmente des Punktes, an dessen Stelle der Winkel berechnet werden soll. Sie sind in Skizze 3.5 dargestellt.

Werden nun statt den Vektoren $\overrightarrow{pq}$ und $\overrightarrow{qr}$ die normalisierten Vektoren $\overrightarrow{pq}'$ und $\overrightarrow{qr}'$, mit $|\overrightarrow{pq}'| = 1$ und $|\overrightarrow{qr}'| = 1$, verwendet, so lässt sich die Gleichung 3.6 folgendermaßen vereinfachen:

$$\overrightarrow{pq}' \cdot \overrightarrow{qr}' = \cos \sphericalangle(\overrightarrow{pq}', \overrightarrow{qr}') \tag{3.7}$$

Auf Grundlage dieser Gleichung lässt sich nun der Algorithmus 3 formulieren, der die Linien einer Linienmenge ab einem bestimmen Winkel auftrennt.

Eine wichtige Optimierung ist die Berechnung des Kosinus in Zeile 1 des Algorithmus 3 vor der Schleife. So wird, statt in jedem Schleifendurchlauf den Winkel aus dem Skalarprodukt zu berechnen und anschließend mit dem Schwellwertwinkel zu vergleichen, der Schwellwertwinkel so umgerechnet, dass er direkt mit dem Skalarprodukt verglichen werden kann.

Grundlegend werden im Algorithmus die Punkte einer Linie gelesen, der Winkel an dieser Stelle gemessen und anschließend in einer neue Linie in die Ausgabelinienmenge hinzugefügt. Sollte an einem Punkt der Winkel den Schwellwert überschreiten, so wird die Linie an diesem Punkt beendet, der Ausgabelinienmenge hinzugefügt und eine

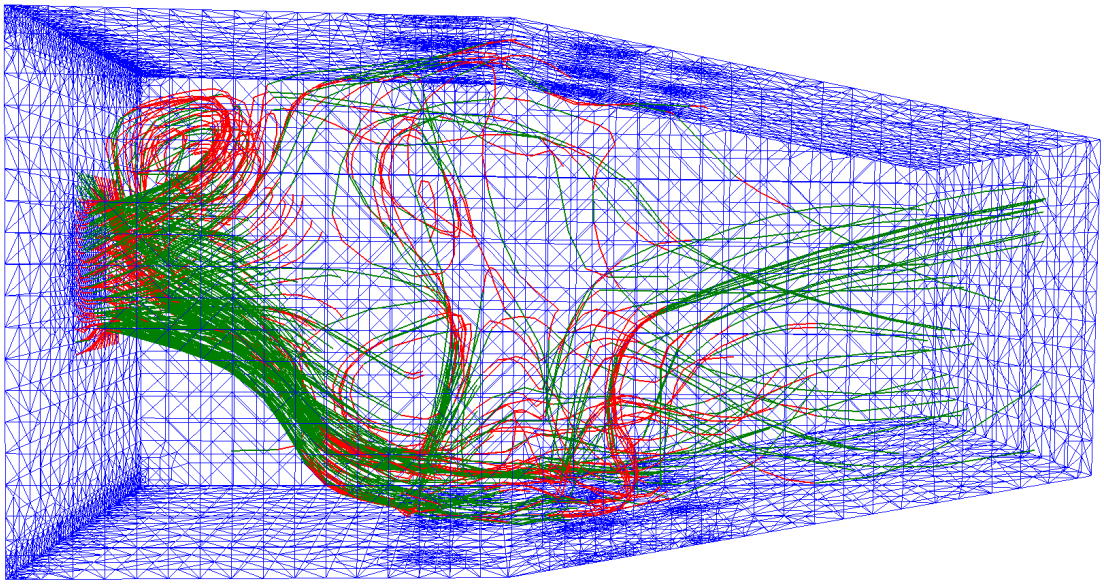


Abbildung 3.7: Segmente mit dicht aufeinander folgenden spitzen Winkeln im Gasbrennkammerdatensatz rot eingefärbt

neue Linie an diesem Punkt begonnen. Durch diese Linearität ist der zeitliche Aufwand $O(n)$.

In den Abbildungen 3.4, 3.6 und 3.7 werden rote Linien dargestellt, welche an einem spitzen Winkel aufgetrennt wurden.

3.4 Trennen und Verbinden

3.4.1 Doppelte Punkte entfernen

In einer Linienmenge ist es möglich, dass zwei oder mehr Linien Punkte mit den gleichen Koordinaten besitzen, beispielsweise zwei sich schneidende Linien. Für einige Analysen, unter anderem das Verschmelzen von Linien in Kapitel 3.4.2, ist es notwendig, dass Punkte mit den gleichen Koordinaten durch das selbe Objekt repräsentiert werden. Möglich wird dies durch die Technik zur Speicherung der Linienmengen in *FAnToM*. In jeder Linienmenge befinden sich zwei Arrays.

Wie in Abbildung 3.8 gezeigt, beinhaltet das Array `mPoints` alle Punkte der gesamten Linienmenge mit ihren Koordinaten. Dabei lässt sich jeder Punkt über einen Index im

Algorithmus 3 : Winkelfilter

Eingabe : Linienmenge M, Schwellwert t
Ausgabe : Linienmenge M

```
1  t ← cos(t ÷ 360 · π)
2  Für jedes L ∈ M
3  |  Für i ← 2 bis L_laenge
4  |  |  a ← normalisiert(L_{i-2} − L_{i-1}) · normalisiert(L_{i-1} − L_i)
5  |  |  Wenn a < t dann
6  |  |  |  Trenne Linie an Punkt L_{i-1}
7  |  |  Ende
8  |  Ende
9  Ende
10 gib zurück M
```

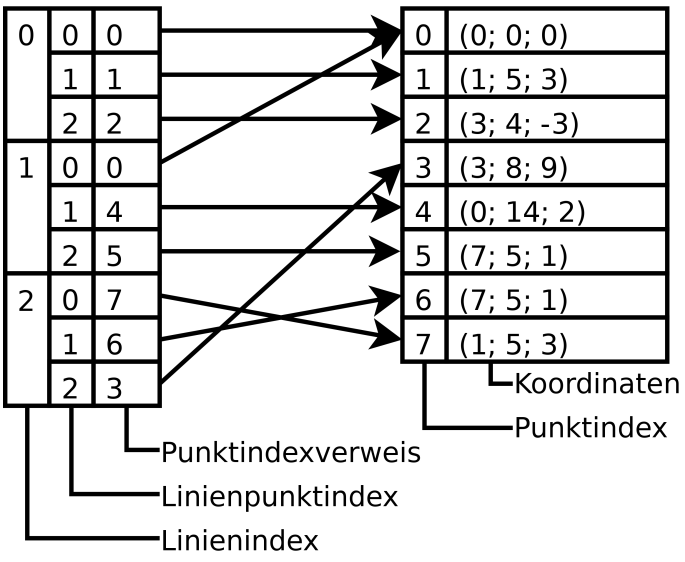


Abbildung 3.8: Das Array `mLines` (links) und das Array `mPoints` (rechts)

Array adressieren. In der Skizze sind die Koordinaten als Ganzzahlentupel dargestellt, in der Software werden Fließkommazahlen verwendet.

Im zweiten Array `mLines` befinden wiederum Arrays. In diesen Arrays sind Indizes von Punkten aus dem Array `mPoints` gespeichert. Sie repräsentieren die Linien der Linienmenge. Dies ist auf der linken Seite der Abbildung 3.8 angedeutet. Der Vorteil ist, dass mehrere Linien eine Referenz auf den selben Punkt haben können, wie in der Abbildung zum Beispiel die beiden jeweils ersten Punkte der Linien 0 und 1 im Linienarray auf Punkt 0 im Punktarray verweisen.

Der Algorithmus „Doppelte Punkte entfernen“ hat die Aufgabe, Punkte in `mPoints` mit den gleichen Koordinaten zu einem Punkt zusammenzufassen und die Indexverweise in `mLines` darauf anzupassen. In der Beispielabbildung 3.8 könnten damit die Punkte 5 und 6 sowie die Punkte 1 und 7 zusammengefasst werden, da diese jeweils die gleichen Koordinaten besitzen.

Im Gegensatz zu den Algorithmen 2 und 3 kann der Algorithmus zum Entfernen doppelter Punkte nicht in linearer Zeit ablaufen, da die Punkte untereinander verglichen werden müssen. Ein erster Ansatz, jeden Punkt mit allen anderen zu vergleichen, sollte schnell verworfen werden, da der zeitliche Aufwand mit $O(n^2)$ deutlich zu groß wäre.

Eine möglicher besserer Ansatz wäre das Sortieren des `mPoints`-Arrays nach Koordinaten und anschließend ein lineares Ablaufen des sortierten Arrays. Das Vergleichen hätte einen Aufwand in $O(n)$, das Sortieren würde bei einer geeigneten Implementierung, beispielsweise durch Mergesort, im Worst-Case einen zeitlichen Aufwand von $O(n \log n)$ benötigen. Jedoch bestünde ein Problem beim Korrigieren der Indexverweise in `mLines`, da sich bei dieser Methode keine Assoziationen zwischen den Indizes der gelöschten, doppelten Punkte und den an ihre Stelle tretenden Punkte ergeben. Es ließe sich also schwer ermitteln, auf welchen Index der Verweis auf einen gelöschten Punkt in `mLines` korrigiert werden müsste.

Aus diesem Grund wird in dieser Arbeit ein anderer Ansatz verfolgt, in dem bei gleichem zeitlichen Aufwand auch die Zuordnung der gelöschten Punkte auf die entsprechenden einzelnen Punkte problemlos möglich ist.

Der Hauptvorteil des verwendeten Algorithmus-Ansatzes ist die Verwendung einer C++-Map. Die C++-Map ist in der Standard-Bibliothek als `std::map` definiert. Sie beschreibt

einen assoziativen Container, der einen Schlüssel mit einem Wert verbindet. Ein Element lässt sich also nicht über einen Index wählen, sondern über einen eindeutigen Schlüssel. Dies hat den Nachteil, dass bei einem Zugriff auf ein Element einer Map der Wert des Schlüssels erst mit einem Aufwand von $O(\log n)$ gesucht werden muss (vgl. [7]). Der Vorteil jedoch ist, dass jedes Koordinaten-Tripel einem bestimmten Punkt-Index zugeordnet werden kann. Durch die Eindeutigkeit des Schlüssels in einer Map kann somit nur Punkte mit einzigartigen Koordinaten existieren.

Ein entsprechender Algorithmus unter Zuhilfenahme einer Map kann wie in Algorithmus 4 dargestellt aussehen. Die `Index`-Funktion liefert einen Zugriff auf den Indexverweis in `mLines`.

