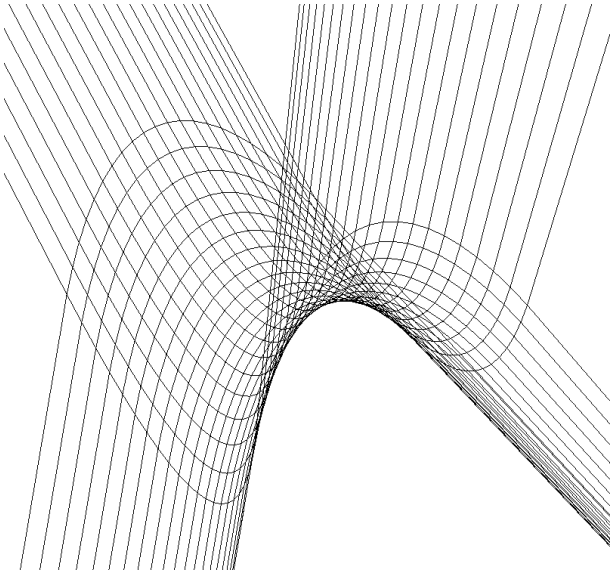
Rheinische Friedrich-Wilhelms-Universität Bonn

Berechnung und Visualisierung von Voronoi-Diagrammen in 3D



Vorgelegt von

Christoph Baudson

Edgar Klein

Erstgutachter:	Zweitgutachterin:
Prof. Dr. R. Klein	Prof. Dr. C. Baier

Danksagung

Wir danken unseren Eltern Margit und Norbert Baudson bzw. Juliane und Josef Klein.

Ein besonderer Dank geht an unsere Ehegattinnen

Marhamah Baudson und Tanja-Gabriele Klein

für Ihre Unterstützung und Geduld während der Erstellung dieser Arbeit.

Hiermit erklären wir, dass wir diese Arbeit selbstständig angefertigt haben und keine anderen als die angegebenen Hilfsmittel verwendet haben.

Weiterhin erklären wir, dass wir die entsprechend gekennzeichneten Teile dieser Arbeit jeweils selbstständig angefertigt haben und keine anderen als die angegebenen Hilfsmittel verwendet haben.

Bonn, 31. März 2006

Christoph Baudson Edgar Klein

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Die vorliegende Diplomarbeit behandelt die Berechnung und Visualisierung des Voronoi-Diagramms (VD) von in allgemeiner Lage befindlichen ungewichteten Punkten des dreidimensionalen Raumes. Ziel dieser Arbeit ist es, ein online lauffähiges Programm[1] anzubieten, das dem Benutzer durch Interaktion mit den dargestellten Geometrien eine bessere Vorstellung über das Diagramm vermittelt. Konkret geht es dabei um die folgenden Punkte: Berechnung und Visualisierung des VDs, Benutzbarkeit des Programms mittels einer intuitiven Oberfläche und Erweiterbarkeit der Algorithmen bzw. Wiederverwendbarkeit der erstellten Pakete respektive Klassen.

Die grobe Arbeitsweise des Algorithmus ist zunächst die inkrementelle Konstruktion der Delaunay-Triangulation (DT) mit dem Algorithmus von Edelsbrunner und Shah [16]. Während der Berechnung der DT wird die Datenstruktur *Augmented Quad Edge* (AQE) aufgebaut, die ihrerseits sowohl die Delaunay-Triangulation als auch ihr Duales, das Voronoi-Diagramm, beinhaltet. Sie bildet die Grundlage für die Darstellung der Geometrien mittels Java3D.

Der von uns gewählte Algorithmus zur Berechnung der Delaunay-Triangulation benötigt im *worst case* $\Omega(n^2)$ Laufzeit und Speicherplatz und bei einer gleichverteilten Punktmenge $O(n \log n)$ erwartete Laufzeit bzw. $O(n)$ erwarteten Speicherplatz. Die Visualisierung erfolgt schließlich mit Hilfe eines Durchlaufs über die AQE-Datenstruktur, deren Laufzeit linear zur Komplexität der DT ist.

[1] Das Programm ist im Internet unter `www.voronoi3d.com` zu finden.

Einleitung

Wo befindet sich der nächste bekannte bewohnbare Planet, den ein Raumschiff ansteuern kann? Wie kann das Wachstum von Kristallen simuliert werden? Wie findet ein Roboter den kürzesten Weg zwischen zwei Anfragepunkten in einem dreidimensionalen Labyrinth? Solche und ähnliche nicht nur informatische, sondern auch in der Natur auftretende Fragestellungen können durch Voronoi-Diagramme elegant und effizient gelöst werden.

Die Geschichte der Forschung zu Voronoi-Diagrammen beginnt Mitte des 19. Jahrhunderts (vgl. Aurenhammer [1]). Trotz einer großen Anzahl an potentiellen Anwendungsgebieten kristallisierten sich insbesondere drei Bereiche heraus: die Modellierung natürlicher Vorkommnisse, die mathematische Untersuchung ihrer geometrischen und stochastischen Eigenschaften und die Repräsentation und Konstruktion von Voronoi-Diagrammen mit Hilfe von Computern.

Diese Arbeit setzt an dem letzten Punkt der geschichtlichen Entwicklung von Voronoi-Diagrammen an. Zur Zeit existiert kein Programm, das dieses geometrische Konstrukt im Raum schrittweise berechnet, visualisiert und das außerdem im Internet frei verfügbar ist. Diese Arbeit hat nun zum Ziel, nicht nur das Voronoi-Diagramm, sondern auch die Delaunay-Triangulation inkrementell zu berechnen und zu visualisieren. Für die Visualisierung stellen wir eine Datenstruktur vor, die beide Geometrien gleichzeitig vorhält. Um Plattformunabhängigkeit und die Verfügbarkeit des Programms im Internet zu gewährleisten, wählten wir als Programmiersprache Java und die Java3D-Klassenbibliothek.

Im theoretischen Teil der Arbeit erfolgt zunächst eine Vorstellung etablierter Bibliothe-

ken zur Berechnung und Visualisierung von Voronoi-Diagrammen. Weiterhin werden vorhandene Algorithmen und Datenstrukturen für Delaunay-Triangulationen und Voronoi-Diagramme vorgestellt, um anschließend den von uns gewählten Algorithmus von Edelsbrunner und Shah [16] zu beschreiben und sein Laufzeitverhalten und seinen Speicherplatzbedarf zu analysieren.

Die genaue Umsetzung der gewählten Algorithmen und Datenstrukturen einschließlich UML-Diagramme, Pakete mit ihren Klassen und deren Variablen und Methoden behandelt der Implementierungsteil dieser Arbeit, in dem die theoretischen Überlegungen aus dem ersten Teil der Arbeit praktisch umgesetzt werden.

Der dritte Teil der Arbeit fasst die erzielten Ergebnisse zusammen und gibt mögliche Ausblicke dieser Resultate bzw. Erweiterungen des Programms.

Im Anhang finden sich die Beschreibung der Programmoberfläche, Hintergrundinformationen zu Java3D, ein Exkurs über Voronoi-Diagramme auf der Kugeloberfläche und die Information, wer welche Teile der Arbeit erstellt hat.

Teil I.

Theorie

1. Motivation

Voronoi-Diagramme sind vielseitig anwendbar und finden dadurch in zahlreichen Gebieten der Forschung Einsatz. Aurenhammer [1] nennt folgende Gründe, weshalb sie in so vielen Bereichen Anklang finden: Zum einen tauchen Voronoi-Diagramme in der Natur in verschiedensten Situationen auf; ferner weisen sie interessante mathematische Eigenschaften und Beziehungen zu anderen geometrischen Strukturen auf, und schließlich können sie auch als mächtige Werkzeuge zur Lösung scheinbar unabhängiger Aufgabenstellungen angewandt werden. Diese drei Bereiche sollen im Folgenden etwas genauer illustriert werden.

1.1. Natürliche Phänomene

Ein natürlich vorkommendes Voronoi-Diagramm kann man sich anhand folgender Überlegung vorstellen: Nehmen wir an, das Weltall sei in kristallförmige Gebilde unterteilt, die aus ihren Mittelpunkten mit derselben Geschwindigkeit und zum gleichen Zeitpunkt zu wachsen beginnen. Diese Partitionierung des Sonnensystems [24] entspricht dem Modell von Descartes (siehe Abbildung 1.1), das zwar in der modernen Astronomie als obsolet gilt, in der Kristallographie hingegen bis heute Anwendung findet (vgl. Aurenhammer [1]). Diese Wissenschaft war es im übrigen auch, die die frühen Arbeiten zu Voronoi-Diagrammen motivierte.

In den Naturwissenschaften können Voronoi-Diagramme ebenfalls zur Lösung verschie-

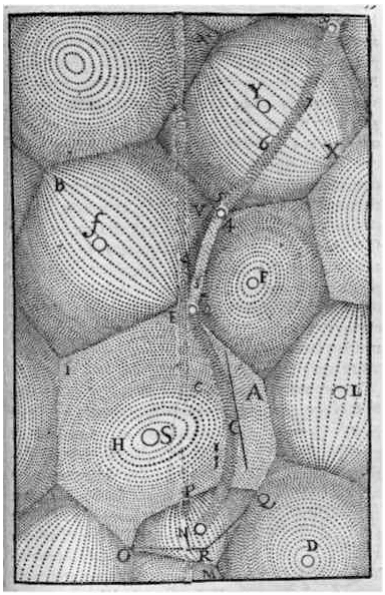


Abbildung 1.1.: Zerlegung des Sonnensystems in Wirbel nach Descartes.

denster Probleme beitragen (vgl. Aurenhammer [1]): Ein physikochemisches Modell, wie beispielsweise das in der Metallurgie verwendeten Wigner-Seitz-Zonen-Modell, besteht aus einer Anzahl von Molekülen, Ionen oder Atomen. Unter anderem determiniert das Gleichgewicht dieser Teilchen, wie der Raum zwischen den einzelnen Modellelementen aufgeteilt ist. Ein anderes Beispiel stammt von Thiessen, einem Klimaforscher: Dieser schätzte den Niederschlag für große Gebiete ab, indem er einzelnen Messpunkten Polygone zuordnete und diese in Nachbarschaftsbeziehung zu anderen Polygonen setzte. Im Rahmen von Problemen der Mustererkennung setzte Blum bereits 1967 die Idee der Nachbarschaftsbeziehungen von Punkten für die Modellierung bzw. Erkennung von biologischen Formen ein, die bestimmte Bereiche definieren.

1.2. Mathematische Sichtweise

Wie oben bereits kurz erwähnt, weisen Voronoi-Diagramme einige mathematisch interessante Eigenschaften auf. An dieser Stelle konzentrieren wir uns insbesondere auf diejenigen

Charakteristika, die für die Lösung unserer Aufgabe von besonderem Interesse sind. Diese werden in den folgenden Abschnitten genauer erläutert.

Zerlegung des Raumes in konvexe Polyeder: Das Voronoi-Diagramm stellt eine konvexe Zerlegung des Raumes dar; dies folgt aus der Definition, auf die wir in Kapitel 2.1 noch im Detail eingehen werden. Aus der Eigenschaft der Konvexität ergibt sich, dass nicht alle Voronoi-Regionen beschränkt sein können.

Beziehung zur konvexen Hülle: Die unbeschränkten Voronoi-Regionen weisen die interessante Beziehung zur konvexen Hülle auf, dass ein Punkt $p \in S$ genau dann eine unbeschränkte Voronoi-Region in $V(S)$ besitzt, wenn er auf dem Rand der konvexen Hülle von S liegt.

Dualität zur Delaunay-Triangulation[1]: Die Dualitätseigenschaft der Delaunay-Triangulation mit dem Voronoi-Diagramm (vgl. Kapitel 2.3) ist für die Lösung unserer Aufgabe essentiell: Sie ermöglicht es, die beiden Geometrien ineinander zu überführen. Somit genügt es, nur eines der beiden zu berechnen, um das andere automatisch zu erhalten.

1.3. Anwendung des Voronoi-Diagramms

1.3.1. Theorie

Das Voronoi-Diagramm kann als Hilfsmittel zur Lösung verschiedenartiger Probleme verwendet werden. Wir konzentrieren uns in diesem Abschnitt exemplarisch auf konkrete Distanzprobleme[2]; eine Betrachtung weiterer Aufgabenstellungen liefert Aurenhammer [1].

[1] Die Delaunay-Triangulation stellt eine winkelmaximierende Triangulation einer gegebenen Punktmenge dar.

[2] Für eine genaue Beschreibung ähnlicher Aufgabenstellungen verweisen wir auf Klein [24].

Nächster-Nachbar-Problem: Gegeben sei eine Menge S von n Punkten, deren Position durch ihre kartesischen Koordinaten im $\mathbb{R}^3$ eindeutig bestimmt ist. Gesucht ist nun derjenige Punkt, der zu einem Anfragepunkt p am nächsten gelegen ist. Um die Aufgabe zu lösen, genügt es, diejenige Voronoi-Region zu finden, in der sich p befindet[3].

Alle-Nächsten-Nachbarn-Problem: Gesucht ist zu jedem in S gegebenen Punkt der jeweilige nächste Nachbar. Zur Lösung der Aufgabe wird die Eigenschaft des Voronoi-Diagramms benötigt, dass für jeweilige nächste Nachbarn auch ihre Voronoi-Regionen benachbart sind.

Größte-Leere-Sphären-Problem: Gegeben sei ein konvexes Polyeder A mit m Ecken und eine Punktmenge S. Gesucht ist nun die größte Sphäre, die ihr Zentrum in A hat und keinen der Punkte aus S enthält. Der gesuchte Mittelpunkt x, der die Aufgabe löst, ist ein Voronoi-Knoten[4] von $V(S)$ innerhalb von A, ein Schnittpunkt einer Voronoi-Kante bzw. -Fläche mit einer Fläche bzw. Kante des Gebietes A oder ein Eckpunkt von A.

1.3.2. Praxis

Praktische Anwendung finden dreidimensionale Voronoi-Diagramme und Delaunay-Triangulationen beispielsweise in der Bioinformatik: Dort können sie zur Analyse und Modellierung von Proteinstrukturen eingesetzt werden. Exemplarisch für diesen Forschungszweig sei hier in vereinfachter Form ein Teilaspekt einer solchen Strukturanalyse vorgestellt; eine detaillierte Diskussion dieses Anwendungsbereichs liefert Poupon [30].

Proteine sind Makromoleküle, die u. a. aus Kohlenstoff-, Wasserstoff-, Sauerstoff- und Stickstoffatomen aufgebaut sind. Diese Atome werden bei einer Strukturanalyse durch eine Punktmenge repräsentiert, deren Delaunay-Triangulation berechnet wird. Die Sphären,

[3] Diese Herangehensweise wird als *Ortsansatz*, im Englischen *locus approach*, bezeichnet.

[4] Eine Definition der Begrifflichkeiten zu Voronoi-Diagrammen folgt in Kapitel 2.1.

die die Tetraeder dieser Triangulation umgeben, sind leere Kugeln, d. h. kein anderes Atom ist in ihnen enthalten. Die Größe und Position dieser leeren Räume sind interessante Charakteristika von Proteinen: So können die Radien dieser Sphären als Maß für die Dichte des Proteins verwendet werden. Durch gewichtete Voronoi-Diagramme (siehe Kapitel 2.6.1) lässt sich außerdem das Volumen der Atome modellieren.

Ein weiteres Anwendungsgebiet für Voronoi-Diagramme im dreidimensionalen Raum stellt die meereskundliche Datenerfassung (siehe Ledoux und Gold [26]) dar. Hier stellt sich das besondere Problem, dass die verschiedenen Untersuchungsobjekte (repräsentiert als Punkte im Voronoi-Diagramm) nicht statisch sind, sondern ihre Standorte im Laufe der Zeit dynamisch verändern. Gegenüber globalen Ansätzen der marinen Datenerfassung, die bei lokalen Änderungen der Messdaten die gesamte Datenstruktur neu berechnen müssen, bieten mittels inkrementeller Algorithmen berechnete Voronoi-Diagramme den Vorteil, dass sie nur lokale Änderungen erfordern.

2. Theoretische Grundlagen

Im Folgenden sollen nun die mathematischen Grundlagen, die wir im vorangegangenen Kapitel z. T. bereits knapp angesprochen haben, im Detail erläutert werden. Von besonderem Interesse ist an dieser Stelle die Beziehung zwischen Voronoi-Diagramm, Delaunay-Triangulation und konvexer Hülle; hierzu sollen diese Strukturen zunächst definiert werden.

2.1. Definition: Voronoi-Diagramm im $\mathbb{R}^3$

Das *Voronoi-Diagramm* V einer Punktmenge $S \subset \mathbb{R}^3$ weist jedem Punkt $p \in S$ eine Menge von Punkten zu, die näher an p als an allen übrigen Punkten aus S liegen. Klein [24] liefert eine formale Definition; zu diesem Zweck führt er zunächst den Bisektor ein.

Unter dem *Bisektor* zweier Punkte p und q verstehen wir die Menge $B(p, q)$ von Punkten, die zu p und q den gleichen Abstand haben, d. h.

$$B(p,q) = \left\{ x \in \mathbb{R}^3 \mid d(p,x) = d(q,x) \right\};$$

dabei ist $d(p, q)$ der euklidische Abstand

$$d(p,q) = L_2(p,q) = |pq| = |p-q| = \sqrt{\sum_{i=1}^{3} (p_i - q_i)^2}.$$

Der Bisektor teilt den $\mathbb{R}^3$ in zwei offene Halbräume $D(p, q)$ und $D(q, p)$ auf. Diese sind wie folgt definiert:

$$D(p,q) = \left\{ x \in \mathbb{R}^3 \mid d(p,x) < d(q,x) \right\},$$

$$D(q,p) = \left\{ x \in \mathbb{R}^3 \mid d(p,x) > d(q,x) \right\}.$$

Wir bezeichnen

$$VR(p,S) = \bigcap_{q \in S \setminus \{p\}} D(p,q).$$

als *Voronoi-Region* von p bzgl. S.

Demnach sind Voronoi-Regionen konvex, offen und bilden eine Partition des Raumes. Damit folgt sofort, dass nicht alle Voronoi-Regionen beschränkt sein können: Genau diejenigen Punkte, die auf der konvexen Hülle $ch(S)$ (das sind die Punkte, die zu der kleinsten konvexen S umschließenden Teilmenge des $\mathbb{R}^3$ beitragen) liegen, besitzen unbeschränkte Voronoi-Regionen, denn nur für einen Punkt $p \in ch(S)$ existieren andere Punkte im $\mathbb{R}^3$, die unendlich weit von p entfernt liegen und trotzdem p als nächsten Nachbarn haben.

Als *Voronoi-Diagramm* bezeichnet man den $\mathbb{R}^3$ ohne die Vereinigung der Voronoi-Regionen, d. h.

$$V(S) := \mathbb{R}^3 \setminus \{ VR(p_i) \mid p_i \in S, 1 \leq i \leq n \}.$$

Sind die Punkte aus S in *allgemeiner Lage*, d. h. sind keine drei Punkte aus S kollinear, keine vier Punkte koplanar und keine fünf Punkte aus S auf einer gemeinsamen Kugeloberfläche, so ergeben sich folgende Vereinfachungen: Es gibt keinen Punkt $x \in \mathbb{R}^3$ mit fünf oder mehr nächsten Nachbarn in S. Besitzt x genau vier nächste Nachbarn aus S, bezeichnen wir x als *Voronoi-Knoten*. Alle Punkte x mit genau drei gleichen nächsten Nachbarn aus S sind kollinear und bilden eine *Voronoi-Kante*. Alle Punkte x mit genau zwei gleichen nächsten Nachbarn aus S sind koplanar und bilden eine konvexe *Voronoi-Fläche*.

2.2. Definition: Delaunay-Triangulation im $\mathbb{R}^3$

Unter einer *Triangulation* T einer Punktmenge $S \subset \mathbb{R}^3$ in allgemeiner Lage versteht man eine Partition der konvexen Hülle von S in Tetraeder, wobei jedes Tetraeder aus vier

verschiedenen Punkten aus S besteht und der Schnitt zweier beliebiger Tetraeder eine Dreiecksfläche aus T, eine Kante aus T, ein Punkt aus S oder leer ist (Edelsbrunner [16]).

Eine *Delaunay-Triangulation* (oder *reguläre Triangulation*) DT einer Punktmenge S ist eine Triangulation, in der für jedes Tetraeder die *Regularitätsbedingung* erfüllt ist. Diese Bedingung besagt, dass die kleinste ein Tetraeder aus DT umgebende Sphäre (die Umkugel des Tetraeders) keine anderen Punkte aus S als die Eckpunkte dieses Tetraeders enthalten darf. Die Knoten der Triangulation nennt man *Delaunay-Knoten*, die Kanten *Delaunay-Kanten* und die Dreiecksflächen der Tetraeder *Delaunay-Flächen* (oder *Delaunay-Dreiecke*).

Die Delaunay-Triangulation stellt eine winkelmaximierende Triangulation der gegebenen Punktmenge S dar. Aus diesem Grund ist die Delaunay-Triangulation DT_S eindeutig.

2.3. Dualität von Voronoi-Diagramm und Delaunay-Triangulation

Befindet sich S in allgemeiner Lage, so ist das Voronoi-Diagramm V einer Punktmenge S dual zu deren Delaunay-Triangulation DT: Jeder Region aus V entspricht genau ein Knoten aus DT, jeder Voronoi-Fläche aus V genau eine Kante aus DT, jeder Kante aus V genau eine Delaunay-Fläche aus DT und jedem Knoten aus V genau ein Tetraeder aus DT. Die Komplexität der Delaunay-Triangulation ist demnach gleich der des Voronoi-Diagramms.

Aus der Annahme der allgemeinen Lage der Punktmenge folgt für das Voronoi-Diagramm, dass alle Voronoi-Knoten vier inzidente Voronoi-Kanten haben: Ein Delaunay-Tetraeder ist dual zu einem Voronoi-Knoten, und jedes zu diesem Tetraeder gehörige Delaunay-Dreieck ist dual zu einer aus dem Voronoi-Knoten ausgehenden Voronoi-Kante. Ebenso haben alle Voronoi-Kanten drei inzidente Voronoi-Flächen, da eine Voronoi-Kante dual zu einem Delaunay-Dreieck ist, das aus drei Kanten besteht, wobei jede wiederum dual zu

einer Voronoi-Fläche ist (siehe Abbildung 2.1). Ebenso besitzt jede Voronoi-Fläche zwei inzidente Voronoi-Regionen, denn eine Voronoi-Fläche entspricht einer Delaunay-Kante, die zwei Knoten miteinander verbindet, die wiederum jeweils dual zu einer Voronoi-Region sind.

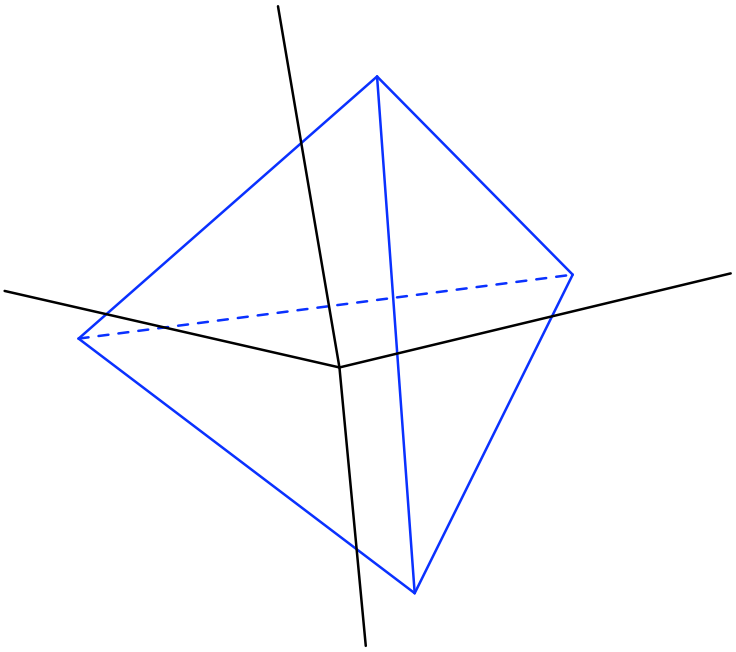


Abbildung 2.1.: Dualität zwischen Delaunay-Triangulation und Voronoi-Diagramm.

Um das Voronoi-Diagramm zu einer Punktmenge zu berechnen, genügt es aufgrund der Dualität, die Delaunay-Triangulation zu bestimmen. Daraus lässt sich das Voronoi-Diagramm in linearer Zeit zur Komplexität der Delaunay-Triangulation ableiten.

2.4. Konvexe Hülle im $\mathbb{R}^{d+1}$ und Delaunay-Triangulation im $\mathbb{R}^d$

Die Konstruktion der Delaunay-Triangulation einer Punktmenge $S \subset \mathbb{R}^d$ lässt sich auf die Berechnung der konvexen Hülle einer Punktmenge $S^+ \subset R^{d+1}$ zurückführen: Zu einem

Punkt $p = (p_1, p_2, \ldots, p_d) \in \mathbb{R}^d$ definieren wir $p^+ = (p_1, p_2, \ldots, p_d, p_{d+1}) \in \mathbb{R}^{d+1}$, wobei $p_{d+1} = \sum_{i=1}^{d} p_i^2$ ist; für eine Punktmenge $S \subset \mathbb{R}^d$ ist demnach $S^+ = \{p^+ | p \in S\}$. Für $d = 2$ entspricht S^+ der auf einen Paraboloiden gelifteten Punktmenge S.

Ein d-Simplex der Delaunay-Triangulation von S entspricht dann der vertikalen Projektion eines d-Simplexes der unteren Kontur von S^+; dabei verstehen wir unter einem *d-Simplex* die konvexe Hülle einer Menge von $k + 1$ $(0 \leq k \leq d)$ affin unabhängigen Punkten des $\mathbb{R}^d$.

Betrachten wir die Regularitätsbedingung für Delaunay-Triangulationen im $\mathbb{R}^2$: Ein Dreieck $tria_{\{a,b,c\}}$ ist regulär, falls kein weiterer Punkt q in seinem Umkreis liegt. Eine äquivalente Behauptung ist, dass kein Punkt q^+ unterhalb der Ebene liegt, die durch das Dreieck $tria^+_{\{a,b,c\}}$ der unteren Kontur definiert ist, welches dem Dreieck in der Ebene entspricht. Diese Regularitätsbedingung kann auf beliebige Dimensionen erweitert werden (siehe Edelsbrunner [16]).

Die Konstruktion des Voronoi-Diagramms im $\mathbb{R}^d$ ist demnach mindestens so aufwändig wie die Berechnung der konvexen Hülle im $\mathbb{R}^{d+1}$.

2.5. Komplexität der Delaunay-Triangulation

Wie zuvor erwähnt, hat das Voronoi-Diagramm dieselbe Komplexität wie die Delaunay-Triangulation. Nun kann man zeigen, dass es Punktmengen der Größe n im $\mathbb{R}^3$ gibt, deren Delaunay-Triangulation aus $\Omega(n^2)$ Tetraedern besteht (siehe Abbildung 2.3); jede Voronoi-Region teilt dabei mit $\Omega(n)$ anderen eine Fläche. Beispiele für Punktmengen, deren Delaunay-Triangulationen eine Komplexität von $\Omega(n^2)$ haben, finden sich bei Edelsbrunner [14] oder bei Dewdney und Vranch [12]. Wir zeigen hier die von Edelsbrunner gegebene Punktkonstellation, bei der sich jeweis $\frac{n}{2}$ Punkte auf zwei windschiefen Geraden (siehe Abbildung 2.2) befinden. Bei einer solchen Punktmenge liegt auch die Laufzeit in $\Omega(n^2)$.

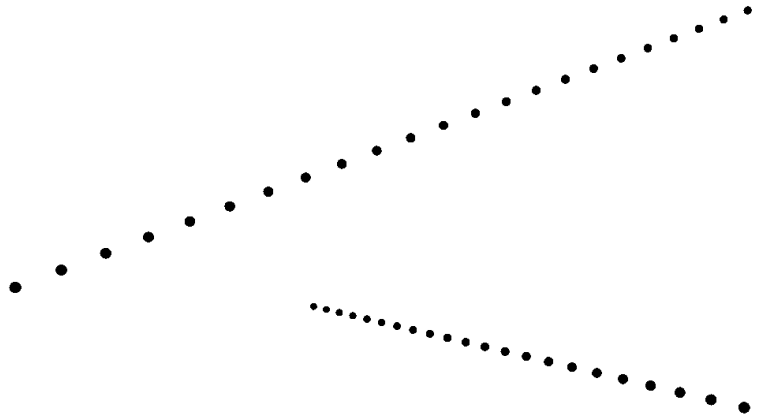


Abbildung 2.2.: Punkte auf zwei windschiefen Geraden.

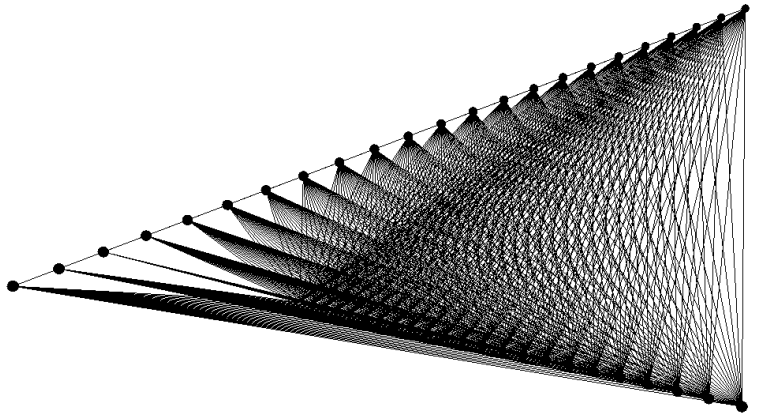


Abbildung 2.3.: Delaunay-Triangulation der Punktkonstellation aus Abbildung 2.2.

Wählt man jedoch eine zufällige, in der Einheitskugel gleichverteilte Punktmenge, so zeigt Dwyer [13], dass die erwartete Anzahl an Delaunay-Tetraedern in $\Theta(n)$ liegt:

Satz 2.5.1 *Sei $S = \{p_1, \ldots, p_n\}$ eine Menge von n gleichverteilten Punkten im Inneren der Einheitssphäre. Dann ist $ES_n \in \Theta(n)$, mit ES_n Erwartungswert für die Anzahl der Simplizes (Tetraeder der Delaunay-Triangulation).*

Beweisskizze: Zunächst berechnet man die erwartete Anzahl von Simplizes in der Delaunay-Triangulation bei einer (beliebigen) zufälligen Punktverteilung. Mit diesem Zwischenergebnis kann nun der Erwartungswert auf eine gleichverteilte Punktmenge in der

Einheitssphäre angewandt werden.

Die Anwendung des berechneten Erwartungswertes für $d = 2$ ergibt $ES_n \sim 2n$; für $d = 3$ erhalten wir $ES_n \sim 6.77n$, und für $d = 4$ ist $ES_n \sim 31.78n$. Diese Werte sind mit den Ergebnissen von Avis und Bhattacharya [2] vereinbar, die ihre Tests aber nur für relativ kleine Punktmengen der Kardinalität 1.000 innerhalb des Einheitswürfels durchführten.

2.6. Erweiterungen

In diesem Abschnitt behandeln wir zwei Erweiterungen des Voronoi-Diagramms: Voronoi-Diagramme mit beliebigen Distanzfunktionen und Voronoi-Diagramme höherer Ordnung.

2.6.1. Voronoi-Diagramme mit beliebigen Distanzfunktionen

Als erste Erweiterung ordnen wir jedem Punkt $p \in S \subset \mathbb{R}^3$ eine Abstandsfunktion

$$d_p : \mathbb{R}^3 \longrightarrow \mathbb{R}$$

zu, der für jeden Punkt $x \in \mathbb{R}^3$ einen Abstand $d_p(x)$ zu p definiert. Nun kann eine Voronoi-Region für beliebige Abstandsfunktionen d_p wie folgt definiert werden (s. Klein [24]):

$$VR\left(\{p, S\}\right) := \left\{x \in \mathbb{R}^3 \mid f_p(x) < f_q(x),\ \forall q \in S\right\}.$$

Wählen wir nun für d_p den Euklidischen Abstand, so erhalten wir unsere Definition aus Kapitel 2.1.

Edelsbrunner und Shah [16] verwenden für die Beschreibung ihres Algorithmus eine *gewichtete Distanzfunktion* $\pi_p(x)$ von x nach p, mit

$$d_p(x) := \pi_p(x) := |xp|^2 - w_p,$$

wobei jedem Punkt $p \in S$ ein additives Gewicht $w_p \in \mathbb{R}$ zugeordnet wird. Wählt man die Gewichte für alle Punkte gleich, so erhalten wir wieder unsere Ursprungsdefinition vom Anfang des Kapitels.

Anschaulich gesehen gibt das additive Gewicht w_p eines Punktes p die Größe des Punktes an, d. h. denjenigen Vorsprung zum Bisektor, den p gegenüber einem anderen Punkt q mit Gewicht w_q hat. Eine Anwendung wäre die in Kapitel 1.3.2 angesprochene Proteinstrukturanalyse; die Gewichte können da das Volumen der Atome simulieren.

2.6.2. Voronoi-Diagramme höherer Ordnung

Als zweite Erweiterung der Definition von Voronoi-Diagrammen betrachten wir nun nicht nur den unmittelbar nächsten Nachbarn, sondern die ersten k nächsten Nachbarn einer endlichen Punktmenge $S = \{p_1, \ldots, p_n\}$. Dann definieren wir für die Punkte $p_1, \ldots, p_k \in S$ die *Voronoi-Region k-ter Ordnung*:

$$VR\left(\{p_1, \ldots, p_k\}, S\right) := \left\{x \in \mathbb{R}^3 \mid \forall 1 \leq i \leq k \; \forall k+1 \leq j \leq n : d(p_i, x) < d(p_j, x)\right\},$$

d. h. $p_1, \ldots, p_k$ sind die k nächsten Nachbarn in S. Analog zu vorher definieren wir damit das *Voronoi-Diagramm k-ter Ordnung*[1] und kürzen dieses mit $V^k(S)$ ab. Für $k = n - 1$ nennen wir $V^{n-1}(S)$ das *furthest point-Voronoi-Diagramm* oder entsprechend *furthest site-Voronoi-Diagramm* und kürzen dieses auch mit $V^{-1}(S)$ ab. Wir schreiben $VR^{-1}(p)$ für die *furthest point-Voronoi-Region* eines Punktes p.

Im Gegensatz zu Voronoi-Regionen erster Ordnung, die nicht leer sein können, gilt Folgendes für *furthest point*-Voronoi-Regionen:

$$p \notin ch\,(S) \Rightarrow VR^{-1}\,(p, S) = \emptyset,$$

d. h. für alle Punkte $p \in S$, die nicht zur konvexen Hülle von S beitragen, sind deren Voronoi-Regionen leer.

Anwendung finden *furthest point*-Voronoi-Diagramme zum Beispiel bei der Berechnung von Voronoi-Diagrammen auf der Kugeloberfläche, wie wir in Anhang C.3.1 beschreiben werden.

[1] Diese Definition ist sinnvoll, denn Voronoi-Diagramme erster Ordnung sind konsistent mit der Definition des Voronoi-Diagramms aus Kapitel 2.1.

3. Softwarepakete zur Berechnung und Visualisierung von Voronoi-Diagrammen

Bevor wir unsere Implementierung vorstellen, präsentieren wir an dieser Stelle bereits vorhandene Geometriesoftwarepakete zur Berechnung und Visualisierung von Voronoi-Diagrammen. Wir möchten sie hier unter den Gesichtspunkten Entstehung, Funktionsumfang, Kosten, Lizenzierung und Dokumentation betrachten, bevor wir im anschließenden Kapitel näher auf die darin verwendeten Algorithmen und Datenstrukturen eingehen.

