Endbericht der Projektgruppe 471

Beyond Graphics
Strömungssimulation in der GPU

Lehrstuhl für
Graphische Systeme
Fachbereich Informatik

Lehrstuhl für Angewandte
Mathematik und Numerik
Fachbereich Mathematik

Universität Dortmund

SoSe 2005, WiSe 2005/06

Teilnehmer
Daniel Bachmann
Przemyslaw Beben
Till Becker-Adam
André Braun
Andreas Ehrenberg
Christian Groß
Michael Hein
Matthias Miemczyk
Raphael Münster
Mark Senne
Mirko Sykorra
Klaus Wohlgemuth

Betreuer
Claus-Peter Alberts
Dominik Göddeke

Inhaltsverzeichnis

Kapitel 1

Einleitung
Motivation, Einordnung, Grenzen

1.1 Anforderung, Ziele, Motivation

Die Aufgabe der Projektgruppe (PG) bestand darin, eine Softwareumgebung zur numerischen Strömungssimulation zu entwickeln. Dabei sollte insbesondere die Leistungsfähigkeit moderner Grafikkarten ausgenutzt und entsprechende 3D-Gittermanipulationstechniken verwendet werden.
Strömungsvorgänge treten in der Natur in einer Vielzahl verschiedenster Situationen auf, man denke beispielsweise an meteorologische Vorgänge wie Wind- und Meeresströmungen, Luft- und Benzinströmungen in Automotoren, Turbinen und Triebwerken sowie Umströmungen zur Analyse des Luft bzw. Wasserwiderstandes komplexer Objekte wie Automobile oder Schiffe. Um diese Vorgänge qualitativ und quantitativ zu untersuchen, gibt es zwei grundsätzliche Vorgehensweisen – das Experiment und die Simulation.
Im Experiment wird ein Modell der Wirklichkeit, häufig in verändertem Maßstab, konstruiert und an diesem werden dann Messungen durchgeführt. In der Automobilindustrie platziert man beispielsweise während der Designphase ein verkleinertes Modell eines Prototyps in einem Windkanal, um seinen Luftwiderstand zu messen.
In der Simulation geht man im Gegensatz dazu abstrakter vor. Man versucht mit Hilfe der Mathematik ein Modell zu bilden und dieses mit Computern auswerten zu lassen. Ein so genannter Simulationszyklus besteht aus vier wesentlichen Schritten:

1. **Modellbildung**: Ein physikalisches Phänomen wird durch eine Reihe von mathematischen Gleichungen beschrieben. Für die Strömungssimulation sind dies die sogenannten Navier-Stokes-Gleichungen [53], ein System partieller Differentialgleichungen. Eine „Lösungsformel" für diese Gleichungen existiert nicht, daher ist man auf numerische Verfahren angewiesen, die eine Näherungslösung iterativ berechnen.

2. **Diskretisierung**: Das Modell wird diskretisiert, d.h. in eine für den Computer verarbeitbare Form gebracht. Dies geschieht beispielsweise mit der Methode der *Finiten Elemente*

(FEM) oder mittels so genannter *Finiter Differenzen* (FD)[54]. Man erhält nach einiger mathematischer Arbeit große (lineare) Gleichungssysteme.

3. **Berechnung**: Die Lösung dieser Gleichungssysteme liefert große Mengen an diskreten Rohdaten.

4. **Aufbereitung**: Mit Hilfe geeigneter Visualisierungstechniken werden die Rohdaten in eine für den Anwender interpretierbare Form gebracht.

Die wissenschaftliche Disziplin des CFD (*computational fluid dynamics*) [28, 63] beschäftigt sich mit der Vorhersage von Strömungsvorgängen durch eben diese Simulationen. CFD ist ein interdisziplinäres Forschungsgebiet, es beinhaltet Aspekte aus so verschiedenen Bereichen wie den klassischen Naturwissenschaften, der angewandten Mathematik, der Informatik und den Ingenieurwissenschaften.
Als Fallbeispiel soll das Szenario des virtuellen Windkanals dienen, bei dem ein dreidimensionales, komplexes Objekt, gegeben durch eine Beschreibung seiner Oberfläche, in einem Kanal fixiert und von einer Seite des Kanals eine Strömung eingeleitet wird. Als Beschreibungsform des Objektes wurde von der PG die in der Computergrafik gängige Form der Oberflächentriangulierung verwendet [39].
Mit Hilfe von FD wird das Differenzvolumen zwischen Objekt und Kanal (also der Bereich, in dem beim Experiment tatsächlich Strömungsvorgänge auftreten würden) in viele kleine hexaederförmige Elemente zerlegt. All diese Elemente überdecken zusammen das Simulationsgebiet, ihre Gesamtheit wird Rechengitter genannt. Dieses Rechengitter bildet zunächst eine rein geometrische Diskretisierung des Strömungsgebiets. Ein gutes Rechengitter muss optimal an die konkrete Situation angepasst sein, es müssen beispielsweise alle relevanten Details der Geometrie aufgelöst werden. So sieht man in Abbildung 1.1 die hochaufgelöste Geometrie einer Kugel und die Approximation der Kugel durch die Seitenflächen der anliegenden Hexaeder.