Algorithmus 4 : Doppelte Punkte entfernen

Eingabe : Linienmenge M
Ausgabe : Linienmenge M'

```
1  Initialisiere m als Map(Point3 → Index)
2  H ← ∅
3  Für jedes L ∈ M
4  |  Für i ← 0 bis L_laenge
5  |  |  Wenn L_i ∈ m dann
6  |  |  |  Index (L_i) ← m(L_i)
7  |  |  sonst
8  |  |  |  Füge L_i zu H hinzu und merke dessen Index als j
9  |  |  |  Füge (L_i → j) zu m hinzu
10 |  |  Ende
11 |  Ende
12 Ende
13 M'.mPoints ← H;
14 gib zurück M'
```

Ziel des Algorithmus 4 ist es, das Array `mPoints` von Punkten mit gleichen Koordinaten zu befreien. Dafür wird in Zeile 2 eine Hilfsmenge H eingeführt, die nur die einzelnen Punkte beinhalten soll. Anschließend wird jeder Punkt jeder Linie durchlaufen und analysiert. Dabei wird in Zeile 5 geprüft, ob sich bereits ein Punkt mit den Koordinaten in der Map m befindet. Sollte dies der Fall sein, so wird der Indexverweis dieses Punktes der Linie auf den in H vorhandenen Punkt angepasst. Anderenfalls wird der Punkt mit den Koordinaten zu H hinzugefügt und seine Position in H zwischengespeichert. Daraufhin wird in der Map m eine neue Assoziation zwischen den Koordinaten des Punktes und der Position

dieses Punktes in H erzeugt. Am Ende des Algorithmus wird das bestehende Array `mLines` durch die Hilfsmenge H ersetzt.

Die Implementierung im Quelltext 4 auf Seite 43 sich sehr nah an die Vorlage des Algorithmus 4. Da allerdings die Eingabelinienmenge `lineset` im *FAnToM*-Algorithmus konstant ist, müssen Punkte und Linien in neue Arrays in die Linienmenge `filteredLineSet` übertragen werden.

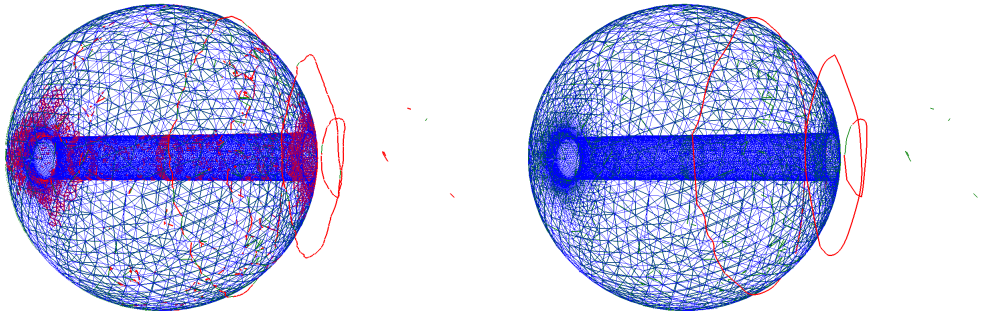
Durch die Verwendung von Standardbibliotheksfunktionen verbleibt nur die Deklaration der Variablen außerhalb der Schleife als Geschwindigkeitsoptimierung. Der zeitliche Aufwand von $O(n \log n)$ entsteht hauptsächlich durch die Suche des Schlüssel in der Map mit $O(\log n)$ (vgl. [7]) und der linearen Iteration über die Punkte.

3.4.2 Linien verschmelzen

Im folgenden Kapitel soll ein Algorithmus vorgestellt werden, der Linien aus Linienmengen verschmilzt und trennt. Zwei Linien, deren Enden sich berühren, sollen verschmolzen werden, an Schnitt- und Kreuzungspunkten sollen die Linien getrennt werden. Dabei sollen möglichst lange Pfade entstehen. Beispielsweise nach Anwendung des Sujudi-Haimes-Algorithmus können so Wirbelkernlinien korrekt verbunden und Artefakte ausgefiltert werden.

Da dieser Algorithmus wesentlich aufwendiger als die bisherigen ist, wird er in verschiedenen Abschnitten erklärt. Im ersten Abschnitt wird jedem Punkt ein Status zugewiesen, um später schnell die entsprechende durchzuführende Aktion wählen zu können. Der Abschnitt danach findet die Linien, die zusammengeschmolzen werden sollen. Im dritten Abschnitt wird der Anfang einer zu bildenden neuen Linie gesucht und die entsprechenden Linien zusammengefügt, außerdem werden die Trennpunkte beachtet. Ein vierter Abschnitt ist notwendig, um eventuelle Sonderfälle zu behandeln.

Damit der Algorithmus korrekt funktioniert, sollten vorher die doppelten Punkte unter Zuhilfenahme des Algorithmus aus Kapitel 3.4.1 verschmolzen worden sein. Als Eingabedaten erhält der Algorithmus die Linienmenge `lineset` mit den zu verbindenden Linien und eine Anweisung `split`, ob auch Linien getrennt werden sollen.



(a) unverschmolzen (b) verschmolzen

Abbildung 3.9: Wirbelkernlinien im Lochkugeldatensatz nach Länge gefiltert mit und ohne Verschmelzung

Status	Beschreibung
ungesehen	Der Punkt wurde noch nicht vom Algorithmus betrachtet
Endpunkt	Der Punkt ist der Endpunkt einer Linie
Innenpunkt	Der Punkt ist Innenpunkt einer Linie
Schmelzpunkt	An diesem Punkt sollen zwei Linien verschmolzen werden
Trennpunkt	An diesem Punkt sollen Linien getrennt werden

Tabelle 3.1: Punktkategorien

Status der Punkte

Im ersten Abschnitt des Algorithmus sollen die Punkte in verschiedene Kategorien eingeordnet werden, die ihren gegenwärtigen Status beschreiben. Die möglichen Status sind in Tabelle 3.1 aufgelistet. Es wird ein Array `punktStatus` angelegt, das zu jedem Punkt den entsprechenden Status angibt. Weiterhin werden zwei Funktionen `StatusLinienEnde` und `StatusLinienMitte` erstellt, die den Status eines Punktes anpassen, wenn dieser in einer weiteren Linie gefunden wird.

Zu Beginn des Algorithmus haben alle Punkte den Status *ungesehen*. Es werden alle Linien durchlaufen und auf deren Punkte die Funktion `StatusLinienMitte` beziehungsweise `StatusLinienEnde`, wenn es sich um einen Endpunkt der Linie handelt, angewendet. Je nach Anzahl der Linien, in denen ein Punkt enthalten ist, sowie in Abhängigkeit der Position des Punktes innerhalb der Linien ergibt sich ein anderer Status für diesen Punkt. In Tabelle 3.2 wird angegeben, wie sich der Status eines Punktes abhängig von seinem bestehenden Status ändert, wenn eine neue Linie gefunden wird, in der dieser Punkt als Linien-Endpunkt oder als Linien-Innenpunkt referenziert wird.

bestehender Status	Punkt ist in der neuen Linie	
	ein Endpunkt	**ein Innenpunkt**
ungesehen	Endpunkt	Innenpunkt
Endpunkt	Schmelzpunkt	Trennpunkt
Innenpunkt	Trennpunkt	Trennpunkt
Schmelzpunkt	Trennpunkt	Trennpunkt
Trennpunkt	Trennpunkt	Trennpunkt

Tabelle 3.2: Neuer Status nach Finden einer weiteren Referenz

Ziel dieser Aktion ist es, nur die passenden Punkte mit dem Status *Schmelz-* oder *Trennpunkt* zu versehen. Es können nur Linien verschmolzen werden, wenn sich genau zwei Endpunkte berühren. Berühren sich mehr als zwei Endpunkte, so können nicht eindeutig zwei Linien verschmolzen werden, daher bekommt dieser Punkt den Status *Trennpunkt*. Berührt ein Endpunkt oder ein Innenpunkt einen Innenpunkt, so sollen die Linien an dieser Stelle getrennt werden, daher bekommen diese Punkte auch den Status *Trennpunkt*. Die beiden Status *Endpunkt* und *Innenpunkt* dienen nur als Zwischenschritt haben keinen Einfluss auf die Trennung und Verschmelzung, spielen jedoch in den folgenden Abschnitten des Algorithmus noch eine Rolle. Der erste Abschnitt ist in Algorithmus 5 dargestellt.

Die Implementierung des ersten Abschnitts erfordert noch keine erwähnenswerten Optimierungen, eine vollständige Übersicht der Implementierung des gesamten Algorithmus befindet sich im Anhang im Quelltext 6.

Linien assoziieren

Nachdem durch Algorithmus 5 die Schmelzpunkte gefunden wurden, soll nun berechnet werden, welche Linien untereinander verschmolzen werden müssen.

Dafür werden zwei weitere Arrays `PaarA` und `PaarB` angelegt, welche zu einem gegebenen Linienindex den Index der zu verschmelzenden Linie ausgeben. Hierbei wurde bewusst der Datentyp *Array* (in der Implementierung `std::vector`) gewählt, da so eine konstante Zugriffszeit auf die Elemente möglich ist. Allerdings wird auch Speicher für Linien benötigt, die nicht verschmolzen werden. Eine Alternative würde der Datentyp *Map* darstellen, der weniger Speicher, aber dafür mit $O(\log n)$ eine höhere Zugriffszeit benötigen würde.