Wir stellen zuerst drei der gängigsten Geometrie-Bibliotheken vor, die die Berechnung von Voronoi-Diagrammen im Raum beherrschen: CGAL, QHull und LEDA. Da diese Bibliotheken nicht zwingend eine Visualisierungskomponente enthalten, präsentieren wir darüber hinaus zwei Programme, die die Möglichkeit der Darstellung und Interaktion mit den Geometrien bieten: zum einen GeomView, das mit CGAL und QHull erstellte Voronoi-Diagramme darstellen kann, zum anderen Cinderella, das eine benutzerfreundliche Manipulation einfacher Geometrien erlaubt.

3.1. CGAL

CGAL (*Computational Geometry Algorithms Library*) [7] ist im Jahr 1998 aus der Zusammenarbeit mehrerer Universitäten und wissenschaftlicher Institute aus Europa und Israel entstanden und versteht sich als eine Programmbibliothek, die Nutzern aus Wissenschaft und Wirtschaft Zugang zu den wichtigsten geometrischen Algorithmen und Datenstrukturen gewähren will.

Die Bibliothek besteht aus drei Bereichen von C++-Klassen: dem Kernel, Basisalgorithmen und Erweiterungen. Algorithmen zu Voronoi-Diagrammen im Raum sind in der zur Zeit aktuellen Version 3.1 nicht implementiert. Zur Delaunay-Triangulation in der Ebene und im Raum findet sich eine Lösung in den Basisalgorithmen, der eine schnelle kombinatorische Datenstruktur zu Grunde liegt; diese werden wir in Kapitel 4 genauer betrachten. CGAL selbst bietet keine eigenen grafischen Ausgaben oder Benutzeroberflächen, lediglich die Möglichkeit des Datenexports in weiterverwendbare Dateiformate. Zur Visualisierung bietet sich z.B. das Programm GeomView [17] an, das wir in Kapitel 3.4 beschreiben.

Seit der Version 3.0 aus dem Jahr 2003 ist CGAL quelloffen. Der Kernel und die Erweiterungen unterliegen der LGPL-Lizenz (*GNU Lesser General Public License*) [18], sind also freie Software, während für die Basisalgorithmen die QPL (*Q Public Licence*) [19] gilt, die nicht mit der GNU GPL kompatibel ist: dies hat zur Folge, dass man die CGAL-Algorithmen nur als Bibliothek verwenden, nicht aber mit anderen GPL-kompatiblen Programmen verschmelzen kann (vgl. [20]).

Die Dokumentation von CGAL besteht aus einem umfangreichen Benutzerhandbuch, das die Klassen technisch ausführlich dokumentiert und die Algorithmen dabei kurz erläutert. Der Quelltext selbst ist nur spärlich kommentiert.

3.2. QHull

QHull ist ein Programm zur Berechnung konvexer Hüllen und basiert auf dem QuickHull-Algorithmus von Barber et. al. [3]. Da die Berechnung der konvexen Hülle im $\mathbb{R}^{d+1}$ der der Delaunay-Triangulation im $\mathbb{R}^d$ entspricht, ist auf diesem Weg auch eine Berechnung von Voronoi-Diagrammen möglich. QHull wurde seit dem Jahr 1993 am Geometriezentrum der University of Minnesota mit Unterstützung der Minnesota Technology Inc. entwickelt. Verwendung finden die Algorithmen inzwischen z.B. in MATLAB oder Mathematica.

QHull beinhaltet mit „qvoronoi“ einen Algorithmus zur Berechnung von Voronoi-Diagrammen in 3D. Laut Angaben der Autoren bewältigt dieser auch höhere Dimensionen sowie *furthest site*-Voronoi-Diagramme (siehe Kapitel 2.6). Auf die in QHull verwendeten Algorithmen und Datenstrukturen gehen wir in Kapitel 4 genauer ein. Ähnlich wie bei CGAL wird ein externes Programm zur Visualisierung benötigt, etwa das schon erwähnte GeomView [17].

Auch QHull ist in C++ geschrieben. Die Software ist urheberrechtlich geschützt, aber quelloffen und darf laut Angaben der Autoren frei benutzt und verändert werden (vgl. QHull Copyrights [32]).

Die Dokumentation in Form eines Benutzerhandbuchs ist sehr knapp. Der Quelltext ist zwar ausführlich dokumentiert, ein grundlegendes Verständnis der Algorithmen wird dadurch aber kaum vermittelt.

3.3. LEDA

LEDA [25] steht für *Library of Efficient Data Types and Algorithms*. Die Bibliothek wurde ab 1988 von Mitarbeitern des Max-Planck-Instituts in Saarbrücken entwickelt. Aus diesem Entwicklerkreis entstand 1995 die Algorithmic Solutions Software GmbH, die LEDA seitdem kommerziell vertreibt.

Die Software, die ebenso in C++ programmiert wurde, beherrscht neben einer Vielzahl geometrischer Algorithmen und Datenstrukturen auch Voronoi-Diagramme im $\mathbb{R}^3$.

LEDA besteht aus mehreren Komponenten wie Datenstrukturen, Geometrien oder Benutzeroberflächen, die in unterschiedlichen Kombinationen als Pakete erhältlich sind. Informationen zu verwendeten Algorithmen und Datenstrukturen werden jedoch nicht gegeben.

Die Pakete unterscheiden sich jedoch nicht nur hinsichtlich der Komponenten, die sie beinhalten, sondern auch in den implizierten Nutzungsrechten: Die Spanne reicht von Paketen für Forschungs- und Lehrzwecke zu Preisen ab 49 EUR bis hin zu kommerziell nutzbaren Paketen bis 19.660 EUR, die auch die Quellcodemanipulation erlauben. Zudem gibt es eine Evaluationsvariante, die für 30 Tage gültig ist und die die Datenstruktur-, Geometrie- und Benutzeroberflächenkomponente enthält, allerdings nur mit einem sehr eingeschränkten Blick in die Quelldateien, die zudem noch unkommentiert sind (Stand: März 2006).

3.4. GeomView

GeomView ist ein 2D- und 3D-Visualisierungsprogramm für Unix-Systeme und ist (ebenso wie QHull) am Geometriezentrum der University of Minnesota zwischen 1992 und 1996 entstanden. Auch wenn das Institut inzwischen geschlossen ist, wird das Programm immer noch von Tausenden von Benutzern weltweit angewandt. Mithilfe der freiwilligen Arbeit der ursprünglichen Autoren wird die Software auch weiterhin verbessert und erneuert.

Der Benutzer kann Objekte mit der Maus rotieren, verschieben und skalieren und auf diese Weise die 3D-Szene manipulieren. In der aktuellen Version 1.8 kann GeomView als eigenständiges Programm statische Objekte anzeigen oder als Darstellungswerkzeug für andere Programme dienen, die selbst dynamisch Objekte manipulieren können; einige dieser externen Programme sind CGAL, QHull, Mathematica, Maple usw.

GeomView ist eine frei verfügbare Software, die unter dem *GNU Lesser General Public*

License (GPL) vertrieben wird.

3.5. Cinderella

Cinderella[1] versteht sich als einfach zu bedienende interaktive Geometriesoftware. Das Programm arbeitet in 2D und auf der Sphäre. Die ursprüngliche Idee von Cinderella war, ein computerbasiertes Werkzeug zu schreiben, das geometrische Gegebenheiten eines gegebenen Szenarios automatisch erkennt und beschreibt (vgl. Richter-Gebert und Hortenkamp [34]).

Die grafische Benutzeroberfläche bietet diverse Möglichkeiten, um vorhandene Objekte zu bearbeiten bzw. neue Objekte anzulegen. So kann z. B. eine Parallele zu einer gezeichneten Linie durch eine Verdopplung der Linie und ihre anschließende Verschiebung gezeichnet werden. Ändert sich der Winkel der Ausgangslinie für die Parallelen, so werden die von dieser Linie aus erzeugten Parallelen auch automatisch angepasst. Berechnungen von konvexen Hüllen, Triangulationen von Polyedern oder Voronoi-Diagrammen zu einer gegebenen Punktmenge kann das Programm nicht ausführen.

Die Programmierung der aktuellen Version 1.4 basiert auf Java, um Plattformunabhängigkeit zu gewährleisten. Der Quellcode von Cinderella ist nicht frei verfügbar, die Software ist urheberrechtlich geschützt.

Cinderella eignet sich wegen seiner intuitiven Handhabbarkeit für Unterrichtszwecke im Bereich Geometrie, ist aber kein Programm zur Berechnung von komplexen Algorithmen zu einer gegebenen Punktmenge. Solche Erweiterungen, z. B. zur Berechnung der Delaunay-Triangulation, werden in der kommenden Version 2.0 hinzugefügt, die zum Zeitpunkt der Erstellung dieser Arbeit in der Beta-Phase ist und 59,95 US-$ kosten wird.

[1] Den Namen erhielt das Programm nach dem Boot, auf dem sich die beiden ursprünglichen Entwickler kennenlernten.

4. Algorithmen und Datenstrukturen für Delaunay-Triangulationen

Wir haben in Kapitel 2.3 gesehen, dass das Voronoi-Diagramm $V(S)$ dual zur Delaunay-Triangulation $DT(S)$ ist. Es genügt demnach, die Delaunay-Triangulation zu berechnen, da das Voronoi-Diagramm schnell von ihr abgeleitet werden kann. Wir möchten in diesem Kapitel zunächst eine Auswahl an Algorithmen und Datenstrukturen zur Konstruktion der Delaunay-Triangulation vorstellen und danach eine Datenstruktur präsentieren, die sowohl die Delaunay-Triangulation als auch das Voronoi-Diagramm speichern kann.

Delaunay-Triangulationen lassen sich auf verschiedenen Wegen berechnen, wobei es keinen Algorithmus gibt, der für jede denkbare Voraussetzung überlegen wäre. Es gibt vielmehr für jeden Anwendungsbereich andere Möglichkeiten, die Berechnung der Delaunay-Triangulation im Bezug auf Laufzeit und Speicherplatzbedarf zu optimieren.

Auch wenn die Vielzahl an vorhandenen Algorithmen endlos scheint, folgen sie doch alle einer überschaubaren Anzahl an Paradigmen (vgl. Klein [24]), wie dem *Divide-and-Conquer*, dem *Sweep*, der geometrischen Transformation[1] oder der inkrementellen Konstruktion.

Aus dieser Menge an Paradigmen erscheint die inkrementelle Konstruktion der Delaunay-Triangulation am attraktivsten: Der Vorteil eines solchen Algorithmus besteht für uns darin, dass ein Hinzufügen von Punkten zur Laufzeit unseres Programms unproblematisch

[1] Die geometrische Transformation nutzt den Zusammenhang zwischen konvexen Hüllen im $\mathbb{R}^{d+1}$ und Delaunay-Triangulationen im $\mathbb{R}^d$, siehe Kapitel 2.4.

ist. Gerade für Lehrzwecke ist eine solche Funktionalität hilfreich, weil sie eine flexiblere Interaktion mit dem Benutzer bietet. Insofern haben wir uns bei der Untersuchung von Algorithmen zur Berechnung der Delaunay-Triangulation auf eine Auswahl an inkrementellen Algorithmen beschränkt.

4.1. Eine Auswahl inkrementeller Algorithmen und ihrer Datenstrukturen

Meist unterscheiden sich inkrementelle Algorithmen nur in der Art und Weise, wie sie die folgenden beiden Teilprobleme lösen: zum einen die *Lokalisierung* eines in die Triangulation $DT_{S_{i-1}}$ einzufügenden Punkts p_i und zum anderen die *Transformation* von $DT_{S_{i-1}}$ hin zu einer regulären Triangulation DT_{S_i} (auch als DT_i bezeichnet). Ein weiterer wichtiger Faktor beim Entwurf eines geometrischen Algorithmus ist seine *Robustheit*, also die Wahl einer Arithmetik sowie die Behandlung von Punktmengen, die sich nicht in allgemeiner Lage befinden.

Unter diesen Gesichtspunkten möchten wir in den folgenden Unterkapiteln verschiedene inkrementelle Algorithmen untersuchen. Dabei wollen wir einerseits dem Leser einen Einblick in die Vielfalt vorhandener Algorithmen geben und andererseits dokumentieren, wie sich der in unserer Implementation verwendete Algorithmus von Edelsbrunner und Shah [16] im Vergleich zu anderen Algorithmen verhält.

Zwei der betrachteten Algorithmen - den in CGAL verwendeten Algorithmus von Devillers et al. [10] und den von Barber et al. [3] konzipierten QHull-Algorithmus - haben wir bereits in den Kapiteln 3.1 und 3.2 vorgestellt. Ebenso werden wir uns mit Shewchuks [36] Pyramid- sowie Clarksons [9] Hull-Algorithmus beschäftigen. Die Auswahl komplettiert Tess3 von Liu und Snoeyink [27], der für die Berechnung von Delaunay-Triangulationen spezieller Punktmengen, die Molekularstrukturen von Proteinen repräsentieren (siehe auch Kapitel 1.3.2), optimiert ist.

4.1.1. Wahl einer Arithmetik

Bei der Wahl eines Zahlensystems gilt es, den gewünschten Grat zwischen Geschwindigkeit und Rechengenauigkeit zu finden: Je genauer ein Zahlensystem ist, desto langsamer erfolgen die Berechnungen. Wir vergleichen hier exemplarisch nur die in CGAL und Tess3 verwendeten Arithmetiken.

CGAL stellt dem Benutzer die Wahl der Arithmetik frei (vgl. Boissonnat et al. [4]). Die berechnete kombinatorische Datenstruktur ist abhängig von der Auswertung von Prädikaten, also rein numerischen Operationen, z. B. Orientierungstests[2]. Die Trennung von kombinatorischen und numerischen Operationen erlaubt es dem Benutzer auch, mehrere Arithmetiken zu benutzen: Beispielsweise kann bei Orientierungstests standardmäßig eine Fließpunktarithmetik verwendet werden und eine exakte Arithmetik nur dann, wenn die Resultate einen Schwellenwert unterschreiten, eine höhere Genauigkeit also wirklich benötigt wird.

Tess3 nutzt die Eigenschaften der zu bearbeitenden Daten, der PDB-Dateien[3], um die Performanz des Zahlensystems zu optimieren: Da einzelne Atome in einer solchen Datei lediglich eine Ausdehnung von wenigen Ångström[4] haben, sind die Messungen nicht sehr genau; die Präzision des Zahlensystems von Tess3 kann deshalb ohne Datenverlust auf Fließpunkt-Datentypen der Größe 20 Bit beschränkt werden. Nachdem die Triangulation fertiggestellt ist, kann Tess3 unter Verwendung exakter Berechnungen kontrollieren, ob die Triangulation in der Tat der Delaunay-Triangulation entspricht.

Eine andere Eigenschaft, die sich Tess3 zunutze macht, ist die Lage der Punktmenge: Die Atome von Proteinen sind relativ gleich im Raum verteilt und haben einen physikalisch bedingten Mindestabstand voneinander. Dadurch ist auch für Orientierungstests nur eine

[2]Ein Orientierungstest im $\mathbb{R}^3$ entscheidet durch Auswertung einer Determinanten, ob ein Punkt oberhalb oder unterhalb einer orientierten Ebene liegt.

[3]Die Dateien stammen aus Datenbanken, wie z. B. der *Worldwide Protein Data Bank* [31].

[4]Ein Ångström entspricht 10^{-10} Metern.

eingeschränkte Genauigkeit vonnöten.

4.1.2. Behandlung von Punktmengen in nicht allgemeiner Lage

Die Behandlung von Punktkonstellationen in nicht allgemeiner Lage zerfällt in zwei Gebiete: das Erkennen einer Verletzung der allgemeinen Lage sowie die Lösung eines solchen Konflikts.

Nur eine exakte Arithmetik ermöglicht das stets fehlerfreie Erkennen von Verletzungen der allgemeinen Lage: Liefert z. B. die Auswertung eines Orientierungstests (wozu lediglich eine Determinante berechnet werden muss) den Wert Null, sind die vier betrachteten Punkte koplanar. Wird kein exaktes Zahlensystem verwendet, kann dieser Orientierungstest aufgrund von Rundungsfehlern falsche Ergebnisse liefern: Die so konstruierte Triangulation entspricht möglicherweise nicht der Delaunay-Triangulation. QHull, das lediglich mit Fließpunktarithmetik arbeitet, bietet die Möglichkeit, die Triangulation im Nachhinein zu reparieren; durch diese Nachtriangulation entsteht dann wieder die Delaunay-Triangulation.

Für Punktmengen, die sich nicht in allgemeiner Lage befinden, gibt es die Möglichkeit, diese zu simulieren. Es gibt mehrere Lösungsansätze, wie z. B. die Definition einer lexikographischen Ordnung auf der Punktmenge: Von zwei gleichen Werten ist derjenige kleiner, der in dieser Ordnung vor dem anderen steht. Mücke und Edelsbrunner [15] beschreiben die *Simulation Of Simplicity*, was einer Perturbation (d. h. einer leichten Manipulation der Koordinaten in einem ε-Bereich) der Punkte entspricht; Devillers und Teillaud [11] wählen für CGAL einen ähnlichen Ansatz.

4.1.3. Datenstrukturen

Die Repräsentation der Delaunay-Triangulation ist in allen Implementationen ähnlich: Alle Algorithmen benutzen Datenstrukturen, die auf Knoten und Tetraedern basieren, wobei

Tetraeder zusätzlich Verweise auf ihre Nachbartetraeder speichern. Wegen der Ähnlichkeit der Implementationen sollen an dieser Stelle lediglich die in CGAL verwendeten Datenstrukturen beschrieben werden (vgl. CGAL-Benutzerhandbuch [8]).

In CGAL werden für ein Tetraeder Verweise auf dessen vier Eckpunkte in einem Array so gespeichert, dass die ersten drei Punkte eine orientierte Ebene definieren und der vierte Punkt über dieser Ebene liegt (siehe Abbildung 4.1). Diese Eigenschaft erleichtert Berechnungen, bei denen die Orientierung einer Ebene entscheidend ist. So lässt sich beispielsweise das Problem lösen, ob sich ein Anfragepunkt innerhalb eines Tetraeders befindet: Befindet er sich oberhalb aller Dreiecksflächen des Tetraeders, liegt er in dessen Inneren.

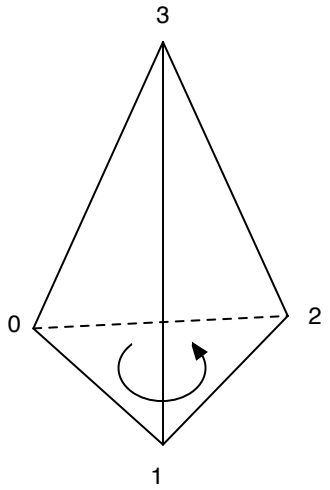


Abbildung 4.1.: Orientierung des Tetraeders gegen den Uhrzeigersinn.

Zudem speichert ein Tetraeder die Referenzen auf seine vier Nachbartetraeder in einem weiteren Array. Dabei stehen ein Eckpunkt und ein adjazentes Tetraeder in ihrem jeweiligen Array genau dann an derselben Indexposition, wenn der Eckpunkt *nicht* zum Nachbartetraeder gehört, d. h. ihm sozusagen gegenüber liegt. In Abbildung 4.2 haben sowohl der oberste Punkt des Tetraeders als auch das eingezeichnete Nachbartetraeder den Index 3.

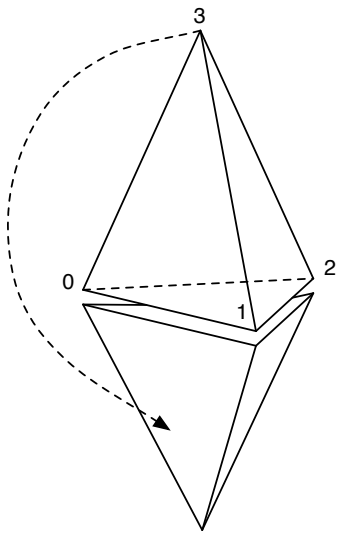


Abbildung 4.2.: Verweise auf Nachbartetraeder.

Ein Punkt in CGAL verweist dazu noch auf das Tetraeder, dem er angehört. So kann ein Punkt als Startknoten einer Navigation fungieren: Man betrachtet sein inzidentes Tetraeder und kann sich anhand der Verweise auf adjazente Tetraeder durch die Triangulation bewegen.

In CGAL werden die Dreiecke eines Tetraeders nicht explizit repräsentiert, da sie durch ein Tetraeder und einen Eckpunkt desselben dargestellt werden können: Es handelt sich dann um die Dreiecksfläche des Tetraeders, der der spezifizierte Eckpunkt nicht angehört.

4.1.4. Punktlokalisierung (Point Location)

Die inkrementelle Konstruktion nimmt für jeden in die Triangulation eingefügten Punkt lokale Änderungen vor: Nicht reguläre Tetraeder werden durch reguläre ersetzt. Um diese lokalen Änderungen durchzuführen, muss ermittelt werden, in welchem Tetraeder ein Anfragepunkt liegt.

Barber et al. [3] verwalten dazu in QHull für jedes Tetraeder eine Konfliktliste: Dort werden alle noch einzufügenden Punkte gespeichert, die das dem Tetraeder entsprechende

Konstrukt in der konvexen Hülle im $\mathbb{R}^4$ sehen können. Jedoch müssen für diesen Ansatz alle einzufügenden Punkte vorab bekannt sein (vgl. Liu und Snoeyink [27]).

Clarkson [9] sowie Edelsbrunner und Shah [16] verwenden als hierarchische Datenstruktur einen gerichteten, azyklischen Graphen (*directed acyclic graph*, kurz: *DAG*), in dem alle Tetraeder, die zu einem Zeitpunkt der Konstruktion Teil der Triangulation waren, verwaltet werden. Diesen Graphen nennt man *History-DAG* oder *Delaunay-DAG*. Er besitzt genau einen Wurzelknoten: Dieser repräsentiert ein künstliches Tetraeder, das die gesamte Punktmenge enthält[5]. Ein Kind eines Knotens beinhaltet das Tetraeder (siehe Abbildung 4.3), das den oder die Elternknoten nach einer Aktualisierung in der Triangulation ersetzt hat. Die Blätter des *DAG* enthalten die aktuell in der Triangulation vorhandenen Tetraeder.

Die Suche nach einem Anfragepunkt beginnt im Wurzelknoten des *DAG*. In jedem Knoten wird entschieden, ob das Tetraeder, das diesem Knoten entspricht, den Anfragepunkt enthält. Falls nein, sucht man in Geschwisterknoten, falls ja, in Kinderknoten weiter, bis der Blattknoten erreicht wird, der das gesuchte Tetraeder beinhaltet. Da die Triangulation einer Partition des Raums innerhalb des künstlichen Tetraeders entspricht, terminiert die Punktlokalisierung stets: Jeder Anfragepunkt ist in genau einem Tetraeder enthalten.

Shewchuks [36] Pyramid-Algorithmus sucht nach einem auf Mücke et al. [29] basierenden *jump and walk*-Schema. Hierbei wird nur in der aktuellen Triangulation gesucht. Diese enthalte zu einem bestimmten Zeitpunkt m Tetraeder. Von einem Anfragepunkt p wird die Distanz zu $m^{1/4}$ zufällig ausgewählten Tetraedern, d. h. zu deren Mittelpunkten[6], gemessen. Von dem zu p nächsten Tetraeder σ geht die Suche weiter: Man folgt einem Strahl, der von σ ausgeht und durch p verläuft. Vom ersten Tetraeder, das der Strahl durchstößt, geht die Suche weiter, bis schließlich das Tetraeder gefunden ist, das p enthält.

[5] Damit ist gewährleistet, dass schon der erste Punkt der Punktmenge in einem Tetraeder lokalisiert werden kann. Kriterien für die Wahl eines solchen Tetraeders werden wir in Kapitel 5.1.1 beschreiben.

[6] Unter dem Mittelpunkt eines Tetraeders verstehen wir den Mittelpunkt der Sphäre, die durch die vier Eckpunkte des Tetraeders definiert ist.

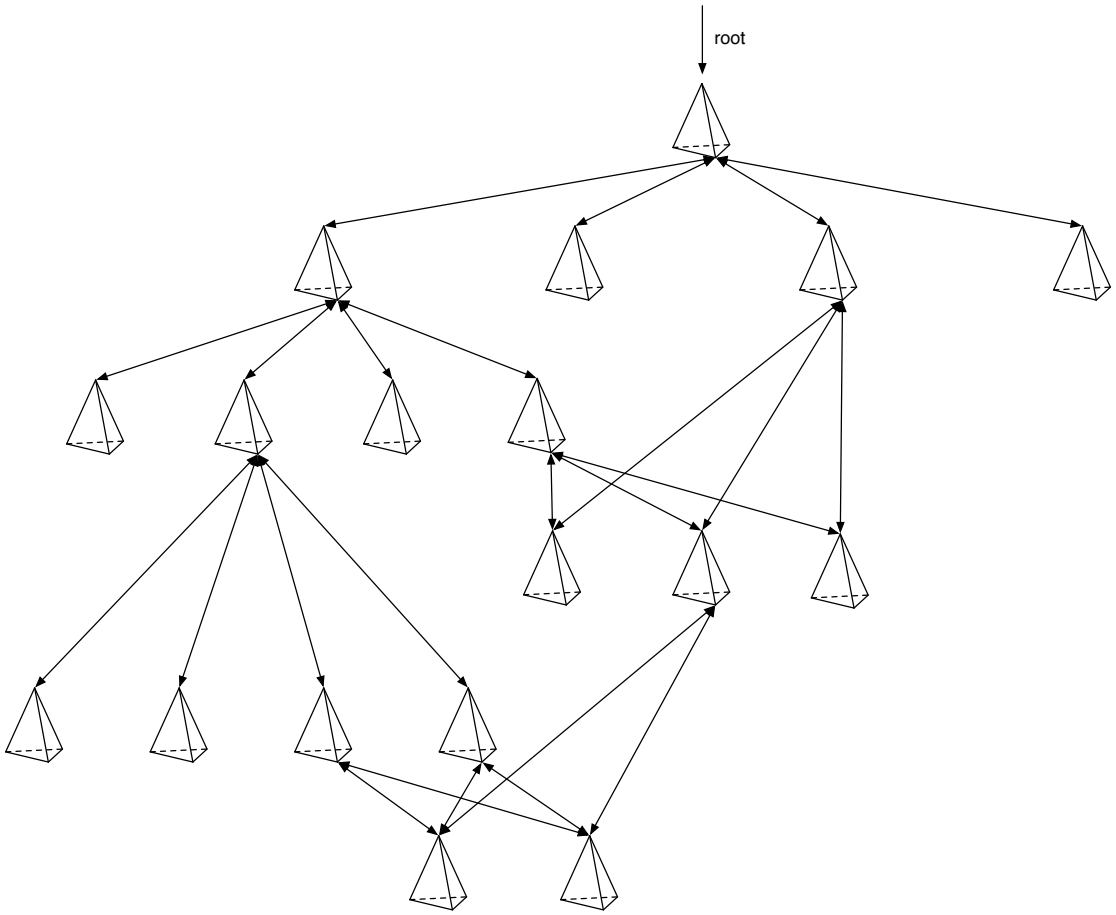


Abbildung 4.3.: Beispiel für einen *History-DAG*.

In CGAL kombinieren Devillers et al. in ihrer Suche eine hierarchische Datenstruktur mit dem *jump and walk*. Dies liefert insbesondere bei nicht gleichmäßig verteilten Punktmengen bessere Ergebnisse. Während Shewchuk einen Strahl für den *walk* benutzt, erfolgt hier direkt ein Übergang von einem Tetraeder zu dessen Nachbartetraeder. Man spricht deswegen von einem *zig-zag walk* und nicht von einem *straight-line walk*. Tess3 verwendet ebenso einen *zig-zag walk*, der aufgrund der Verwendung von Hilbertkurven aber noch kürzer ist als bei Devillers et al. Dies gelingt, weil die zugrundeliegenden Daten gleichmäßig verteilte Punktmengen beinhalten, wobei ein Lauf über Hilbertkurven bessere Resultate liefert.

Die *jump-and-walk*-Ansätze versprechen eine schnellere Laufzeit, wogegen ein *Delaunay-DAG* unkomplizierter und anschaulicher ist, jedoch den Nachteil eines hohen Speicher-

platzbedarfs hat. Eine ausführliche Laufzeit- und Speicherplatzanalyse des *History-DAG* folgt in den Kapiteln 5.3.2.2 und 5.3.2.1.

4.1.5. Aktualisierung der Triangulation

Nachdem ein einzufügender Punkt p_i in $DT_{S_{i-1}}$ gefunden wurde, muss die Triangulation $DT_{S_{i-1}}$ so aktualisiert werden, dass wieder eine Delaunay-Triangulation DT_{S_i} entsteht. Hier unterscheiden sich die Algorithmen fundamental voneinander.

Zum einen gibt es den Ansatz von Bowyer-Watson (vgl. Bowyer [6] und Watson [39]), der abgewandelt in Tess3, CGAL, QHull und Hull verwendet wird. Hierbei werden die Tetraeder, die nach dem Einfügen eines neuen Punkts in Konflikt mit diesem stehen (d. h. wenn die Umkugeln dieser Tetraeder den neuen Punkt enthalten), entfernt. Die Vereinigung dieser zu entfernenden Tetraeder nennt Watson das *Einfügepolyeder*, da neue Tetraeder in dieses Polyeder eingefügt werden müssen, um wieder eine Triangulation zu erhalten. Der neue Punkt bildet dabei mit jedem Dreieck des Rands dieses Einfügepolyeders ein neues Tetraeder.

Sowohl Shewchuk als auch Edelsbrunner und Shah arbeiten dagegen mit *Flipping*. Das bedeutet, dass bestimmte Delaunay-Dreiecke bildlich gesprochen „gedreht“ werden[7]. Ziel ist, durch Flipping wieder eine reguläre Triangulation zu gewinnen.

An einem Flip im $\mathbb{R}^3$ sind immer fünf Tetraeder beteiligt, wobei $5 - i$ Tetraeder verschwinden und i neue Tetraeder entstehen ($0 < i < 5$). Die Vereinigung der an einem Flip beteiligten Tetraeder muß dabei stets konvex sein.

Bei einem 1-zu-4-Flip (siehe Abbildung 4.4) verschwindet ein Tetraeder, und vier neue entstehen. Dies ist der Fall, wenn ein neuer Punkt eingefügt wird: Man fügt Kanten zu den übrigen vier Knoten des einen Tetraeders ein, wodurch sich vier neue Tetraeder ergeben.

[7] Bei Delaunay-Triangulationen in der Ebene spricht man von *edge flips*; wir verwenden der Einfachheit halber lediglich die Bezeichnung *Flip*.

Einen 4-zu-1-Flip zu implementieren (bei dem ja ein Punkt verschwinden würde) ist für eine korrekte Berechnung der Delaunay-Triangulation nicht notwendig. Eine effiziente Methode zum Entfernen von Punkten aus Triangulationen wird z. B. von Devillers und Teillaud [11] behandelt.

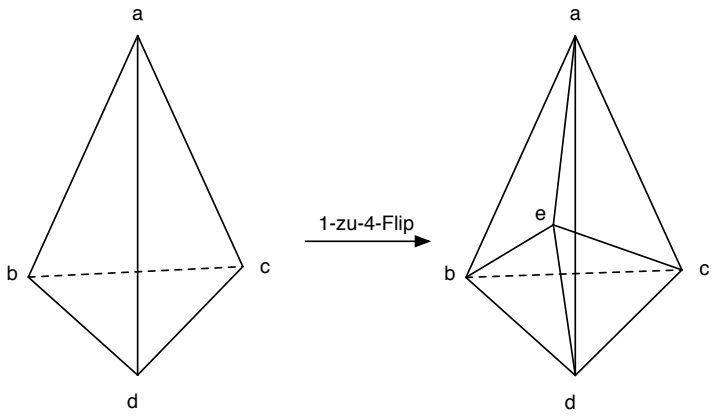


Abbildung 4.4.: Der 1-zu-4-Flip.

Die beiden übrigen Fliptypen sind 2-zu-3 und 3-zu-2 (siehe Abbildung 4.6). Der 2-zu-3-Flip ersetzt zwei Tetraeder, indem er das Delaunay-Dreieck, das beide gemein haben, durch drei neue Dreiecksflächen ersetzt. Diese neuen Flächen bestehen jeweils aus einem Punkt des alten Dreiecks und den beiden Punkten der Vereinigung der Tetraeder, die ursprünglich nicht zu diesem Dreieck gehörten. Der 3-zu-2-Flip funktioniert genau umgekehrt: Aus drei Tetraedern werden zwei.

Bei Edelsbrunner und Shah beeinflußt das Flipping den Aufbau des *History-DAG*: Jeder Flip fügt neue Blattknoten hinzu (siehe Abbildung 4.5). Ein 1-zu-4-Flip hängt vier neue Blätter an einen einzelnen Knoten, der zuvor ein Blatt war. Der 2-zu-3-Flip fügt drei Blätter hinzu, die jeweils dieselben beiden Elternknoten haben. Analog sind es zwei Blätter mit jeweils drei gleichen Eltern bei einem 3-zu-2-Flip. Da ein Kindknoten mehrere Elternknoten haben kann, ist ein Baum zur Darstellung der Historie nicht ausreichend.

Für eine genaue Laufzeit- und Speicherplatzanalyse des Flippings sei auch hier auf die Kapitel 5.3.2.1 und 5.3.2.2 verwiesen.

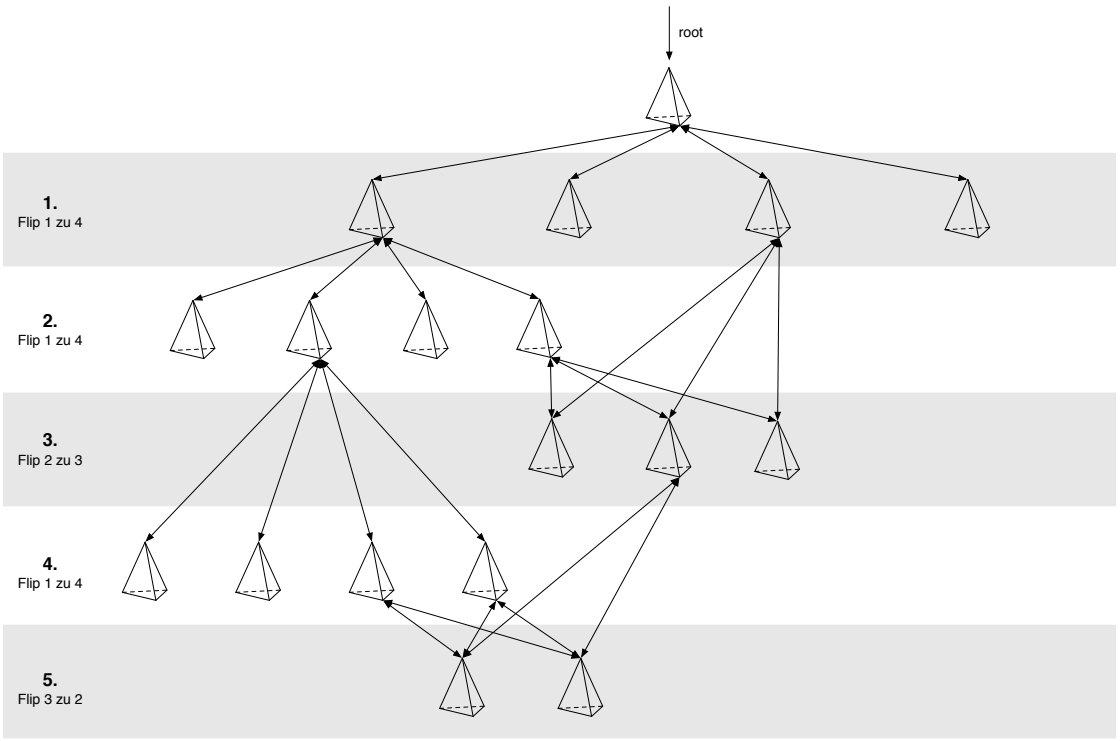


Abbildung 4.5.: Beispiel für den Aufbau eines *History-DAG* durch Flipping.

4.2. Repräsentation des Voronoi-Diagramms durch die Augmented Quad Edge (AQE)-Datenstruktur

Die bisher vorgestellten Datenstrukturen der Delaunay-Triangulation machen es erforderlich, das Voronoi-Diagramm aus der Delaunay-Triangulation zu berechnen; es liegt nicht explizit vor. Für eine Berechnung aus der Delaunay-Triangulation wäre ein zu deren Komplexität linearer Aufwand erforderlich.