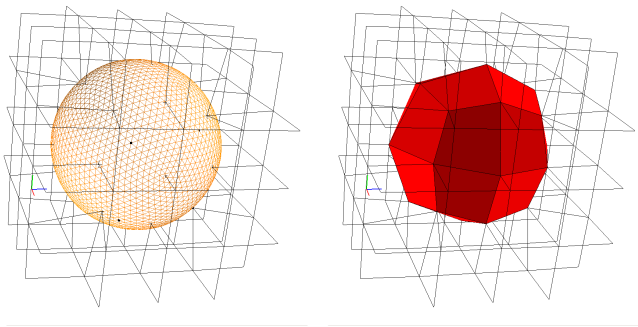


Abbildung 1.1: *Links: Hochaufgelöste Oberflächentriangulierung einer Kugel. Rechts: Durch 24 Hexaederseitenflächen approximierte Oberfläche der Kugel.*

Die Hauptanforderung der PG bestand darin, eine Software zur Erzeugung und Deformation geeigneter Volumengitter zu entwickeln. Insbesondere die Qualität der Gitterelemente (Hexaeder) stellte eine enorme Herausforderung dar. So war es z.B. wichtig, dass sich einzelne Gitterzellen nicht gegenseitig durchdringen.

Die Projektgruppe verfolgte weiterhin einen aktuellen Trend, der in der Evaluierung der Rechenkraft der Grafikkarte für allgemeine Anwendungen außerhalb des Renderings (*general purpose computation using graphics hardware*, [17]) besteht. Diese können als *streaming processor* aufgefasst werden und in Hochsprachen komfortabel programmiert werden. Erste Veröffentlichungen, wie [5, 10, 27], übertragen einfache numerische Lösungsverfahren in eine GPU-Implementierung und versprechen im Vergleich zu einer reinen CPU-Implementierung eine Beschleunigung um einen Faktor 5 bis 15.

1.2 Einordnung des Projektes

Eine erste Sichtung der zur Verfügung stehenden Software zur Gittergenerierung zeigte folgende Schwachpunkte: Die Programme beschränkten sich auf zweidimensionale Problemstellungen, waren nicht frei verfügbar oder waren für die Anforderungen der PG nicht geeignet [52]. Die in der PG erstellte Gittersoftware *InGrid3D* erlaubt es, ein für die Strömungssimulation taugliches Gitter zu erzeugen, welches eine vorgegebene Geometrie möglichst gut approximiert und ein allgemeintaugliches Ergebnisgitter zur Verfügung stellt. Dieses Gitter könnte nun an eine CFD-Software wie z.B. FEATFLOW [11] weitergegeben werden. Um im Kontext der PG die Praxistauglichkeit der erzeugten Gitter zu demonstrieren, wurde ein Löser auf GPU-Basis implementiert. Da der Löser nur einen Konzeptnachweis (engl. *proof of concept*) darstellen sollte, beschränkte sich die PG bei der Umsetzung auf vollständig reguläre Gitter [46]. Die implementierten Gitterunterteilungsalgorithmen sind aber auch problemlos auf unregelmäßige Gitter anwendbar, ebenfalls wäre es möglich, durch einige Modifikationen des Programms eine komplette Hierarchie für ein geometrisches Mehrgitterverfahren [14] zu erzeugen und in entsprechende Löser zu exportieren.
Ursprünglich sollte die GPU in allen Bereichen zur Beschleunigung eingesetzt werden. Es stellte sich jedoch heraus, dass die zur Gitterdeformation umgesetzten Techniken aufgrund mangelnder Parallelisierbarkeit nicht GPU-tauglich waren (vgl. Kap. 4).