Algorithmus 5 : Status der Punkte analysieren

Eingabe : Linienmenge M
Ergebnis : Linienmenge M, `punktStatus`

```
1  Initialisiere Array punktStatus mit Status UNGESEHEN für alle Punkte.
2  Für jedes L ∈ M
3  |  punktStatus [L_erstes] ← StatusLinienEnde(L_erstes)
4  |  Für i ← 1 bis L_laenge − 1
5  |  |  punktStatus [L_i] ← StatusLinienMitte(L_i)
6  |  Ende
7  |  punktStatus [L_letztes] ← StatusLinienEnde(L_letztes)
8  Ende
9  Funktion StatusLinienEnde( status )
10 |  Falls status gleich
11 |  |  Wert UNGESEHEN
12 |  |  |  gib zurück ENDPUNKT
13 |  |  Wert ENDPUNKT
14 |  |  |  gib zurück SCHMELZPUNKT
15 |  |  sonst gib zurück TRENNPUNKT
16 |  Ende
17 Ende
18 Funktion StatusLinienMitte( status )
19 |  Wenn status = UNGESEHEN dann
20 |  |  gib zurück INNENPUNKT
21 |  sonst
22 |  |  gib zurück TRENNPUNKT
23 |  Ende
24 Ende
```

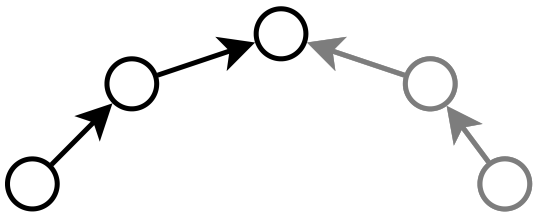


Abbildung 3.10: Zu verschmelzende Linien A und B

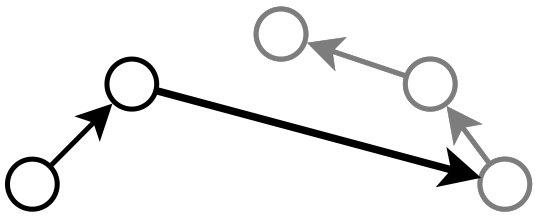


Abbildung 3.11: Falsch verschmolzene Linien *A* und *B*

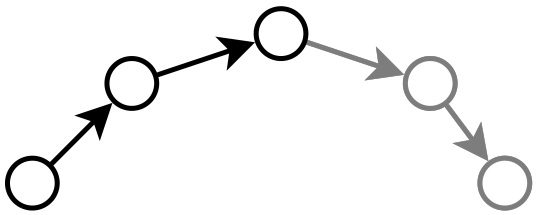


Abbildung 3.12: Korrekt verschmolzene Linien *A* und *B*

Ein Problem besteht, wenn zwei Linien mit unterschiedlichen Richtungen verschmolzen werden sollen, wie in Abbildung 3.10 gezeigt. In diesem Beispiel soll die dunkle Linie *B* mit der hellem Linie *A* verschmolzen werden, indem sie an das Ende gehängt wird. Aufgrund der unterschiedlichen Richtungen ergäbe sich allerdings bei diesem nativen Ansatz eine in Abbildung 3.11 dargestellte zusammengesetzte Linie.

Es ist also darauf zu achten, welches Ende jeder Linie den Schmelzpunkt beinhaltet. Sollten beide zu verschmelzende Punkte jeweils die letzten, beziehungsweise beide die ersten Punkte der Linien sein, so muss genau eine der Linien in der Richtung umgekehrt werden. Dabei ändert sich die Linie optisch nicht, da die Richtung nur innerhalb der Datenstruktur vorhanden ist. Im Beispiel wird die Richtung von Linie *B* geändert und dann der letzte Punkt von Linie *A* mit dem ersten Punkt von Linie *B* verschmolzen. Das Ergebnis ist die korrekt verschmolzene Linie in Abbildung 3.12.

Für den Algorithmus heißt das, dass für jeden Schmelzpunkt die beiden zu verschmelzenden Linien gespeichert werden. So kann vom jeder zu verschmelzenden Linie die entsprechende Partnerlinie gefunden werden.

In Algorithmus 5 wurde den Schmelzpunkten der Status *Schmelzpunkt* zugeteilt. Der folgende Abschnitt des Algorithmus analysiert nun diese Punkte der Reihe nach und findet die beiden entsprechenden Linien dazu.

Algorithmus 6 : Linienpaare finden

Eingabe : Linienmenge M
Ergebnis : Linienmenge M, `PaarA`, `PaarB`, `Drehen`

```
1  Für jedes L ∈ M
2      Wenn punktStatus [L_erstes] = SCHMELZPUNKT dann
3          Wenn PaarA [L_erstes] ≠ ∅ dann
4              PaarA [L_erstes] ← L
5          sonst
6              PaarB [L_erstes] ← L
7          Ende
8      Ende
9      Wenn punktStatus [L_letztes] = SCHMELZPUNKT dann
10         Wenn PaarA [L_letztes] ≠ ∅ dann
11             PaarA [L_letztes] ← L
12         sonst
13             PaarB [L_letztes] ← L
14         Ende
15     Ende
16 Ende
```

Algorithmus 6 zeigt, wie das Finden der Linienpaare ohne Berücksichtigung der Linienrichtungen abläuft. Dabei werden alle Linien der Linienmenge abgelaufen (Zeile 1) und der erste Punkt sowie der letzte Punkt analysiert.

Hat einer der analysierten Punkte den Statuts *Schmelzpunkt*, so wird zu diesem Punkt die Nummer der Linie als ein Partner im Paartupel eingetragen. Am Ende des Algorithmus existiert zu jedem Schmelzpunkt ein Paartupel aus `PaarA` und `PaarB`, in dem die beiden an diesem Punkt zu verschmelzenden Linien eingetragen sind.

Damit wurde auch dieser Abschnitt des Algorithmus vollständig behandelt. Die Linienmenge besteht nun aus Linien, deren Punkte verschiedene Status haben. Für jeden Schmelzpunkt ist der Status nun *bereit* und ein Index einer zugehörigen Linie als *Paar* angegeben. Weiterhin ist für jede Linie im Array `drehen` hinterlegt, ob diese Linie umgekehrt werden muss.

Linien verschmelzen

Nachdem die Vorbereitungen abgeschlossen wurden, kann nun mit dem Verschmelzen der Linien begonnen werden. Dafür wird ein weiteres Array `fertig` benötigt, der den Bearbeitungsstatus jeder Linie beinhaltet. Dies ist notwendig, da die Linien der Linienmenge nicht in der Reihenfolge bearbeitet werden, in der sie im Speicher abgelegt sind. Zu Beginn ist noch keine Linie berechnet, daher ist `fertig` für alle Linien *falsch*. Außerdem wird ein Zustand `drehen` eingeführt, der anzeigt, ob die aktuell bearbeitete Linie umgekehrt betrachtet werden muss.

Der dritte Abschnitt des Algorithmus analysiert jede Linie und betrachtet den ersten sowie den letzten Punkt der Linie. Haben beide Punkte nicht den Status *Schmelzpunkt*, so kann die Linie einfach in die Ausgabelinienmenge übernommen werden, wie in Zeile 4 des Algorithmus 7 gezeigt.

Ist jedoch genau einer der Punkte ein *Schmelzpunkt*, so bedeutet dies, das dort eine neue zu verschmelzende Linie beginnt. Je nachdem, an welchem Ende der Linie sich der *Schmelzpunkt* befindet, wird die Linie unverändert oder richtungsverkehrt an die Ausgabelinienmenge gefügt. Dies spielt sich in den Zeilen 6 und 7 des Algorithmus 7 unter Zuhilfenahme der Variable `drehen` ab. Während die Punkte der Reihe nach angefügt werden, wird deren `punktStatus` geprüft. Solle der `punktStatus` eines Punktes Trennpunkt sein, so wird die Linie an dieser Stelle getrennt und eine neue mit dem selben Punkt begonnen.

Nun wird in `PaarA` und `PaarB` für den Schmelzpunkt nach der passenden zu verschmelzenden Linie gesucht. Diese ist leicht zu finden, da in `PaarA` und `PaarB` nur die zu verschmelzende Linie und die aktuelle Linie hinterlegt sind. Sollte in `PaarA` die derzeit bearbeitete Linie hinterlegt sein, so ist die zu verschmelzende Linie in `PaarB`. Die zu verschmelzende Linie wird als N zwischengespeichert (Zeilen 15 und 23).

Anschließend wird geprüft, ob N vor dem Anfügen gedreht werden muss. Danach wird N als aktuelle Linie gesetzt und der Algorithmus sucht die nächste zu verschmelzende Linie.

Die Schleife terminiert erst, wenn kein weiterer Schmelzpunkt gefunden wird. Jede bearbeitete Linie wird im Array `fertig` mit *wahr* markiert. Der Algorithmus 7 fügt also ausgehend von einem Startpunkt die Linien zusammen, die jeweils als Paar eingetragen waren.

Algorithmus 7 : Linien verschmelzen

Eingabe : Linienmenge M
Ausgabe : Linienmenge R

```
1  Für jedes L ∈ M
2      Wenn fertig [L] = falsch dann
3          Wenn punktStatus [L_letztes] ≠ SCHMELZPUNKT und punktStatus [L_erstes] ≠
           SCHMELZPUNKT dann
4              Füge L an die Ausgabelinienmenge
5          sonst
6              Wenn punktStatus [L_letztes] ≠ SCHMELZPUNKT und punktStatus [L_erstes] =
               SCHMELZPUNKT dann Drehen ← wahr
7              Wenn punktStatus [L_letztes] = SCHMELZPUNKT und punktStatus [L_erstes] ≠
               SCHMELZPUNKT dann Drehen ← falsch
8              wiederhole
9                  Wenn fertig [L] dann unterbrich Schleife
10                 Wenn Drehen = wahr dann
11                     Füge L richtungsverkehrt an newLine
12                     fertig[L] ← wahr
13                     Wenn punktStatus [L_erstes] = SCHMELZPUNKT dann
14                         Finde N als Paar
15                     sonst
16                         N ← ∅
17                     Ende
18                 sonst
19                     Füge L an newLine
20                     fertig[L] ← wahr
21                     Wenn punktStatus [L_letztes] = SCHMELZPUNKT dann
22                         Finde N als Paar
23                     sonst
24                         N ← ∅
25                     Ende
26                 Ende
27                 Wenn N ≠ ∅ dann
28                     Berechne Drehen
29                     L ← N
30                     Wenn fertig [L] dann unterbrich Schleife
31                 sonst
32                     Unterbrich Schleife
33                 Ende
34             bis es unterbrochen wird
35             Füge newLine der Ausgabelinienmenge R hinzu.
36         Ende
37     Ende
38 Ende
39 gib zurück R
```

Algorithmus 8 : Kreisförmige Linien verschmelzen

Eingabe : Linienmenge M
Ausgabe : Linienmenge R

Für jedes $L \in M$
  Wenn `fertig` *[L]* = *falsch* **dann**
    `Drehen` ← falsch
    wiederhole
      Wenn `fertig` *[L]* **dann** unterbrich Schleife
      Wenn `Drehen` = *wahr* **dann**
        Füge L richtungsverkehrt an *newLine*
        `fertig`$[L]$ ← wahr
        Wenn `punktStatus` $[L_{erstes}] =$ `SCHMELZPUNKT` **dann**
          Finde N als Paar
        sonst
          $N \leftarrow \emptyset$
        Ende
      sonst
        Füge L an *newLine*
        `fertig`$[L]$ ← wahr
        Wenn `punktStatus` $[L_{letztes}] =$ `SCHMELZPUNKT` **dann**
          Finde N als Paar
        sonst
          $N \leftarrow \emptyset$
        Ende
      Ende
      Wenn $N \neq \emptyset$ **dann**
        Berechne `Drehen`
        $L \leftarrow N$
        Wenn `fertig` *[L]* = *wahr* **dann** unterbrich Schleife
      sonst
        Unterbrich Schleife
      Ende
    bis es unterbrochen wird
    Füge *newLine* der Ausgabelinienmenge R hinzu.
  Ende
Ende
gib zurück R

Mit Algorithmus 7 sind fast alle Linien verschmolzen worden, doch es existiert noch ein unbehandelter Sonderfall. Sollten Linien einer Linienmenge zu einem Kreis zusammengeschlossen sein, werden sie im Algorithmus noch nicht verschmolzen. Das hat den Grund, dass kein Anfangspunkt im Kreis gefunden werden konnte. Eine einfache Lösung ist nun, die verbliebenen Linien, die also noch den Bearbeitungsstatus `falsch` haben, zu wählen und an Schmelzpunkten aneinander zu hängen, ohne vorher einen Anfang der Linie zu suchen. Das wird im Algorithmus 8, der auf Algorithmus 7 basiert, verwirklicht. Lediglich die Prüfung, ob eine Linie den Anfang einer zu verschmelzenden Linie darstellt, wurde im Algorithmus 8 entfernt.