Im Folgenden erörtern wir die Frage, wie eine Datenstruktur beschaffen sein sollte, um ein Voronoi-Diagramm optimal zu repräsentieren. Es wäre wünschenswert, anstatt einer impliziten, tetraederbasierten, wie sie zuvor vorgestellt wurde, eine explizite, kantenbasierte Datenstruktur zur Navigation durch das Voronoi-Diagramm zu besitzen.

Eine Datenstruktur, die diesen Anforderungen gerecht wird, ist die *Augmented Quad Ed-*

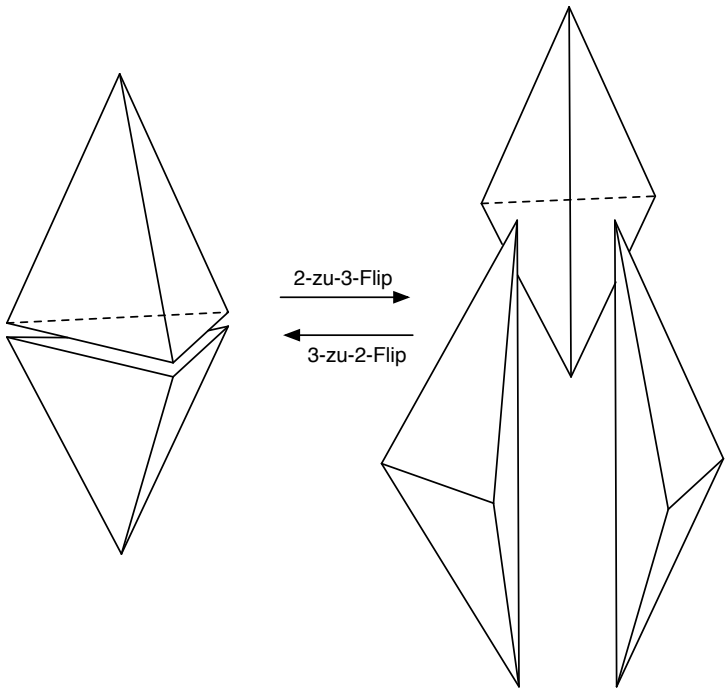


Abbildung 4.6.: Der 2-zu-3- und 3-zu-2-Flip.

ge-Datenstruktur von Gold, Ledoux und Dzieszko [21]. Sie basiert auf der von Guibas und Stolfi [22] vorgestellten *Quad Edge* (*QE*)-Datenstruktur. Dabei handelt es sich um eine kantenbasierte Datenstruktur, die für das Navigieren durch Graphen in der Ebene (oder Topologien, die einer Ebene entsprechen) konzipiert ist. *Quad Edges* eignen sich für Polyederoberflächen, d. h. in unserem Fall auch für einzelne Delaunay-Tetraeder bzw. Voronoi-Regionen.

Die Idee der *AQE* besteht nun darin, diese einzelnen Polyederoberflächen mit *Quad Edges* darzustellen und darüber hinaus Verbindungen zwischen benachbarten Polyedern herzustellen. So ist es in unserem Fall möglich, mit *Quad Edges* über Oberflächen von Voronoi-Zellen bzw. Delaunay-Tetraedern und mit Hilfe der Erweiterungen der *AQE* auch von Voronoi-Zelle zu Voronoi-Zelle oder von Delaunay-Tetraeder zu Delaunay-Tetraeder zu navigieren.

Die Navigation zwischen gleichartigen Zellen erfolgt über die duale Kante der gemein-

samen Fläche: Bei der Navigation zwischen Voronoi-Zellen läuft man über Delaunay-Kanten, beim Übergang von einem Delaunay-Tetraeder zu einem benachbarten bewegt man sich auf Voronoi-Kanten. Aufgrund der Dualität ist es möglich, Voronoi-Diagramm und Delaunay-Triangulation in einer Datenstruktur zu speichern. Eine explizite Berechnung des Voronoi-Diagramms aus der Delaunay-Triangulation ist nicht erforderlich.

Wir betrachten in dem folgenden Unterkapiteln zunächst die *Quad Edge*-Datenstruktur; danach werden wir demonstrieren, wie man diese Datenstruktur zu einer *AQE*-Datenstruktur erweitern kann.

4.2.1. Navigation auf Polyederoberflächen mit einer Quad Edge-Datenstruktur

Man betrachte einen in eine Ebene eingebetteten kreuzungsfreien Graphen: Jede ungerichtete Kante entspricht in einer *QE*-Datenstruktur vier Kanten. So existieren zu einer Kante, die zwei Punkte p und q miteinander verbindet, folgende Kanten: eine von p nach q und eine von q nach p gerichtete Kante sowie zwei gerichtete duale Kanten, die die beiden an die Kante pq angrenzenden Flächen A und B miteinander verbinden (siehe Abbildung 4.7).

Die vier Kanten einer *Quad Edge* sind untereinander verbunden: Mit dem Operator *rotate* (`rot`) wechselt man von einer die zwei Punkte p und q verbindenden Kante zu der gegen den Uhrzeigersinn liegenden dualen Kante, die die rechte angrenzende Region (B) mit der linken (A) verbindet. Wendet man auf diese Kante wieder `rot` an, so erhält man die Kante von q nach p. Eine weitere `rot`-Operation liefert wieder eine die zwei Flächen verbindende Kante, diesmal von A nach B gerichtet. Die vierte `rot`-Operation liefert die Identität der Ursprungskante (siehe Abbildung 4.8).

Die Operation *origin* (`org`) liefert den Startknoten bzw. die Startfläche jeder Kante (siehe Abbildung 4.9): Von jedem Knoten eines Graphen führen also gerichtete Kanten zu den Nachbarknoten und von jeder Region gerichtete Kanten zu den Nachbarflächen. Mit der

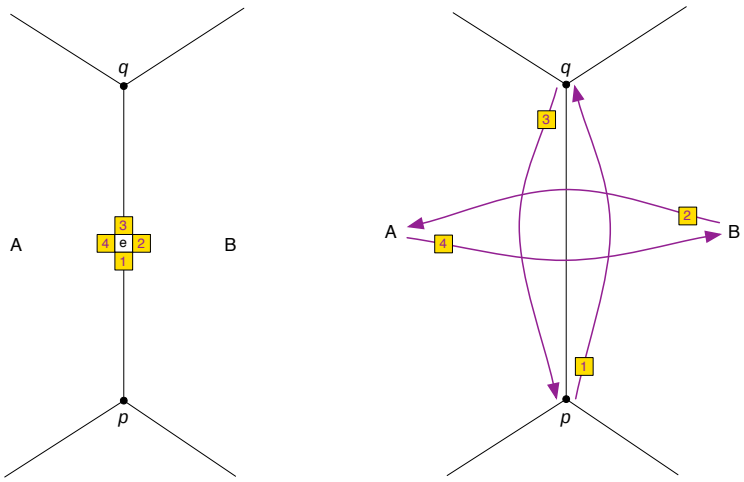


Abbildung 4.7.: Eine *Quad Edge*.

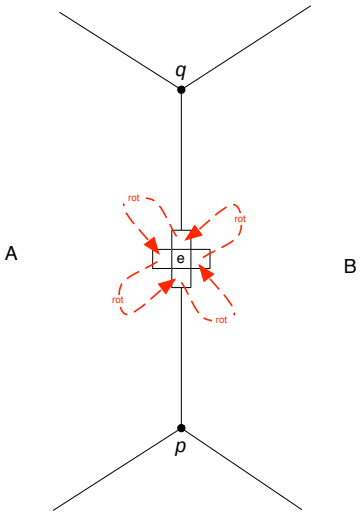


Abbildung 4.8.: Die `rot`-Verweise einer *Quad Edge*.

Operation *origin next* (`onext`) erhält man die gegen den Uhrzeigersinn nächste Kante, die denselben Ursprung hat wie die Ausgangskante. Damit ist es nun möglich, alle Kanten, die von einem Knoten ausgehen, gegen den Uhrzeigersinn zu besuchen. Ebenso ist es möglich, eine Fläche gegen den Uhrzeigersinn zu umrunden (siehe Abbildung 4.10 und 4.11).

Um zum Beispiel eine einer Kante entgegengerichtete Kante (`twin`) zu ermitteln, bedarf es nun keiner gesonderten Operation: Sie ist durch zweimaliges anwenden von `rot` eindeutig

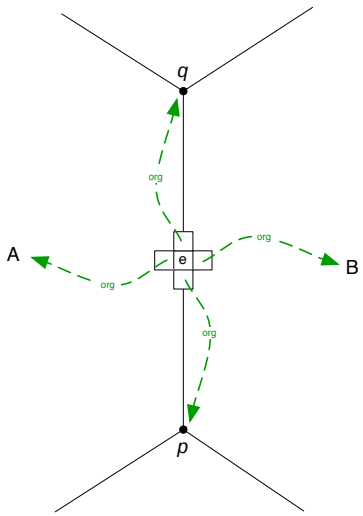


Abbildung 4.9.: Die `org`-Verweise einer *Quad Edge*.

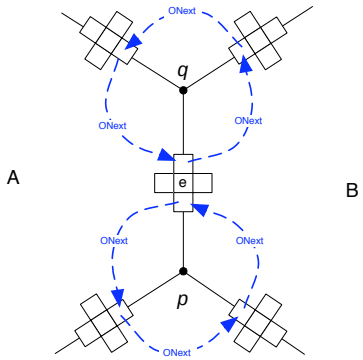


Abbildung 4.10.: Die `onext`-Verweise einer *Quad Edge*, die zwei Knoten verbindet.

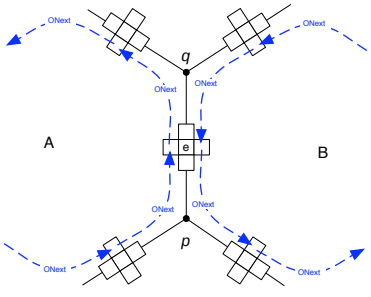


Abbildung 4.11.: Die `onext`-Verweise einer *Quad Edge*, die zwei Flächen verbindet.

definiert. Der Zielknoten einer Kante wird so durch die Operationen `twin` und `org` vom Startknoten aus erreicht.

Die im Uhrzeigersinn nächste rotierte Kante (`invRot`) wird durch dreifache Anwendung von `rot` gefunden. Außerdem kann man, um die im Uhrzeigersinn nächste Kante zu einem Ausgangsknoten zu finden (`invOnext`), so oft (n-mal) `onext` anwenden, bis wieder auf die Ursprungskante verwiesen wird. Das Ergebnis der Anwendung von `invOnext` entspricht dann dem von $(n-1)$-mal `onext`.

Die Beschränkung auf lediglich drei Operatoren ist zwar zeitintensiver, spart aber Speicherplatz. Mit `rot`, `onext` und `org` sind also ausreichend Navigationsmöglichkeiten für Graphen in der Ebene bzw. auf ebenenähnlichen Topologien wie Polyederoberflächen gegeben.

4.2.2. Navigation zwischen zwei benachbarten Polyedern mit einer Augmented Quad Edge-Datenstruktur

Während bei der *QE*-Datenstruktur die rotierten Kanten zwei Flächen miteinander verbinden, so sind es bei der *AQE* die zwei zu den Flächen dualen Kanten, die verbunden werden: Der Ursprung einer solchen rotierten Kante ist wiederum eine Kante. Dies ermöglicht den Übergang von Polyeder zu Polyeder.

Man stelle sich eine Kante eines Delaunay-Tetraeders vor. Die rotierte Kante verbindet hier also nicht zwei Dreiecksflächen miteinander, sondern die zu den Dreiecken dualen Voronoi-Kanten (siehe Abbildung 4.12). Eine zu einem Delaunay-Dreieck duale Voronoi-Kante sei dabei immer so gerichtet, dass ihr Ursprung der Mittelpunkt des Delaunay-Tetraeders ist, auf dem sich die Ursprungskante (die Kante des Delaunay-Tetraeders) befindet (siehe Abbildung 4.13 und 4.14[8]). Der Zielknoten ist der Mittelpunkt des Nachbartetraeders. Werden also auf eine Delaunay-Kante die Operatoren `rot` und `org` ange-

[8]Die in den Illustrationen verwendeten Kantenbezeichnungen werden in Kapitel 7 erläutert

wandt, so erhält man die Voronoi-Kante, die zum einen auf der Voronoi-Region um den Ursprungsknoten der Delaunay-Kante liegt und zum anderen das Delaunay-Dreieck durchstößt, das rechts von dieser Delaunay-Kante liegt. Zu jedem Delaunay-Dreieck existieren also drei duale Kanten, da jeder Punkt des Dreiecks einer Voronoi-Zelle entspricht.

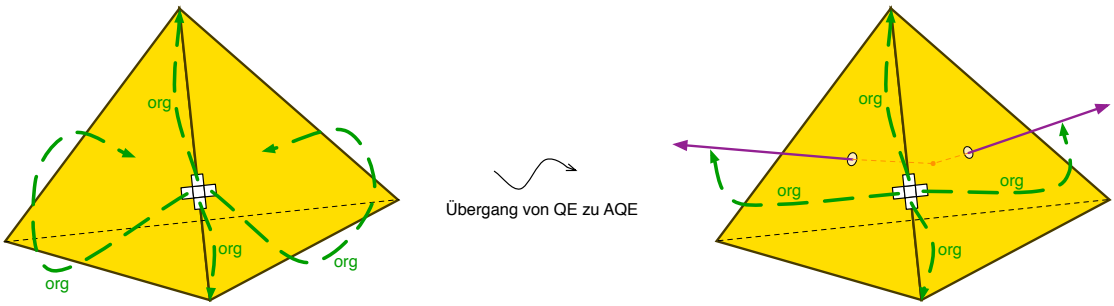


Abbildung 4.12.: Der Übergang von *Quad Edge* zu *Augmented Quad Edge*.

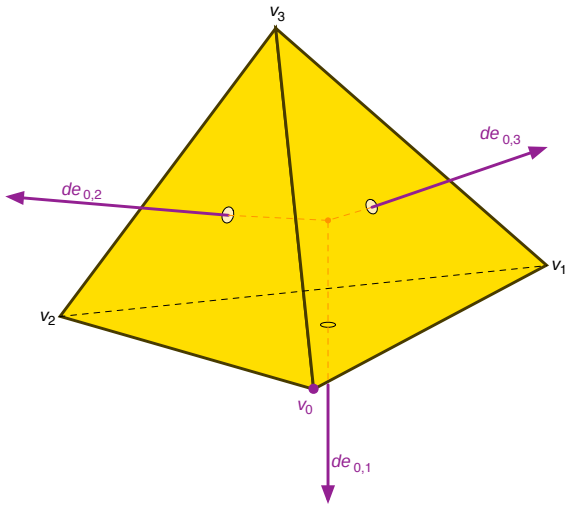


Abbildung 4.13.: Die zur Voronoi-Region von v_0 gehörenden dualen Kanten.

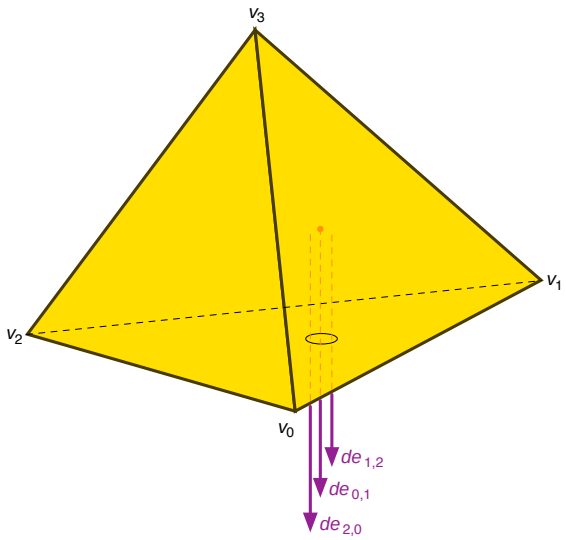


Abbildung 4.14.: Alle zur Fläche v_0, v_1 und v_2 gehörenden dualen Kanten.

5. Analyse des implementierten Algorithmus

In diesem Kapitel analysieren wir den Algorithmus von Edelsbrunner und Shah [16], der in unserer Implementation zur Berechnung der Delaunay-Triangulation verwendet wird.

Wir beschreiben im Detail, wie die Delaunay-Triangulation konstruiert wird, und schätzen die zu erwartenden Werte für Laufzeit und Speicherplatzbedarf ab. Danach werden wir zeigen, wie man das Voronoi-Diagramm und die Delaunay-Triangulation aus der *AQE*-Datenstruktur ableiten kann.

5.1. Konstruktion der Delaunay-Triangulation

Der Algorithmus zur Konstruktion der Delaunay-Triangulation arbeitet für jeden einzufügenden Punkt in zwei Schritten: Zuerst wird das Tetraeder lokalisiert, das diesen neuen Punkt enthält, und anschließend durch einen 1-zu-4-Flip eine Triangulation hergestellt. Danach werden lokale Änderungen durch eine Folge von 2-zu-3- und 3-zu-2-Flips vorgenommen, um neu entstandene, nicht reguläre Tetraeder zu beseitigen; so entsteht wieder eine Delaunay-Triangulation (siehe Algorithmus 5.1).

Algorithmus 5.1: Berechnung der Delaunay-Triangulation

1 Konstruiere das allumfassende Tetraeder
2 Für alle Punkte $p_i \in S$
3 Lokalisiere Punkt p_i in $DT_{S_{i-1}}$
4 Führe einen 1-zu-4-Flip aus
5 Transformiere $DT_{S_{i-1}}$ nach DT_{S_i} durch 2-zu-3- und 3-zu-2-Flips

5.1.1. Wahl des allumfassenden Tetraeders

Da zu Beginn des Algorithmus noch kein Tetraeder existiert, in dem einzufügende Punkte lokalisiert werden können, wird ein künstliches, unendlich großes Tetraeder konstruiert, das die Punktmenge S vollständig enthält (vgl. Schritt 1 von Algorithmus 5.1).

Als Eckpunkte dieses Tetraeders schlägt Edelsbrunner [16] folgende Punkte vor:

$$\begin{aligned} p_{-1} &= (\ -\infty\ ,\ -\infty\ ,\ -\infty\ ), \\ p_{-2} &= (\ +\infty\ ,\ -\infty\ ,\ -\infty\ ), \\ p_{-3} &= (\ 0\ ,\ +\infty\ ,\ -\infty\ ), \\ p_{-4} &= (\ 0\ ,\ 0\ ,\ +\infty\ ). \end{aligned}$$

Das Problem bei der Konstruktion dieses Tetraeders ist, dass sich der Wert $+\infty$ nicht mittels der Standard-Java-Zahlen repräsentieren lässt. Würde man lediglich einen sehr großen Wert wählen, könnte es passieren, dass sich ein Knoten dieses künstlichen Tetraeders zu nahe an der Punktmenge befände: Der Knoten könnte innerhalb einer Sphäre liegen, die durch ein inneres Tetraeder definiert wäre, und einen ungewollten Flip auslösen.

Shah [35] beschreibt die Problematik der Wahl dieses Initialisierungstetraeders und berechnet anhand der Präzision des verwendeten Zahlensystems, wie weit dessen Eckpunkte von der Punktmenge entfernt liegen müssten, um die oben beschriebenen Probleme zu

vermeiden. Da die Koordinaten dieser Punkte in der Tat sehr groß werden und so bei Orientierungstests schnell Überläufe entstehen, bedient er sich des chinesischen Restesatzes bei der Berechnung der Determinanten.

5.1.2. Point Location

Um festzustellen, in welchen Teilen der Triangulation lokale Änderungen erfolgen müssen, suchen wir das Tetraeder, das den nächsten in die Triangulation einzufügenden Punkt enthält (siehe Schritt 3 des Algorithmus 5.1). Die Funktionsweise des hierzu verwendeten *History-DAG* haben wir bereits in Kapitel 4.1.4 beschrieben. Er lässt sich am einfachsten durch eine Rekursion implementieren (siehe Algorithmus 5.2).

Algorithmus 5.2: `lokalisierePunkt(`p_i, w`)`

```
1 Falls p_i im Tetraeder zum History-DAG-Knoten w enthalten ist
2     Falls w Kinderknoten besitzt
3         Für alle Kinderknoten w_j von w
4             Setze w_neu = lokalisierePunkt(p_i, w_j)
5             Falls w_neu ungleich null ist
6                 Liefere w_neu zurück
7     Sonst liefere w zurück
8 Sonst liefere null zurück
```

5.1.3. Aktualisierung der Triangulation

Nachdem der einzufügende Punkt in $DT_{S_{i-1}}$ gefunden wurde, muß ein 1-zu-4-Flip ausgeführt werden, damit der Punkt ein Knoten in DT_{S_i} wird. Das Tetraeder, das den Anfragepunkt enthielt, wird also durch vier neue Tetraeder ersetzt. Diese Tetraeder verletzen aber möglicherweise die Regularitätsbedingung, d. h. die Triangulation ist noch keine

Delaunay-Triangulation: Einzelne Delaunay-Dreiecke der neuen Tetraeder sind womöglich nicht regulär. Die Berechnung dieser beiden Schritte beschreibt Algorithmus 5.3.

Algorithmus 5.3: Transformiere $DT_{S_{i-1}}$ nach DT_{S_i}

```
1  Initialisiere Stapel φ_l mit den Link Facets von p_i
2  Solange Stapel φ_l nicht leer ist
3      Setze λ = oberstes Link Facet von φ_l
4      Entferne λ vom Stapel φ_l
5      Falls λ in DT_{S_{i-1}} enthalten ist
6          Setze q = Anzahl der reflexen Kanten von λ bzgl.
               der Vereinigung der λ enthaltenden Tetraeder
7          Falls q < 2 ist
8              Falls λ nicht regulär ist
9                  Falls q = 0 ist
10                     Führe einen 2-zu-3-Flip für λ aus
11                     Füge neue Link Facets von p_i dem Stapel φ_l hinzu
12                 Falls q = 1 ist
13                     Falls λ flipbar ist
14                         Führe einen 3-zu-2-Flip für λ aus
15                         Füge neue Link Facets von p_i dem Stapel φ_l hinzu
```

Die Delaunay-Dreiecke, die zwei neue Tetraeder trennen, sind in jedem Fall regulär: Wären sie es nicht, so könnte der neu eingefügte Punkt nicht innerhalb des Tetraeders liegen. Es müssen also lediglich die Delaunay-Dreiecke geprüft werden, die neue von alten Tetraedern trennen. Diese Delaunay-Dreiecke nennen wir *Link Facets*. Die Bezeichnung „link“ bezieht sich auf die Zugehörigkeit zum zuletzt eingefügten Punkt: Allen *Link Facets* ist gemein, dass sie zu Tetraedern gehören, deren übriger vierter Punkt dieser zuletzt eingefügte ist. Die *Link Facets* werden auf einem Stapel φ_l verwaltet, der abgearbeitet werden muss, bevor ein neuer Punkt p_i eingefügt werden kann. Erst dann ist die Triangulation wieder

eine Delaunay-Triangulation.

Das Abarbeiten des Stapels φ_l beschreibt Algorithmus 5.3.

Falls ein *Link Facet* nicht regulär ist, muss es durch einen Flip aus der Triangulation entfernt werden. Ein 2-zu-3-Flip ist nötig, falls die Vereinigung der Nachbartetraeder des *Link Facets* konvex ist, also alle drei Kanten des Link Facets bzgl. der Vereinigung der beiden Nachbartetraeder konvex sind (siehe Abbildung 5.1).

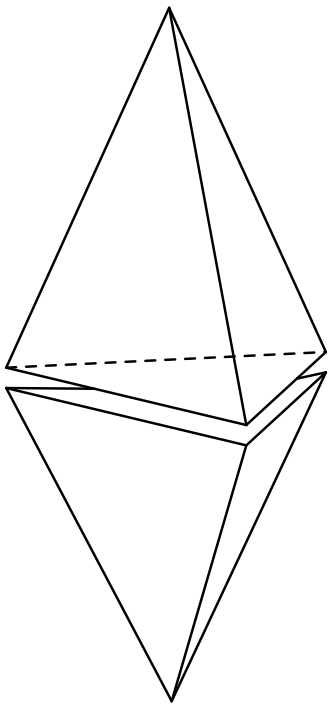


Abbildung 5.1.: Alle Kanten des *Link Facet* sind konvex.

Ist genau eine Kante des *Link Facets* reflex (siehe Abbildung 5.2), ist zu prüfen, ob der Grad dieser Kante in der Triangulation genau 3 ist, d. h. ob genau drei Tetraeder an diese Kante angrenzen. Ist dies der Fall (siehe Abbildung 5.2), so kann ein 3-zu-2-Flip durchgeführt werden, da die Vereinigung der drei Tetraeder der konvexen Hülle der fünf beteiligten Punkte entspricht.

Ansonsten ist das *Link Facet* nicht flipbar und wird verworfen. Durch einen 2-zu-3-Flip

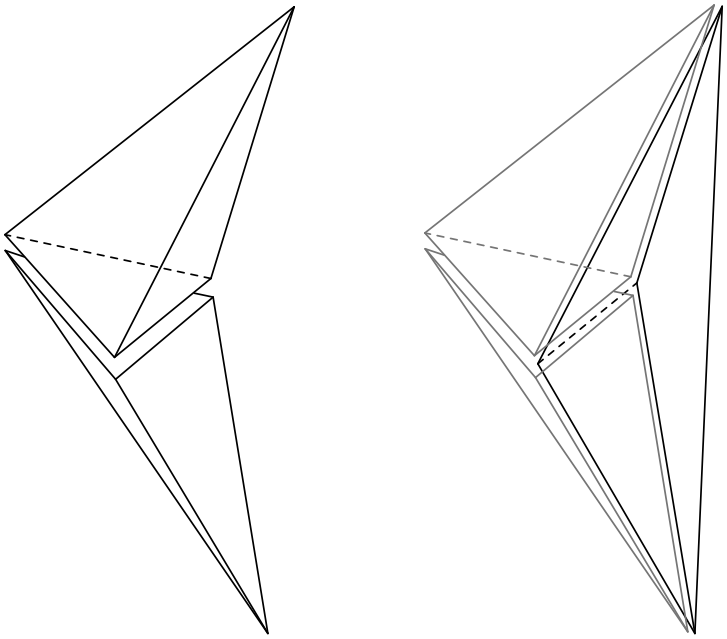


Abbildung 5.2.: Links: Genau eine Kante des *Link Facet* ist reflex. Rechts: Das *Link Facet* ist flipbar.

entstehen drei neue *Link Facets*, durch einen 3-zu-2-Flip zwei. Diese sind auf den *Link Facet*-Stapel zu legen, da sie durch die lokalen Änderungen ebenso irregulär sein könnten.

5.2. AQE-Durchlauf zur Darstellung der Diagramme

Der Durchlauf der Diagramme (vgl. Algorithmus 5.4)entspricht einer Tiefensuche in der AQE-Datenstruktur. Dabei werden alle zu zeichnenden Flächen auf einem Stapel gesammelt.

Algorithmus 5.4: Finde alle Flächen ausgehend vom Startpunkt p

```
1  Lege Stapel φ_f für die zu findenden Flächen an
2  Lege p auf den Stapel φ_b noch nicht besuchter Knoten
3  Solange φ_b nicht leer ist
4      Setze q = oberster Punkt von φ_b
5      Entferne q vom Stapel φ_b
6      Markiere q als „besucht“
7      Setze e = inzidente Kante von q
8      regionAbarbeiten(e, φ_b, φ_f)
9      Entferne die Markierung „endeckt“ für alle Knoten
10 Entferne die Markierung „besucht“ für alle Knoten
```

Wir verwenden zwei Arten von Knotenmarkierungen: Eine Markierung („besucht“) dient dazu, schon während des Durchlaufs abgearbeitete Zellen (Voronoi-Regionen bzw. Delaunay-Tetraeder) zu kennzeichnen, eine andere („entdeckt“), um von einem besuchten Knoten direkt erreichbare, noch nicht besuchte Knoten auszuzeichnen.

Ebenso benutzen wir zwei Stapel, auf denen zum einen die entdeckten, zum anderen die schon besuchten Knoten verwaltet werden. Dies ermöglicht, deren Markierungen wieder zu entfernen, sobald der Durchlauf abgeschlossen ist.

Die Suche beginnt in einem Startknoten: Dieser wird als schon „besucht“ markiert und auf den Stapel der schon besuchten Knoten gelegt. Man folgt nun allen Kanten, die von diesem Punkt ausgehen[1], um alle Flächen der zu diesem Punkt dualen Region zu finden (vgl. Algorithmus 5.5). Das Abarbeiten dieser Region geschieht rekursiv und ist ebenso eine Tiefensuche.

[1] Da es in der *AQE* mehrere Kanten mit gleichen Start- und Zielpunkten gibt, wählen wir davon repräsentativ eine aus.

Algorithmus 5.5: `regionAbarbeiten`$(e, \varphi_b, \varphi_f)$

```
1  Setze r = Zielpunkt der Kante e
2  Falls r weder als „besucht“ noch als „entdeckt“ markiert ist
3      Füge r dem Stapel φ_b noch nicht besuchter Knoten hinzu
4      Markiere r als „entdeckt“
5      Setze e_dual = duale Fläche zu e
6      Lege e_dual auf den Stapel φ_f
7      Für alle Nachbarflächen e_dual,i von e_dual
8          Setze e_i = duale Kante zu e_dual,i
9          Führe regionAbarbeiten(e_i, φ_b, φ_f) aus
```

Als Startkante bezeichnen wir eine von dem aktuellen Knoten ausgehende Kante. Ist der Zielpunkt dieser Startkante noch nicht als „besucht“ oder „entdeckt“ markiert, wird er auf einen Stapel für noch nicht besuchte Knoten abgelegt und als entdeckt gekennzeichnet. Von dieser ersten Kante ausgehend, besucht man danach gegen den Uhrzeigersinn alle Kanten, die den dualen Flächen entsprechen, die mit der schon gefundenen dualen Fläche der Startkante inzident sind. Für jede dieser neuen Kanten wird ebenso verfahren. Wenn alle Kanten gefunden wurden, wird die „entdeckt“-Markierung für die Punkte, die auf dem Stapel der schon entdeckten Punkte liegen, wieder gelöscht, der Stapel danach ebenso.

Die Suche geht nun mit dem Knoten weiter, der auf dem Stapel der noch nicht besuchten Knoten oben liegt. Die Suche terminiert, sobald alle Knoten als besucht markiert sind.

Um die Flächen der Diagramme für eine Visualisierung aus der *AQE*-Datenstruktur zu gewinnen, navigiert man jeweils über die Kanten des Dualen: Bei der Suche nach Voronoi-Flächen folgt man Kanten der Delaunay-Triangulation, beim Auffinden von Delaunay-Dreiecken entsprechend Voronoi-Kanten. Die Datenstruktur wird demnach zweimal vollständig durchlaufen. Wird bei einem Durchlauf wie zuvor beschrieben ein Knoten als „entdeckt“ markiert, wird die Kante, die zu diesem Knoten hinführt, repräsentativ für ihre duale Fläche einem Stapel hinzugefügt. Da der Startknoten dieser Kante

schon als besucht markiert ist, stellen wir sicher, dass eine Fläche, obwohl sie zu zwei Zellen gehört, nur einmal gefunden wird.

Ist der Durchlauf beendet, wird zu jeder Kante des Stapels die zugehörige duale Fläche berechnet (siehe Abbildung 5.3).

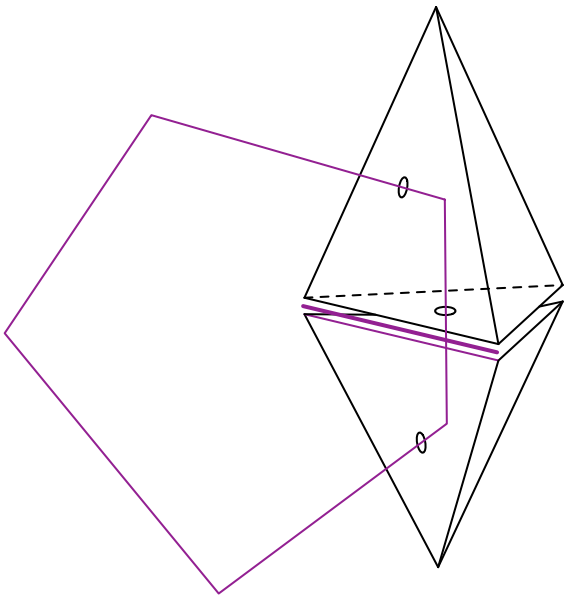


Abbildung 5.3.: Die zu einer Delaunay-Kante duale Voronoi-Fläche.

5.3. Analyse des Algorithmus

5.3.1. Korrektheit

Um zu zeigen, dass der von uns gewählte Algorithmus eine korrekte Delaunay-Triangulation liefert, sind folgende Aussagen zu beweisen (vgl. Edelsbrunner [16]):

1. Reguläre Tetraeder erzeugen größtmögliche leere Sphären.
2. Das Betrachten von *Link Facets* ist ausreichend.

3. Der Algorithmus terminiert.

4. Nichtreguläre Tetraeder sind stets flipbar.

Um die erste Behauptung zu beweisen, betrachten wir eine Menge $U \subset S$ von 4 Punkten und einen Punkt y, der im Inneren des Simplexes σ_U liegt. Nun betrachten wir den Schnitt der in y senkrecht nach oben startenden Halbgeraden mit dem auf die Oberfläche des vierdimensionalen Hyperparaboloids gelifteten Tetraeders (siehe Kapitel 2.3). Dann wird der Abstand zwischen y und dem Zentrum der Umkugel um σ_U maximal, wenn das geliftete Tetraeder ein Delaunay-Tetraeder ist, d. h. wenn es zur unteren konvexen Hülle im $\mathbb{R}^4$ beiträgt.

Um die zweite Behauptung zu zeigen, benötigen wir zunächst folgenden

Satz 5.3.1 *Alle Flips, die berechnet werden müssen, um den nächsten Punkt p_i zur Delaunay-Triangulation hinzuzufügen, genügen folgenden Eigenschaften:*

1. *$p_i \in U$, $U \subset S$ eine Mengen von 5 Punkten, die geflippt werden,*

2. *$\sigma_{\text{alt}} = \sigma_{U \setminus \{p_i\}}$ ist ein Tetraeder aus $\text{DT}_{S_{i-1}}$ und der Flip entfernt es.*

Für den Beweis verweisen wir auf Shah [35]. Damit hätten wir gezeigt, dass jeder Flip einen 3-Simplex σ_{alt} aus $DT_{S_{i-1}}$ entfernt. Alle weiteren 3-Simplizes, die bei einem Flip entfernt werden, teilen den Punkt p_i; die einzige Ausnahme spielt hier der 1-zu-4-Flip, bei dem das Tetraeder $\tilde{\sigma}$ den Punkt p_i nicht als Ecke aufweist. σ_{alt} und $\tilde{\sigma}$ teilen jedoch einen 2-Simplex (also eine Fläche), der gerade ein *Link Facet* ist, und bevor der Flip ausgeführt wird, ist dieses *Link Facet* flipbar und nicht regulär. Damit haben wir also gezeigt, dass es genügt, sich auf die *Link Facets* zu beschränken.

Da jeder Flip den oben erwähnten Abstand für alle Punkte $y \in ch(U)$ maximiert, aber für $y \notin ch(U)$ unverändert lässt, kann die Vergrösserung des Abstandes als Maß für den Fortschritt des Algorithmus interpretiert werden. Damit wird ein bereits entferntes *Link*

Facet kein weiteres Mal eingefügt. d. h. im Schritt 2 des Algorithmus 5.3 kann sich keine Endlosschleife bilden.

Nun bleibt nur noch der vierte Punkt zu zeigen, d. h. wenn nicht reguläre *Link Facets* existieren, dann gibt es mindestens ein flipbares. Sei T die aktuelle Triangulation (noch nicht die Delaunay-Triangulation DT_{S_i}) bei der Abarbeitung des Punktes p_i. Dann reicht es zu zeigen, dass T mindestens ein reflexes und flipbares *Link Facet* enthält.