1.3 Vorgehensweise der Projektgruppe

Als Basis für die Berechnung dient ein Modell des zu umströmenden Objektes und ein an dieses Objekt angepasstes Grobgitter. Das Grobgitter, das anfänglich aus wenigen Hexaederelementen besteht, wird wiederholt unterteilt. Die dabei neu entstehenden Randpunkte müssen an die Oberfläche des Objektes angepasst werden, um eine bessere Approximation zu erhalten. Die Beschreibung des Objektes wird über eine Oberflächentriangulierung vorgegeben, auf dieser ist die Bestimmung der neuen Randpunkte nicht trivial. Nachdem dieses Problem bereits von Göddeke mit Hilfe geometrischer Projektionstechniken untersucht wurde [14], entschied sich die PG dazu, einen alternativen Ansatz auf Basis einer Parametrisierung zu entwickeln. Dabei wird die Oberflächentriangulierung sukzessive vergröbert, ohne dass seine Struktur verloren geht. Während der Unterteilung dient dieses grobe Netz als Parameterraum für die Randanpassung. Das so berechnete Gitter muss häufig noch nachbearbeitet werden, da die einzelnen Hexaderzellen unter Umständen von ungünstiger Qualität sein könnten. Das Resultat ist ein optimal an das Objekt angepasstes Hexaedergitter. Auf diesem Gitter wird mittels eines GPU-basierten Lösungsverfahrens eine Strömungssimulation durchgeführt, um prototypisch die Nutzbarkeit dieser Gitter zu demonstrieren.

1.4 Struktur des Berichtes

Nach der allgemeinen Einleitung, der genauen Aufgabenstellung und der Einordnung des Berichtes stellt Kapitel 2 zunächst die benötigten theoretischen Grundlagen bereit:

- Kapitel 2.1 bietet eine Einführung in die benötigten mathematischen Grundlagen der numerischen Strömungssimulation. Dabei wird insbesondere das zugrundeliegende mathematische Modell *„potential flow"* auf der Basis der Poisson-Gleichung erklärt.
- In Kapitel 2.2 wird gezielt die Methode *„Multiresolution Adaptive Parametrization of Surfaces"* (MAPS) beleuchtet, welche in der Implementierung von *InGrid3D* zum Einsatz kommt.
- Kapitel 2.3 beschreibt anschließend die Anforderungen an ein Gitter zur numerischen Strömungssimulation, hierbei wird vor allem auf die verschiedenen Netztypen, die Qualitätskriterien und die verschiedenen Glättungsmethoden eingegangen.
- Kapitel 2.4 führt in Gitterkorrekturmethoden zur Behebung von Selbstdurchdringungen und Elementinvertierungen (engl. *mesh untangling*) ein.
- Anschließend erläutert Kapitel 2.5 die theoretischen Grundlagen von *aktiven Konturen*, die allgemein auch als *„snakes"* bezeichnet werden. Diese dienen als Hilfe bei der Berechnung eines kürzesten geodätischen Weges zwischen zwei Punkten.
- Kapitel 2.6 bietet eine kompakte Übersicht zum Themenkomplex *„general-purpose computation using graphics hardware"* (GPGPU).

Das darauf folgende Kapitel 3 ergänzt das Grundlagenkapitel um eine kurze Übersicht der verwendeten Werkzeuge und Bibliotheken. Im Anschluss folgt ein kurzer Bericht über die geleisteten Vorarbeiten der PG in Kapitel 4. Insbesondere Konzepte, die untersucht, im Laufe des Projektes aber als untauglich bzw. ungeeignet befunden wurden, werden hier erwähnt.
Nachfolgend stellt Kapitel 5 das von der Projektgruppe entwickelte System vor, und zwar zunächst das allgemeine Konzept und danach die konkrete Arbeitsweise der verschiedenen Module und Teilsysteme. Eine objektive Einschätzung der Resultate bietet Kapitel 6, wobei zunächst die einzelnen Teilsysteme und Module betrachtet werden, und anschließend an Hand eines komplexen Testfalles das Gesamtsystem evaluiert wird.
Kapitel 7 befasst sich mit dem zeitlichen Ablauf des Projektes und bietet einen kurzen Überblick der erstellten Prototypen.
Der Anhang gliedert sich in drei Bereiche: Im Benutzerhandbuch, Anhang A, wird die Bedienung des Systems in Form von Anleitungen erklärt. Das Pflichtenheft, Anhang B, beinhaltet eine detaillierte Beschreibung der in der PG erstellten Software. Die Programmierschnittstelle, Anhang C, des Systems wird in diesem Bericht in Kurzform auf Klassenebene und auf Methodenebene dokumentiert. Der Bericht schließt mit dem Abdruck der verwendeten Lizenz und einem umfassenden Literaturverzeichnis.
Eine kurze Projektvorstellung und der Quellcode finden sich auf der PG-Homepage[1].