Jeder dieser vier Schritte läuft in $O(n)$ für die Anzahl der Punkte der Linien ab. In den letzten beiden Schritten werden die Linien zwar nicht der Reihe nach bearbeitet, doch durch die Verwendung der Bearbeitungsstatus `fertig` kein Punkt mehrmals bearbeitet werden.

3.5 Linienglättung und -vereinfachung

3.5.1 Linienvereinfachung mithilfe des Douglas-Peucker-Algorithmus

Um den Speicher des Computers nicht an die Grenzen zu bringen und nachfolgende Algorithmen zu beschleunigen, kann es hilfreich sein, die Linienmengen zu vereinfachen und mit weniger Punkten darzustellen. Eine Vereinfachung hat jedoch zur Folge, dass die Linien an Genauigkeit verlieren. Ziel ist es also, einen Kompromiss zwischen Genauigkeitsverlust und Geschwindigkeitserhöhung sowie Speicherverbrauchsminimierung zu finden.

Einen sehr guten Ansatz zum Vereinfachen von Linien verfolgt der Algorithmus von Douglas und Peucker [3]. Dabei werden nur die Punkte entfernt, die den Verlauf der Linie wenig beeinflussen. Als Eingabe wird neben der Linienmenge ein Grenzwert ϵ übergeben, der angibt, bis zu welchem Abstand die vereinfachte Linie von den ursprünglichen Punkten abweichen darf, wie in Abbildung 3.13 dargestellt ist.

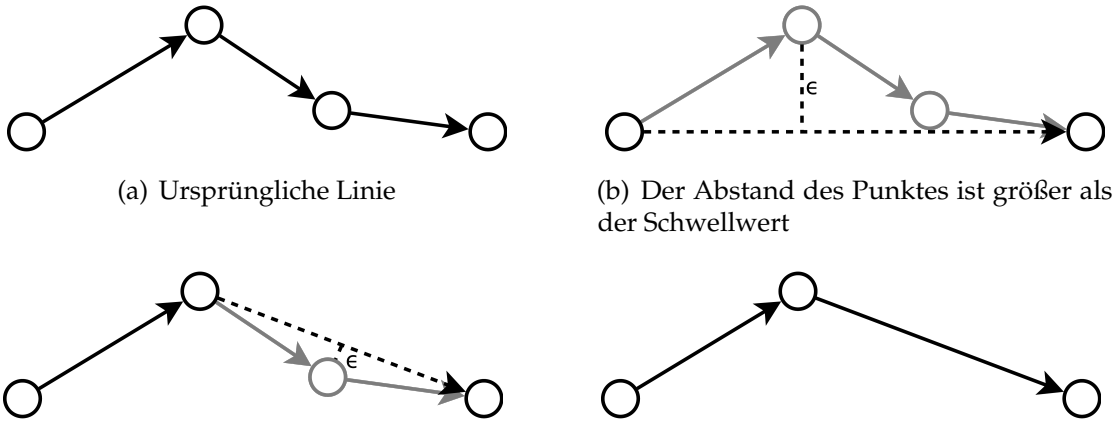
Der Algorithmus läuft rekursiv ab. Zu Beginn befinden sich nur der erste sowie der letzte Punkt in der Ausgabelinienmenge. Zwischen diesen Punkten wird nun eine imaginäre Linie gezogen. Anschließend werden alle Punkte, die in der Eingabelinienmenge zwischen

(a) Punkt außerhalb des Schwellwerts ϵ bleibt erhalten

(b) Punkt innerhalb des Schwellwerts ϵ wird entfernt

Abbildung 3.13: Punktentfernung nach Schwellwert ϵ

diesen beiden Punkten liegen, abgearbeitet und die Distanz zur imaginären Linie ermittelt. Für den Punkt mit dem größten Abstand wird geprüft, ob dessen Distanz zur imaginären Linie größer als ϵ ist. Sollte dies der Fall sein, so wird der Punkt zur Ausgabeliniemenge hinzugefügt und für die Linienabschnitte zu beiden Seiten dieses Punktes der Algorithmus rekursiv aufgerufen. In Abbildung 3.14 ist der Ablauf des Algorithmus illustriert.



(a) Ursprüngliche Linie

(b) Der Abstand des Punktes ist größer als der Schwellwert

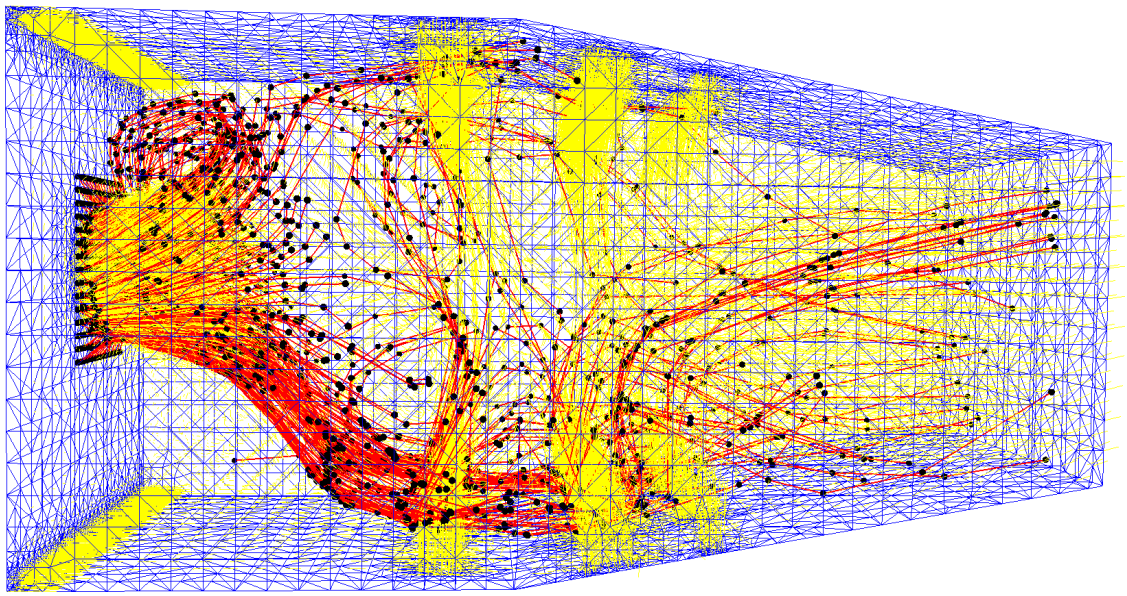
(c) Der Punkt liegt innerhalb des Schwellwerts und wird entfernt

(d) Vereinfachte Linie

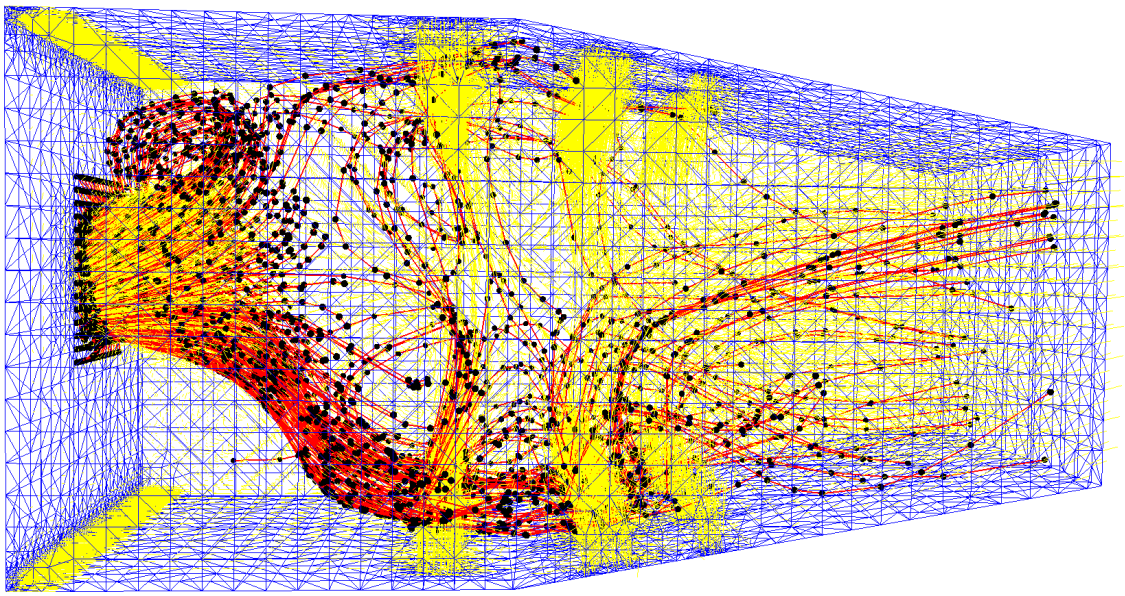
Abbildung 3.14: Beispielanwendung des Douglas-Peucker-Algorithmus

Der Douglas-Peucker-Algorithmus terminiert, wenn kein Punkt mehr gefunden werden kann, dessen Distanz zur vereinfachten Linie größer als der Grenzwert ϵ ist.

Im Worst Case kann dieser rekursive Algorithmus einen Aufwand von $O(n \cdot m)$ haben, wobei n die Anzahl der Eingabepunkte und m die Anzahl der Punkte auf der vereinfachten Linie ist. In den Abbildungen 3.15 und 3.17 sind Ergebnisse des Douglas-Peucker-Algorithmus mit verschiedenen Werten für ϵ dargestellt.