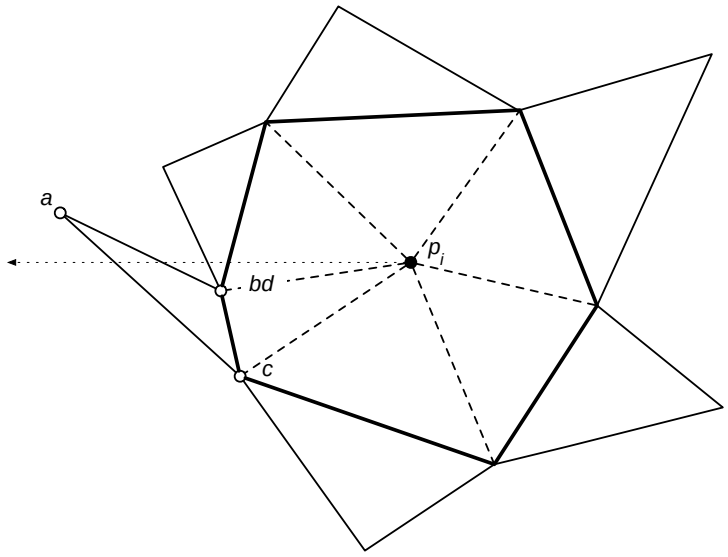


Abbildung 5.4.: Die breiten Linien repräsentieren *Link Facets* von p_i, und die außerhalb der breiten Linien eingezeichneten Tetraeder gehören zu L (nach Edelsbrunner [14]).

Wir verwenden einen Widerspruchsbeweis und nehmen dafür an, dass alle reflexen *Link Facets* nicht flipbar sind. Sei L die Menge aller Tetraeder, die außerhalb des Sterns von p_i liegen, d. h. alle Tetraeder, die den Knoten p_i nicht als Ecke haben und mindestens ein *Link Facet* als ein Seitendreieck besitzen. Sei $L' \subseteq L$ diejenige Menge von Tetraedern, deren *Link Facets* zu p_i nicht regulär sind; dann gilt nach Annahme $L' \neq \emptyset$. Betrachte nun jedes Tetraeder aus L und fasse die Distanz zwischen dem Umkugelzentrum und p_i als die Funktion f auf. Sei f für das Tetraeder $\sigma_{\{a,b,c,d\}}$ minimal.

Mit der obigen Terminologie bleibt zu zeigen, dass das *Link Facet* $\lambda_{\{b,c,d\}}$ flipbar ist. Dafür nehmen wir an, dass $\lambda_{\{b,c,d\}}$ nicht flipbar ist und bringen dies zum Widerspruch. Sei bd eine reflexe Kante der Tetraeder $\sigma_{\{a,b,c,d\}}$ und $\sigma_{\{b,c,d,p_i\}}$ und sei $\sigma_{\{a,b,d,x\}}$ das Tetraeder auf der anderen Seite des Dreiecks $tria_{\{a,b,d\}}$. Wäre die Kante bd als einzige nicht reflex, dann muss $x \neq p_i$ sein, ansonsten wäre $\lambda_{\{b,c,d\}}$ flipbar. Sei also bc eine weitere reflexe Kante. Wenn nun ein Tetraeder $\sigma_{\{a,b,c,y\}}$ auf der anderen Seite von $tria_{\{a,b,c\}}$ und $x = y = p_i$ wäre, könnte $\lambda_{\{b,c,d\}}$ geflippt werden. Also sei $x \neq p_i$, mit anderen Worten: $tria_{\{a,b,d\}}$ ist kein *Link Facet* von p_i. Wenn wir nun eine Halbgerade betrachten, die aus p_i startet und einen Punkt aus dem Inneren von $tria_{\{a,b,d\}}$ durchläuft (siehe Abbildung 5.4), dann schneidet sie, bevor $\sigma_{\{a,b,c,d\}}$ erreicht wird, ein reguläres Tetraeder aus $DT_{S_{i-1}}$. Aber die Funktion f für dieses Tetraeder ist wegen Lemma 3.2 nach Shah [35] größer als für $\sigma_{\{a,b,c,d\}}$, und dies steht im Widerspruch zur Minimalität von f bzgl. des Tetraeders $\sigma_{\{a,b,c,d\}}$. Damit haben wir also gezeigt, dass stets Flips bis zum Erreichen von DT_{S_i} durchgeführt werden.

5.3.2. Aufwandsabschätzung

5.3.2.1. Speicherplatz

Satz 5.3.2 *Der Speicherplatzbedarf der* AQE*-Datenstruktur des Voronoi-Diagramms und der Delaunay-Triangulation einer beliebigen Punktmenge $S \subset \mathbb{R}^3$ liegt in $O(n^2)$. Ist die Punktmenge im Raum gleichverteilt, erwarten wir einen Speicherplatzbedarf von $O(n)$.*

Beweis: Die Speicherplatzbedarf der *AQE*-Datenstruktur hängt von der Gesamtanzahl der Kanten und Knoten im Voronoi-Diagramm und in der Delaunay-Triangulation ab. Die Anzahl der Knoten ist n; die Gesamtanzahl der Kanten schätzen wir durch die Anzahl der Tetraeder der Delaunay-Triangulation ab. Dazu verteilen wir die *AQE*-Kanten wie folgt auf alle Tetraeder:

Ein Tetraeder besteht aus sechs Kanten; jede dieser Kanten entspricht einer *Quad Edge*. Darüber hinaus existieren zu jedem der vier Dreiecke jeweils drei duale Kanten. Deren

Quad Edges können dem Tetraeder aber nur zur Hälfte angerechnet werden: Die `twin`-Kante einer aus dem Tetraeder herausführenden dualen Kante wird ebenso wie die Kante `twin rot` dem Nachbartetraeder angerechnet.

Damit erhalten wir $6 \cdot 4 + \frac{(3 \cdot 4) \cdot 4}{2} = 48$ Kanten mit jeweils drei Pointern `rot`, `org` und `onext`. Für ein Tetraeder müssen also insgesamt 48 Kanten und $3 \cdot 48 = 144$ Verweise gespeichert werden.

Die Anzahl der Tetraeder ist im *worst case* $O(n^2)$, der Speicherplatzbedarf der *AQE*-Datenstruktur ist deshalb ebenso quadratisch zur Anzahl der Punkte aus S.

Für beliebig im Raum verteilte Punktmengen hilft uns der Satz 2.5.1 aus Kapitel 2.5: Die erwartete Anzahl an Tetraedern liegt in $O(n)$. Deshalb ist der Speicherplatzbedarf der *AQE*-Datenstruktur in diesem Fall ebenso linear zur Anzahl der Punkte aus S.

Da aber die erwartete Anzahl an Tetraedern einer Delaunay-Triangulation einer im Raum gleichverteilten Punktmenge S mit n Punkten bei $6.77n$ liegt (vgl. Dwyer [13]), ist der konstante Faktor der O-Notation mit etwa 975 trotzdem sehr hoch.

Nun schätzen wir den Speicherplatzbedarf der Delaunay-Triangulation ab.

Satz 5.3.3 *Sei $S \subset \mathbb{R}^3$ eine beliebige Punktmenge. Die Komplexität des* History-DAG *einer Delaunay-Triangulation von S ist $O(n^2)$; die erwartete Komplexität für eine im Raum gleichverteilte Punktmenge S ist $O\left(n^{\varepsilon} + f(n)\right), \varepsilon > 0$, falls die erwartete Anzahl an Tetraedern der Delaunay-Triangulation durch eine Funktion $f(n)$ beschränkt ist, wobei $\frac{f(n)}{n^{\varepsilon}}$ monoton steigt.*

Beweis: Der Satz ist eine Folgerung aus Lemma 3.6 und Lemma 3.7 von Shah [35]. Er betrachtet dabei $(d+1)$-dimensionale Teilmengen T von S und deren d-Simplizes σ_T (bei uns also Tetraeder), die mit genau k anderen Punkten aus S in Konflikt stehen und $\omega = |\Omega| = |T \cap S_0|$ Eckpunkte des allumfassenden Simplexes (σ_{S_0}, also bei uns das allumfassende Tetraeder) besitzen; die Menge all dieser Teilmengen bezeichnet er mit G_k^{Ω}. Er berechnet die Wahrscheinlichkeit, dass diese Simplizes regulär sind (also zu

einem Zeitpunkt der Konstruktion in einer regulären Triangulation vorkommen). Damit ein solcher Simplex regulär ist, müssen die $d+1$ Punkte des Simplexes (bei uns die vier Punkte des Tetraeders) in die Triangulation eingefügt werden, bevor die k Punkte eingefügt werden, mit denen der Konflikt besteht.

Der Erwartungswert ist dann eine dreifache Summe:

$$E = \sum_{\Omega \subseteq S_0} \sum_{k=0}^{n} \sum_{T \in G_k^\Omega} \textit{Wahrsch.}(\sigma_T \textit{ ist Teil einer regulären Triangulation } DT_k).$$

Durch Umformen erhält man letztendlich

$$E \leq \sum_{\Omega \subseteq S_0} l(l!) \sum_{k=1}^{n} \frac{cn^{\lfloor l/2 \rfloor}}{k^{1+\lfloor l/2 \rfloor}} = O\left(n^{\lceil d/2 \rceil}\right),$$

wobei c eine positive Konstante und $l \leq d+1$ ist. Der zweite Teil folgt aus der Annahme für im Raum gleichverteilte Punktmengen, dass, falls die Anzahl der Flächen auf der konvexen Hülle durch $f(n)$ beschränkt ist, gilt:

$$E\left(|G_{k \leq j}^\Omega|\right) \leq O\left(j^l f(n/j)\right).$$

Daraus folgt dann die zweite Behauptung:

$$E \leq \sum_{\Omega \subseteq S_0} l(l!) \sum_{k=1}^{n} \frac{c \cdot f(\frac{n}{k})}{k} = O\left(f(n)\right).$$

Die Anzahl der Tetraeder haben wir schon in Satz 2.5.1 aus Kapitel 2.5 abgeschätzt. Daraus folgt die Behauptung.

5.3.2.2. Laufzeit

Satz 5.3.4 *Der Durchlauf durch die* AQE*-Datenstruktur einer Delaunay-Triangulation und eines Voronoi-Diagrammes einer Punktmenge S ist linear zum Speicherplatzbedarf*

der AQE-*Datenstruktur.*

Beweis: Bei einem Durchlauf durch die *AQE*-Datenstruktur (siehe Algorithmen 5.2 und 5.2) wird jeder Knoten genau einmal besucht. Beim Abarbeiten eines einzelnen Knotens läuft man über alle Flächen der Zelle (alle Delaunay-Dreiecke eines Tetraeders bzw. alle Bisektorflächen einer Voronoi-Region). Die Komplexität einer Zelle lässt sich abschätzen, indem man die Anzahl der Kanten der *AQE*-Datenstruktur (siehe Satz 5.3.2) durch die Größe von S teilt. Daraus folgt die Behauptung.

Satz 5.3.5 *Der Zeitaufwand zur Durchführung aller Flips bei der Berechnung einer Delaunay-Triangulation einer Punktmenge S ist linear zum Speicherplatzbedarf des* History-DAG.

Beweis: Der *History-DAG* wird nur durch Flips erweitert. Für jeden Flip vergrößert sich der *DAG* um konstant viele Knoten, und die Bearbeitung eines Flips benötigt konstant viel Zeit. Daraus folgt die Behauptung.

Satz 5.3.6 *Der Zeitaufwand zur* Point Location *bei der Berechnung einer Delaunay-Triangulation einer beliebigen Punktmenge $S \subset \mathbb{R}^3$ ist $O(n \log n + n^{\lceil \frac{n}{2} \rceil})$. Ist S im Raum gleichverteilt, ist der Zeitaufwand $O\left(\sum_{k=1}^{n} f(\frac{n}{k})\right)$, wobei die erwartete Anzahl der Tetraeder der Delaunay-Triangulation durch $f(n)$ beschränkt ist.*

Beweis: Der Satz ist eine Folgerung aus Lemma 3.8 von Shah [35], das auf Lemma 3.7 aufbaut. Da für die Mengen $T \in G_k^\Omega$ jeweils k Punkte innerhalb der Umkugeln der Tetraeder σ_T liegen, müssen k Flips ausgeführt werden, um die Regularität wieder herzustellen.

$$
\begin{aligned}
E &= \sum_{\Omega \subseteq S_0} \sum_{k=0}^{n} \sum_{T \in G_k^\Omega} k \cdot \textit{Wahrsch.}(\sigma_T \textit{ ist Teil einer regulären Triangulation } DT_k). \\
&\leq \sum_{\Omega \subset S_0} l(l!) \sum_{k=1}^{n} \frac{cn^{\lfloor l/2 \rfloor}}{k^{\lfloor l/2 \rfloor}} \\
&= O(n \log n + n^{\lceil d/2 \rceil}).
\end{aligned}
$$

Für im Raum gleichverteilte Punktmengen gilt analog zu Satz 5.3.3:

$$E \leq \sum_{\Omega \subseteq S_0} l(l!) \sum_{k=1}^{n} c \cdot f\left(\frac{n}{k}\right) = O\left(\sum_{k=1}^{n} f\left(\frac{n}{k}\right)\right).$$

Da $f(n)$ in unserem Fall in $O(n)$ liegt, können wir eine Laufzeit von $O(n \log n)$ erwarten.

6. Rahmenbedingungen unserer Implementation

Wir haben gesehen, dass eine Vielzahl an Programmen und Geometriebibliotheken zur Berechnung und Visualisierung von Voronoi-Diagrammen existiert. Dabei handelt es sich sowohl um freie als auch um kommerzielle Software. Die hier vorgestellten Programme bieten eine große Bandbreite an geometrischen Algorithmen und beschränken sich nicht auf Voronoi-Diagramme und Delaunay-Triangulationen bestimmter Dimensionen. Der Preis dieses Funktionsumfangs ist die hohe Einarbeitungszeit, die zur Nutzung der Bibliotheken nötig ist, zumal eine Dokumentation meist nur oberflächlich vorhanden ist. Bei kommerziellen Produkten schrecken zudem die Kosten ab; bei freier Software wie CGAL ist die Lizenzierung teilweise nicht vollkommen transparent.

Die hier vorgestellten Programme verstehen sich in der Regel als reine Geometriebibliotheken oder Visualisierungsprogramme. Unsere Konzeption sieht eine Vereinigung dieser beiden Elemente vor. Der Funktionsumfang umfasst sowohl die Berechnung als auch die Visualisierung von Voronoi-Diagrammen, Delaunay-Triangulationen und konvexen Hüllen von Punktmengen im Raum. Die Bedienung soll intuitiv und ohne ein tiefes Verständnis der Materie möglich sein.

Punktlokalisierung und Aktualisierung der Triangulation Das Programm sollte als Lehrmedium einsetzbar sein, also die Möglichkeit bieten, die Entstehung des Voronoi-Diagramms und der Delaunay-Triangulation sichtbar zu machen. Die Punktlokalisierung

und Aktualisierung der Triangulation haben wir deshalb denen des Algorithmus von Edelsbrunner und Shah nachempfunden; er erschien uns seiner Eleganz wegen am besten vermittelbar.

Andere Algorithmen bieten für spezielle Punktkonstellationen eine effizientere Berechnung der Delaunay-Triangulation, doch sie verwenden Techniken, die auf den Konzepten der klassischen Algorithmen aufbauen, zu denen auch der von Edelsbrunner und Shah zählt. In unseren Augen bildet das Verständnis dieses Algorithmus für Studenten der Algorithmischen Geometrie eine solide Basis.

Der Speicherplatzbedarf ist wegen der Verwendung eines *DAG* zur Punktlokalisierung für beliebig verteilte Punktmengen quadratisch zur Größe der Punktmenge. Doch da wir in der Regel mit zufällig erzeugten, im Raum gleichverteilten Punktmengen arbeiten werden, können wir einen Speicherplatzbedarf in $O(n)$ erwarten (vgl. Satz 5.3.3).

Die über alle Einfügereihenfolgen gemittelte Laufzeit des Algorithmus liegt in $O(n \log n + n^{\lceil d/2 \rceil})$. Für die Berechnung von Voronoi-Diagrammen im Raum bedeutet dies für beliebig verteilte Punktmengen einen quadratischen Zeitaufwand, doch da der Algorithmus output-sensitiv arbeitet (und der *History-DAG* in diesem Fall nur $O(n)$ viele Knoten besitzt), kann bei der Verwendung gleichverteilter Punktmengen eine Laufzeit von $O(n \log n)$ erwartet werden.

Da das Primärziel unseres Programm die Visualisierung ist, dürften ohnehin kaum große Punktmengen in Frage kommen; Laufzeitverhalten und Speicherplatzbedarf erachten wir als weniger relevant.

Robustheit Im Bezug auf Arithmetik wollen wir unser Programm flexibel gestalten. Es soll die Möglichkeit bieten, zusätzliche Nummernklassen hinzuzufügen. Für eine Standardklasse erscheint uns eine auf dem Double-Datentyp aufbauende Klasse sinnvoll: Sie bietet mit 64 Bit eine sehr hohe Genauigkeit, und da wir in der Regel mit zufällig generierten, im Raum gleichverteilten Punktmengen arbeiten, wird auch die Wahrscheinlichkeit,

nicht in allgemeiner Lage befindliche Punktmengen zu generieren, sehr gering. Eine Überprüfung dieses Postulats findet sich in Kapitel 11. Die hohe Genauigkeit (und die damit einhergehenden Geschwindigkeitseinbußen) von `double` ist in unseren Augen insofern kein Nachteil, als selten große Punktmengen bearbeitet werden sollen.

Falls sich Punkte nicht in allgemeiner Lage befinden sollten, beschränken wir uns darauf, dies bestmöglich zu erkennen und eine Fehlermeldung anzuzeigen. Da die Wahrscheinlichkeit dafür bei zufällig erzeugten Punktmengen gering ist, haben wir auf die Möglichkeit, die allgemeine Lage zu simulieren, verzichtet; es würde aber durchaus eine sinnvolle Ergänzung darstellen.

Repräsentation des Voronoi-Diagramms Eine geeignete Datenstruktur, die eine nachträgliche Berechnung des Voronoi-Diagramms aus der Delaunay-Triangulation überflüssig macht, ist die *Augmented Quad Edge*-Datenstruktur. Sie wird während der Konstruktion der Delaunay-Triangulation aufgebaut und enthält beide Diagramme. Sobald die Berechnung der Delaunay-Triangulation abgeschlossen ist, liegt auch das Voronoi-Diagramm vor.

Fazit Eine freie, webbasierte und erweiterbare Software zur Berechnung und Visualisierung der Delaunay-Triangulation und des Voronoi-Diagramms im Raum, die eine unkomplizierte Bedienbarkeit gewährleistet und zudem die Konstruktion der Diagramme erlebbar macht, ist nach unserer Recherche nicht existent. Diese Lücke möchten wir mit unserem Programm schließen.

Teil II.

Implementierung / Umsetzung

7. Implementierung der Delaunay-Triangulation

In diesem Kapitel beschreiben wir die Art und Weise der Umsetzung der theoretischen Überlegungen aus Kapitel 5. Dafür erläutern wir zunächst unsere abstrakte Zahlenklasse und die darauf aufbauenden grundlegenden Geometrien, also die Punkte und die Tetraeder; schließlich folgen die für die Konstruktion der Delaunay-Triangulation benötigten Elemente, wie z. B. der *History-DAG*.

7.1. Das Geometriepaket

Das Geometriepaket stellt grundlegende arithmetische und geometrische Klassen zur Verfügung (siehe Abbildung 7.1), die später bei der Berechnung der Delaunay-Triangulation verwendet werden. Es dient nicht nur als Grundbaustein dieser Arbeit, sondern ist für die weitere Verwendung für arithmetische bzw. geometrische Algorithmen konzipiert. Es ist vollkommen unabhängig von den restlichen Klassen dieser Arbeit, wie dies aus dem UML-Diagramm auf Seite 93 zu ersehen ist. Es enthält die folgenden Klassen:

`V3D_GEOM_Number` Stellt eine abstrakte Zahlenklasse dar, die die Schnittstelle für später benötigte arithmetische Operationen definiert.

`V3D_GEOM_FastNumber` Implementiert die abstrakten Zahlenklasse, die in unserem Programm verwendet wird. Diese ist auch als Standardzahlenklasse des Programms

definiert.

`V3D_GEOM_Point3D` Repräsentiert einen Punkt bzw. Vektor im dreidimensionalen Raum und stellt notwendige Methoden wie Skalarmultiplikation, Vektoraddition, Vektorsubtraktion, Vektorprodukt etc. zur Verfügung.

`V3D_GEOM_Tetrahedron` Repräsentiert ein gegen den Uhrzeigersinn orientiertes Tetraeder (siehe Abbildung 7.2), das vier Punkte enthält und Methoden bereitstellt wie die Entscheidung, ob ein Punkt innerhalb des Tetraeders liegt, die Berechnung des Mittelpunktes der tetraederumschließenden Sphäre etc.

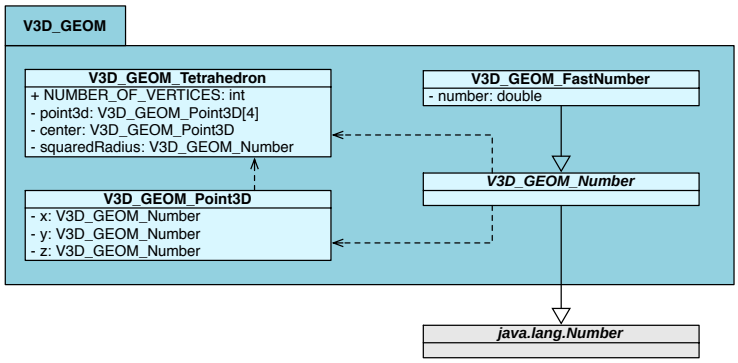


Abbildung 7.1.: UML-Diagramm des Geometriepaketes.

7.1.1. Arithmetische Klassen

Dieser Abschnitt beschreibt die dieser Arbeit zugrundeliegenden arithmetischen Klassen.

7.1.1.1. Die abstrakte Zahlenklasse

Diese Klasse erbt von der abstrakten Java-Zahlenklasse `java.lang.Number` und erweitert diese um Operationen, die auch für unser Programm benötigt werden. Unter anderen haben wir Schnittstellen für Vergleichsoperationen (`isSmaller`, `isGreater`, `equals`), Rechnungsoperationen (`plus`, `minus`, `times`, `dividedBy`, `square`, `squareRoot`), Operationen,

um abstrakte Zahlen mit gegebenen primitiven Zahlendatenstrukturen wie `int`, `float` etc. belegen zu können. Die Methode `isInRange` überprüft, ob eine Zahl innerhalb eines gegebenen ε-Streifens liegt. Mit der Methode `random` kann eine Zufallszahl, die zwischen den gegebenen minimalen und maximalen Werten liegt, erzeugt werden.

Um die für unsere Arbeit verwendeten Algorithmen, die weiter hinten im Dokument beschrieben werden, möglichst flexibel zu implementieren, setzen diese auf die abstrakte Zahlenklasse auf. Dadurch ist es möglich, andere (z. B. robuste) Zahlenklassen zur Berechnung zu verwenden; diese brauchen lediglich von unserer abstrakten Zahlenklasse zu erben.

7.1.1.2. Die schnelle Zahlenklasse

Die schnelle Zahlenklasse erbt von der abstrakten Zahlenklasse und benutzt als Halter für den Zahlenwert die primitive Java-Datenstruktur `double`. Sie implementiert die vorgegebenen Schnittstellen der abstrakten Oberklassen `java.lang.Number` und `V3D_GEOM_Number` und dient bei allen Algorithmen im gesamten Verlauf dieser Arbeit als die Umsetzung der abstrakten Zahlenklasse.

7.1.2. Geometrische Klassen

In diesem Abschnitt beschreiben wir den zweiten Teil des Geometriepaketes, das die Punktklasse `V3D_GEOM_Point3D` und die Tetraederklasse `V3D_GEOM _Tetrahedron` enthält. Diese beiden Klassen bilden die geometrische Grundstruktur der späteren Berechnungen und Klassen, die für die Algorithmen verwendet werden. Schließlich beschreiben wir eine für die Implementation der Delaunay-Triangulation ausschlaggebende Methode des Orientierungstests.

7.1.2.1. Die Punktklasse

Die Klasse `V3D_GEOM_Point3D` repräsentiert einen dreidimensionalen Punkt bzw. Vektor, dessen Koordinaten die in Kapitel 7.1.1.1 beschriebene abstrakte Zahlenklasse verwendet. Sie stellt Berechnungsmethoden (`plus`, `minus`, `times`, `dividedBy`, `cartesianProdukt`) und Vergleichsmethoden (`isLonger`, `isShorter`, `isAsLongAs`, `equals`) für Vektoren zur Verfügung. Sie besitzt weiterhin Konvertierungsmethoden in andere Punktformate wie die von Java3D verwendeten Klassen `Vector3d`, `Vector3f`, `Point3d` und `Point3f`. Weitere Methoden sind vorhanden, um

- den Mittelpunkt des Umkreises eines durch drei Punkte gegebenen Dreiecks (`getCircumcenter`),
- den Winkel zwischen zwei Vektoren (`angle`),
- einen oder eine Menge von zufälligen Punkten innerhalb einer Sphäre bzw. innerhalb eines Bereichs (`getRandomPoints`)

zu berechnen.

Eine besonders wichtige Methode ist `isPositiveOriented` mit zwei unterschiedlichen Signaturen, denn diese entscheidet, ob entweder vier gegebene Punkte im Sinne der Abbildung 7.2 positiv orientiert sind oder ob ein gegebener Punkt oberhalb bzw. unterhalb dreier weiterer gegebener Punkte liegt (siehe Kapitel 7.1.2.3).

7.1.2.2. Die Tetraederklasse

Die Klasse `V3D_GEOM_Tetrahedron` repräsentiert ein Tetraeder, das hauptsächlich aus vier Punkten der Klasse `V3D_GEOM_Point3D`, dem Mittelpunkt der das Tetraeder umschließenden Sphäre und deren Radius besteht. Die letzten beiden werden nur dann berechnet, wenn die aktuell vorhandenen Werte nicht mehr korrekt sind; diese Aktualitätsinformation wird

mittels eines *Flags* über die Lebenspanne eines Tetraeders mitgeführt. Die vier Punkte des Tetraeders sind gemäß Abbildung 7.2 gegen den Uhrzeigersinn orientiert.

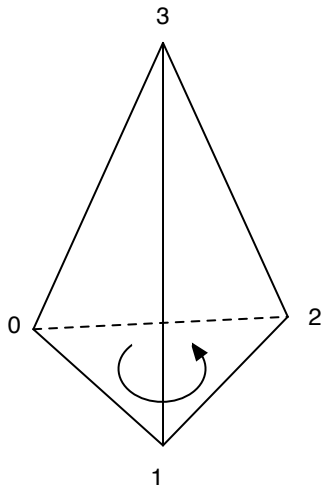


Abbildung 7.2.: Ausrichtung des Tetraederpunktes gegen den Uhrzeigersinn.

Um die Sphäre zu berechnen, die das Tetraeder umfasst, gehen wir analog zu Bourke [5] vor:

Gegeben seien vier Punkte p_1, p_2, p_3 und p_4, mit $p_i := \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$. Dann ergibt sich die Sphärengleichung durch die Lösung des folgenden Gleichungssystems:

$$\begin{vmatrix} x^2+y^2+z^2 & x & y & z & 1 \\ x_1^2+y_1^2+z_1^2 & x_1 & y_1 & z_1 & 1 \\ x_2^2+y_2^2+z_2^2 & x_2 & y_2 & z_2 & 1 \\ x_3^2+y_3^2+z_3^2 & x_3 & y_3 & z_3 & 1 \\ x_4^2+y_4^2+z_4^2 & x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0.$$

Dabei muss beachtet werden, dass die Punkte in allgemeiner Lage sind; ansonsten wirft das Programm die Ausnahme `V3D_EXCEPTION_GeneralPosition`. Lösen wir die Determinante nach der ersten Zeile auf, so erhalten wir:

$$(x^2+y^2+z^2) \cdot M_1 - x \cdot M_2 + y \cdot M_3 - z \cdot M_4 + M_5 = 0;$$

dabei bezeichnen M_1 bis M_5 die entsprechenden Unterdeterminanten.

Da die Sphärengleichung mit Radius r und Mittelpunkt $c = \begin{pmatrix} x_0 \\ z_0 \\ y_0 \end{pmatrix}$ der folgenden Gleichung

$$(x - x_0)^2 + (x - x_0)^2 + (x - x_0)^2 = r^2$$

entspricht, folgt für die Lösung unseres Problems

$$\begin{aligned} x_0 &= \frac{M_2}{2 \cdot M_1} \\ y_0 &= \frac{M_3}{2 \cdot M_1} \\ z_0 &= \frac{M_4}{2 \cdot M_1} \\ r^2 &= x_0^2 + y_0^2 + z_0^2 - \frac{M_5}{M_1}. \end{aligned}$$

Zu beachten ist, dass die obigen Gleichungen nur für $M_1 \neq 0$ zu lösen sind, was einer Verteilung entspricht, bei der sich die Punkte in allgemeiner Lage befinden.

7.1.2.3. Der Orientierungstest

Die beiden Methoden des Orientierungstests sind für die Ausführung der entsprechenden Flips bei der Berechnung der Delaunay-Triangulation entscheidend.

Der Orientierungstest wird zunächst zur Überprüfung der positiven Orientierung eines Tetraeders herangezogen. Dabei wird überprüft, ob jeder Punkt oberhalb der durch das Dreieck (das durch die restlichen Punkte des Tetraeders bestimmt ist) definierten Fläche liegt. Sollte dies nicht der Fall sein, so werden zwei Punkte des Tetraeders ausgetauscht, wodurch sichergestellt wird, dass eine positive Orientierung vorhanden ist.

Bei der Delaunay-Triangulation spielt die schnelle Lokalisierung eines Punktes in den bereits vorhandenen Tetraedern eine zentrale Rolle. Um diese Aufgabe zu lösen, verwenden wir einen *History-DAG*, in dem der aktuelle Anfragepunkt in den jeweiligen Kno-

ten des *DAG*, die einzelnen Tetraedern entsprechen, lokalisiert wird. Hierfür verwenden wir die Methode `isInTetrahedron`, die für den Anfragepunkt überprüft, ob dieser jeweils oberhalb der vier Dreiecksflächen des Tetraeders liegt. Dafür werden die Flächen – bei vorausgesetzt positiv orientierten Tetraedern – wie folgt definiert: $\Delta_0 = \{p_1, p_3, p_2\}$, $\Delta_1 = \{p_0, p_2, p_3\}$, $\Delta_2 = \{p_0, p_3, p_1\}$ und $\Delta_3 = \{p_0, p_1, p_2\}$. Ist der Punkt oberhalb jeder Fläche, so befindet sich der Punkt im Tetraeder; ist jedoch die Bedingung für mindestens eine Fläche verletzt, so liegt der Punkt außerhalb.

Dieselbe Methode verwenden wir ebenfalls bei der Überprüfung, ob eine Kante reflex oder konvex ist. Diese Tatsache ist wiederum zentral bei der Entscheidung über die Art des auszuführenden Flips bzw. ob ein Flip überhaupt stattfindet (vgl. Kapitel 5.1.3).

Bei dem Orientierungstest verwenden wir einen ε-Streifen. Da die standardmäßig verwendete Zahlenklasse nicht exakt arbeitet, kann es zu Rundungsfehlern kommen. Falls der Wert der Determinante innerhalb dieses ε-Streifens liegt, interpretieren wir das Ergebnis als Null: In diesem Fall erachten wir die vier Punkte als koplanar. Die Größe des ε-Streifens kann in der Konfigurationsdatei festgelegt werden; der von uns voreingestellte Wert, der sich als guter Kompromiss zwischen Robustheit und Ausnahme von „Punkten in nicht allgemeiner Lage“ ergeben hat, beträgt 10^{-20}.

7.2. Das Delaunay-Paket

Das Delaunay-Paket besteht aus fünf Klassen:

`V3D_DT` Repräsentiert und berechnet die Delaunay-Triangulation und stellt Methoden zur Überprüfung ihrer Korrektheit zur Verfügung.

`V3D_DT_Tetrahedron` Erbt von der Klasse `V3D_GEOM_Tetrahedron` und erweitert sie: Diese Tetraeder kennen die Nachbartetraeder, die inzidenten Kanten und den auf dieses Tetraeder verweisenden *History-DAG*-Knoten. Weiterhin weiß es, ob es ein Teil der

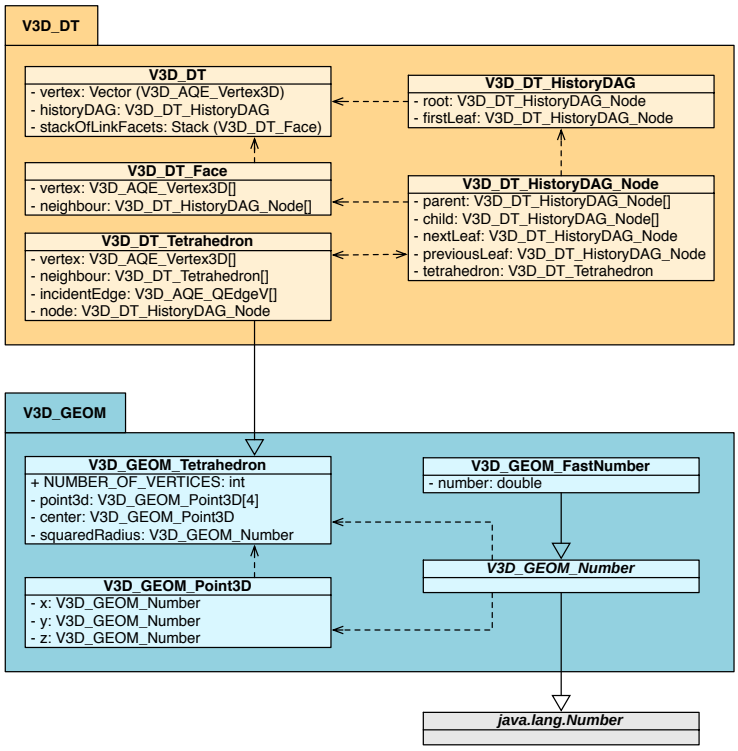


Abbildung 7.3.: UML-Diagramm des Delaunay-Paketes.

aktuellen Triangulation und ob es ein Hilfstetraeder ist, d. h. ob es einen Punkt mit dem allumfassenden Tetraeder gemeinsam hat.

`V3D_DT_Face` Repräsentiert ein *Link Facet* und wird zur Berechnung der Flips benötigt. Ein Objekt dieser Klasse kennt die das Dreieck definierenden Eckpunkte, die Nachbartetraeder und die Indizes derjenigen Punkte in den Nachbartetraedern, die nicht auf dem Dreieck liegen; weiterhin verfügt es über Methoden zur Überprüfung der Regularität.

`V3D_DT_HistoryDAG` Repräsentiert einen *History-DAG* und wird für die schnelle Lokalisierung von Anfragepunkten in der aktuellen Delaunay-Triangulation verwendet. Der *DAG* verweist auf den Wurzelknoten und den ersten Blattknoten; darüber hinaus werden Methoden zur Suche und der Manipulation des *DAG* zur Verfügung gestellt.

`V3D_DT_HistoryDAG_Node` Repräsentiert einen Knoten im *History-DAG* und kennt seine Eltern und Kinder bzw. seine Tiefe im *DAG*. Weiterhin kennt der Knoten, falls er selbst ein Blatt ist, noch die Vorgänger- bzw. Nachfolgerblätter, was eine Navigation durch alle Blattknoten in linearer Zeit zur Anzahl der Blätter ermöglicht.

7.2.1. `V3D_DT`

Die Klasse `V3D_DT` übernimmt die Berechnung der Delaunay-Triangulation unter Verwendung der Klassen `V3D_DT_Tetrahedron`, `V3D_DT_Face`, `V3D_DT_HistoryDAG` und `V3D_DT_HistoryDAG_Node`. Sie wird mit einer Punktmenge, d. h. einem `Vector` von `V3D_GEOM_Point3D` initialisiert.

7.2.1.1. Klassenvariablen

Eine Delaunay-Triangulation wird im Wesentlichen durch eine Punktmenge (einen `Vector` mit Namen `vertex`, der aus Instanzen von `V3D_AQE_Vertex3D` besteht) und einen *History-DAG* (einen `V3D_DT_HistoryDAG` mit Namen `historyDAG`) repräsentiert. Um nach dem Einfügen eines neuen Punktes aus der entstandenen Triangulation wieder eine Delaunay-Triangulation zu gewinnen, bedarf es eines Stapels (`stackOfLinkFacets`), der die *Link Facets* vom Typ `V3D_DT_Face` verwaltet. Darüberhinaus wird der Index des nächsten einzufügenden Punkts in `indexOfNextPoint` festgehalten.

Eine weitere Variable der Klasse ist `stepCount`. Für jeden durchgeführten Flip wird dieser Wert um eins erhöht. Mit Hilfe von `stepCount` ist es möglich, das Rückgängigmachen von Flips zu simulieren: Anstatt den k-ten Flip ungeschehen zu machen, wird die Triangulation noch einmal von vorne berechnet, bis der `stepCount` dem Wert $k-1$ entspricht. Um gleich zu dem Zustand zurückzuspringen, der vor dem Einfügen eines neuen Punktes herrschte, werden solche Werte des Schrittzählers auf einem Stapel abgelegt (`stackOfPointMarkers`). Will man nun den zuletzt eingefügten Punkt entfernen, wird

der oberste Wert des Stapels entfernt und der Triangulationsaufbau bis zu dem besagten Schritt wiederholt.

7.2.1.2. `getSmallestTetrahedronSurroundingAllPoints`

Unser Ansatz sieht vor, die Punktmenge so zu skalieren, dass sie wahlweise innerhalb einer Sphäre um den Ursprung mit Radius 1 oder innerhalb eines Würfels um den Ursprung mit Kantenlänge zwei lebt.

Wir wählen hier ein Tetraeder, das die Punktmenge enthält: Damit wird der Algorithmus initialisiert. Die Punkte des allumfassenden Tetraeders wählen wir wie folgt:

$$\begin{aligned} p_0 &= \rho \cdot (\quad -\sqrt{6} \quad , \quad -\sqrt{2} \quad , \quad -1 \quad) \\ p_1 &= \rho \cdot (\quad \sqrt{6} \quad , \quad \sqrt{2} \quad , \quad 1 \quad) \\ p_2 &= \rho \cdot (\quad 0 \quad , \quad 2 \cdot \sqrt{2} \quad , \quad -1 \quad) \\ p_3 &= \rho \cdot (\quad 0 \quad , \quad 0 \quad , \quad 3 \quad); \end{aligned}$$

dabei ist $\rho = \sqrt{6}$ der Skalierungsfaktor, damit die im Intervall $]-1, 1[$ zufällig erzeugten Punkte innerhalb dieses Tetraeders liegen.

Darüberhinaus wählen wir eine Konstante `SCALE_FACTOR`, um die wir die Eckpunkte des Tetraeders skalieren: Wie in Kapitel 5.1.1 beschrieben, hängt die Korrektheit der berechneten Delaunay-Triangulation von der Größe dieses Tetraeders ab. In unseren Berechnungen waren Triangulationen, die mit zu kleinen umfassenden Tetraedern konstruiert wurden, zwar oft regulär, aber selten konvex (siehe Abbildung 7.4).

Durch die Wahl einer großen Konstanten werden Fehler zwar nicht ausgeschlossen, aber sehr unwahrscheinlich. Um eine geeignete Konstante zu finden, haben wir für verschieden große, zufällige, im Raum gleichverteilte Punktmengen Delaunay-Triangulationen mit steigendem Skalierungsfaktor berechnet und die Korrektheit der entstandenen Triangulation überprüft.

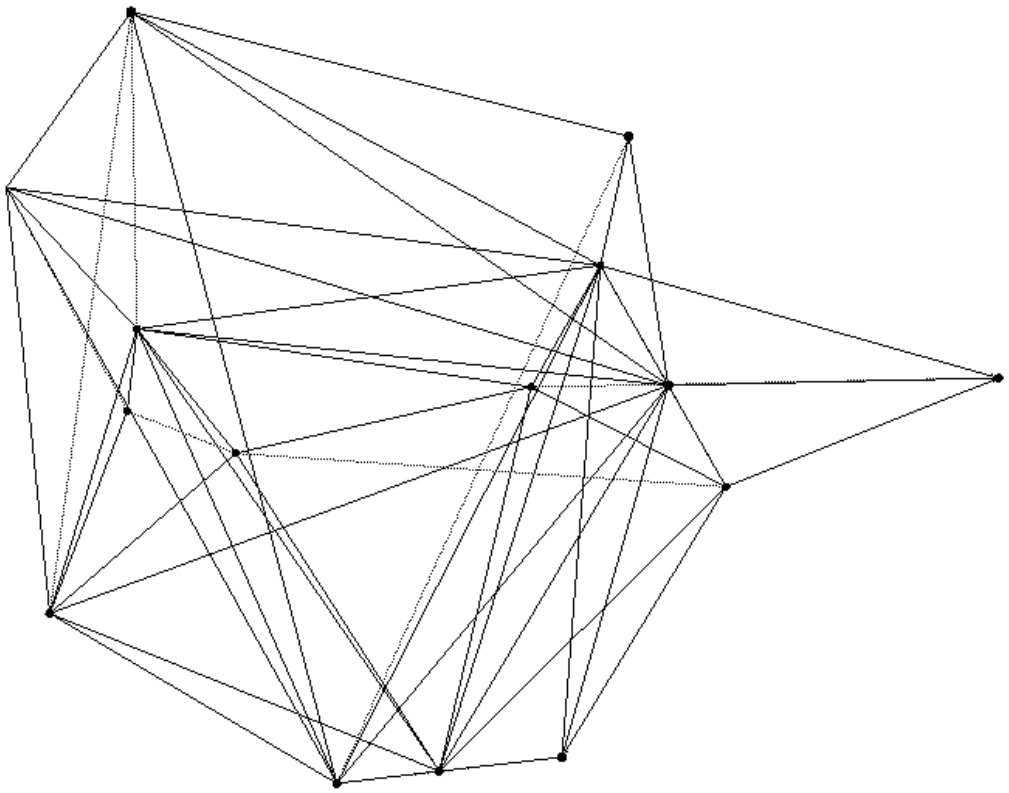


Abbildung 7.4.: Bei der Wahl eines zu kleinen allumfassenden Tetraeders ist die berechnete Delaunay-Triangulation ggf. nicht konvex.

Ermittlung eines geeigneten Skalierungsfaktors Um einen ausreichend großen Wert empirisch zu ermitteln, haben wir für exponentiell wachsende Skalierungsfaktoren jeweils mehrere Testläufe durchgeführt und anschließend geprüft, ob die entstandene Triangulation wirklich der Delaunay-Triangulation entspricht. Diese Korrektheitsüberprüfung wird in Kapitel 7.2.1.5 genauer erläutert. Ein Testlauf bestand aus der Generierung von zehn zufälligen Punktmengen der gleichen Größe und der Berechnung ihrer Delaunay-Triangulationen. Insgesamt haben wir für jeden Skalierungsfaktor zehn Testläufe durchgeführt, wobei die Punktmengengröße anfangs bei 2.000 lag und in Zweitausenderschritten bis auf 20.000 erhöht wurde.

Das Diagramm in der Abbildung 7.5 zeigt, wie sich die Korrektheit der Berechnungen

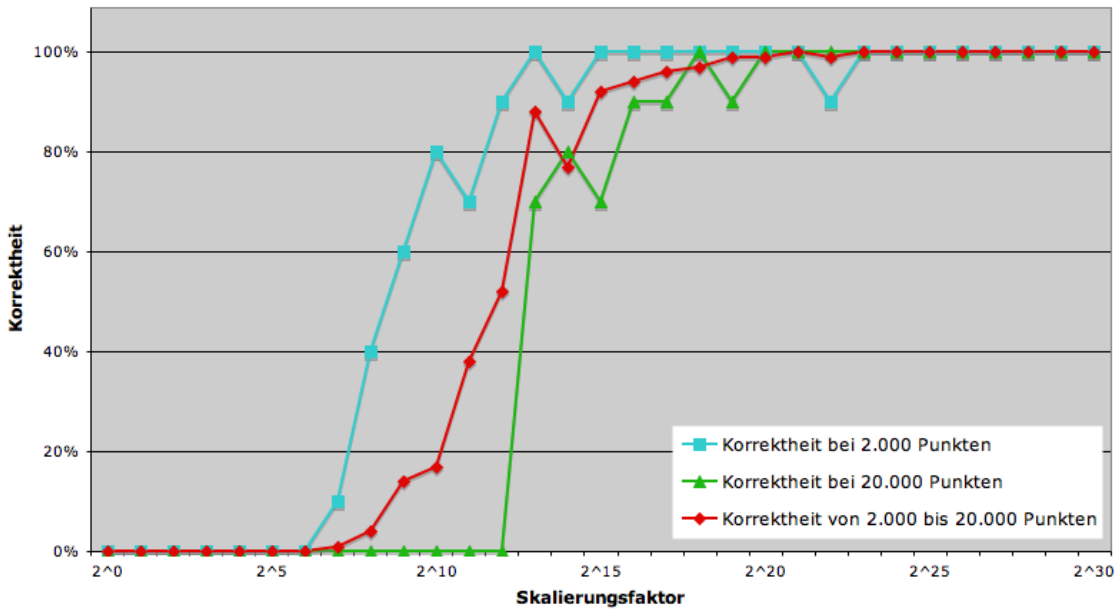


Abbildung 7.5.: Korrektheit der Berechnung abhängig vom Skalierungsfaktor.

bei 2.000, 20.000 Punkten und gemittelt über alle berechneten Punkte in Abhängigkeit vom Skalierungsfaktor entwickelt. Je größer die Anzahl der Punkte in einer Berechnung, desto höher ist die Wahrscheinlichkeit, dass eine Punktkonstellation entsteht, zu der eine inkorrekte Triangulation berechnet wird. Ab einer Skalierungsgröße von 2^{23} werden alle Triangulationen für alle Punktmengen bis einschließlich 20.000 korrekt berechnet. An dieser Stelle traten Speicherprobleme mit der *Java Virtual Machine* auf und wurden wie im Kapitel 11.1 auf Seite 103 beschrieben gelöst.

Letztendlich entschieden wir uns dafür, als Skalierungsfaktor 2^{30} zu wählen. Er ist in der Konfigurationsdatei angegeben. Wie unsere späteren Performanztests (siehe Kapitel 11.1) zeigten, war damit stets eine korrekte Berechnung der Delaunay-Triangulation möglich.

7.2.1.3. `addNextPointAndProcessFirstFlip` und `flip1to4`

Da der Stapel der *Link Facets* zu Beginn leer ist, wird mit der Methode `addNextPointAndProcessFirstFlip` begonnen: Der nächste einzufügende Punkt wird

darin als Anfragepunkt der Punktlokalisierung des *History-DAG* (`locatePoint`) übergeben; der erhaltene Knoten des *History-DAG* ist neben dem Anfragepunkt Parameter für den danach folgenden 1-zu-4-Flip (`flip1to4`).

Dieser 1-zu-4-Flip kreiert vier neue Tetraeder (`V3D_DT_Tetrahedron`) und setzt zunächst deren Nachbarschaftsbeziehungen untereinander, dann die mit bereits bestehenden Tetraedern. Danach werden aus den neuen Tetraedern in der Methode `addNode` der Klasse `V3D_DT_HistoryDAG` die neuen Knoten des *History-DAG* erzeugt und als Kinder an das Blatt gehängt, das zuvor die Punktlokalisierung zurückgeliefert hatte.

Die neu entstandenen Knoten werden an `addNextPointAndProcessFirstFlip` zurückgegeben. Aus den in den Knoten enthaltenen Tetraedern werden die *Link Facets* gewonnen; diese sind zunächst auf Regularität zu prüfen und dann auf den dafür vorgesehenen Stapel abzulegen.

7.2.1.4. `processNextLinkFacet`, `flip2to3` und `flip3to2`

Um die nun auf dem Stapel befindlichen *Link Facets* abzuarbeiten, wird die Methode `processNextLinkFacet` aufgerufen. Darin wird das oberste *Link Facet* des Stapels betrachtet. Zunächst wird geprüft, ob es noch Teil der Triangulation ist. Falls ja, ist zu ermitteln, ob es regulär ist. Dazu wird die Methode `isRegular` der Klasse `V3D_DT_Face` aufgerufen. Ist das *Link Facet* nicht regulär, ist zu ermitteln, welcher Flip nun durchzuführen ist. Dazu betrachtet man die Vereinigung der beiden Tetraeder, die an das *Link Facet* angrenzen. Ist in diesem Kontext keine Kante reflex, ist ein 2-zu-3-Flip auszuführen (`flip2To3`), bei genau einer reflexen Kante ein 3-zu-2-Flip (`flip3To2`). Bei genau zwei reflexen Kanten ist kein Flip anwendbar (vgl. Kapitel 5.1.3).

Die Methode `flip2To3` erhält als Parameter das zu flippende `V3D_DT_Face`. Wie beim 1-zu-4-Flip werden die Nachbarschaftsbeziehungen gesetzt und die Tetraeder als neue Knoten an den *History-DAG* angebunden.

Neben dem *Link Facet* wird `flip3To2` noch der Index der reflexen Kante übergeben.

Der Index bezeichnet dabei den Index des Knotens, der nicht der zu der reflexen Kante dazugehört. Auch hier werden nun die Nachbarschaftsbeziehungen gesetzt und die neuen Tetraeder als Kinder dem *History-DAG* hinzugefügt.

Auch diese beiden Flips liefern die neuen Knoten des History-DAGs an `processNextLinkFacet` zurück. Aus den darin enthaltenen Tetraedern werden die neuen *Link Facets* gewonnen und auf den Stapel gelegt.

Die Methode `processNextLinkFacet` muss solange aufgerufen werden, bis der Stapel der Link Facets wieder leer ist. Falls noch Punkte einzufügen sind, ist wieder `addNextPointAndProcessFirstFlip` auszuführen.

7.2.1.5. Überprüfung der Korrektheit einer Delaunay-Triangulation

Um zu gewährleisten, dass auch wirklich eine Delaunay-Triangulation berechnet wird, stellen wir Methoden zur Verfügung, die einen solchen Test optional durchführen können. Wir untersuchen dabei, ob alle *Link Facets* regulär sind (`isRegularTetrahedralization`) und die errechnete Triangulation konvex ist (dieser Test wird bei dem Durchlauf durch die *AQE*-Datenstruktur durchgeführt, da nur dort die zur konvexen Hülle gehörenden Delaunay-Dreiecke bekannt sind). Diese Tests laufen nach der vollständigen Berechnung der Delaunay-Triangulation.

Um Rechenzeit zu sparen, ist der Konvexitätstest in die Methoden zur Anforderung der zu zeichnenden Flächen verwoben. Hierbei ist zu beachten, dass er $O(n^2)$ in Anspruch nimmt, da für jedes Dreieck der konvexen Hülle der Triangulation geprüft wird, ob der Rest der Punktmenge auf derselben Seite dieses *Link Facets* liegt. Dieser Test ist optional; er kann über die Konfigurationsdatei ein- und ausgeschaltet werden.

Eine sinnvolle Erweiterung wäre eine Durchführung dieser Tests mit einer exakten Arithmetik.

7.2.2. `V3D_DT_Tetrahedron`

Zu einer Instanz der Klasse `V3D_DT_Tetrahedron` gehört in erster Linie ein Array `vertex` von vier Knoten des Typs `V3D_AQE_Vertex3D`. Des weiteren wird in einem Array `neighbour` die Nachbarschaftsbeziehung zu anderen Tetraedern gespeichert: An Index 0 gibt es einen Verweis auf das Nachbartetraeder, das sich mit dem Tetraeder selbst das Dreieck teilt, welches aus den Knoten besteht, die im Knotenarray an Index 1, 2 und 3 stehen. Es sind also genau die Indizes, die nicht 0 sind. Die Indexpositionen 1 bis 3 von `neighbour` sind mit Verweisen belegt, die der gleichen Logik folgen (siehe Abbildung 7.6).

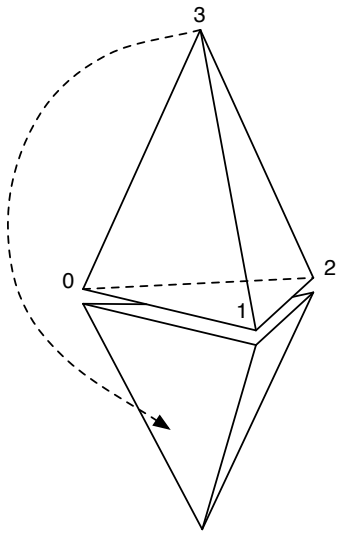


Abbildung 7.6.: Verweise auf Nachbartetraeder.

Dazu kommt ein Verweis `node` zu dem Knoten des *History-DAG*, dem das Tetraeder entspricht, sowie zwei *Flags*, `isPartOfCurrentTriangulation` und `isAuxiliaryTetrahedron`. Das erste *Flag* speichert, ob das Tetraeder noch Teil der aktuellen Triangulation ist, also der dem Tetraeder entsprechenden Knoten ein Blatt des *History-DAG* ist, während das zweite signalisiert, ob das Tetraeder einen Knoten des allumfassenden Tetraeders besitzt. In diesem Fall ist das Tetraeder ja auch kein Teil der Delaunay-Triangulation, sondern lediglich ein Hilfskonstrukt.

Die Klasse besitzt neben Funktionen zur Verwaltung dieser Daten keine weiteren Fähigkeiten.

7.2.2.1. `getE`, `getRE`, `getDE` und `getRDE`

Man bezeichne eine Delaunay-Kante mit $e_{i,j}$ (*edge*), wobei i und j die Indizes der Tetraederknoten bezeichnet, die durch die Kante verbunden werden. Weiter sei die von $e_{i,j}$ durch `rot` erreichte Kante mit $re_{i,j}$ (*rotated edge*), und die durch `rot` und `org` resultierende Voronoi-Kante mit $de_{i,j}$ (*dual edge*) bezeichnet. Die Kante $rde_{i,j}$ (*rotated dual edge*) bezeichne nun das Resultat der Operation `rot` auf die Voronoi-Kante $de_{i,j}$. Diese Kante verbindet zwei Delaunay-Kanten und hat als Ursprung wieder die Ausgangskante $e_{i,j}$. Die Identität ist also durch die Operationenfolge `rot org rot org` gegeben (siehe Abbildung 7.7).

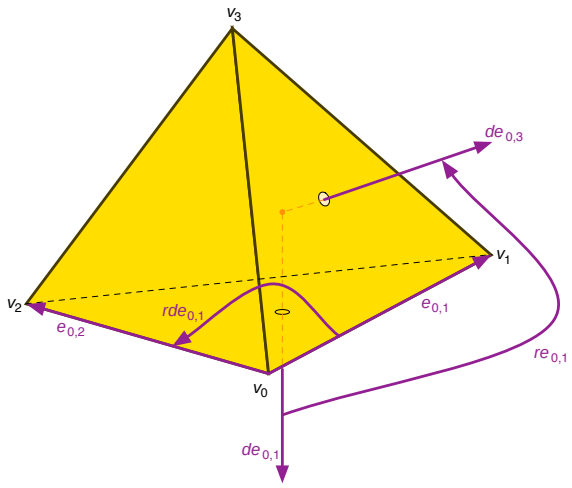


Abbildung 7.7.: Die Kanten $e_{0,1}$, $re_{0,1}$, $de_{0,1}$ und $rde_{0,1}$.

7.2.2.2. `deletePointers`

Da Java alle Objekte auf einem *Heap* verwaltet, ist es möglich, dass dieser für große Punktmengen überläuft. Standardmäßig ist die Heapgröße auf 64 MByte gesetzt. Deshalb

werden nicht mehr benötigte Verweise der *AQE* nach einer Aktualisierung der Delaunay-Triangulation gelöscht. Die Eckpunkte eines Tetraeders, das zu einem Knoten des *History-DAG* gehört, der kein Blattknoten ist, verweisen z. B. nicht mehr auf inzidente Kanten, ebenso wenig der Mittelpunkt des Tetraeders. Außerdem werden die Nachbarschaftsbeziehungen zwischen solchen Tetraedern gelöscht; sie werden nicht mehr benötigt.

7.2.3. `V3D_DT_Face`

Unter einer Instanz der Klasse `V3D_DT_Face` verstehen wir keine Dreiecksfläche eines Tetraeders, sondern eine Dreiecksfläche als Teil einer Triangulation, also als *Link Facet*. Deswegen besteht ein Tetraeder des Typs `V3D_DT_Tetrahedron` auch nicht aus Instanzen dieser Klasse. Wir verwenden die *Link Facets* nur temporär, um eine reguläre Triangulation herzustellen. Dazu werden sie in `V3D_DT` auf dem Stapel `stackOfLinkFacets` verwaltet. Sobald sie dort bearbeitet worden sind, verschwinden sie wieder.

7.2.3.1. Klassenvariablen

Ein `V3D_DT_Face` wird mit einem Knoten des *History-DAG* (`V3D_DT_HistoryDAG_Node`) und einem Index initialisiert. Das *Link Facet* entspricht der Dreiecksfläche des im Knoten enthaltenen Tetraeders, zu dem nicht der Punkt gehört, der in dem Tetraeder an dem übergebenen Index steht.

Die drei Punkte des *Link Facet* werden in dem Array `vertex` gespeichert. Dazu kommt ein Array mit Verweisen auf die Knoten des *History-DAG*, die den an das *Link Facet* angrenzenden Tetraedern entsprechen (`neighbour`).

Da die beiden Punkte, die nicht Teil des *Link Facets*, wohl aber der beiden angrenzenden Tetraeder sind, häufig benötigt werden, entschieden wir uns, die Indizes der beiden Punkte in ihren Tetraedern ebenso abzuspeichern. `remainingVertexIndexOfNeighbour` ist ein

Array, an dessen i-ter Position der Punktindex des Tetraeders steht, auf dessen *History-DAG*-Knoten auch in `neighbour` and Stelle i verwiesen wird.

7.2.3.2. `isRegular`

Diese Methode prüft, ob der Punkt des zweiten Nachbartetraeders, der nicht zum *Link Facet* gehört, innerhalb der Umkugel des ersten Nachbartetraeders liegt. Dazu wird die Funktion `isInSphere` der Tetraederklasse verwendet. Liegt der Punkt innerhalb der Sphäre, ist das *Link Facet* nicht regulär und muss geflippt werden.

7.2.3.3. `isEdgeReflexByVertexIndex`

Als Parameter erhält diese Methode den Index des Knotens, der nicht zu der zu testenden Kante gehört. Um zu entscheiden, ob die betroffene Kante nun in der Vereinigung der beiden Nachbartetraeder reflex ist, wird ein Orientierungstest verwendet. Wir prüfen, ob der Punkt des ersten Nachbartetraeders, der nicht zum *Link Facet* dazugehört, und der Punkt des *Link Facets*, der nicht zur zu testenden Kante gehört, auf verschiedenen Seiten der Ebene liegen, die durch die Punkte der Kante sowie den Punkt des zweiten Nachbartetraeders, der nicht zum *Link Facet* gehört, definiert ist. Liegen beide Punkte tatsächlich auf verschiedenen Seiten, ist die Kante reflex, andernfalls konvex (siehe Abbildung 5.2).

7.2.4. `V3D_DT_HistoryDAG`

7.2.4.1. Klassenvariablen

Der *History-DAG* besteht aus einem Wurzelknoten `root` des Typs `V3D_DT_HistoryDAG_Node` und einem Verweis `firstLeaf` auf das erste Blatt der doppelt verketteten Blattliste. Die Liste ermöglicht ein Abfragen der Tetraeder der Triangulation in $O(k)$ Zeit, falls der *DAG* über k Blätter verfügt. Ein Beispiel für eine solche verkettete Blattliste wird in Abbildung 7.8 gezeigt.

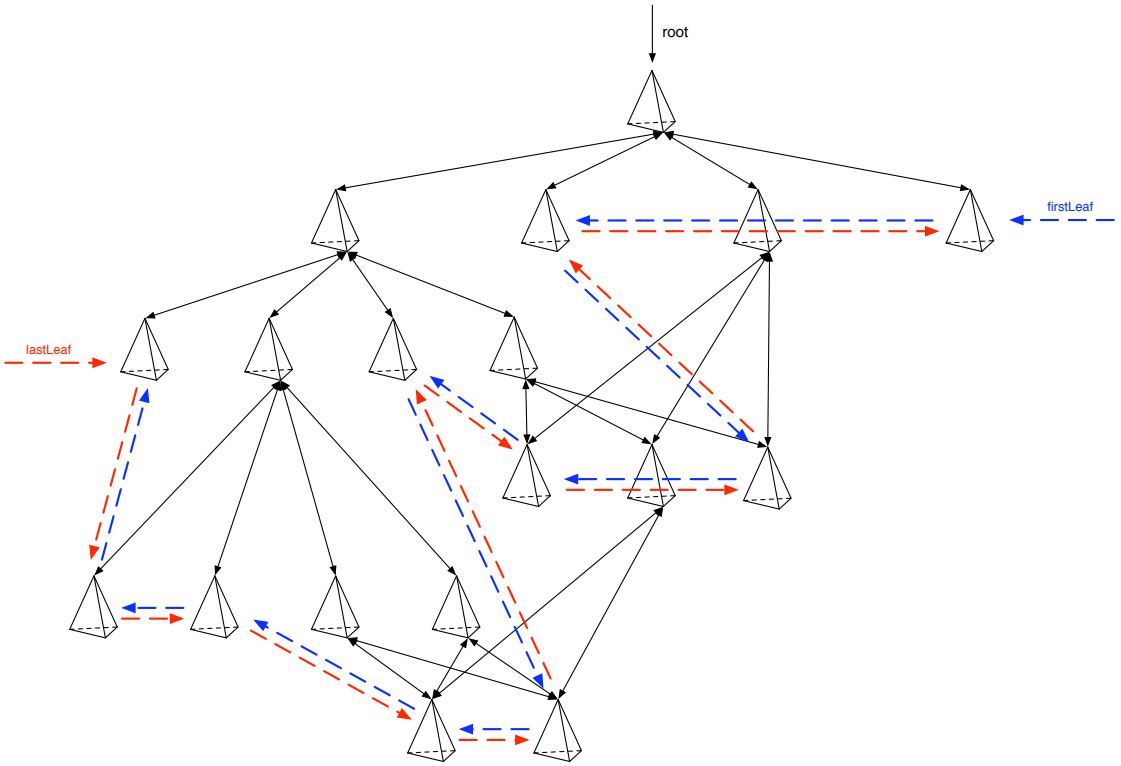


Abbildung 7.8.: Die Blattliste des *History-DAG*.

Daneben werden noch einige statistische Werte wie absolute und durchschnittliche Tiefe, Anzahl der Knoten bzw. Blätter und der durchgeführten Flips verwaltet.

7.2.4.2. `locatePoint`

`locatePoint` ist das Herzstück der Klasse `V3D_DT_HistoryDAG` und liefert den Blattknoten des *DAG*, dessen Tetraeder einen Anfragepunkt enthält. Es handelt sich um eine rekursive Funktion, die lediglich prüft, ob ein Anfragepunkt innerhalb eines Tetraeders liegt. Beides wird der Funktion übergeben, wobei das Tetraeder durch seinen im *History-DAG* entsprechenden Knoten repräsentiert wird. Liegt der Punkt nicht innerhalb des Tetraeders, wird `null` zurückgeliefert. Andernfalls wird getestet, ob der Knoten Kinderknoten besitzt. Falls ja, wird `locatePoint` mit dem Anfragepunkt und jeweils allen Kinderknoten als Parameter ausgeführt; falls nein, wird der Knoten selbst zurückgeliefert

(vgl. Algorithmus 5.2 auf Seite 43).

7.2.4.3. `addNode`

Je nachdem, ob ein 1-zu-4-Flip oder ein 2-zu-3- bzw. 3-zu-2-Flip durchgeführt wurde, werden verschiedene Funktionen aufgerufen, um die neuen Tetraeder als neue Knoten an den *History-DAG* anzuhängen: Ist nur ein Elternknoten vorhanden (1-zu-4-Flip), wird `addNode(V3D_DT_Tetrahedron, V3D_DT_HistoryDAG_Node)` aufgerufen, ansonsten `addNode(V3D_DT_Tetrahedron, V3D_DT_HistoryDAG_Node[])`.

Prinzipiell arbeiten beide Funktionen identisch: Es werden neue Knoten, die das entsprechende Tetraeder beinhalten, generiert und an den *DAG* angehängt und die doppelt verkettete Liste der Blätter aktualisiert. Zudem wird `isAuxiliaryTetrahedron` des übergebenen Tetraeders auf `true` gesetzt, falls das Tetraeder einen Knoten mit dem allumfassenden Tetraeder teilt, und `isPartOfCurrentTriangulation` der Tetraeder der Elternknoten mit `false` belegt.

7.2.5. `V3D_DT_HistoryDAG_Node`

Eine Instanz der Klasse `V3D_DT_HistoryDAGNode` repräsentiert einen Knoten des *History-DAG*. Sie besitzt einen Array `parent` und einen Array `child` des Typs `V3D_DT_HistoryDAG_Node` um den Knoten im *History-DAG* zu integrieren. Sind alle Verweise von `parent` gleich `null`, handelt es sich um den Wurzelknoten; sind alle Verweise von `child` gleich `null`, handelt es sich um einen Blattknoten. Ist ein Knoten ein Blatt, verweisen `previousLeaf` bzw. `nextLeaf` auf das vorangehende bzw. nachfolgende Blatt in der doppelt verketteten Liste der Blätter. Zusätzlich wird die Tiefe des Knotens im *DAG*, also die Entfernung zum Wurzelknoten in `level`, festgehalten.

8. Implementierung der AQE-Datenstruktur

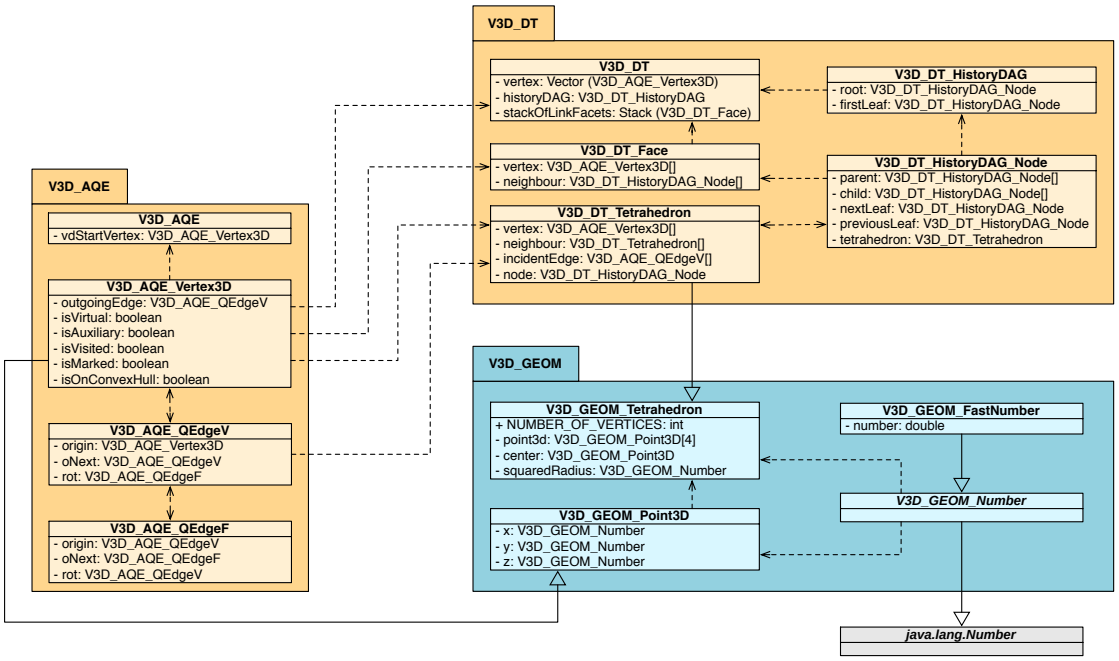


Abbildung 8.1.: UML-Diagramm des *AQE*-Paketes.

Das Paket `V3D_AQE` enthält vier Klassen, die eine *AQE*-Datenstruktur ergeben. Es handelt sich um:

`V3D_AQE_Vertex3D` Repräsentiert einen Knoten in einer *AQE*-Datenstruktur.

V3D_AQE_QEdgeV Stellt eine Kante dar, die zwei Knoten des Typs V3D_AQE_Vertex3D miteinander verbindet.

V3D_AQE_QEdgeF Steht für eine Kante, die zwei Kanten des Typs V3D_AQE_QEdgeV miteinander verbindet.

V3D_AQE Die *AQE*-Datenstruktur enthält einen Startknoten des Typs V3D_AQE_Vertex3D, von dem eine Navigation durch die Diagramme ausgeht.

8.1. V3D_AQE_Vertex3D

Eine Instanz dieser Klasse repräsentiert einen Knoten einer *AQE*-Datenstruktur. Die Klasse erbt von V3D_GEOM_Point3D. Es sind lediglich Methoden zum Setzen und Auslesen der Klassenvariablen vorhanden.

Ein Knoten der *AQE*-Datenstruktur besitzt lediglich einen Verweis auf eine von diesem Knoten ausgehende Kante outgoingEdge des Typs V3D_AQE_QEdgeV und mehrere *Flags*:

- isVirtual
- isAuxiliary
- isVisited
- isMarked
- isOnConvexHull

Das *Flag* isVirtual wird gesetzt, falls der Knoten dual zu einem Tetraeder ist, das ein Nachbartetraeder des allumfassenden Tetraeders simuliert. Diese Knoten haben keine Lokalität und sind nötig, um nicht aus der Datenstruktur herausnavigieren zu können.

Die anderen *Flags* werden während des Durchlaufs der *AQE* gesetzt: Ein gesetztes *Flag* isAuxiliary haben die Knoten, die zum allumfassenden Tetraeder gehören, virtuell sind

oder Mittelpunkte von Hilfstetraedern (die ebenso ein *Flag* `isAuxiliary` besitzen) sind. Mit `isOnConvexHull` werden die Knoten belegt, die auf der konvexen Hülle der zugrundeliegenden Punktmenge liegen. `isVisited` bezeichnet die Knoten, deren duale Polyeder bei einem Durchlauf schon entdeckt wurden. `isMarked` schließlich wird für die Knoten gesetzt, von deren dualen Polyedern schon Flächen beim Besuch eines Nachbarpolyeders abgearbeitet wurden.

8.2. V3D_AQE_QEdgeV

Dieser Kantentyp stellt eine Kante der *AQE*-Datenstruktur dar, die zwei Knoten des Typs `V3D_AQE_Vertex3D` miteinander verbindet. Eine solche Kante besitzt lediglich drei Verweise: Der Pointer `rot` verweist auf die gegen den Uhrzeigersinn nächste Kante des Typs `V3D_AQE_QEdgeF`, `onext` auf die gegen den Uhrzeigersinn nächste Kante `V3D_AQE_QEdgeV` und `org` auf den Knoten `V3D_AQE_Vertex3D`, von dem diese Kante ausgeht (vgl. Kapitel 4.2).

8.3. V3D_AQE_QEdgeF

Eine Instanz dieser Klasse verbindet die beiden zu zwei benachbarten Flächen eines Polyeders dualen Kanten `V3D_AQE_QEdgeV`. Auch dieser Kantentyp hat keine weitere Funktionalität inne; er besitzt lediglich die drei schon genannten Verweise `rot`, `onext` und `org`. Hier verweist `rot` auf die gegen den Uhrzeigersinn nächste Kante `V3D_AQE_QEdgeV`, `onext` auf die gegen den Uhrzeigersinn nächste Kante `V3D_AQE_QEdgeF` und `org` auf die duale Kante `V3D_AQE_QEdgeV` der Ursprungsfläche.