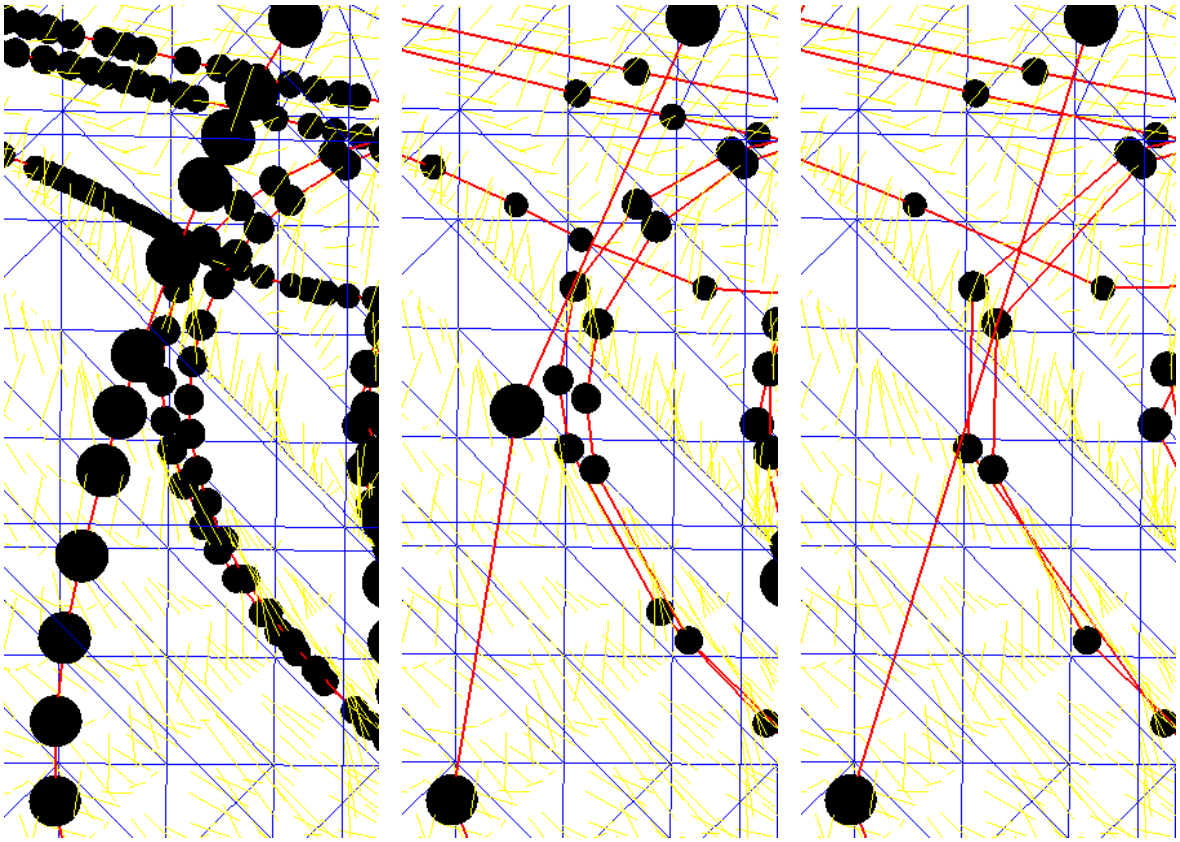


(a) Linienvereinfachung $\epsilon = 1$



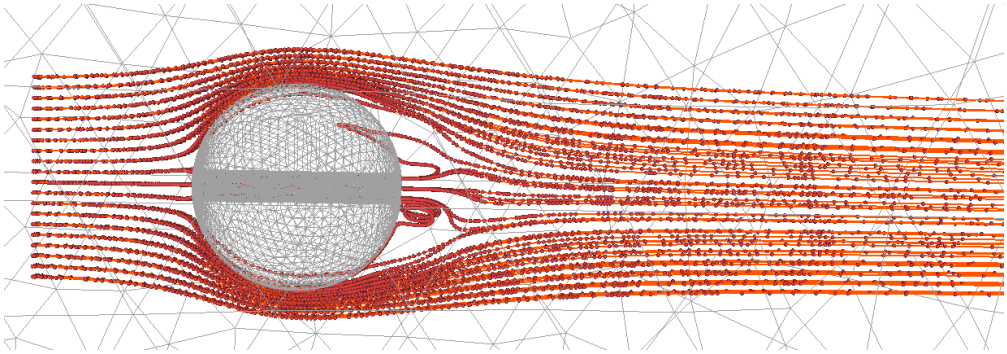
(b) Linienvereinfachung $\epsilon = 0.3$

Abbildung 3.15: Douglas-Peucker-Linienvereinfachung im Gasbrennkammerdatensatz

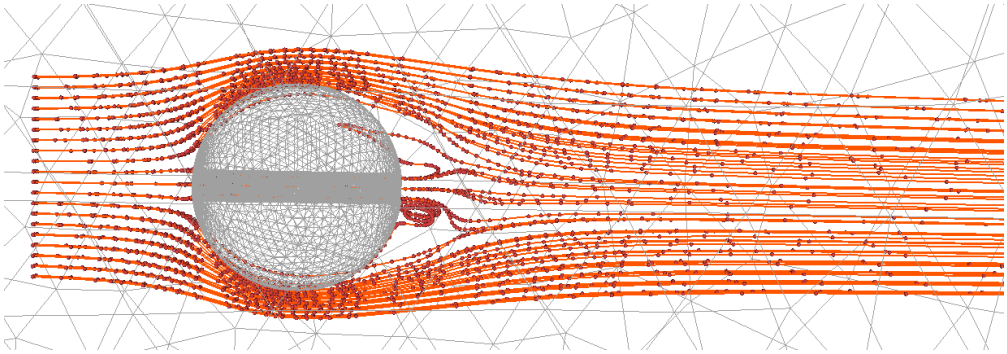


(a) unvereinfachte Linienmenge (b) Linienvereinfachung $\epsilon = 0.3$ (c) Linienvereinfachung $\epsilon = 1$

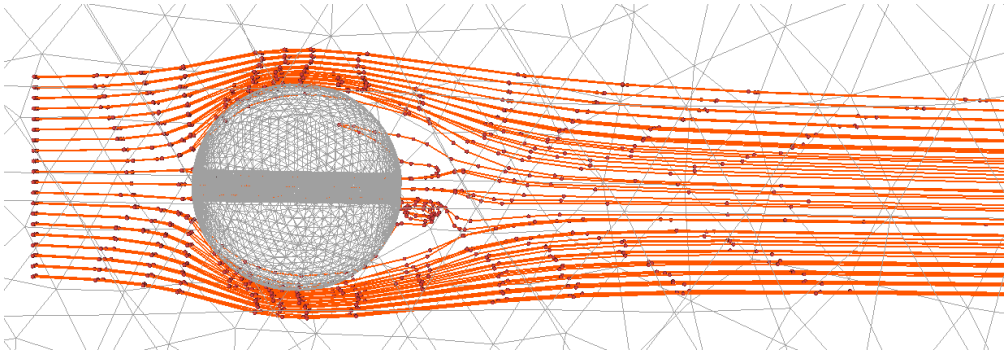
Abbildung 3.16: Vergrößerter Ausschnitt der Douglas-Peucker-Linienvereinfachung im Gasbrennkammerdatensatz



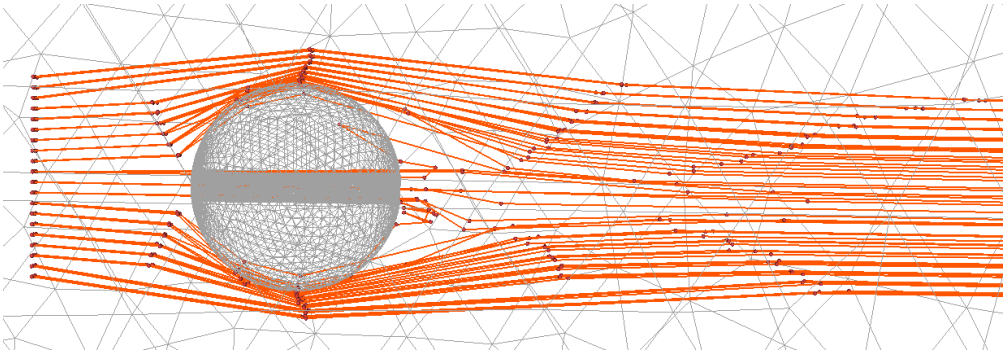
(a) Unvereinfachte Linien



(b) Linienvereinfachung $\epsilon = 0.1$



(c) Linienvereinfachung $\epsilon = 1.0$



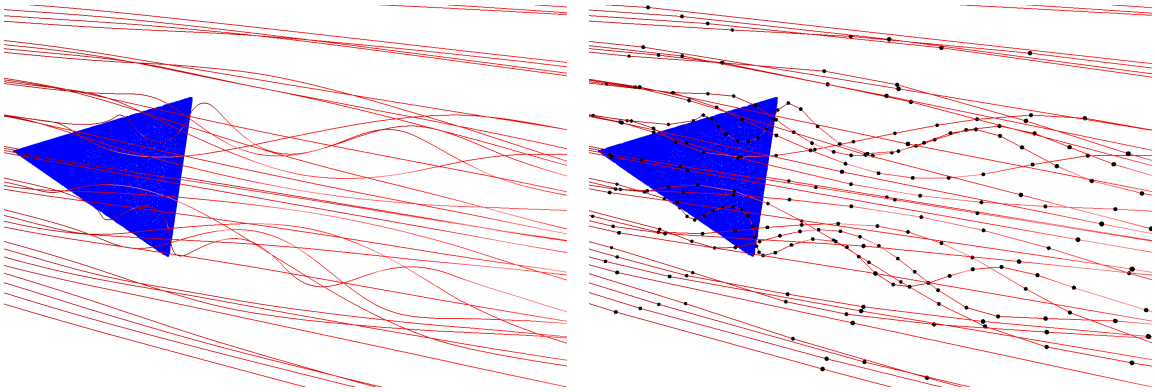
(d) Linienvereinfachung $\epsilon = 10$

Abbildung 3.17: Douglas-Peucker-Linienvereinfachung bei Stromlinien um die Lochkugel

Algorithmus 9 : Linienvereinfachung nach Douglas und Peucker

Eingabe : Linienmenge M, Schwellwert ϵ
Ausgabe : Linienmenge R

```
1  gib zurück DouglasPeucker (M, 0, M_laenge, ε)
2  Funktion DouglasPeucker (Linie L, Start s, Ende e, Grenzwert ε)
3      d_max ← 0
4      i_max ← 0
5      R ← ∅
6      Wenn e - s < 2 dann
7          Füge L_s und L_e zu R hinzu
8          gib zurück R
9      sonst
10         Für i ← s + 1 bis e − 1
11             d ← kleinster Abstand von L_i zur Linie zwischen L_s und L_e
12             Wenn d > d_max dann
13                 d_max ← d
14                 i_max ← i
15             Ende
16         Ende
17         Wenn d_max ≥ ε dann
18             Füge DouglasPeucker (L, s, i_max, ε) zu R hinzu
19             Füge DouglasPeucker (L, i_max, e, ε) zu R hinzu
20             gib zurück R)
21         sonst
22             Füge L_s und L_e zu R hinzu
23             gib zurück R
24         Ende
25     Ende
26 Ende
```



(a) Ursprüngliche Linienmenge (b) Vereinfachte Linienmenge

Abbildung 3.18: Douglas-Peucker-Linienvereinfachung im Dreiecksdatensatz

3.5.2 Bézierglättung

Neben der Linienvereinfachung, zum Beispiel durch Douglas-Peucker, kann auch eine Linienglättung für eine bessere Darstellung gewünscht sein.