8.4. V3D_AQE

8.4.1. Klassenvariablen

Ähnlich wie eine Liste verweist die *AQE*-Datenstruktur nur auf ein Element, von dem aus alle anderen Elemente erreicht werden können. `V3D_AQE` besitzt einen Pointer `vdStartVertex` des Typs `V3D_AQE_Vertex3D`. Dies ist ein Knoten der Delaunay-Triangulation. Von diesem Knoten aus kann der Graph komplett durchlaufen werden. Eine Instanz vom Typ `V3D_AQE` kann mit einem solchen Knoten als Parameter generiert, der Startknoten kann aber auch manuell gesetzt werden.

Da der durch die AQE-Datenstruktur beschriebene Graph Kreise enthält, besteht die Notwendigkeit, Knoten zu markieren, will man ihn in schnellstmöglich komplett durchlaufen. Wir verwenden für diesen Zweck sowohl Markierungen für die einzelnen Punkte als auch einen Stapel `visitedVertices`, auf dem die schon besuchten Knoten abgelegt werden. Mit Hilfe des Stapels ist es nach einem Durchlauf möglich, die Markierungen der Punkte wieder zu entfernen. Wir haben uns dagegen entschieden, Kanten anstatt Knoten zu markieren, da z.B. ein Delaunay-Dreieck drei duale Kanten besitzt.

8.4.2. `initializeEdges` und `initializeAdditionalEdges`

Sobald in der Delaunay-Klasse ein Flip durchgeführt wird, werden neue Tetraeder generiert. Die Klasse `V3D_DT` kümmert sich nur um die Aktualisierung der Nachbarschaftsbeziehungen zwischen den Tetraedern; darüber hinaus muss aber auch die AQE-Datenstruktur aktualisiert werden. Beim Kreieren eines neuen Tetraeders wird deshalb die Methode `initializeEdges` aufgerufen.

Für jedes neue Tetraeder werden hier die *Quad Edges* angelegt und die Verweise `rot`, `org` und `onext` zwischen Kanten dieses Tetraeders gesetzt. Die einzigen Verweisarten, die nicht bei der Konstruktion eines neuen Tetraeders angelegt werden können, sind `onext`

und rot des Kantentyps rde, da sie auf Kanten anderer, zu diesem Zeitpunkt unbekannter Tetraeder zeigen. Um diese Verweise zu setzen, exisitieren die Methoden setRotForRDE und SetONextForRDE, die von den entsprechenden Flips aufgerufen werden.

Ist das zu konstruierende Tetraeder das allumfassende Tetraeder, wird zusätzlich die Methode initializeAdditionalEdges aufgerufen. Für das allumfassende Tetraeder existieren keine Nachbartetraeder, d.h. die Verweise onext und rot für Kanten des Typs rde sind nicht definiert. Dieses Tetraeder ist ja das einzige, welches nicht aus Flips entsteht, sondern künstlich erzeugt wird.

Damit z. B. die Operation twin auf eine de-Kante (dies entspricht ja der Operation rot auf eine rde-Kante) des allumfassenden Tetraeders möglich ist, definieren wir einen virtuellen AQE_Vertex3D. Ein virtueller Knoten ist ein Hilfsknoten, dessen *Flag* isVirtual zusätzlich auf true gesetzt wird. Wir definieren für jede de-Kante des allumfassenden Tetraeders zusätzlich eine Kante rrde und rrrde: Die Kante rrde ist dabei der twin der de-Kante (also der rot von rde), die also von dem virtuellen Knoten aus (dieser ist demnach org von rrde) in das allumfassende Tetraeder hineinzeigt. rrrde entsteht aus rrde durch die Operation rot. Wird rot wiederum auf rrrde angewandt, so erhält man wieder die de-Kante. Ein org auf rrrde liefert eine e-Kante. Die rrde-Kanten werden, analog zu den de-Kanten, untereinander mit onext verbunden.

Damit fehlt nur noch die Operation onext für die Kantenklasse rde: Eigentlich müsste hier jede rde-Kante mit onext auf sich selbst verweisen, da es keine Nachbartetraeder gibt. Doch wegen der Existenz der Hilfskanten rrrde können wir eine Verbindung herstellen, sodass alle Operatoren sinnvoll belegt sind und man das Urtetraeder nicht verlassen kann. Für die Kante $\texttt{rde}_{0,1}$ ist so z. B. der $\texttt{rrrde}_{0,3}$ die onext-Kante. $\texttt{rrrde}_{0,3}$ verweist mit dem onext-Operator wiederum auf $\texttt{rde}_{0,1}$.

8.4.3. setRotForRDE und setONextForRDE

Es existieren jeweils zwei Implementationen von `setRotForRDE` und `setONextForRDE`: zum einen für die Aktualisierung der AQE-Datenstruktur nach einem 1-zu-4-Flip, zum anderen für die Flips 2-zu-3 und 3-zu-2.

Die Methoden verbinden die neuen Kanten der Tetraeder mit den bereits existierenden Kanten der Triangulation.

8.4.4. getAllFaces

Unter einem *Face* verstehen wir ein zu zeichnendes Flächenstück der Delaunay-Triangulation oder des Voronoi-Diagramms.

Wir haben sowohl eine öffentliche als auch eine private Methode `getAllFaces`: Die öffentliche Methode erwartet einen String, der den Diagrammtyp spezifiert, und ruft die private Methode auf, für die der angegebene Diagrammtyp in einen entsprechenden Startknoten übersetzt wird. Darüber hinaus ist die einzige Aufgabe der öffentlichen Methode, die Markierungen der besuchten Punkte wieder zu entfernen.

Die private Methode führt den Durchlauf durch die *AQE*-Datenstruktur aus. Dazu wird für jeden noch nicht besuchten Knoten die Methode `getHullAroundOrigin` aufgerufen. Sie berechnet das Duale zu dem an sie übergebenen Knoten, d. h. die Voronoi-Region bzw. das Delaunay-Tetraeder, und fügt neu entdeckte Knoten dem Stapel der unbesuchten Knoten hinzu. Zurückgeliefert wird ein `Vector`, der die zu zeichnenden Flächen enthält.

Die Suche nach Voronoi-Flächen startet in dem Knoten `vdStartVertex`. Um die Delaunay-Flächen zu finden, startet man in einem Voronoi-Knoten, der nicht explizit gespeichert wird; man ermittelt ihn, indem man auf die von `vdStartVertex` ausgehende Kante die Operationen `rot` und zweimal `org` anwendet. `rot` und `org` liefern ja eine Voronoi-Kante; die zweite `org`-Operation liefert deren Ursprungsknoten.

Bei den gefundenen *Faces* unterscheiden wir insgesamt fünf Arten, zwei für Voronoi-Flächen und drei für Delaunay-Flächen. Bei Voronoi-Flächen sind dies Bisektoren beschränkter bzw. unbeschränkter Voronoi-Regionen. Dabei sind die Voronoi-Flächen Teil von unbeschränkten Voronoi-Regionen, die zwei solche trennen. Dazu muß die Kante, die dual zu einem solchen *Face* ist, zwei Knoten miteinander verbinden, die auf der konvexen Hülle liegen. Alle übrigen *Faces* gehören zu beschränkten Voronoi-Regionen.

Bei Delaunay-Dreiecken findet man drei Arten von *Faces*: Zum einen sind dies Dreiecke, die auf dem Rand der konvexen Hülle der Punktmenge liegen. Um diese zu identifizieren, prüft man deren duale Voronoi-Kante: Ist genau einer der beiden Knoten, die durch die Kante verbunden werden, ein Hilfsknoten, ist das duale *Face* ein Teil des Randes der konvexen Hülle. Des weiteren gibt es *Faces*, die innerhalb der konvexen Hülle, aber nicht auf dem Rand liegen. Bei diesen *Faces* verbindet die zum *Face* duale Kante zwei Knoten, von denen keiner ein Hilfsknoten ist. Zur letzten Gruppe von *Faces* gehören die Dreiecke, die zu den Knoten des allumfassenden Tetraeders inzident sind. Deren duale Kante verbindet zwei Hilfsknoten.

Diese Einteilung ist eine Partition aller *Faces*; jedes *Face* kann eindeutig einer dieser Klassen zugeordnet werden. Das Ziel dieser Klassifizierung ist eine flexible Visualisierung, d. h. der Benutzer kann zum genaueren Studium die Darstellung dieser Gruppen aktivieren und deaktivieren.

8.4.5. `getDualFace`

Man betrachte eine Kante `e` eines Tetraeders. Um die zu dieser Kante duale Voronoi-Fläche zu umlaufen, verwendet man die zugehörige Kante `rde`. Sie hat `e` als Ursprung und zeigt auf die Delaunay-Kante, die denselben Ursprung wie `e` und als Nachbardreieck das von `de` durchstoßene hat. Die sukzessive Anwendung von `onext` liefert also gegen den Uhrzeigersinn alle Kanten, die `e` als Ursprung haben. Diese Kanten zeigen alle auf Delaunay-Kanten, die auf unterschiedlichen Tetraedern liegen. Mit der Operation `rot org`

auf die `rde`-Kante erreicht man die Mittelpunkte dieser Tetraeder. Diese Tetraedermittelpunkte entsprechen den Knoten der zur Kante `e` dualen Voronoi-Fläche. Durch sukzessive Anwendung von `onext` auf eine Kante des Typs `rde` berechnet man also die zu einer Kante `e` duale Voronoi-Flache. Analog verfährt man, wenn man das zu einer Voronoi-Kante duale Delaunay-Dreieck sucht (siehe Abbildung 5.3).

8.4.6. `getVoronoiRegion`

Wir haben die Funktionalität vorgesehen, dass ein Benutzer einzelne Voronoi-Regionen hervorheben kann. Die Methode `getVoronoiRegion` liefert mit Hilfe von `getHullAroundOrigin` die *Faces* einer einzelnen Voronoi-Region zurück, um sie erneut zeichnen zu können.

9. Implementierung des Voronoi-Diagramms

In diesem Kapitel erläutern wir, wie das Voronoi-Diagramm in unserer Implementierung umgesetzt ist und wie die bestehenden Pakete untereinder vernetzt sein müssen.

Das Voronoi-Paket besteht lediglich aus der Klasse `V3D_VD`. Alle bisher vorgestellten Pakete fließen in das Voronoi-Paket ein (siehe Abbildung 9.1).

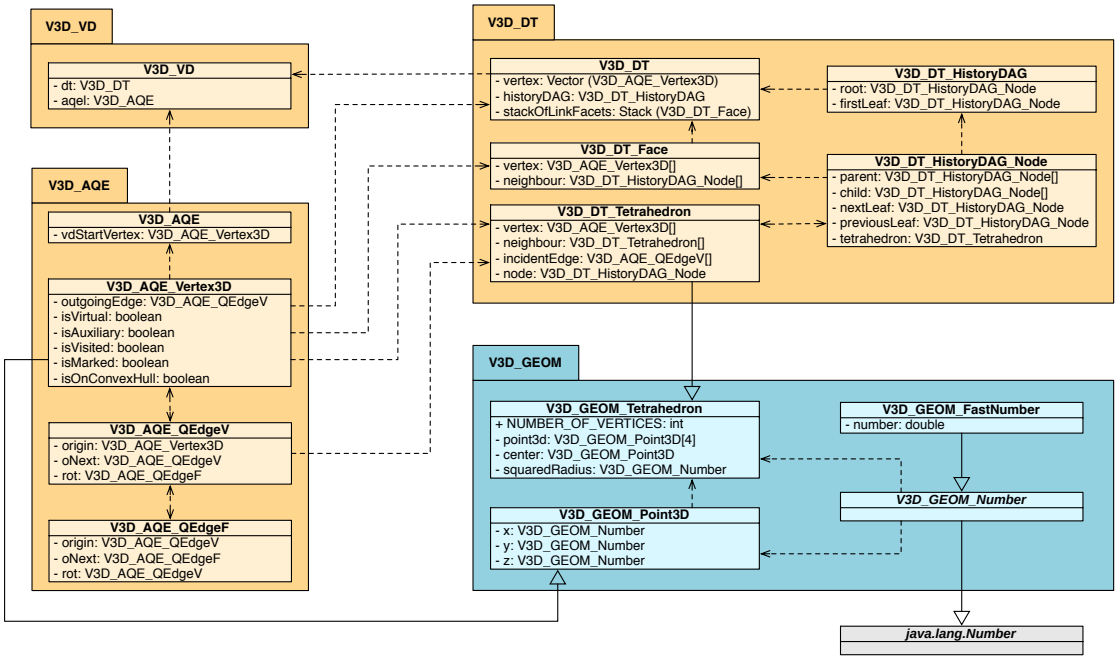


Abbildung 9.1.: UML-Diagramm des Voronoi-Paketes.

9.1. V3D_VD

9.1.1. Klassenvariablen

Eine Instanz der Klasse `V3D_VD` enthält zwei Variablen: zum einen eine Delaunay-Triangulation `dt` des Typs `V3D_DT`, zum anderen eine AQE-Datenstruktur `aqe` des Typs `V3D_AQE`.

Die Klasse wird mit einer Punktmenge initialisiert; sie wird aber nicht in dieser Klasse, sondern in `V3D_DT` gespeichert.

Neben der Delaunay-Triangulation `dt` wird im Konstruktor des Voronoi-Diagramms die AQE-Datenstruktur `aqe` angelegt. Der Startknoten der AQE-Datenstruktur ist dabei der erste Punkt der in der Delaunay-Triangulation gespeicherten Punktmenge.

9.1.2. getShuffledVertices

Es existieren Punktmengen, für die bei einer festen Einfügereihenfolge die Hinzunahme jedes einzelnen Punkts lineare Laufzeit benötigt und die Gesamtlaufzeit somit $\Omega(n^2)$ beträgt. Um von der Einfügereihenfolge unabhängig zu sein, muss sie zu Beginn randomisiert werden.

Diese Methode permutiert die Einfügereihenfolge der Punktmenge der Delaunay-Triangulation. Diese Funktion wird ausgeführt, sobald eine Randomisierung der Einfügereihenfolge durch den Benutzer eingeleitet wurde. Die Delaunay-Triangulation muss danach erneut von Beginn an berechnet werden.

Wir bieten nur die Möglichkeit einer vom Benutzer ausgelösten Randomisierung. Zum einen sind die von uns verwendeten zufällig erzeugten Punktmengen an sich schon randomisiert, andererseits soll die Ordnung von Punktmengen, die ein Benutzer eingegeben hat, nicht zerstört werden, da die vorgegebene Einfügereihenfolge möglicherweise Voraussetzung für die Veranschaulichung eines geometrischen Sachverhaltes ist. Eine automatische

Randomisierung würde z. B. die gezielte Vorbereitung einer Lehrveranstaltung unnötig erschweren.

9.2. `getAllFaces` und `getVoronoiRegion`

Hier werden die zur Darstellung erforderlichen Flächen der Diagramme berechnet. Sind zu diesem Zeitpunkt weniger als vier Punkte in der Delaunay-Triangulation enthalten, können die Methoden `getAllFaces` und `getVoronoiRegion` der Klasse `V3D_AQE` nicht verwendet werden, da zu diesem Zeitpunkt alle Tetraeder der Triangulation noch Hilfstetraeder sind, also Knoten mit dem allumfassenden Tetraeder gemein haben. Die Berechnung der Bisektorflächen dieser Spezialfälle erfolgt in den Methoden `getBisectorsOfFirstThreePoints` und `getBisectorsOfFirstTwoPoints`; sie werden sowohl von `getAllFaces` als auch von `getVoronoiRegion` aufgerufen. Um die Delaunay-Triangulation von zwei oder drei Punkten darzustellen, erzeugen wir manuell eine Kante bzw. ein Dreieck.

9.3. `processAllPoints`, `processNextPoint` und `processNextLinkFacet`

Sobald der Benutzer die Berechnung der Delaunay-Triangulation über die grafische Benutzeroberfläche befiehlt, werden diese Methoden angesprochen. `processAllPoints` löst eine Berechnung der kompletten Delaunay-Triangulation aus, `processNextPoint` fügt den nächsten Punkt in die Triangulation ein, und `processNextLinkFacet` prüft das oberste *Link Facet* des Stapels auf Regularität und leitet ggf. einen Flip ein. Damit sind die elementaren Funktionen der Delaunay-Triangulation gekapselt; das Voronoi-Diagramm steuert den Aufbau der Delaunay-Triangulation.

10. Implementierung der Visualisierung und GUI

In diesem Kapitel widmen wir uns der Umsetzung der Darstellung und der Benutzerschnittstelle. Dabei werden die Pakete `V3D_GUI` und `V3D_VIEW` und die Programmklasse `V3D_Applet` behandelt (siehe Abbildung 10.1).

10.1. Das Paket `V3D_GUI`

Das Paket `V3D_GUI` besteht aus einer Java-Klasse `V3D_GUI_Canvas` und ist für die Speicherung und Darstellung der zu zeichnenden Geometrien und die Interaktion mit den Diagrammen zuständig. Die Klasse erbt von der Java3D-Klasse `javax.media.j3d.Canvas3D` und kennt als wichtige Variablen das Voronoi-Diagramm, einen Vektor von Geometrien und ein Array von Flächen der Klasse `V3D_VIEW_Face`.

10.1.1. `createSceneGraph`

Die Methode `createSceneGraph` initialisiert den Wurzelknoten aus dem Java3D-Zeichnungsbaum. Dabei werden die Sichtbeschränkung (`BoundingSphere`), Hintergrundfarbe, Transparenz, Erscheinung und Verhaltensweisen bei Rotation, Skalierung und Translation definiert und an die entsprechenden Knoten gebunden. Der an dieser Stelle

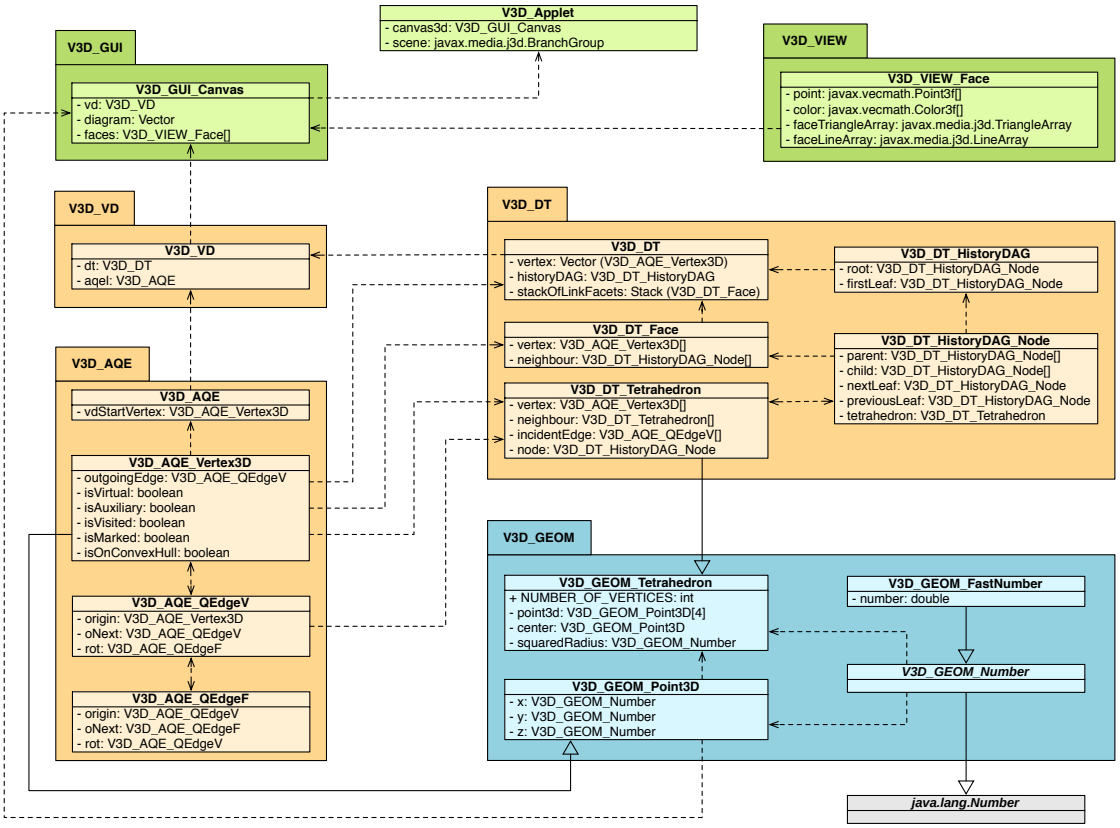


Abbildung 10.1.: UML-Diagramm des Programms.

erzeugte Wurzelknoten wird in der Java3D-Terminologie auch als Szenengraph bezeichnet; in Anhang B.3 werden wir ausführlicher auf ihn eingehen.

10.1.2. `getAllFaces`

Diese Methode ruft die `getAllFaces`-Methode der Delaunay-Triangulation zweimal auf. Beim ersten Mal wird der Vektor mit den Flächen gespeichert, die für die Delaunay-Triangulation wichtig sind, nämlich die außerhalb der konvexen Hülle befindlichen Tetraederflächen, die Flächen auf der konvexen Hülle und die innerhalb der konvexen Hülle. Der zweite Durchlauf ruft die Flächen der beschränkten und unbeschränkten Regionen des Voronoi-Diagramms ab. Diese Methode erhält einen booleschen Parameter, der angibt, ob die Korrektheit der Delaunay-Triangulation überprüft werden soll. In diesem Fall

werden die *Flags* `isConvex` und `isValid` in `V3D_DT` auf die entsprechenden Werte gesetzt, je nachdem, wie diese Korrektheitsprüfung ausgefallen ist.

10.1.3. `updateFacesOfScenegraph`

Diese Methode aktualisiert die Zeichnung der darzustellenden Diagramme. Dazu hat sie sieben Parameter:

`showVD` Ein boolescher Wert zur Angabe, ob das Voronoi-Diagramm gezeichnet werden soll.

`hideUnboundedVR` Ein boolescher Wert zur Angabe, ob die unbeschränkten Voronoi-Regionen ausgeblendet werden sollen.

`showCH` Ein boolescher Wert zur Angabe, ob die konvexe Hülle gezeichnet werden soll.

`showDT` Ein boolescher Wert zur Angabe, ob die Delaunay-Triangulation gezeichnet werden soll.

`showAuxiliaryParts` Ein boolescher Wert zur Angabe, ob die Tetraederflächen, die einen Knoten des allumfassenden Tetraeders beinhalten, gezeichnet werden sollen.

`howToDisplay` Ein String zur Angabe, ob die gezeichneten Flächen ausgefüllt oder nur die Ränder der Flächen gezeichnet werden sollen.

`showAllPoints` Ein boolescher Wert zur Angabe, ob die berechneten Punkte als kleine Sphären gezeichnet werden sollen.

Bei jedem Aufruf der Methode wird die Zeichenfläche von den nicht gewünschten Diagrammen geleert und mit den gewünschten gefüllt.

10.1.4. `updateLinkFacetsOfScenegraph`

Analog zur Zeichnung der Diagramme funktioniert auch die Aktualisierung der *Link Facets*, die je nach gegebenem booleschen Wert ein- bzw. ausgeblendet werden. In der Konfigurationsdatei kann eingestellt werden, ob nur das aktuell zu bearbeitende *Link Facet* oder alle sich auf dem Stapel befindlichen *Link Facets* gezeichnet werden sollen. Ersteres ist die Standardeinstellung, denn nur dieses *Link Facet* ist für den nächsten Schritt relevant.

10.1.5. `highlightRegions`

Die Methode `highlightRegions` hebt Voronoi-Regionen hervor oder blendet eine solche existierende Hervorhebung aus. Sie wird durch die folgenden Parameter gesteuert:

`highlight` Ein boolescher Wert zur Angabe, ob Regionen hervorgehoben bzw. ob bereits hervorgehobene ausgeblendet werden sollen.

`indexOfPoints` Ein Array zur Angabe der Punkteindizes, die hervorgehoben werden sollen.

`howToDisplay` Ein String zur Angabe, ob die gezeichneten Flächen der Regionen ausgefüllt oder nur die Ränder der Flächen gezeichnet werden sollen. In der Konfigurationsdatei regelt die Konstante `ALLOW_HIGHLIGHTED_REGIONS_TO_CHANGE_FILLING`, ob dieser Wert ausgewertet oder ob die Region stets ausgefüllt wird. Aufgrund der besseren Sicht ist der Standardwert dieser Konstante auf `false` gesetzt.

10.2. Der Szenengraph

Der Szenengraph ist ein innerer Knoten (in der Abbildung 10.2 „scene“ genannt) in der Baumhierarchie des darstellenden Teils des Programms. An ihn sind alle weiteren Objekte

gebunden, die für die Zeichnung, Belichtung und Interaktion mit der Zeichnung gebraucht werden (siehe Abbildung 10.2). Direkt an ihm hängt die nächste *Branch Group* (`objR`), an die die Rotations-, Skalierungs- und Translationslogik verbunden ist. Die darunterliegende `objT` *Transform Group*[1] hat zwei Kinderknoten: `lf` speichert die *Link Facets* und `diagr`, die für den gesamten Rest der Zeichnung zuständig ist.

Die *Branch Group* `diagr` enthält diejenigen Knoten, die Zeichnungsobjekte für das Voronoi-Diagramm, Delaunay-Triangulation, konvexe Hülle und die Sphären für die berechneten Punkte speichern. Dabei wird das Voronoi-Diagramm in zwei Gruppen aufgeteilt: einmal für die beschränkten (`vd`) und auf der anderen Seite für die unbeschränkten Regionen (`uVR`). Die Delaunay-Triangulation wird ihrerseits in drei Gruppen aufgeteilt, nämlich in die äußere Triangulation (`ax`), die mindestens einen Hilfsknoten enthält, die konvexe Hülle (`ch`) und die Triangulation innerhalb der konvexen Hülle (`dt`). Durch dieser Unterteilungen ist es möglich, mittels zweier linearer Durchläufe durch die *AQE*-Datenstruktur beliebige Kombinationen der Geometrien zu zeichnen. Da alle genannten Zeichnungen bis auf die Punktmenge dieselbe Transparenz und gleichartige Erscheinungen haben, benutzen sie dieselbe *Appearance*, wogegen die Sphären eine eigene benötigen.

10.3. Das Paket `V3D_VIEW`

Da Java3D keine Klasse zur Verfügung stellt, die ein beliebiges konvexes Polygon darstellt, entwarfen wir das Paket `V3D_VIEW`, das aus einer Java-Klasse `V3D_VIEW_Face` besteht. Es repräsentiert eine beliebige konvexe Fläche, die intern aus Dreiecken zusammengestellt ist (siehe Abbildung 10.3). Dabei werden aus n gegebenen Punkten ein konvexes n-Eck gebaut, indem eine Triangulation stattfindet. Dabei ist das erste Dreieck $tria_{\{p_1,p_2,p_3\}}$ und das k-te Dreieck $tria_{\{p_1,p_{k+1},p_{k+2}\}}$. Da Java3D die Vorder- bzw. Rückseite von Dreiecken nach der Orientierung der Punkte definiert, müssen

[1] Es ist notwendig, dass `objT` eine *Transform-* und keine *Branch Group* ist; ansonsten würde sich die Rotations-, Skalierungs- und Translationslogik nicht auf ihre Kinder auswirken.

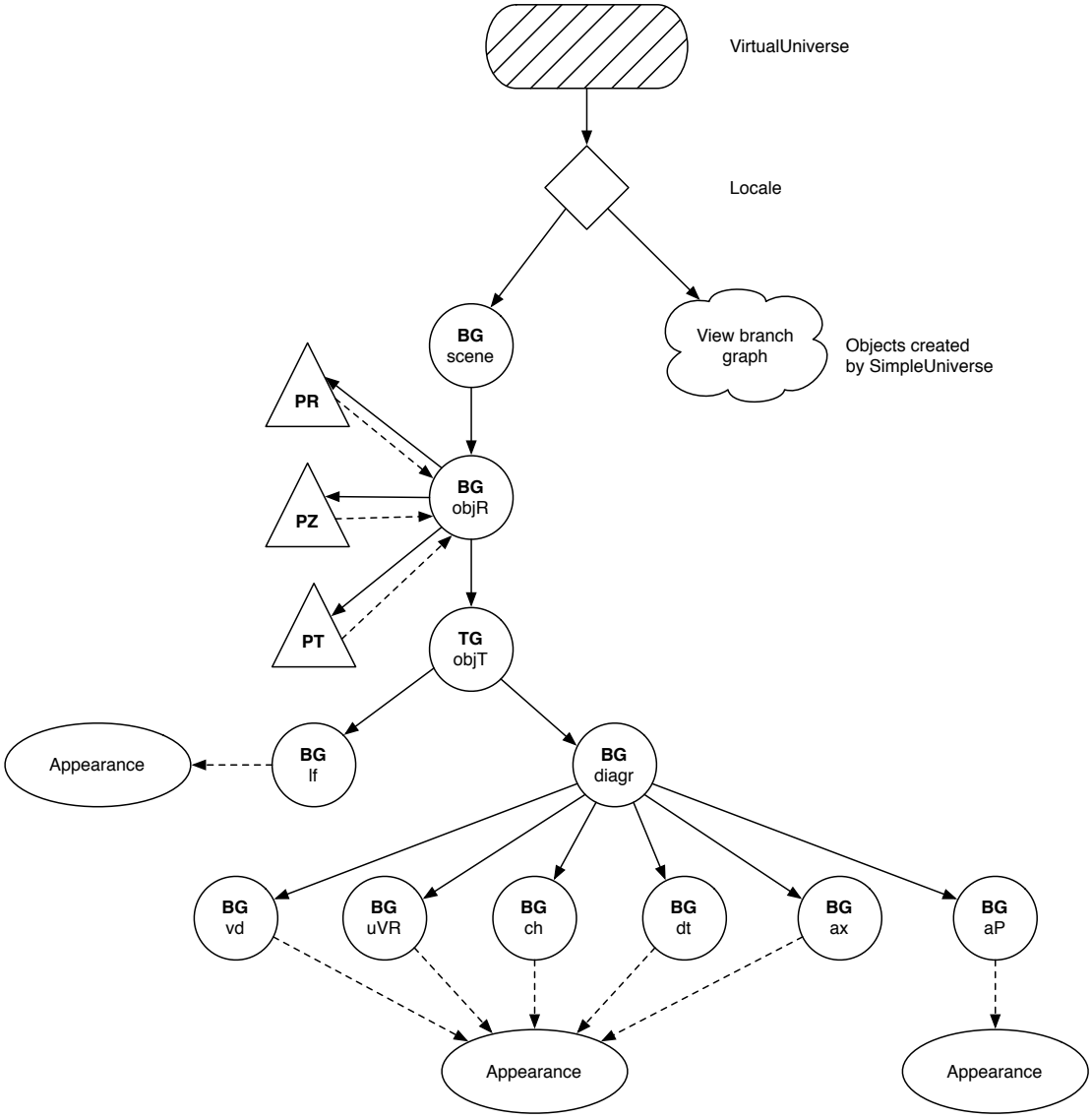


Abbildung 10.2.: Der Aufbau des Szenengraphen.

die gegebenen Punkte gegen den Uhrzeigersinn orientiert sein. `V3D_VIEW_Face` erbt von der Java3D-Klasse `javax.media.j3d.Shape3D` und kennt als wichtige Variablen ein Array von `javax.vecmath.Point3f` für die Randpunkte, ein Array von Farben `javax.vecmath.Color3f`, ein `javax.media.j3d.TriangleArray` für die die Fläche definierenden Dreiecke und ein `javax.media.j3d.LineArray` für den Rand der Gesamtfläche. Die Objekte der Klasse werden hauptsächlich durch die Konstruktoren komplett

aufgebaut; aus diesem Grund besitzt sie auch keine nennenswerten Methoden.

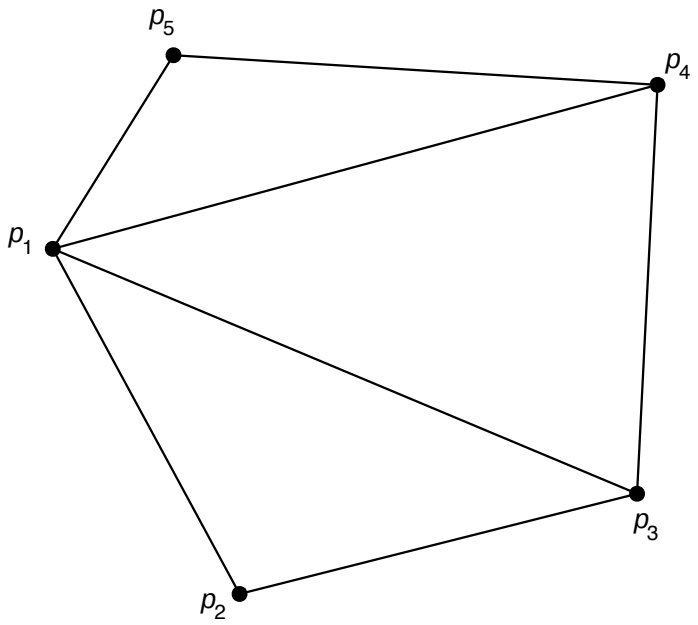


Abbildung 10.3.: Aufbau eines konvexen Fünfecks aus Dreiecken.

An dieser Stelle trat das Problem auf, dass unbeschränkte Voronoi-Regionen in Java3D nicht ohne Weiteres abgebildet werden können. Um dieses Problem zu lösen, berechnen wir für die Kanten der unbeschränkten Voronoi-Regionen Endpunkte, die außerhalb des sichtbaren Bereiches projiziert werden (siehe Abbildung 10.4). Dafür verwenden wir die private Methode `liftVertexToInfinity` der Klasse `V3D_AQEL` während des Durchlaufs von AQE. Diese Methode projiziert den Endpunkt – dies ist der Mittelpunkt des Umkreises der zur Voronoi-Kante dualen Delaunay-Dreiecksfläche $tria_{\{a,b,c\}}$ – der unbeschränkten Voronoi-Kante durch Beibehaltung der Kantenrichtung außerhalb des dargestellten Bereichs. Damit verlängern wir die unbeschränkten Voronoi-Kanten, die dann außerhalb des Zeichnungsbereichs verbunden werden, und stellen somit sicher, dass die Verbindungskanten e' zwischen solchen nach außen projizierten Punkten vom Renderer nicht gezeichnet werden. Dadurch werden zwar für Java3D ganz normale, darstellbare Flächen berechnet, die dem Betrachter aber als unendliche Flächen bzw. Kanten erscheinen.

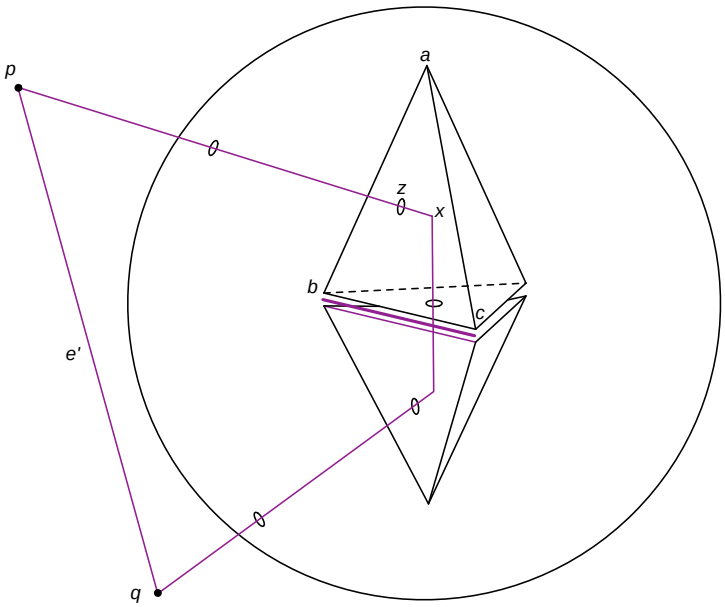


Abbildung 10.4.: Skizze zur Lösung des Darstellungsproblems für unbeschränkte Voronoi-Regionen.