Für die Glättung von Linien bietet sich die Verwendung der nach Pierre Bézier benannten Bézier-Kurven an, die von Foley et al. im Buch *Computer Graphics: Principles and Practice*[6] erklärt werden. In dieser Arbeit werden die Bézier-Kurven mithilfe eines angepassten De-Casteljau-Algorithmus erstellt. Da der Algorithmus in diesem Fall nur kubische Bézierkurven berechnet, wurden die zwei dafür notwendigen Rekursionsschritte in der Implementierung zu einer Gleichung zusammengefasst. Die Darstellung einer vollständigen Bézierkurve ist durch die Datenstruktur nicht möglich, der Verlauf der Linie kann lediglich durch Hinzufügen zusätzlicher Punkte an den Verlauf einer Bézierkurve angenähert werden. Die Anzahl der pro Segment einzufügenden Punkte wird dem Algorithmus dabei als Parameter übergeben.

Da immer nur eine Linie mit drei Punkten durch eine quadratische Bézierkurve geglättet werden kann, werden die Linien der Linienmenge in Abschnitte unterteilt und einzeln geglättet. In der Mitte jedes Liniensegments wird ein Hilfspunkt erstellt. Für jedes Tupel aus einem Punkt b der Linienmenge und seine angrenzenden Hilfspunkte a und c wird eine Bezierkurve berechnet. Gegeben sei eine solche Linie mit den Punkten a, b und c, wie beispielsweise in Abbildung 3.21(a) dargestellt. Weiterhin sei $f \in [0, 1]$ eine Laufvariable, deren Schrittweite den Detailgrad der geglätteten Linie darstellt. Dann werden im ersten

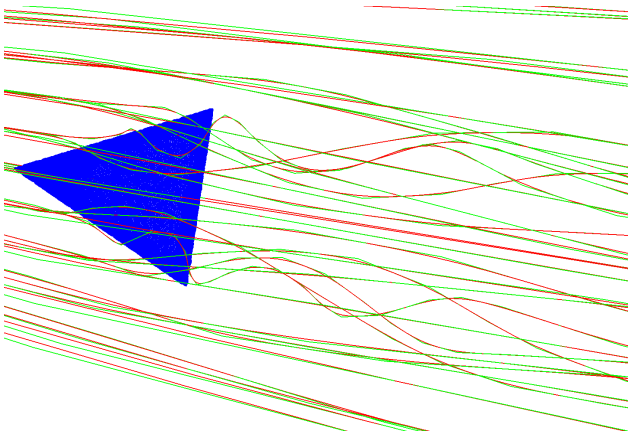
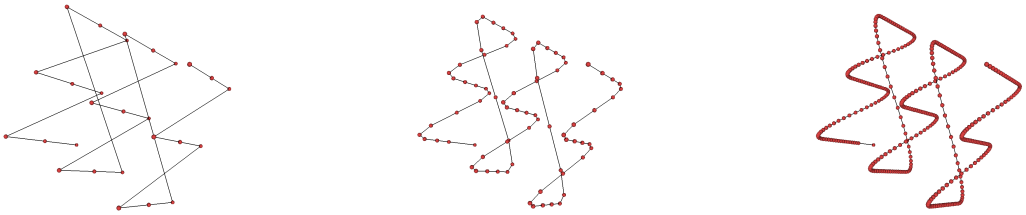


Abbildung 3.19: Ungeglättete (grün) und Bézier-geglättete (rot) Linien im Dreiecksdatensatz



(a) Ungeglättete Linien (b) Detailgrad 3 (c) Detailgrad 15

Abbildung 3.20: Verschiedene Detailgrade der Bézier-Kurvenglättung

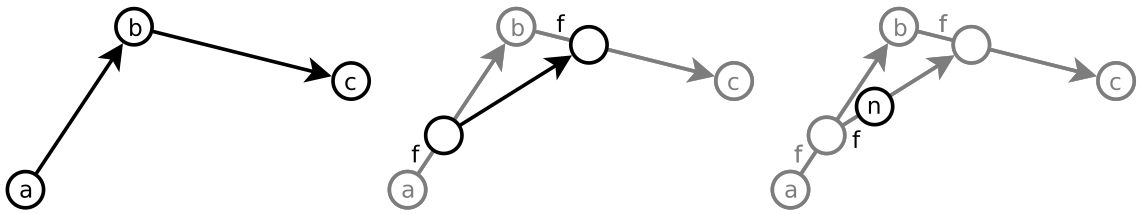
Rekursionsschritt zwei weitere Hilfspunkte p und q mit dem Verhältnis f auf den Segmenten $\overrightarrow{ab}$ und $\overrightarrow{bc}$ wie folgt erstellt:

$$\vec{p} = \vec{a} + f \cdot \overrightarrow{ab} \tag{3.8}$$

$$\vec{q} = \vec{b} + f \cdot \overrightarrow{bc} \tag{3.9}$$

Die beiden entstandenen Hilfspunkte sind in Abbildung 3.21(b) gezeigt. Im nächsten Rekursionsschritt wird auf der Verbindung $\overrightarrow{pq}$ zwischen den zuvor erstellten Hilfspunkten mit dem Verhältnis f die Position des neu einzufügenden Punktes n berechnet, wie in Abbildung 3.21(c) gezeigt wird. Es ergibt sich folgende Gleichung:

$$\vec{n} = \vec{p} + f \cdot \overrightarrow{pq} \tag{3.10}$$



(a) Ausgangslinien (b) zwei Hilfspunkte (c) Neuer Punkt

Abbildung 3.21: Ablauf der Bézier-Kurvenglättung

Diese Schritte lassen sich zu einer Gleichung zusammenfassen:

$$n = \vec{a}f^2 - 2\vec{a}f + \vec{a} - 2\vec{b}f^2 + 2\vec{b}f + \vec{c}f^2 \tag{3.11}$$

Der Aufwand dieses Algorithmus liegt bei $O(n \cdot d)$, wobei n die Anzahl der Punkte ist und d der Detailgrad.

Der Einfluss des Detailgrads lässt sich in Abbildung 3.20 erkennen.

Algorithmus 10 : Bézier-Linienglättung

Eingabe : Linienmenge M, Detailgrad d
Ausgabe : Linienmenge R

```
1  s ← 1 ÷ d
2  Für jedes L ∈ M
3  |  Füge L_erstes zu N hinzu
4  |  a ← L_0
5  |  Für i ← 1 bis L_laenge − 1
6  |  |  b ← L_i
7  |  |  c ← b + (L_{i+1} − b) ÷ 2
8  |  |  Für f ← 0 bis 1 mit Schrittweite s
9  |  |  |  n ← af² − 2af + a − 2bf² + 2bf + cf²
10 |  |  |  Füge n zu N hinzu
11 |  |  Ende
12 |  |  a ← c
13 |  Ende
14 |  Füge L_letztes zu N hinzu
15 |  Füge N zu R hinzu
16 Ende
17 gib zurück R
```

4 Diskussion

Durch diese sechs *Datenalgorithmen* konnte ein Einblick in die Vielseitigkeit von Linienmengen gewonnen werden. Die Linien der Linienmengen können nach Länge gefiltert und an Winkeln getrennt werden. Doppelte Punkte können entfernt und Linien verschmolzen werden. Außerdem ist es durch die Algorithmen möglich, die Linien in ihrem Lauf zu vereinfachen oder zu glätten.

Für einige Algorithmen, wie den Algorithmus zur Linienvereinfachung in Kapitel 3.5.1 oder dem Linienglättungsalgorithmus in Kapiel 3.5.2 konnte auf bereits bestehende Algorithmen zurückgegriffen werden, die nur an die entsprechenden Gegebenheiten angepasst werden mussten. Andere Algorithmen, wie der Algorithmus zum Verschmelzen von Linien in Kapitel 3.4.2, wurden von Grund auf neu entwickelt.

Der Längenfilter aus Kapitel 3.3.1 ist trotz seiner Einfachheit ein Algorithmus, der im Programm einige Zeit in Anspruch nimmt. Hauptursächlich dafür ist die aufwändige Berechnung für jedes Liniensegment. An dieser Stelle könnte in einer zukünftigen Arbeit eine Methode entwickelt werden, die die Länge der Linien durch einfache Berechnungen abschätzt und nur bei Bedarf die genaue Länge ermittelt. Beispielsweise könnte geprüft werden, ob eine Linie A, deren Punkte eine Untermenge der Punkte einer Linie B sind, immer kürzer als diese Linie B ist und ob sich daraus ein schnellerer Algorithmus ableiten lässt.

Der Winkelfilter in Kapitel 3.3.2 funktioniert sehr effizient. Allerdings werden nur die Winkel zwischen den einzelnen Liniensegmenten betrachtet. Setzt sich eine Krümmung der Linie aus mehreren Liniensegmenten zusammen, wird dies nicht unbedingt vom Algorithmus erkannt. Eine zukünftige Arbeit könnte sich also damit befassen, eine Krümmung der Linie unabhängig von der Anzahl der Liniensegmente zu erkennen.

Der Algorithmus zum Entfernen doppelter Punkte in Kapitel 3.4.1 funktioniert durch die Map effizient, entfernt jedoch nur Punkte, die exakt die selben Koordinaten haben.

In der Praxis werden jedoch Fließkommawerte verwendet, daher werden Punkte schon bei kleinsten Abweichungen in der Berechnung nicht mehr als doppelt erkannt. Eine Weiterentwicklung des Algorithmus könnte einen ϵ-Wert einführen, der eine Abweichung angibt, bis zu der Punkte auseinander liegen dürfen und dennoch als doppelt erkannt werden. Dabei könnte aber die Datenstruktur der Map nicht weiter verwendet werden, da sich in dieser nicht einfach der Abstand der Punkte berechnen lässt. Als Lösung könnte stattdessen ein Plane-Sweep-Algorithmus Anwendung finden, der den Datensatz durchläuft und Punkte mit kleinem Abstand zusammenfasst.

Da der Algorithmus zum Verschmelzen der Linien einer Linienmenge aus Kapitel 3.4.2 neu entwickelt wurde, sind noch weitere Arbeiten notwendig, um den Algorithmus zu optimieren. Als Voraussetzung steht nur das Entfernen doppelter Punkte. Dabei werden jedoch noch keine Liniensegmente beachtet, die sich schneiden. Für optimale Ergebnisse sollte also zuvor ein Line-Intersect-Algorithmus ausgeführt werden. Ein solcher Algorithmus könnte als einfacher Plane-Sweep-Algorithmus implementiert werden. Weiterhin ist der Speicherverbrauch des Linienverschmelzalgorithmus nicht optimal. Dies fällt jedoch im Vergleich zu den Datenmengen der übrigen Simulationsdaten (zum Beispiel dem Gitter) kaum ins Gewicht. Der Algorithmus könnte erheblich vereinfacht werden, wenn bei der Verschmelzung die Richtung der Linien berücksichtigt werden würde.