Das Beispiel in Abbildung 10.4 stellt den Zeichnungsbereich von Java3D als den großen Kreis bzw. Sphäre dar, der die beiden Tetraeder umfasst. Der Punkt z wird nach p projiziert, indem wir die Kante von x nach z mit einem Skalierungsfaktor aus der Konfigurationsdatei verlängern. Um sicherzustellen, dass der projizierte Punkt stets außerhalb des Darstellungsbereiches liegt (und nicht außerhalb des Wertebereichs für das Zahlensystem), normalisieren wir zunächst den zu projizierenden Punkt und skalieren erst dann. Analog verfahren wir, um den Punkt q zu erhalten. Die Punkte werden dann durch die Kante e' verbunden, die außerhalb des Darstellungsbereichs verläuft. Damit entsteht für den Betrachter der Eindruck, dass die Kanten nach p bzw. q ins Unendliche gehen und dass die dazwischen verlaufende Fläche auch unbeschränkt ist.

10.4. V3D_Applet

Die Hauptklasse des Programms erbt von `java.applet.Applet` und kann als Java-Applet oder als Java-Applikation ausgeführt werden. Sie stellt die gesamte Benutzeroberfläche und die Interaktion mit dieser zur Verfügung. Als wichtige Variablen enthält sie `V3D_GUI_Canvas` (siehe Kapitel 10.1) und den Szenengraphen (siehe Kapitel 10.2).

10.4.1. Der Back-Button

Bei jedem ausgeführten Flip wird ein globaler Schrittzähler inkrementiert. Gleichzeitig führen wir noch einen Stapel mit, der den Zählerstand für die bereits berechneten Punkte enthält (siehe Kapitel 7.2.1.1). Damit ist es möglich, von jedem beliebigen Berechnungsstatus zu jedem vorherigen zurückzukehren. Aus Gründen der Übersichtlichkeit und Handhabung haben wir uns jedoch nur auf die jeweils letzten Schritte beschränkt. Geht man flipweise zurück, so berechnen wir den gesamten Vorgang vom Anfang bis zu der um eins dekrementierten Position; bei einem punktweisen Zurückschritt wird die Position durch das Auslesen des obersten Stapelelementes erfragt.

10.4.2. Normalisierung manuell eingefügter Punkte

Zu der bereits vorhandenen Punktliste können noch einzelne Punkte hinzugefügt werden. Hierfür werden zufällige Werte im Bereich $[-999, 999]$ für die Koordinaten generiert; diese werden anschließend normalisiert, indem sie mit 10^{-n} multipliziert werden, wobei n die Anzahl der Ziffern der größten zufälligen Koordinate des aktuellen Punktes bezeichnet. Gibt der Benutzer eigene Zahlen für die Koordinaten ein, so werden diese analog zu dem o. a. Verfahren normalisiert und somit auf das offene Intervall $]-1, 1[$ skaliert.

11. Performanz des Programms

11.1. Die Testumgebung zur Performanzanalyse

Um die zu erwartenden Werte für Laufzeit und Speicherplatzbedarf zu bestätigen, wurde der Applikation eine Testumgebung hinzugefügt. Sie ermöglicht die Ausführung von Voronoi-Diagramm-Berechnungen ohne anschließende Visualisierung, wobei die für eine Performanzanalyse relevanten Daten protokolliert und ausgewertet werden.

Damit wir Daten erhalten, die von der Punktmengengröße unabhängig sind, wird die Berechnung für schrittweise steigende Punktmengengrößen durchgeführt. Zu statistisch aussagekräftigen Werten gelangen wir, indem wir in jedem Schritt die Berechnung für eine bestimmte Anzahl zufälliger Punktmengen konstanter Größe durchführen.

Der Benutzer kann die Rahmenbedingungen dieser Performanzanalyse festlegen, indem er zum einen die Anfangs- und Endgrößen der Punktmengen sowie den Abstand zwischen zwei Punktmengengrößen festlegt und zum anderen die Anzahl der Berechnungen von zufälligen Punktmengen konstanter Größe angibt.

Da wir gleichverteilte Punktmengen innerhalb des Einheitswürfels verwenden, können wir davon ausgehen, dass wir Daten erhalten, die den im Kapitel 5.3.2 postulierten Erwartungswerten entsprechen.

Wir haben uns dazu entschieden, die Performanzanalyse für Punktmengen der Größe 2.000 bis 20.000 durchzuführen. Für größere Punktmengen benötigt die Berechnung zu viel Zeit, um innerhalb eines realistischen Zeitfensters eine befriedigende Anzahl an Durchläufen

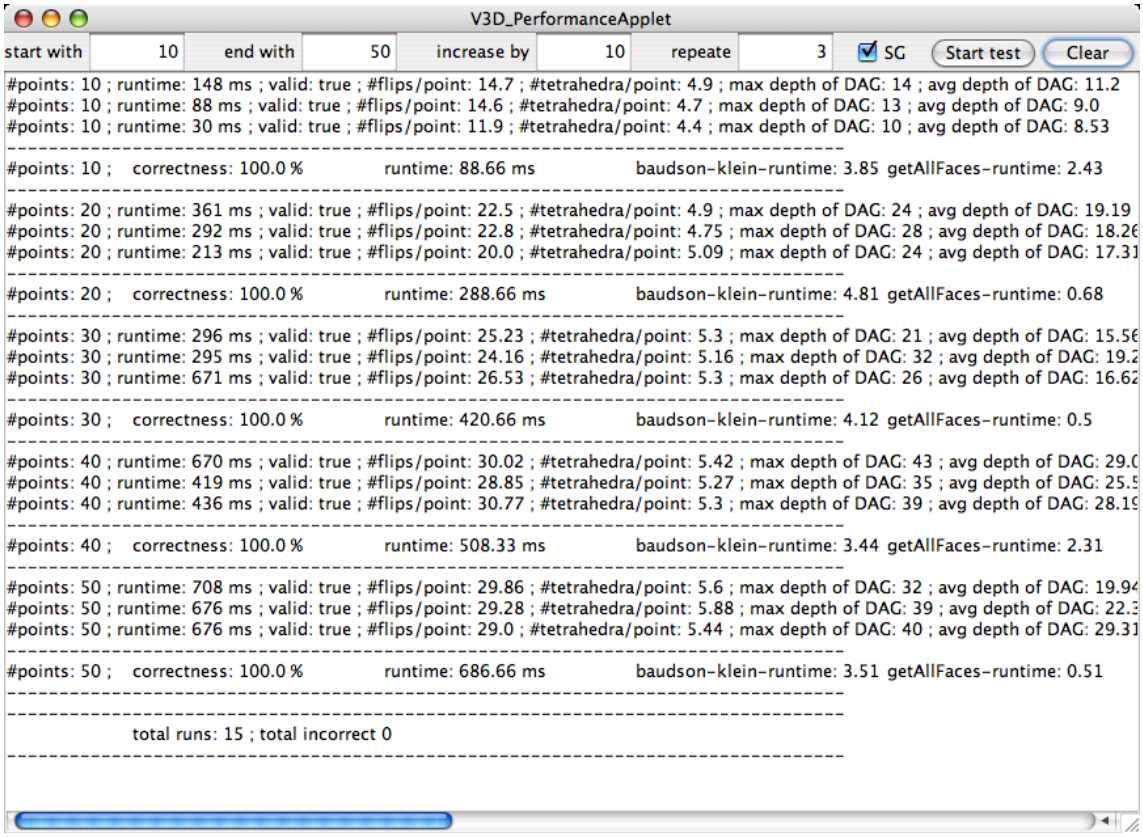


Abbildung 11.1.: Die Testumgebung zur Performanzanalyse.

abzuschließen. Für jede Punktmengengröße haben wir 50 zufällige Punktmengen erzeugen und deren Voronoi-Diagramm berechnen lassen. Damit haben wir insgesamt 500 Voronoi-Diagramme berechnet.

Als Skalierungsfaktor für das allumfassende Tetraeder wählten wir 2^{30} (siehe Kapitel 7.2.1.2). Alle 500 Voronoi-Diagramme konnten damit korrekt berechnet werden.

Die Annahme, die Punkte befänden sich in allgemeiner Lage, wurde durch keine der 500 erzeugten Punktkonstellationen verletzt.

Die Performanzanalyse wurde auf einem iMac G5 2GHz mit 1,25 GByte Arbeitsspeicher durchgeführt und nahm ca. 5 Tage und 15 Stunden in Anspruch. Da die Standardeinstellung bei der *Java Virtual Machine* (*JVM*) für die Berechnung von über 10.000 Punkten

nicht mehr ausreichte[1], haben wir das Programm mit dem Parameter `-Xmx1000m` aufgerufen, so dass der *JVM* 1 GByte Arbeitsspeicher zur Verfügung gestellt wird.

11.2. Laufzeit

Zu den gemessenen Werten gehört die Gesamtlaufzeit, d. h. der zur Berechnung und Visualisierung des Voronoi-Diagramms notwendige Zeitaufwand. Er setzt sich aus den Einzellaufzeiten für die Lokalisierung einzufügender Punkte, der Aktualisierung der Triangulation und dem Durchlauf durch die *AQE*-Datenstruktur zusammen (siehe Abbildung 11.2).

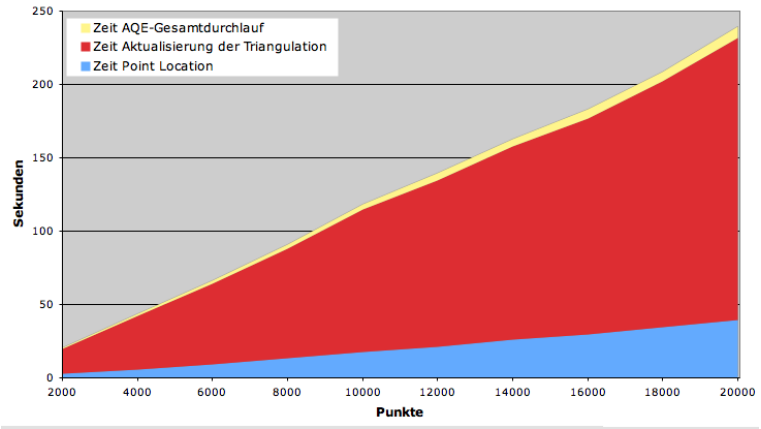


Abbildung 11.2.: Gesamtlaufzeit der Berechnung und Visualisierung des VDs.

Absolut gesehen überwiegt der Anteil der Aktualisierung der Triangulation durch Flips hier deutlich. Doch wir werden sehen, dass für größere Punktmengen die *Point Location* schwerer wiegen dürfte: Nach Shah [35] (siehe Kapitel 5.3.2.2) nimmt sie $O(n \log n)$ Zeit in Anspruch, die beiden anderen Aufgaben können dagegen in linearer Zeit gelöst werden.

[1] Da eine Visualisierung des Voronoi-Diagramms zusätzlichen Speicher für die Instanzen der Klasse `V3D_VIEW_Face` benötigt, können mit den Standardeinstellungen nur Diagramme mit bis zu 600 Punkten konstruiert und dargestellt werden. Der Speicherplatzbedarf einer Instanz dieser Klasse ist von uns nicht beeinflussbar, da Datentypen aus Java3D verwendet werden müssen (siehe Kapitel 10.3).

11.2.1. Point Location

Um die geforderte Laufzeit von $O(\log n)$ pro Punkt zu verifizieren, wird zuerst die Laufzeit gemessen (siehe Abbildung 11.3).

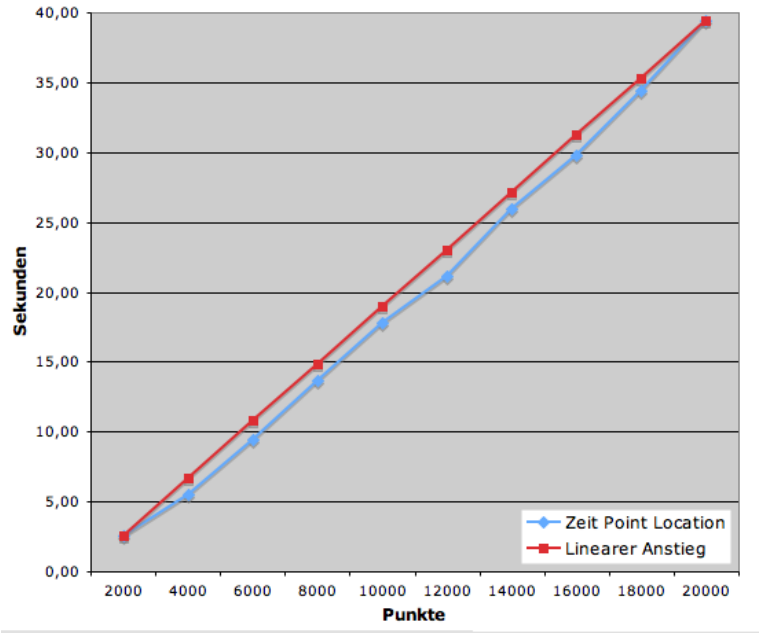


Abbildung 11.3.: Entwicklung der Zeit für die *Point Location*.

Anschließend werden wir die Tiefe des Delaunay-DAG bestimmen. Zum einen wird die maximale Tiefe festgehalten, zum anderen die durchschnittliche Tiefe, d. h. der über alle Blätter gemittelte Wert. Das Ergebnis zeigt Abbildung 11.4.

Die Laufzeit wächst leicht schneller als linear. Dies könnte ein Indiz für eine Laufzeit in $O(n \log n)$ sein. Die Daten des DAG geben genauere Auskünfte: Die durchschnittliche und maximale DAG-Tiefe scheinen in einem logarithmischen Verhältnis zur Anzahl der Punkte zu stehen. Dies bestätigt die durch die Laufzeitanalyse gewonnenen Erkenntnisse: Wir können von einer Laufzeit in $O(n \log n)$ ausgehen.

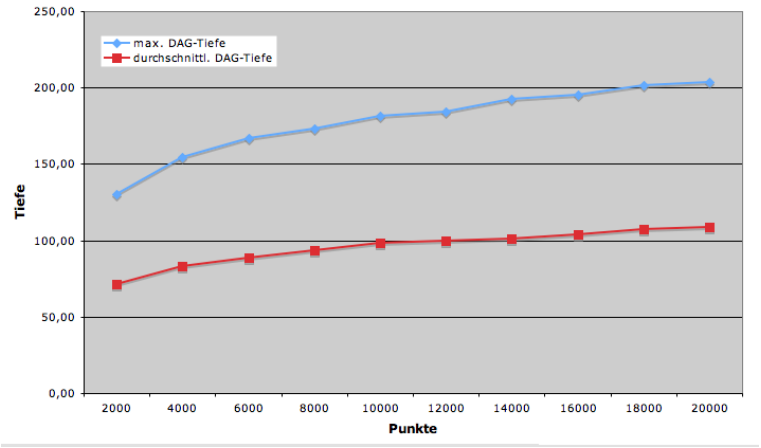


Abbildung 11.4.: Entwicklung der maximalen und der durchschnittlichen Tiefe des *DAG*.

11.2.2. Aktualisierung der Triangulation

Die Aktualisierung der Triangulation sollte $O(n)$ Zeit in Anspruch nehmen. Für jeden neu eingefügten Punkt steht also im Schnitt eine konstante Anzahl an Flips zur Verfügung, bis wieder eine Delaunay-Triangulation entsteht. Jeder einzelne Flip benötigt nur konstante Laufzeit. Neben der Laufzeit zählen wir die Gesamtanzahl der vollzogenen Flips und teilen sie durch die Größe der Punktmenge (siehe Abbildung 11.5).

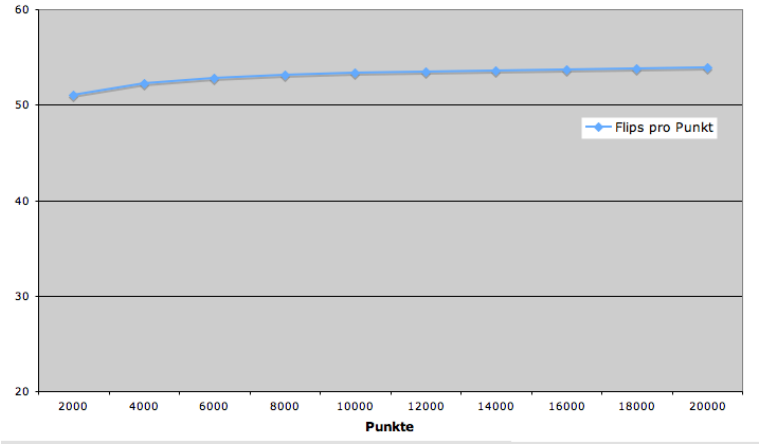


Abbildung 11.5.: Anzahl der Flips pro Punkt.

Die Werte nähern sich sehr schnell einer Konstanten. Wir können auch hier die geforderten

Werte bestätigen.

11.2.3. Durchlauf durch die AQE-Datenstruktur

Ein *AQE*-Durchlauf sollte ebenfalls nur $O(n)$ Zeit benötigen. Da jeder Punkt bei einem Durchlauf nur einmal besucht wird, hängt die Laufzeit von der Komplexität einer Voronoi-Region ab, da alle Flächen dieser Region durchlaufen werden. Wir berechnen deshalb die durchschnittliche Anzahl an Nachbarregionen (siehe Abbildung 11.6) pro Voronoi-Region aus der Anzahl der zu zeichnenden Voronoi-Flächen. Dieser Wert muss, nachdem er durch die Anzahl der Punkte geteilt wurde, noch verdoppelt werden, da eine Voronoi-Fläche zu zwei Regionen zählt, wir sie aber nur einmal zeichnen.

Da eine Region nach Meijering [28] bei entsprechender Verteilung der Punkte konstant viele Nachbarregionen besitzt, ist die Komplexität der Region ebenfalls nur konstant. Meijering erwartet für jede Region ≈ 15.54 Nachbarregionen im dreidimensionalen Raum; diese Konstante wird durch unsere Messungen bestätigt, denn der Graph konvergiert deutlich.

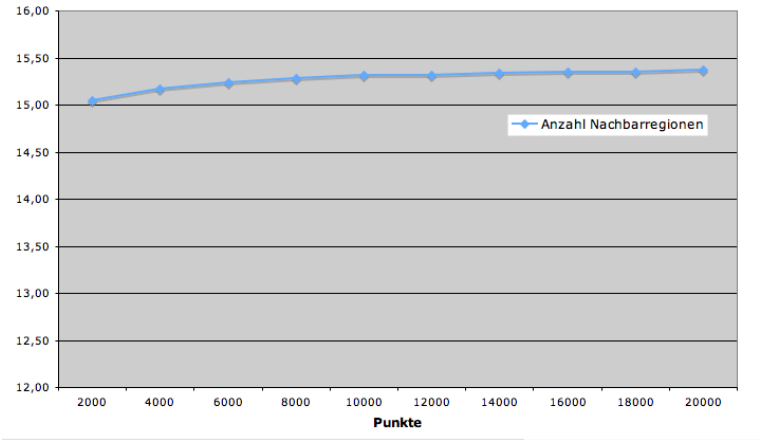


Abbildung 11.6.: Anzahl der Nachbarregionen für eine Voronoi-Region.

Unsere Werte nähern sich Meijerings Konstante, sind jedoch für kleinere Punktmengen deutlich geringer. Da unbeschränkte Voronoi-Regionen weniger Nachbarregionen haben

als beschränkte, vermuten wir, dass das Verhältnis von unbeschränkten zu beschränkten Voronoi-Regionen für größer werdende Punktmengen abnimmt. Diese Vermutung wird durch Har-Peled [23] bestätigt: Er zeigt, dass von einer innerhalb eines Würfels gleichverteilten Punktmenge der Größe n der Erwartungswert für die Anzahl der Punkte, die auf dem Rand der konvexen Hülle dieser Punktmenge liegen, in $O(\log^2 n)$ liegt. Da die Anzahl der Punkte auf der konvexen Hülle der Anzahl der unbeschränkten Voronoi-Regionen entspricht, muss die durchschnittliche Anzahl an Nachbarregionen für größer werdende Punktmengen ansteigen und sich einer Konstante nähern. Die Konstante würde erreicht werden, falls nur beschränkte Voronoi-Regionen betrachtet werden würden.

11.3. Speicherplatz

Der Gesamtspeicherplatz setzt sich aus dem für die *AQE*-Datenstruktur und für den *History-DAG* benötigten Speicherplatz zusammen. Wir werden beide im Folgenden betrachten.

11.3.1. Speicherplatzbedarf der AQE-Datenstruktur

Da der *AQE*-Speicherplatz linear zur Anzahl der Delaunay-Tetraeder ist, können wir ihn durch die Anzahl der Blätter des *DAG* abschätzen. Nach Dwyer [13] werden $O(n)$ Delaunay-Tetraeder für eine im Einheitswürfel gleichverteilte Punktmenge der Größe n erwartet. Um dies zu bestätigen, zeigen wir in Abbildung 11.7 die Anzahl der Tetraeder pro Punkt. Sie ist für die von uns gewählten Punktmengen durch eine Konstante beschränkt; wir erwarten, wie in Kapitel 2.5 erwähnt, den Wert 6.77. Auch an dieser Stelle stellt man fest, dass sich unsere Messergebnisse, wie schon im Kapitel davor, dem erwarteten Wert nähern.

Unsere Werte liegen stets unterhalb der geforderten Konstante, doch sind sie auch hier für kleinere Punktmengen merklich geringer. Avis und Bhattacharya [2] erhielten mit 6.31

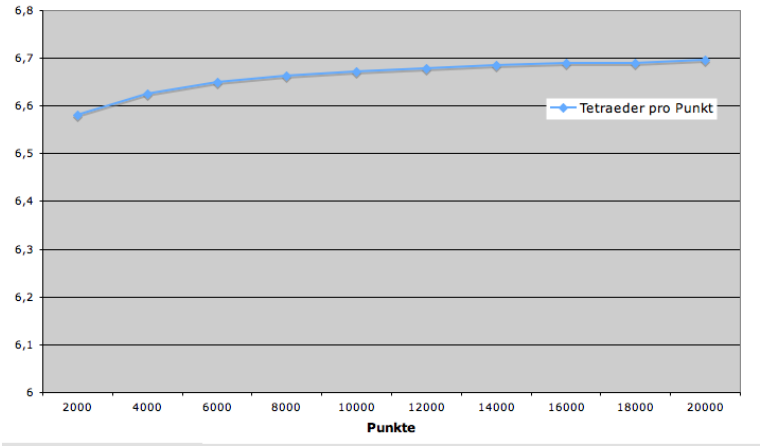


Abbildung 11.7.: Entwicklung des Speicherplatzbedarfs für die Tetraeder.

auch eine kleinere Konstante für Punktmengen der Größe 1000. Ähnlich wie bei der durchschnittlichen Anzahl der Nachbarregionen könnte es am Verhältnis zwischen Punkten auf dem Rand zu Punkten innerhalb der konvexen Hülle liegen: Weniger Nachbarregionen bedeuten weniger Delaunay-Kanten, also auch weniger Tetraeder.

11.3.2. Speicherplatzbedarf des History-DAG

Da die Größe des *History-DAG* der Anzahl seiner Knoten entspricht, ist diese der dominierende Faktor für den Gesamtspeicherplatzbedarf: Die AQE-Datenstruktur ist schließlich nur von der Anzahl der Blätter des DAG abhängig. Wir erwarten für die von uns gewählten Punktmengen einen zur Größe der Punktmenge linearen Speicherplatzbedarf. Abbildung 11.8 zeigt die von uns ermittelten Werte.

Auch hier bestätigen sich unsere Vermutungen: Die Werte nähern sich einer Konstanten an; die leicht steigenden Werte lassen sich wieder durch das Verhältnis zwischen Punkten auf dem Rand zu Punkten innerhalb der konvexen Hülle erklären.

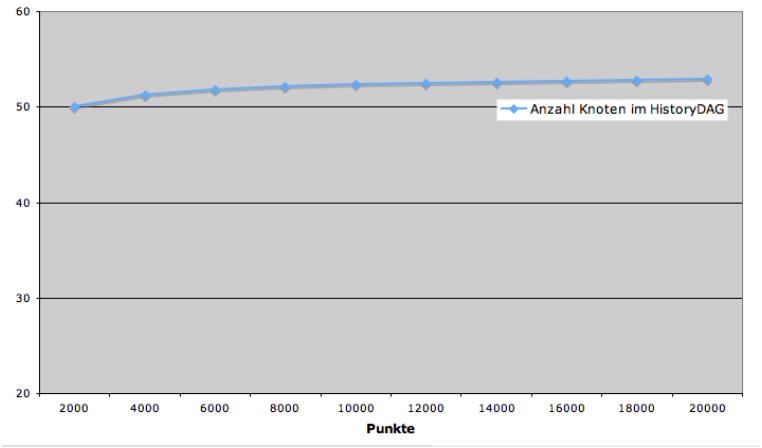


Abbildung 11.8.: Entwicklung des Speicherplatzbedarfes für den *History-DAG*.

11.4. Fazit

Die bei den Testläufen ermittelten Resultate decken sich mit den zuvor postulierten Prognosen. Die von uns an ein Programm zur Berechnung und Visualisierung des Voronoi-Diagramms in 3D gestellten Forderungen konnten wir damit erfüllen.

Teil III.

Ausblick

Die vorliegende Diplomarbeit stellt ein Programm zur Berechnung und Visualisierung von Voronoi-Diagrammen und Delaunay-Triangulationen in 3D zur Verfügung. Es bietet durch die vielfältigen Interaktionsmöglichkeiten mit den erzeugten Geometrien und durch die flexible Darstellung der inkrementellen Konstruktion ein effizientes Werkzeug zum besseren Verständnis dieser Diagramme. Die Verfügbarkeit des Applets im Internet (`http://www.voronoi3d.com`) ermöglicht eine zeit-, orts- und plattformunabhängige Verwendung.

Zur Berechnung der Delaunay-Triangulation implementierten wir den Algorithmus von Edelsbrunner und Shah [16], der mit einem *History-DAG* zur *Point Location* arbeitet und die *Flipping*-Technik zur Aktualisierung der Triangulation verwendet. Das Voronoi-Diagramm muss nicht explizit berechnet werden: Wir erhalten es durch die *Augmented Quad Edge*-Datenstruktur, die während der Konstruktion der Delaunay-Triangulation aufgebaut wird. Diese kantenbasierte Datenstruktur speichert sowohl das Voronoi-Diagramm als auch die duale Delaunay-Triangulation. Mit ihr ist zudem eine elegante Navigationsmöglichkeit über diese Diagramme gegeben.

Das im theoretischen Teil unserer Arbeit geforderte Laufzeit- und Speicherplatzverhalten konnten wir in der Performanzanalyse empirisch bestätigen. Für innerhalb einer Einheitssphäre gleichverteilte Punktmengen in allgemeiner Lage lässt sich das Voronoi-Diagramm in $O(n \log n)$ Zeit berechnen, dabei wird $O(n)$ Speicherplatz benötigt.

Die Basispakete sind so konzipiert, dass sie für weitere geometrische Algorithmen im Raum wiederverwendbar sind; wir verstehen sie als erweiterbare Geometriebibliothek.

Beim Testen des Programms mit großen Punktmengen stellte sich heraus, dass der beschränkende Faktor der Speicherplatzbedarf ist, wofür in erster Linie der *History-DAG* verantwortlich ist. Aus diesem Grund wäre ein anderer Ansatz zur Punktlokalisierung überlegenswert, wie z. B. ein auf der *jump-and-walk*-Technik (vgl. Mücke et al. [29]) basierender Algorithmus: Dafür werden schließlich nur die Tetraeder der aktuellen Triangulation und nicht die der gesamten Historie benötigt.

Die Robustheit unseres Programms könnte ebenfalls verbessert werden: Eine exakte Arithmetik würde eine sinnvolle Erweiterung darstellen. Durch ein geeignetes Zahlensystem wäre es zum einen möglich, das allumfassende Tetraeder analog zu Shah [35] so groß zu wählen, dass die Delaunay-Triangulation stets korrekt berechnet werden würde. Zum anderen wäre man damit in der Lage, Verletzungen der allgemeinen Lage zuverlässig zu erkennen. Um Punkte in nicht allgemeiner Lage zu behandeln, wäre es wünschenswert, die allgemeine Lage zu simulieren (vgl. Edelsbrunner und Mücke [15]).

Da mit diesem Programm Voronoi-Diagramme berechnet werden können, wäre es außerdem erstrebenswert, diese durch Bereitstellung entsprechender Funktionen zur Lösung von geometrischen Problemen, wie dem Nächsten-Nachbar-Problem, einzusetzen.

Die Ergonomie könnte durch Verbesserungen und Erweiterungen der grafischen Ausgabe und der Benutzeroberfläche gesteigert werden. Es wäre z. B. sinnvoll, das Entfernen von Punkten aus der Triangulation zu implementieren; dafür böte sich ein Ansatz von Devillers und Teillaud [11] an. Bei der inkrementellen Konstruktion der Delaunay-Triangulation wird diese bisher vollständig neu berechnet, sobald ein Benutzer einen Schritt der Konstruktion rückgängig macht. Besonders bei großen Punktmengen vergeht dadurch unnötig Zeit, bis mit der Bedienung des Programms fortgefahren werden kann.

Zudem ist bisher das Einfügen von Punktmengen nur über die Eingabe von Koordinaten möglich. Für den Benutzer wäre es intuitiver, Punkte durch Mausklicks direkt in die Zeichenfläche eingeben zu können; die Herausforderung dabei besteht darin, zweidimensionale Eingaben in Punkte im Raum zu übersetzen. Ebenso wäre es interessant, in der Triangulation vorhandene Punkte durch Mausbewegungen zu verschieben, um den Effekt auf die Diagramme dynamisch beobachten zu können.

Die Projektion der dreidimensionalen Diagramme auf die Bildschirmoberfläche vermindert den räumlichen Eindruck: Eine stereoskopische Visualisierung wäre eine reizvolle Erweiterung und würde die Vorstellungskraft des Benutzers erheblich beflügeln.

Wir hoffen mit unserem Programm ein Werkzeug zur Verfügung zu stellen, das zu ei-

nem tieferen Verständnis des Voronoi-Diagramms beiträgt; es würde uns freuen, wenn wir durch unsere Arbeit das Interesse anderer an der algorithmischen Geometrie wecken könnten.

Teil IV.

Anhang

A. Beschreibung / Präsentation / Erläuterung der GUI

In diesem Kapitel stellen wir die Benutzung und die grafische Oberfläche des Programms vor. Diese ist in drei Bereiche aufgeteilt (siehe Abbildung A.1):

Navigationsleiste Im oberen Bereich kann der Benutzer Punkte von der Festplatte laden bzw. speichern, neue Punkte generieren, die Berechnung der Diagramme beeinflussen und sich eine kurze Benutzerführung zum Programm anzeigen lassen.

Änderungsbereich Im linken Bereich ist es dem Benutzer möglich, die Art und Weise der gewünschten Diagramme einzustellen. Zusätzlich wird die Liste der Punkte angezeigt. Im untersten Teil können außerdem neue Punkte hinzugefügt bzw. alte, noch nicht in die Berechnung eingebundene Punkte entfernt werden.

Visualisierungsbereich Im zentralen Bereich werden die berechneten Diagramme mit den eingestellten Visualisierungsoptionen angezeigt.

A.1. Die Navigationsleiste

Das Programm kann als Applet wie auch als Applikation gestartet werden. Der Unterschied zwischen den beiden Ausführungsmethoden ist, dass die Appletversion weder eine Datei laden noch speichern kann. Aus diesem Grund werden die entsprechenden

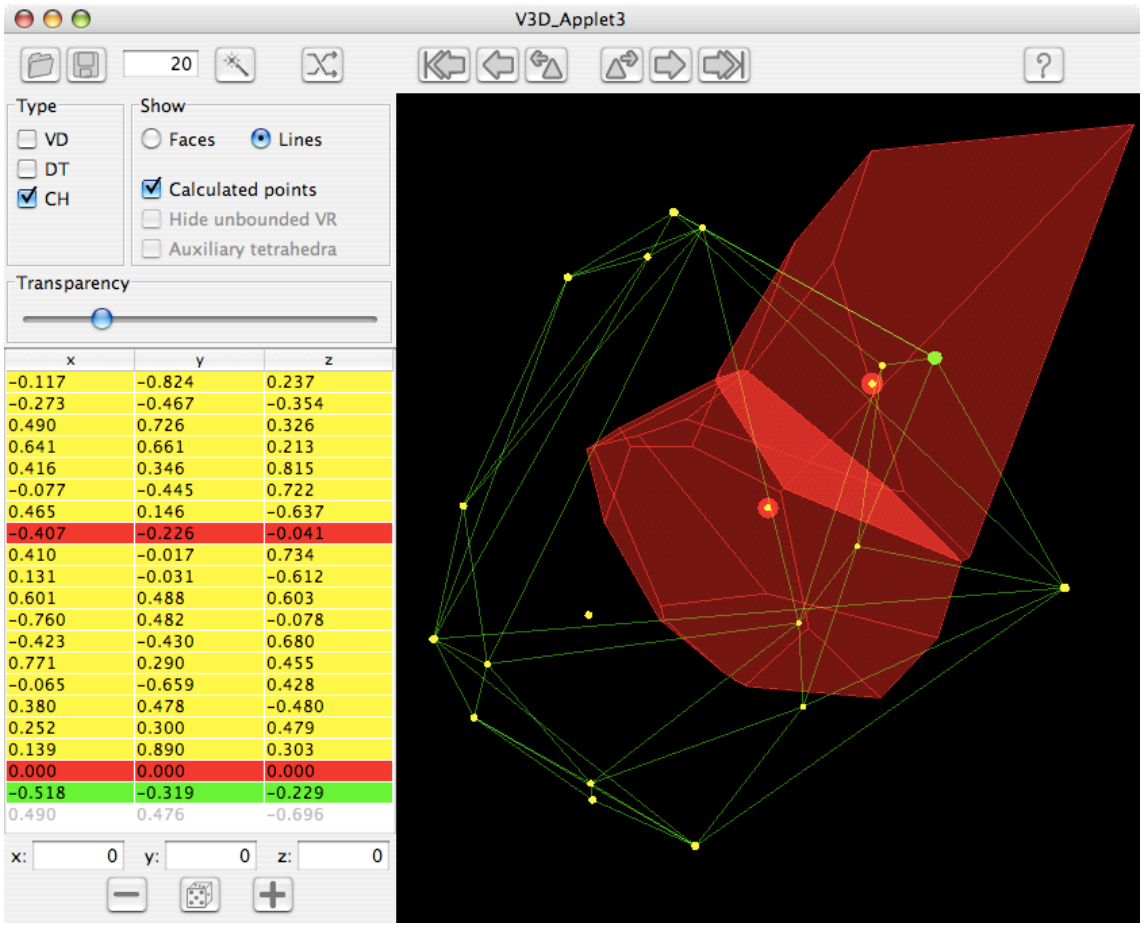


Abbildung A.1.: Grafische Oberfläche des Programms.

Schaltflächen im linken Bereich der Navigationsleiste (siehe Abbildung A.1) bei dieser Ausführungsart ausgeblendet. Das Dateiformat[1] ist schlicht gehalten, denn eine Zeile enthält lediglich die drei Werte für die x-, y- respektive z-Koordinaten eines Punktes, die jeweils mit einem Tabulatorzeichen oder einem Leerzeichen getrennt sind und mit einem Zeilenumbruch beendet werden. Enthält eine Zeile korrupte Daten, so wird diese übersprungen, und alle Daten aus der Zeile werden ignoriert. Eine Zeile ist korrupt, falls sie mindestens eine der folgenden Voraussetzungen erfüllt:

- enthält mindestens eine Koordinate mit einem Wert $\gtrsim |1.0|$,

[1]Die Datei muss als Endung `.v3d` haben, damit ein Öffnen prinzipiell möglich ist.

- enthält mindestens ein Zeichen, das nicht in eine Ziffer konvertiert werden kann,
- enthält „,“ anstatt „.“ als Dezimaltrenner,
- enthält weniger als drei Koordinatenwerte,
- enthält einen bereits eingelesenen Punkt.