Die Algorithmen zur Linienvereinfachung (Kapitel 3.5.1) und Linienglättung (Kapitel 3.5.2) wurden auf bereits vorhandenen Algorithmen basierend implementiert. Über ihre Optimierung wurde bereits in anderen Arbeiten diskutiert. Zu Beachten ist, dass die Bezierglättung keine Kreisformen beachtet. Eine zukünftige Arbeit könnte sich generell mit kreisförmigen Linien, also geschlossenen Polygonzügen beschäftigen.

Freilich sind dies nur Beispiele und ein Bruchteil von den Möglichkeiten, die die Datenstruktur *Linienmenge* bildet. Als neu eingeführte Datenstruktur in *FAnToM* liefert sie noch genügend Raum für Forschung und Experimente, die auch an diese Arbeit anknüpfen können.

Beispielsweise könnte die Anzeige der Linien erweitert werden, indem die Richtung der Linien mit Pfeilen angezeigt wird oder verschiedene Farben für Winkel und Längen verwendet werden. Außerdem könnten weitere Filter entwickelt werden, die zum Beispiel stark verwirbelte Linien auswählen oder Linien filtern, die einen bestimmten Bereich schneiden.

Der Bedarf nach weiteren Algorithmen können auch abhängig von den Eingabedaten während der Benutzung von *FAnToM* auftauchen.

Durch die in *FAnToM* verwendete Methode des *Datenflussnetzwerke* können die Algorithmen miteinander verknüpft werden und somit komplexere Algorithmen gebaut werden.

Die in dieser Arbeit vorgestellten Algorithmen wurden noch nicht auf Parallelisierbarkeit untersucht. Bei den meisten Algorithmen sollte eine parallele Abarbeitung verschiedener Linien möglich sein.

Zusammenfassend ist mit der Einführung von *Linienmengen* in *FAnToM* eine sinnvolle Erweiterung des Programms geglückt, welche noch Platz für weitere Arbeiten liefert.

A Quellcodes

```
double getLineLength( size_t lineNr )
{
   if ( getNrPoints() < 2) return 0.0;

   double lineLength = 0.0;

   Point3 lastUsedPoint = getPoint( mLines.at( lineNr ).at( 0 ) );
   for ( auto point = mLines.at( lineNr ).begin()+1;
         point != mLines.at( lineNr ).end(); point++ )
   {
      lineLength += norm( getPoint( *point ) - lastUsedPoint );
      lastUsedPoint = getPoint( *point );
   }
   return lineLength;
}
```

Quellcode 1: Berechnung der Länge einer Linie

```
std::shared_ptr<const FtLineSet> lineset = options.get< FtLineSet >("lineset");
double threshold = options.get<double>("threshold");

double lineLength;
std::vector< size_t > newLine;
size_t nrPtsOfLine, currPointIdx;

for (size_t currLineIdx=0; currLineIdx < lineset->getNrLines(); currLineIdx++)
{
  nrPtsOfLine = lineset->getNrPointsOfLine( currLineIdx );
  newLine.clear();
  newLine.reserve( nrPtsOfLine );

  lineLength = lineset->getLineLength( currLineIdx );

  if (lineLength < threshold) {
    for ( currPointIdx = 0; currPointIdx < nrPtsOfLine; currPointIdx++ )
    {
      newLine.push_back( filteredLineSetBelow->addPointUnique( lineset->getPointOnLine( currLineIdx, currPointIdx ) ) );
    }
    filteredLineSetBelow->addLine( newLine );
  } else {
    for ( currPointIdx = 0; currPointIdx < nrPtsOfLine; currPointIdx++ )
    {
      newLine.push_back( filteredLineSetAbove->addPointUnique( lineset->getPointOnLine( currLineIdx, currPointIdx ) ) );
    }
    filteredLineSetAbove->addLine( newLine );
  }
}
```

Quellcode 2: Längenfilter

A Quellcodes

```
std::shared_ptr<const FtLineSet> lineset = options.get< FtLineSet >("lineset");
double threshold = options.get<double>("threshold");

threshold = cos( threshold / 360 * M_PI );

std::vector< size_t > newLine;
Point3 ptA, ptB, ptC;
double angle;

for (size_t currLineIdx=0; currLineIdx < lineset->getNrLines(); currLineIdx++ )
{
  newLine.clear();
  newLine.reserve( lineset->getNrPointsOfLine( currLineIdx ) );

  ptA = lineset->getPointOnLine( currLineIdx, 0 );
  ptB = lineset->getPointOnLine( currLineIdx, 1 );
  newLine.push_back( filteredLineSet->addPointUnique( ptA ) );
  newLine.push_back( filteredLineSet->addPointUnique( ptB ) );

  for (size_t currPtIdx=2; currPtIdx < lineset->getNrPointsOfLine(currLineIdx); currPtIdx++ )
  {
    ptC = lineset->getPointOnLine( currLineIdx, currPtIdx );

    angle = normalized(ptA - ptB) * normalized(ptB - ptC);

    if ( angle < threshold )
    {
      newLine.push_back( filteredLineSet->addPointUnique( ptB ) );
      newLine.shrink_to_fit();
      filteredLineSet->addLine( newLine );
      newLine.clear();
      newLine.reserve( lineset->getNrPointsOfLine( currLineIdx ) - currPtIdx );
    }

    newLine.push_back( filteredLineSet->addPointUnique( ptB ) );

    ptA = ptB;
    ptB = ptC;
  }

  newLine.push_back( filteredLineSet->addPointUnique( ptB ) );
  newLine.shrink_to_fit();
  filteredLineSet->addLine( newLine );
}
```

Quellcode 3: Winkelfilter

```
std::shared_ptr< const FtLineSet > lineset = options.get<FtLineSet>("lineset");
std::map< Point3, size_t, comparePoint3 > map;
std::map< Point3, size_t >::const_iterator mapIterator;

std::vector< size_t > newLine;
Point3 currentPoint;
size_t ptIdx;

for ( size_t lineNr = 0; lineNr < lineset->getNrLines(); lineNr++ )
{
  newLine.clear();
  for ( ptIdx = 0; ptIdx < lineset->getNrPointsOfLine( lineNr ); ptIdx++ )
  {
    currentPoint = lineset->getPointOnLine( lineNr, ptIdx );
    mapIterator = map.find( currentPoint );
    if ( mapIterator == map.end() )
    {
      newLine.push_back(
        map[currentPoint] = filteredLineSet->addPoint(currentPoint)
      );
    } else {
      newLine.push_back( map[ currentPoint ] );
    }
```

```
  }
  filteredLineSet->addLine(newLine);
}
```

Quellcode 4: Doppelte Punkte entfernen

```
std::vector< size_t > DouglasPeucker( std::shared_ptr< const FtLineSet > lineset, std::vector< size_t > line, size_t start, size_t end, double
    epsilon )
{
  double dmax = 0;
  size_t index = 0;
  std::vector< size_t > results;

  // invalid epsilon value
  if ( epsilon <= 0 )
  {
    return line;
  }

  // line too short
  if ( end - start < 2 )
  {
    results.reserve( 2 );
    results.push_back( line.at( start ) );
    results.push_back( line.at( end ) );
    return results;
  }

  // find longest distance point
  for ( size_t i = start+1; i < end; i++ )
  {
    double d = distance( lineset->getPoint(line.at(i)), lineset->getPoint(line.at( start )), lineset->getPoint(line.at( end )) );
    if (d > dmax)
    {
      dmax = d;
      index = i;
    }
  }

  // if distance is longer than threshold
  if ( dmax >= epsilon )
  {
    // insert point and call algorithm recursive
    std::vector< size_t > resultA = DouglasPeucker( lineset, line, start, index, epsilon );
    std::vector< size_t > resultB = DouglasPeucker( lineset, line, index, end, epsilon );
    results.reserve( resultA.size() + resultB.size() - 1 );
    for ( size_t i = 0; i < resultA.size(); i++ )
    {
      results.push_back( resultA.at( i ) );
    }
    for ( size_t i = 1; i < resultB.size(); i++ )
    {
      results.push_back( resultB.at( i ) );
    }
  } else {
    results.reserve( 2 );
    results.push_back( line.at( start ) );
    results.push_back( line.at( end ) );
  }
  return results;
}
```

Quellcode 5: Linien glätten nach Douglas und Peucker

```
// IMPORTANT: The algorithm only works if all duplicate points were removed before

bool split = options.get<bool>("split"); // get split value

// copy points to output
```

```
6   for ( size_t pointNr = 0; pointNr < lineset->getNrPoints(); pointNr++ )
7   {
8      filteredLineSet->addPoint( lineset->getPoint(pointNr) );
9   }
10
11  // variables
12  size_t lineNr, linePtIdx, currLine, nextLine;
13
14  // (1) Put points into different categories
15  std::vector< Status > pointStatus ( lineset->getNrPoints(), UNSEEN );
16
17  Progress progress(*this, "categorize", lineset->getNrLines());
18  for ( lineNr = 0; lineNr < lineset->getNrLines(); lineNr++ )
19  {
20     progress++;
21     pointStatus[ lineset->getLine( lineNr ).front() ] = changeStatusEnd( pointStatus[ lineset->getLine( lineNr ).front() ] );
22     pointStatus[ lineset->getLine( lineNr ).back() ] = changeStatusEnd( pointStatus[ lineset->getLine( lineNr ).back() ] );
23
24     for ( linePtIdx = 1; linePtIdx < lineset->getNrPointsOfLine(lineNr) - 1; linePtIdx++ )
25     {
26   pointStatus[ lineset->getLine( lineNr )[ linePtIdx ] ] = changeStatusMid( pointStatus[ lineset->getLine( lineNr )[ linePtIdx] ] );
27     }
28  }
29
30  // (2) get pairs
31  std::vector< size_t > linePairA ( lineset->getNrPoints(), lineset->getNrLines() ); // line pairs
32  std::vector< size_t > linePairB ( lineset->getNrPoints(), lineset->getNrLines() ); // line pairs
33  progress.reset("detecting", lineset->getNrLines());
34  for ( lineNr = 0; lineNr < lineset->getNrLines(); lineNr++ )
35  {
36     progress++;
37     if ( pointStatus[ lineset->getLine( lineNr ).front() ] == MERGE )
38     {
39   if ( linePairA[ lineset->getLine( lineNr ).front() ] == lineset->getNrLines() )
40   {
41      linePairA[ lineset->getLine( lineNr ).front() ] = lineNr;
42   } else {
43      linePairB[ lineset->getLine( lineNr ).front() ] = lineNr;
44   }
45     }
46     if ( pointStatus[ lineset->getLine( lineNr ).back() ] == MERGE )
47     {
48   if ( linePairA[ lineset->getLine( lineNr ).back() ] == lineset->getNrLines() )
49   {
50      linePairA[ lineset->getLine( lineNr ).back() ] = lineNr;
51   } else {
52      linePairB[ lineset->getLine( lineNr ).back() ] = lineNr;
53   }
54     }
55  }
56
57  std::vector< size_t > newLine;            // line to be pushed back to newLines
58  std::vector< bool > done ( lineset->getNrLines(), false );
59  bool reverse = false; // reverse line
60
61  // (3) merge and split lines
62  progress.reset("merging lines", lineset->getNrLines());
63  for ( lineNr = 0; lineNr < lineset->getNrLines(); lineNr++ )
64  {
65     progress++;
66     if ( !done[ lineNr ] ) // if line was not processed yet
67     {
68   // copy single lines
69   if ( pointStatus[ lineset->getLine( lineNr ).back() ] != MERGE && pointStatus[ lineset->getLine( lineNr ).front() ] != MERGE )
70   {
71      newLine.reserve( newLine.size() + lineset->getNrPointsOfLine( lineNr ) );
72      for ( linePtIdx = 0; linePtIdx < lineset->getNrPointsOfLine( lineNr ); linePtIdx++ )
73      {
74     newLine.push_back( lineset->getLine( lineNr )[ linePtIdx ] );
75     if ( split && pointStatus[ lineset->getLine( lineNr )[ linePtIdx ] ] == SPLIT )
76     {
```

```
77        if ( newLine.size() > 1 )
78        {
79      filteredLineSet->addLine(newLine);
80      newLine.clear();
81      newLine.push_back( lineset->getLine( lineNr )[ linePtIdx ] );
82        }
83    }
84      }
85      if ( newLine.size() > 1 )
86      {
87    filteredLineSet->addLine(newLine);
88      }
89      newLine.clear();
90  }
91
92  // find beginning of connected lines
93  currLine = lineset->getNrLines();
94  if ( pointStatus[ lineset->getLine( lineNr ).front() ] == MERGE && pointStatus[ lineset->getLine( lineNr ).back() ] != MERGE ) // beginning of
        new line
95  {
96      currLine = lineNr; // set current line
97      reverse = true;
98      newLine.clear();
99      newLine.push_back(lineset->getLine( lineNr ).back());
100 }
101 if ( pointStatus[ lineset->getLine( lineNr ).back() ] == MERGE && pointStatus[ lineset->getLine( lineNr ).front() ] != MERGE ) // beginning of
        new line
102 {
103     currLine = lineNr; // set current line
104     reverse = false;
105     newLine.clear();
106     newLine.push_back(lineset->getLine( lineNr ).front());
107 }
108 if (currLine != lineset->getNrLines())
109 {
110     while (true) // loop endlessly
111     {
112   if (done[ currLine ]) break;
113   nextLine = lineset->getNrLines();
114   newLine.reserve( newLine.size() + lineset->getNrPointsOfLine( currLine ) );
115   if (reverse)
116   {
117       for ( linePtIdx = lineset->getNrPointsOfLine( currLine ) - 1; linePtIdx > 0; linePtIdx-- ) // process line beginning from back
118       {
119     newLine.push_back( lineset->getLine( currLine )[ linePtIdx-1 ] );
120     if ( split && pointStatus[ lineset->getLine( currLine )[ linePtIdx-1 ] ] == SPLIT ) // split on split points if desired
121     {
122         if ( newLine.size() > 1 )
123         {
124       filteredLineSet->addLine(newLine);
125       newLine.clear();
126       newLine.push_back( lineset->getLine( currLine )[ linePtIdx-1 ] );
127         }
128     }
129       }
130       done[ currLine ] = true; // this line is done
131       if (pointStatus[ lineset->getLine( currLine ).front() ] == MERGE)
132       {
133     nextLine = linePairA[ lineset->getLine( currLine ).front() ];
134     if (nextLine == currLine) nextLine = linePairB[ lineset->getLine( currLine ).front() ];
135       } else {
136     nextLine = lineset->getNrLines();
137       }
138   } else {
139       for ( linePtIdx = 1; linePtIdx < lineset->getNrPointsOfLine( currLine ); linePtIdx++ )
140       {
141     newLine.push_back( lineset->getLine( currLine )[ linePtIdx ] );
142     if ( split && pointStatus[ lineset->getLine( currLine )[ linePtIdx ] ] == SPLIT )
143     {
144         if ( newLine.size() > 1 )
145         {
```

```
146       filteredLineSet->addLine(newLine);
147       newLine.clear();
148       newLine.push_back( lineset->getLine( currLine )[ linePtIdx ] );
149         }
150      }
151       }
152       done[ currLine ] = true;
153       if (pointStatus[ lineset->getLine( currLine ).back() ] == MERGE)
154       {
155      nextLine = linePairA[ lineset->getLine( currLine ).back() ];
156      if (nextLine == currLine) nextLine = linePairB[ lineset->getLine( currLine ).back() ];
157       } else {
158      nextLine = lineset->getNrLines();
159       }
160
161    }
162    if (nextLine != lineset->getNrLines())
163    {
164       if (linePairA[ lineset->getLine( nextLine ).front() ] == currLine)
165       {
166      reverse = false;
167       } else {
168      if (linePairB[ lineset->getLine( nextLine ).front() ] == currLine)
169      {
170         reverse = false;
171      } else {
172         reverse = true;
173      }
174       }
175       currLine = nextLine;
176       if (done[ currLine ])
177       {
178      break;
179       }
180    } else {
181       break;
182    }
183      }
184      if ( newLine.size() > 1 )
185      {
186    filteredLineSet->addLine(newLine);
187      }
188      newLine.clear();
189  }
190    }
191 }
192 newLine.clear();
193 // (4) special case: circles
194 progress.reset("merging lines", lineset->getNrLines());
195 for ( lineNr = 0; lineNr < lineset->getNrLines(); lineNr++ )
196 {
197    progress++;
198    if ( !done[ lineNr ] ) // if line was not processed yet
199    {
200  currLine = lineNr; // set current line
201  reverse = false;
202  newLine.clear();
203  newLine.push_back(lineset->getLine( lineNr ).back());
204  while (true) // loop endlessly
205  {
206       if (done[ currLine ]) break;
207       nextLine = lineset->getNrLines();
208       newLine.reserve( newLine.size() + lineset->getNrPointsOfLine( currLine ) );
209       if (reverse)
210       {
211    for ( linePtIdx = lineset->getNrPointsOfLine( currLine ) - 1; linePtIdx > 0; linePtIdx-- ) // process line beginning from back
212    {
213       newLine.push_back( lineset->getLine( currLine )[ linePtIdx-1 ] );
214       if ( split && pointStatus[ lineset->getLine( currLine )[ linePtIdx-1 ] ] == SPLIT ) // split on split points if desired
215       {
216       if ( newLine.size() > 1 )
```

```
217       {
218          filteredLineSet->addLine(newLine);
219          newLine.clear();
220          newLine.push_back( lineset->getLine( currLine )[ linePtIdx-1 ] );
221       }
222         }
223    }
224    done[ currLine ] = true; // this line is done
225    if (pointStatus[ lineset->getLine( currLine ).front() ] == MERGE)
226    {
227       nextLine = linePairA[ lineset->getLine( currLine ).front() ];
228       if (nextLine == currLine) nextLine = linePairB[ lineset->getLine( currLine ).front() ];
229    } else {
230       nextLine = lineset->getNrLines();
231    }
232      } else {
233    for ( linePtIdx = 1; linePtIdx < lineset->getNrPointsOfLine( currLine ); linePtIdx++ )
234    {
235       newLine.push_back( lineset->getLine( currLine )[ linePtIdx ] );
236       if ( split && pointStatus[ lineset->getLine( currLine )[ linePtIdx ] ] == SPLIT )
237       {
238      if ( newLine.size() > 1 )
239      {
240         filteredLineSet->addLine(newLine);
241         newLine.clear();
242         newLine.push_back( lineset->getLine( currLine )[ linePtIdx ] );
243      }
244        }
245    }
246    done[ currLine ] = true;
247    if (pointStatus[ lineset->getLine( currLine ).back() ] == MERGE)
248    {
249       nextLine = linePairA[ lineset->getLine( currLine ).back() ];
250       if (nextLine == currLine) nextLine = linePairB[ lineset->getLine( currLine ).back() ];
251    } else {
252       nextLine = lineset->getNrLines();
253    }
254      }
255      if (nextLine != lineset->getNrLines())
256      {
257    if (linePairA[ lineset->getLine( nextLine ).front() ] == currLine)
258    {
259       reverse = false;
260    } else {
261       if (linePairB[ lineset->getLine( nextLine ).front() ] == currLine)
262       {
263      reverse = false;
264       } else {
265      reverse = true;
266       }
267    }
268    currLine = nextLine;
269    if (done[ currLine ])
270    {
271       break;
272    }
273      } else {
274    break;
275       }
276  }
277  if ( newLine.size() > 1 )
278  {
279     filteredLineSet->addLine(newLine);
280  }
281  newLine.clear();
282    }
283 }
```

Quellcode 6: Linien verbinden

Abbildungsverzeichnis

Literatur

[1] Mark de Berg u. a. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008, S. 143–144. ISBN: 3540779736, 9783540779735.

[2] Thomas H Cormen. *Algorithmen: eine Einführung*. Oldenbourg Verlag, 2007.

[3] David H Douglas und Thomas K Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2 (1973), S. 112–122.

[4] Gerald E Farin. *Curves and surfaces for CAGD [electronic resource]: a practical guide*. Morgan Kaufmann, 2002.

[5] Gerald E Farin und Dianne Hansford. *The essentials of CAGD*. AK Peters Natick, 2000.

[6] James D Foley. *Computer graphics: Principles and practice, in C*. Bd. 12110. Addison-Wesley Professional, 1996.

[7] Stefan Kuhlins und Martin Schader. *Die C++-Standardbibliothek: Einführung und Nachschlagewerk*. Springer DE, 2005.

[8] Charles E Leiserson u. a. *Introduction to algorithms*. The MIT press, 2001.

[9] David Sujudi und Robert Haimes. *Identification of swirling flow in 3D vector fields*. Techn. Ber. Technical report, Department of Aeronautics und Astronautics, MIT, 1995. AIAA Paper 95-1715, 1995.

[10] Alexander Wiebel u. a. "FAnToM - Lessons Learned from Design, Implementation, Administration, and Use of a Visualization System for Over 10 Years". In: *Refactoring Visualization from Experience (ReVisE) 2009*. co-located with IEEE Visualization 2009. Atlantic City, NJ, USA, 2009.

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig	2. Januar 2014	
Ort	*Datum*	*Unterschrift*