Das Textfeld des Änderungsbereichs enthält die Anzahl der zu erzeugenden Punkte und nimmt nur positive ganze Zahlen entgegen. Wird nach Eingabe der gewünschten Anzahl von Punkten die Bestätigungstaste oder der Generier-Button (Zauberstab) gedrückt, so wird eine zufällige Punktmenge der gewünschten Größe erzeugt; diese erscheint schließlich in der Punktliste des Änderungsbereichs. Möchte man die Reihenfolge der Punkte aus der Liste ändern, so kann man die Punkte durch einen Druck auf den Shuffle-Button (sich kreuzende Pfeile) zufällig permutieren.

Nun kann mit den diversen Vorwärts- (drei Rechtspfeile) bzw. Rückwärtsschaltflächen (drei Linkspfeile) in den gewünschten Diagrammen flipweise (Pfeil mit Dreieck), punktweise (einzelner Pfeil) oder absolut zum Anfang bzw. Ende (doppelter Pfeil) der Berechnung navigiert werden.

Die am weitesten rechts liegende Schaltfläche (Fragezeichen) der Navigationsleiste zeigt bei einem Druck einen kurzen Hilfetext zur Benutzung des Programms.

A.2. Der Änderungsbereich

Im Änderungsbereich (siehe Abbildung A.1) kann die Art und Darstellungsweise der anzuzeigenden Geometrien bestimmt werden.

A.2.1. Die Geometriearten

Im Kasten „Type“ kann der Benutzer die anzuzeigenden Geometriearten Voronoi-Diagramm („VD“), Delaunay-Triangulation („DT“) oder konvexe Hülle („CH“) einstellen. Dabei ist es möglich, keine, eine, mehrere oder alle diese Geometrien anzeigen zu lassen. Zur besseren Erkennbarkeit wurden dabei für die einzelnen Geometrien verschiedene Farben gewählt.

A.2.2. Anzeige der Geometriearten

Im Kasten „Show„ kann der Benutzer Parameter zur Darstellung der Diagramme ändern. Anhand der Radiobuttons „Faces“ und „Lines“ ändert sich das Aussehen, nämlich, ob die Flächen ausgefüllt bzw. ob nur die Kanten der Flächen gezeichnet werden. Die Checkbox „Calculated points“ zeigt bzw. entfernt die für die berechneten Punkte gezeichneten Sphären. „Hide unbounded VR“ versteckt die unendlichen Voronoi-Regionen, was insbesondere bei vielen Punkten und bei der Darstellung mit ausgefüllten Flächen der besseren Erkennbarkeit dient. Die Checkbox „Auxiliary tetrahedra“ regelt, ob die Hilfstetraeder, die mindestens einen Punkt aus dem allumfassenden Tetraeder beinhalten, ein- oder ausgeblendet werden sollen.

Der Transparenzregler im Kasten „Transparency“ regelt das Maß der Durchsichtigkeit der Zeichnungen. In der Konfigurationsdatei kann der minimale und maximale Wert des Reglers eingestellt werden. Zusätzlich kann noch bestimmt werden, ob die Transparenzwerte logarithmisch oder linear berechnet werden sollen; wir haben die logarithmische Berechnung eingestellt, da bei einer linearen die Zeichnungen schon bei recht wenigen Punkten zu schnell entweder ganz undurchsichtig oder ganz unsichtbar werden.

A.2.3. Die Punktliste

Die Liste zeigt die Punkte mit ihren x-, y- und z-Koordinaten. Diese Liste ist in der Reihenfolge, wie die Punkte generiert wurden, kann aber mittels des Shuffle-Buttons (siehe Seite 117) zufällig permutiert werden. Die Zeilen mit den Punkten, die noch nicht berechnet wurden, sind ausgeblendet, wogegen die mit den bereits berechneten Punkten gelb – wie die Farbe der angezeigten Punkte aus dem Visualisierungsbereich – und die des zuletzt berechneten Punktes grün – wie die Farbe des letzten Punktes aus dem Visualisierungsbereich – hinterlegt sind. Wählt man nun eine oder mehrere Zeilen von berechneten Punkten aus, so werden diese rot hinterlegt und die dazugehörigen Voronoi-Regionen farbig hervorgehoben.

Unterhalb der Liste befinden sich Felder für die manuelle Eingabe von einzelnen Punkten; diese Felder können aber auch mit zufälligen ganzzahligen Werten im Bereich $[0, 999]$ durch einen Klick auf den Würfel-Button gefüllt werden. Ein Klick auf den „+“-Button fügt nach vorheriger Normalisierung der Punkte (siehe Seite 100) die Koordinatenwerte als einen neuen Punkt hinter dem gerade berechneten Punkt aus der Liste ein. Mittels des „−“-Buttons können noch nicht berechnete Punkte aus der Liste entfernt werden.

A.3. Der Visualisierungsbereich

Der Visualisierungsbereich stellt die durch die Spezifikationen aus den beiden oben beschriebenen Bereichen gegebenen Diagramme dar. Nun kann das dreidimensionale Gebilde rotiert, translatiert oder skaliert werden. Zunächst wählt man mit dem Mauszeiger eine Fläche, Linie oder einen Punkt der Grafik aus und hält die entsprechende Maustaste während des Vorgangs gedrückt: die linke Taste für die Rotation, die rechte für die Trans-

lation und die mittlere für die Skalierung[2] des gesamten dargestellten Gebildes.[3]

Ist die Delaunay-Triangulation nicht komplett für eine Punktmenge berechnet – wenn z. B. flipweise vorgegangen wird –, so ist das Voronoi-Diagramm ausgeblendet, denn die Zeichnung wäre nicht korrekt. Aus diesem Grunde können während eines solchen Zwischenstadiums keine Voronoi-Regionen für Punkte angezeigt werden.

[2]Bei der Skalierung muss der Mauszeiger nach oben bzw. unten bewegt werden, um die Zeichnung zu vergrößern respektive zu verkleinern.

[3]Sollte keine Dreitastenmaus zur Verfügung stehen, so können rechte bzw. mittlere Maustasten durch das Festhalten der Tastaturtaste `ctrl` bzw. `alt` und gleichzeitiges Drücken der linken Maustaste simuliert werden.

B. Grafische Darstellung

Die grafische Darstellung der Diagramme verlangt besondere Aufmerksamkeit, da zum Zeitpunkt der Erstellung dieser Arbeit noch keine ausgereiften Bibliotheken für dreidimensionale Programmierung existieren. Wie schon in der Einleitung beschrieben, sind zwar verschiedene Ansätze auf dem Markt, aber keine von ihnen verfügt über die gestellten Anforderungen wie Plattformunabhängigkeit, Lauffähigkeit in einem Browser oder als Programm. Auch wegen dieser Gründe entschieden wir uns für die Programmiersprache Java und ihre Bibliothek Java3D.

B.1. Was ist Java3D?

Java3D ist eine Java-Bibliothek zur Erstellung von dreidimensionalen Grafiken. Sie stellt eine Menge von Klassen zur Verfügung, die eine Schnittstelle zur Darstellung und interaktiven Benutzung von dreidimensionalen Programmen bietet. Die von der Hardware unabhängigen Klassen stellen eine Abstraktionsebene dar, mit der ein dreidimensionales Programm mit komplexen räumlichen Datenstrukturen elegant und effizient gelöst werden kann. Der Entwickler benutzt eine Menge von abstrakten Objekten, um die gewünschte Geometrie zu erzeugen; diese Objekte sind in einem sogenannten virtuellen Universum (siehe unten) eingebettet, welches von Java3D automatisch gerendert werden kann.

Ein solches virtuelles Universum stellt den Kern eines Java3D-Programms dar. Dieses ist in eine Baumhierarchie von sogenannten *Branch Groups* gegliedert, die ihrerseits die Wur-

zeln von Subgraphen sind. Diese Untergraphen können ein einzelnes Objekt oder ganze komplexe dreidimensionale Gebilde repräsentieren und darstellen. Damit ermöglicht diese Baumhierarchie erst die übersichtliche Programmstruktur und das schnelle Rendering, das durch die gegebene Baumstruktur automatisch vom Renderer parallelisiert ausgeführt werden kann [37]; dieses parallele Randering wird mit Hilfe von Java-Threads realisiert.

B.2. Die Java3D-API

Jedes Java3D-Programm enthält zumindest einige Objekte aus der Java3D-Klassenhierarchie. Diese Objektsammlung wird auch *virtuelles Universum* genannt und muss im Laufe des Programms gerendert werden. Die API beinhaltet über 100 sogenannten Kernklassen (*core classes*) im Paket `javax.media.j3d`. Diese Klassen bieten eine gewisse Grundfunktionalität, sind jedoch nicht allzu komfortabel zu benutzen.

Aus diesem Grund stellt das Paket `com.sun.j3d.utils` sogenannten Dienstklassen (*utility classes*) zur Verfügung, die auf die Kernklassen aufbauen und die bloß notwendige Funktionalität erweitern. Dieses Paket kann seinerseits in vier Hauptkategorien unterteilt werden: *content loaders*, *scene graph construction aids*, *geometry classes* und *convenience utilities*.

Zusätzlich zu diesen beiden Paketen verwenden alle Java3D-Programme Klassen aus den `java.awt`- und `javax.vecmath`-Paketen. Das erste stellt die Funktionalität zur Verfügung, Fenster und Rendering anzuzeigen; das zweite definiert mathematische Vektorklassen wie Punkte, Vektoren, Matrizen und andere mathematische Strukturen.

B.3. Aufbau des Szenengraphen

In Java3D ist ein virtuelles Universum ein Szenengraph, der aus Objekten zur Definition von Geometrien, Tönen, Lichtern, Platzierungen, Ausrichtungen und Erscheinungen be-

steht. Wie oben erwähnt, bildet der Szenengraph eine Baumstruktur, sodass jedes Kind maximal einen Elterknoten besitzt bzw. ein Elterknoten beliebig viele Kinderknoten haben kann. Der Szenengraph ist ein Kind des sogenannten Schauplatzes (*locale*).

Da die Java3D-Objekte eines Programms stets eine Baumstruktur aufweisen, existiert jeweils genau ein Pfad vom Wurzelknoten zu den einzelnen Blättern, den wir *Pfad im Szenengraphen* nennen. Da diese Pfade eindeutig sind, definieren sie den Informationsstatus der Blätter des Szenengraphen. Diese Informationen betreffen die Platzierung, die Orientierung und die Größe des Objektes; folglich hängen diese Attribute eines Objektes nur von seinem Pfad im Szenengraphen ab. Diese Eigenschaft macht sich der Renderer zu Nutze, indem der eine optimierte Vorgehensweise bei der Zeichnung wählen kann. Weiterhin ermöglicht diese Eigenschaft ebenfalls, dass sich der Programmierer nicht um die Zeichnungsdetails kümmern braucht bzw. auch nicht ohne Weiteres die Arbeitsweise des Renderers beeinflussen kann.

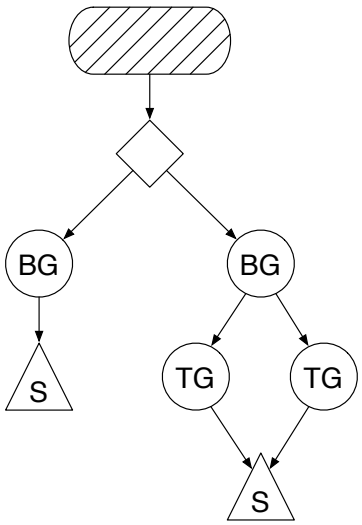


Abbildung B.1.: Ein illegaler Szenengraph (nach Java3D-Tutorials von Sun [37]).

Es ist möglich, illegale Szenengraphen aufzubauen (siehe Abbildung B.1). Der Compiler wird keinen Fehler melden, aber während der Laufzeit wird eine „Mehrfache-Eltern“-Ausnahme geworfen, und der Szenengraph wird nicht gerendert: Eine mögliche Auflösung

des Zyklus könnte wie in Abbildung B.2 dargestellt aussehen.

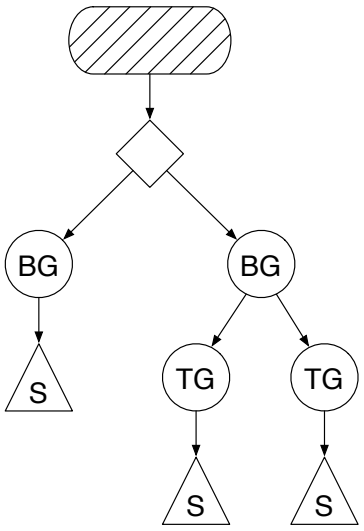


Abbildung B.2.: Eine mögliche Korrektur des illegalen Szenengraphen aus Abbildung B.1 (nach Java3D-Tutorials von Sun [37]).

Ein *Branch Group*-Objekt ist die Wurzel eines Untergraphen und ist in zwei unterschiedlichen Kategorien von Untergraphen aufgeteilt: der Sichtuntergraph (*view branch graph*) und Inhaltsuntergraph (*content branch graph*). Letzterer definiert die Inhalte des virtuellen Universums, wie Geometrie, Erscheinung, Verhalten, Platzierung, Ton und Licht, wogegen ersterer die Sichtparameter wie Sichtplatzierung und Sichtrichtung spezifiziert.

B.4. Das einfache Universum

Die Erstellung eines virtuellen Universums benötigt eine Menge von Java3D-Objekten. Da häufig ein Programm kein besonderes Universum benötigt, stellt die Java3D-Bibliothek ein einfaches Universum (*simple universe*) (siehe Abbildung B.3) zur Verfügung, das die benötigten Elemente in korrekter Weise zusammenstellt.

Das einfache Universum definiert den vollständigen Sichtuntergraphen eines virtuellen Universums, sodass sich der Programmierer nur noch um den zweiten Graphen zur De-

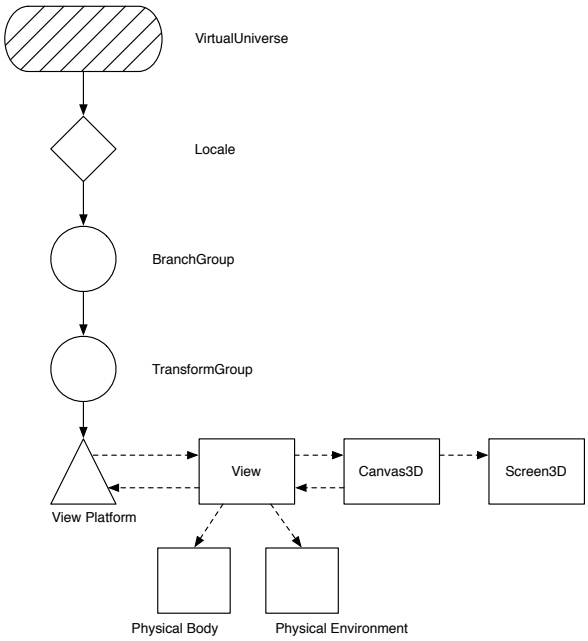


Abbildung B.3.: Der Aufbau des einfachen Universums (nach Java3D-Tutorials von Sun [37]).

finition der Inhalte kümmern braucht. Dieser Sichtuntergraph beinhaltet die sogenannte Bildschirmscheibe (*image plate*) (siehe Abbildung B.4), auf die der Inhalt projiziert wird, um das gerenderte Bild darzustellen. Die Bildschirmscheibe wird in Java3D durch ein Canvas3D-Objekt repräsentiert.

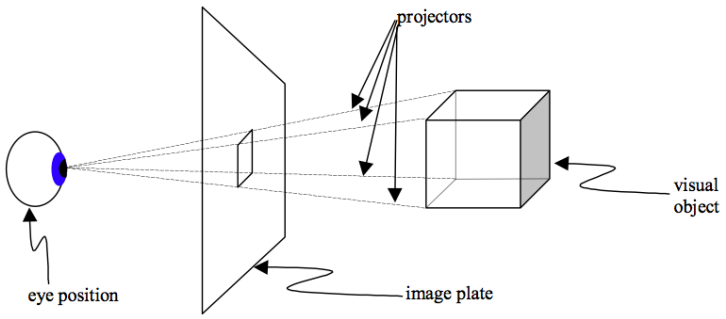


Abbildung B.4.: Skizzenhafte Darstellung des Betrachters eines virtuellen Universums (aus Kapitel 1 des Java3D-Tutorials von Sun [37]).

Standardmäßig ist die Bildschirmscheibe relativ zum Ursprung des einfachen Universums

zentriert angeordnet (siehe Abbildung B.5). Dabei ist die Standardausrichtung, dass man auf die z-Achse schaut, d. h. die positive x-Achse verläuft nach rechts und die positive y-Achse nach oben. Damit befindet sich der Nullpunkt des Koordinatensystems in der Mitte der Bildschirmscheibe.

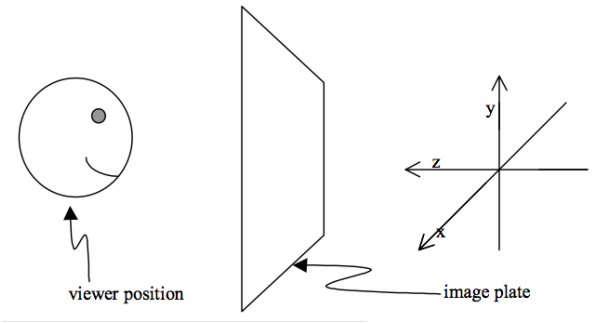


Abbildung B.5.: Koordinatensystem in Java3D aus der Sicht des Betrachters (aus Kapitel 2 des Java3D-Tutorials von Sun [38]).

C. Voronoi-Diagramme auf der Kugeloberfläche

C.1. Definition

Gegeben sei eine Kugel U mit dem Ursprung als Mittelpunkt und Radius 1:

$$U = p \in \mathbb{R}^3 : \|p\| = 1.$$

$\|\cdot\|$ sei die euklidische Norm des $\mathbb{R}^3$. Eine Metrik ist durch die Distanzfunktion

$$d(p,q) = \arccos\langle p,q\rangle, \forall p,q \in U$$

definiert, wobei $\langle\cdot,\cdot\rangle$ das Skalarprodukt im $\mathbb{R}^3$ ist. Die Distanz zweier Punkte $p,q \in U$ ist dabei stets im Intervall $[0,\pi]$.

Ein Kreis auf U mit einem Mittelpunkt $v \in U$ und Radius $r \in [0,\pi/2]$ enthält demnach alle Punkte $p \in U$, für die $d(v,p) \leq r$ gilt.

Das Voronoi-Diagramm auf einer Kugeloberfläche einer Punktmenge S ist eine Partition dieser Sphärenoberfläche in Voronoi-Regionen. Eine Voronoi-Region zu einem Punkt $p \in S$ besteht aus allen Punkten der Kugeloberfläche U, die näher an p liegen als an allen übrigen Punkten aus S, d.h.

$$VR(p,S) := \{z \mid d(p,z) < d(q,z), q \in S \setminus \{p\}, z \in U\}.$$

C.2. Lösbarkeit mit dem Algorithmus von Edelsbrunner und Shah

Der vorgestellte Algorithmus von Edelsbrunner und Shah [16] ist in unserer Implementation nicht dafür geeignet, Voronoi-Diagramme auf der Kugeloberfläche zu berechnen, da jeweils andere Regularitätsbedingungen für die Delaunay-Triangulation gelten: Während auf der Kugeloberfläche geprüft werden muss, ob ein anderer Punkt der Punktmenge innerhalb des Umkreises eines Dreiecks liegt, wird im Raum getestet, ob ein anderer Punkt innerhalb der das Tetraeder umgebenden Sphäre liegt.

Der Algorithmus von Edelsbrunner und Shah maximiert also die Winkel eines Tetraeders. Dabei werden gegebenenfalls Winkel eines Tetraeders maximiert, die keine Rolle spielen, da ja nur die Innenwinkel der Dreiecke auf der Kugeloberfläche maximiert werden müssen.

Selbst wenn die Regularitätsbedingung anwendbar wäre, könnte es vorkommen, dass auf der Kugeloberfläche befindliche Dreiecke benachbart sind, die zugehörigen Tetraeder aber nicht. Also könnte unter Umständen ein notwendiger Flip nicht durchgeführt werden.

Zudem ist eine wesentliche Voraussetzung für die Verwendung des Algorithmus, dass sich die Punktmenge in allgemeiner Lage befindet: Punkte auf einer Kugeloberfläche verletzen diese Bedingung aber stets.

Ein weiteres Problem tritt auf, wenn sich die Punktmenge nur auf einer Hemisphäre befindet. Ist dies der Fall, ist die Delaunay-Triangulation keine Partition der Kugeloberfläche: Es gibt ein Gebiet außerhalb der konvexen Hülle. Dadurch existieren mehr Voronoi-Knoten als Delaunay-Dreiecke, da sich die Bisektoren, die im $\mathbb{R}^2$ Halbgeraden wären, in zusätzlichen Voronoi-Knoten treffen müssen.

C.3. Ein Algorithmus für Voronoi-Diagramme auf der Kugeloberfläche

Ein inkrementeller Algorithmus zur Berechnung von Delaunay-Triangulationen und Voronoi-Diagrammen auf der Kugeloberfläche wurde 1997 von Renka [33] vorgestellt. Er ähnelt dem Algorithmus von Edelsbrunner und Shah insofern, als auch die *Flipping*-Technik verwendet wird.

Allerdings müssen zusätzliche Voronoi-Knoten berechnet werden, wenn sich die Punktmenge nur auf einer Hemisphäre befindet. Nach dem Einfügen eines Punktes und der Durchführung der *Edge Flips* ist also ein zusätzlicher Schritt notwendig, um ein korrektes Voronoi-Diagramm zu berechnen.

C.3.1. Berechnung der Voronoi-Knoten

Man stelle sich den einfachsten Fall vor: Drei Punkte, p_1, p_2 und p_3, liegen auf einer Kugel U und bilden ein Dreieck $tria_{\{p_1,p_2,p_3\}}$. Dieses Dreieck befindet sich natürlich nur auf einer Hemisphäre der Kugel. Da es drei Punkte sind, besteht das Voronoi-Diagramm aus drei Regionen. Die drei Dreieckseiten entsprechen ihren drei dualen Voronoi-Kanten. Bei einem Dreieck in der Ebene wären diese drei Bisektoren Halbgeraden; auf der Kugeloberfläche treffen sie sich in einem zweiten Voronoi-Knoten. Der erste Voronoi-Knoten $v_{1,2,3}$ ist der Mittelpunkt des Dreiecks $tria_{\{p_1,p_2,p_3\}}$, der zweite Voronoi-Knoten ist $-v_{1,2,3}$. Man sieht schon jetzt, dass die Anzahl der Voronoi-Knoten bei Voronoi-Diagrammen auf der Kugeloberfläche nicht immer der Anzahl der Dreiecksmittelpunkte entspricht.

Wir unterscheiden folgende Typen von Delaunay-Knoten: Äußere Knoten seien solche, die auf dem Rand der konvexen Hülle von S liegen, innere Punkte diejenigen, die im Inneren der konvexen Hülle liegen. Man stelle sich ein Delaunay-Dreieck $tria_{\{p_i,p_j,p_k\}}$ vor: Der Dreiecksmittelpunkt $v_{i,j,k}$ ist nach Definition ein Voronoi-Knoten. Doch ist auch $-v_{i,j,k}$ ein Voronoi-Knoten? Für Dreiecke, die aus mindestens einem inneren Knoten bestehen,

ist dies ausgeschlossen, denn der Dreiecksmittelpunkt liegt in jedem Fall näher an einem Punkt $p_l \in S$ als an den Punkten p_i, p_j oder p_k. Damit liegt er in der Voronoi-Region von p_l und kann kein Voronoi-Knoten sein. Wählt man nämlich einen Knoten p_l, der außerhalb des Umkreises von $tria_{\{p_i,p_j,p_k\}}$ liegt, so gilt folgendes:

$$d(v_{i,j,k}, p_l) > r_{i,j,k} \Rightarrow d(-v_{i,j,k}, p_l) = \pi - d(v_{i,j,k}, p_l) < \pi - r_{i,j,k} = d(-v_{i,j,k}, p_m), m \in i, j, k.$$

Um die zusätzlichen Voronoi-Knoten zu bestimmen, berechnet man also eine Triangulation der äußeren Knoten, für deren Dreiecke $tria_{\{p_i,p_j,p_k\}}$ gilt, dass jeder Punkt $p_l \in S, l \neq i, j, k$ in dessen Umkreis enthalten ist. Diese Triangulation entspricht der *furthest site*-Delaunay-Triangulation, sie ist dual zum *furthest site*-Voronoi-Diagramm. Es folgt, dass die negativen Dreiecksmittelpunkte $-v_{i,j,k}$ der Dreiecke dieser *furthest site*-Delaunay-Triangulation Voronoi-Knoten sind:

$$d(v_{i,j,k}, p_l) \leq r_{i,j,k} \Rightarrow d(-v_{i,j,k}, p_l)) = \pi - d(v_{i,j,k}, p_l) \geq \pi - r_{i,j,k} = d(-v_{i,j,k}, p_m), m \in i, j, k.$$

Die *furthest site*-Delaunay-Triangulation der äußeren Punkte wird inkrementell berechnet. Dabei wird der Regularitätstest umgekehrt ausgeführt: Ein Flip erfolgt, wenn ein vierter Punkt nicht im Umkreis eines Dreiecks enthalten ist.

Wir gehen davon aus, dass eine Datenstruktur verwendet wird, bei der die Punkte der konvexen Hülle zu jeder Zeit in zyklischer Ordnung sortiert vorliegen. Der Algorithmus kann dann wie folgt vereinfacht werden: Das erste Dreieck sei $tria_{\{p_1,p_2,p_3\}}$. Ein k-ter Punkt wird eingefügt; er sieht nur die Punkte p_1 und p_{k-1}, da die Punkte ja die konvexe Hülle von S bilden. Das Dreieck $tria_{\{p_1,p_{k-1},p_k\}}$ wird also hinzugefügt. Danach werden, wie im Algorithmus von Edelsbrunner und Shah beschrieben, die *Link Facets* zu p_k auf einen Stapel gelegt und, falls die zuvor beschriebene abweichende Regularitätsbedingung verletzt wird, durch Flips eine neue *furthest site*-Delaunay-Triangulation erstellt.

Die Menge der Voronoi-Knoten besteht also aus den Mittelpunkten der Dreiecke der Delaunay-Triangulation und den negativen Mittelpunkten der Dreiecke der *furthest site*-Delaunay-Triangulation der Knoten der konvexen Hülle der Punktmenge.

C.3.2. Berechnung der Voronoi-Kanten

Die Voronoi-Kanten verbinden die zuvor berechneten Voronoi-Knoten.

Innere Delaunay-Kanten der Delaunay-Triangulation DT, also solche mit zwei Dreiecken als Nachbarn, entsprechen den Voronoi-Kanten, die die beiden Dreiecksmittelpunkte verbinden. Innere Delaunay-Kanten der *furthest site*-Delaunay-Triangulation der Punkte der konvexen Hülle entsprechen Voronoi-Kanten, die die jeweiligen negativen Dreiecksmittelpunkte verbinden.

Für äußere Kanten von Dreiecken der Delaunay-Triangulation verbindet die zugehörige Voronoi-Kante den Dreiecksmittelpunkt des Dreiecks mit dem negativen Dreiecksmittelpunkt der Dreiecks aus der *furthest site*-Delaunay-Triangulation, das ebenfalls diese äußere Kante besitzt.

D. Aufgabenteilung

In diesem Kapitel listen wir auf, welche Bereiche hauptverantwortlich von wem erstellt bzw. bearbeitet wurden. Eine strikte Trennung der Aufgabenbereiche ist kaum möglich, da wir den Großteil der Arbeiten gemeinsam erledigt haben; dennoch möchten wir eine wie folgt geartete Unterteilung vornehmen:

Edgar Klein hat neben der Zusammenfassung und Einleitung die Kapitel 1, 7.1, 9 und 10 sowie die Anhänge A und B verfaßt. Christoph Baudson hat die Kapitel 3, 4, 6, 7.2, 8 sowie Anhang C und den Ausblick geschrieben. Die Kapitel 2, 5 und 11 wurden vollkommen gemeinschaftlich erstellt.

Bei der Programmierung war Edgar Klein für die grafische Benutzeroberfläche und die Darstellung der Geometrien, Christoph Baudson für die AQE-Datenstruktur und die Datenerhebung für den Performanztest verantwortlich. Die Konzeption und die Umsetzung der theoretischen Grundlagen wurde gemeinsam erarbeitet, ebenso die Implementation der Basisklassen und der Aufbau der Delaunay-Triangulation.

Literaturverzeichnis

[1] Aurenhammer, F. (1991). Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys, 23(3),* 345-405.

[2] Avis, D. & Bhattacharya, B. K. (1983). Algorithms for computing *d*-dimensional Voronoi diagrams and their duals. In F. P. Preparata (Ed.), *Advances in Computing Research: Computational Geometry* (pp. 159-180). Greenwich, CT: JAI Press Inc.

[3] Barber, C. B., Dobkin, D. P. & Huhdanpaa, H. T. (1996). The Quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software, 22(4),* 469-483.

[4] Boissonnat, J. D., Devillers, O., Teillaud, M. & Yvinec, M. (2000). Triangulations in CGAL (extended abstract). In: *Proceedings of the Sixteenth Annual Symposium on Computational Geometry (Clear Water Bay, Kowloon, Hong Kong, June 12-14, 2000)* (pp. 11-18). New York, NY: ACM Press.

[5] Bourke, P. (2002). Equation of a sphere from four points.

Online zugänglich unter:
`http://astronomy.swin.edu.au/ pbourke/geometry/spherefrom4`

[6] Bowyer, A. (1981). Computing Dirichlet tessellations. *The Computer Journal, 24(2),* 162-166.

[7] CGAL Homepage.

Online zugänglich unter:
`http://www.cgal.org`
(Zuletzt abgerufen am 31.03.2006)

[8] CGAL User and Reference Manual: All Parts, Release 3.1 (10. Dezember 2004).

[9] Clarkson, K. L. (1992). Safe and effective determinant evaluation. In: *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science* (pp. 387-395). New York, NY: ACM Press.

[10] Devillers, O. (1998). Improved incremental randomized Delaunay triangulation. In: *Proceedings of the 14th Annual ACM Symposium on Computational Geometry* (pp. 106-115). New York, NY. ACM Press.

[11] Devillers, O. and Teillaud, M. (2003). Perturbations and vertex removal in a 3D Delaunay triangulation. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, Maryland, January 12 - 14, 2003). Symposium on Discrete Algorithms* (pp. 313-319). Philadelphia, PA: Society for Industrial and Applied Mathematics.

[12] Dewdney, A. K. & Vranch, J. K. (1977). A convex partition of $\mathbb{R}^3$ with applications to Crum's problem and Knuth's post-office problem. *Utilitas Mathematica, 12,* 193-199.

[13] Dwyer, R. A. (1989). Higher-dimensional Voronoi diagrams in linear expected time. In: *Proceedings of the Fifth Annual Symposium on Computational Geometry (Saarbrücken, West Germany, June 05-07, 1989)* (pp. 326-333). New York, NY: ACM Press.

[14] Edelsbrunner, H. (2000). Triangulations and meshes in computational geometry. *Acta Numerica, 9,* 133-213.

[15] Edelsbrunner, H. & Mücke, E. P. (1990). Simulation of simplicity: a technique to cope

with degenerate cases in geometric algorithms. *ACM Transactions on Graphics, 9(1)*, 66-104.

[16] Edelsbrunner, H. & Shah, N. R. (1996) Incremental topological flipping works for regular triangulations. *Algorithmica, 15,* 223-241.

(Erstveröffentlichung 1992, in: *Proceedings of the 8th Annual ACM Symposium on Computational Geometry* (pp. 43-52).)

[17] GeomView Homepage.

Online zugänglich unter:
`http://www.geomview.org`

(Zuletzt abgerufen am 31.03.2006)

[18] GNU lesser general public license.

Online zugänglich unter:
`http://www.gnu.org/copyleft/lesser.html`

(Zuletzt abgerufen am 31.03.2006)

[19] GNU public license.

Online zugänglich unter:
`http://www.trolltech.com/licenses/qpl.html`

(Zuletzt abgerufen am 31.03.2006)

[20] Various licenses and comments about them.

Online zugänglich unter:
`http://www.gnu.org/licenses/license-list.html#GPLIncompatibleLicenses`

(Zuletzt abgerufen am 31.03.2006)

[21] Gold, C., Ledoux, H. & Dzieszko, M. (2005). A data structure for the construction and navigation of 3D Voronoi and Delaunay cell complexes. Poster presented at the WSCG 2005, January 31-February 4, 2005, Plzen, Czech Republic.

Online zugänglich unter:
`http://wscg.zcu.cz/wscg2005/Papers_2005/Poster/C97-full.pdf`

(Zuletzt abgerufen am 31.03.2006)

[22] Guibas, L. & Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics, 4(2)*, 74-123.

[23] Har-Peled, S. (1997). On the expected complexity of random convex hulls. *Technical Report, 330, Tel-Aviv University.* Tel Aviv: School of Mathematical Sciences.

[24] Klein, R. (1997). *Algorithmische Geometrie.* Bonn: Addison Wesley.

[25] LEDA Homepage.

Online zugänglich unter:
`http://www.algorithmic-solutions.com`

(Zuletzt abgerufen am 31.03.2006)

[26] Ledoux, H. & Gold, C. M. (2004). Modelling oceanographic data with the three-dimensional Voronoi diagram. In: *Proceedings of ISPRS 2004 - XXth Congress, Vol. II,* (pp. 703-708). Istanbul, Turkey: ISPRS.

[27] Liu, Y. & Snoeyink, J. (submitted). A comparison of five implementations of 3D Delaunay tessellation. *Combinatorial and Computational Geometry, MSRI series.*

Online zugänglich unter:
`http://wwwx.cs.unc.edu/~mceuen/twiki/pub/TModeling/DelaunayTessellation/MSRIDel3d-liusnoe.pdf`

(Zuletzt abgerufen am 31.03.2006)

[28] Meijering, J. L. (1953). Interface area, edge length, and number of vertices in crystal aggregates with random nucleation. *Philips Research Reports, 8,* 270-290.

[29] Mücke, E. P., Saias, I. & Zhu, B. (1996). Fast randomized point location without preprocessing in two-and three-dimensional Delaunay triangulations. In: *Proceedings of the 12th Annual ACM Symposium on Computational Geometry* (pp. 274-283). New York, NY: ACM Press.

[30] Poupon, A. (2004). Voronoi and Voronoi-related tessellations in studies of protein structure and interaction. *Current Opinion in Structural Biology, 14(2),* 233-241.

[31] Protein Data Bank Homepage.

Online zugänglich unter:
`http://www.wwpdb.org`

(Zuletzt abgerufen am 31.03.2006)

[32] QHull. Copyrights.

Online zugänglich unter:
`http://www.qhull.org/COPYING.txt`

(Zuletzt abgerufen am 31.03.2006)

[33] Renka, R. J. (1997). Algorithm 772: STRIPACK, Delaunay Triangulation and Voronoi Diagram on the Surface of a Sphere. *ACM Trans. Math. Software, Vol. 23, No. 3*, 416-434.

[34] Richter-Gebert, J. & Kortenkamp, U. H. (1999). *The Interactive Geometry Software Cinderella.* Berlin: Springer.

[35] Shah, N. R. (1994). *Topological modeling with simplicial complexes.* (Ph. D. Thesis.) Urbana, IL: University of Illinois, Department of Computer Science.

[36] Shewchuck, J. R. (1998). Tetrahedral mesh generation by Delaunay refinement. In: *Proceedings of the 14th Annual ACM Symposium on Computational Geometry* (pp. 86-95). New York, NY: ACM Press.

[37] Sun. Homepage for the Java3D tutorials. Chapter 1.

Online zugänglich unter:
`http://java.sun.com/products/java-media/3D/collateral/j3d_tutorial_ch1.pdf`

(Zuletzt abgerufen am 31.03.2006)

[38] Sun. Homepage for the Java3D tutorials. Chapter 2.

Online zugänglich unter:
`http://java.sun.com/products/java-media/3D/collateral/j3d_tutorial_ch2.pdf`

(Zuletzt abgerufen am 31.03.2006)

[39] Watson, D. F. (1981). Computing the n-dimensional Delaunay tessellation with applications to Voronoi polytopes. *The Computer Journal, 24(2)*, 167-172.