[1] http://ls7-www.cs.uni-dortmund.de/students/projectgroups/pg471.shtml

Kapitel 2

Grundlagen

Grundbegriffe und benötigtes Hintergrundwissen

2.1 Gundlagen des CFD

Hier sollen zunächst mathematische Grundlagen im Bereich der numerischen Strömungssimulation aufgezeigt werden. Die Darstellungen gehen zum überwiegenden Teil auf Göddeke [14] zurück.

2.1.1 Notationen

Es werden offene Gebiete $\Omega \subseteq \mathbb{R}^d$ betrachtet, für $d = 1, 2, 3$ mit dem Rand $\partial\Omega$. Ortspunkte $\mathbf{x} \in \Omega$ werden mit $\mathbf{x} = (x_1, \ldots, x_d)^T$, Zeitpunkte mit $t \in I \subseteq \mathbb{R}$ bezeichnet. Für d-dimensionale Vektoren $\mathbf{a}, \mathbf{b}$ wird das übliche euklidische Skalarprodukt mit $\mathbf{a} \cdot \mathbf{b}$, und die euklidische Norm mit $||\mathbf{a}||$ bezeichnet:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &:= \sum_{i=1}^{d} a_i b_i \\ ||\mathbf{a}|| &:= \sqrt{\mathbf{a} \cdot \mathbf{a}} \end{aligned}$$

Funktionen treten entweder skalarwertig oder vektorwertig auf:

$$u = u(\mathbf{x}),\ u = u(\mathbf{x}, t), \mathbf{u} = \begin{pmatrix} u_1(\mathbf{x}, t) \\ \vdots \\ u_d(\mathbf{x}, t) \end{pmatrix}$$

Partielle Ableitungen, d.h. Ableitungen nach einer der vorkommenden Variablen, werden wie folgt geschrieben:

$$\begin{aligned} \partial_t u &:= \frac{\partial u}{\partial t} \\ \partial_i u &:= \frac{\partial u}{\partial x_i} \end{aligned}$$

$\partial_i^p \mathbf{u}$ bezeichnet die p-te Ableitung nach der Variable x_i, $\mathbf{u}_{x_i}$ die Approximation der ersten Ableitung nach der Variable x_i und $\mathbf{u}_{x_i x_i}$ die Approximation der zweiten Ableitung.
Mit dem Nabla-Operator ∇ werden der Gradient (Vektor aller partiellen Ableitungen) einer skalaren sowie die Divergenz einer Vektorfunktion geschrieben als

$$\text{grad } u := \nabla u := (\partial_1 u, \ldots, \partial_d u)^T$$

$$\text{div } \mathbf{u} := \nabla \cdot \mathbf{u} := \partial_1 u_1 + \cdots + \partial_d u_d.$$

Die Kombination von Divergenz- und Gradientenoperator ergibt den **Laplace-Operator**:

$$\Delta u := \nabla \cdot (\nabla u) = \partial_1^2 u + \cdots + \partial_d^2 u.$$

2.1.2 Modellierung

Ziel der Modellierung ist es, für Naturvorgänge eine – oft stark vereinfachte – mathematische Beschreibung zu finden, d.h. Vorgänge wie Diffusion, Stofftransport oder Temperaturverteilung mit Hilfe mathematischer Gleichungen zu beschreiben. Dies ist ein höchst kreativer Akt, der viel Erfahrung erfordert und für den es kein Standardkochrezept gibt. An dieser Stelle wird exemplarisch die Herleitung einer partiellen Differentialgleichung (PDE) für die Auslenkung einer Membran dargestellt. Es wird in Form einer partiellen Differentialgleichung (PDE) beschrieben und basiert auf dem Lehrbuch von Grossman und Roos [18].

Die Auslenkung einer Membran ist ein zweidimensionaler Anwendungsfall für das so genannte **Poissonproblem**. Die Membran sei am Rand eingespannt und werde durch eine vertikale Kraft $f(\mathbf{x})$, $\mathbf{x} \in \Omega$, belastet. Gesucht ist die Auslenkung $u(\mathbf{x})$, mit $\mathbf{x} \in \Omega$ unter der Einspannbedingung $u(\mathbf{x}) = 0$ für $x \in \partial\Omega$. Zur Herleitung betrachtet man die Gesamtenergie $J(u)$ des Systems, die sich zusammensetzt aus Spannungsenergie $J_1(u)$ und potentieller Energie $J_2(u)$. Die Spannungsenergie ist proportional zur Oberflächenänderung und zusätzlich abhängig von einem Elastizitätsfaktor α. Eine ausführliche Herleitung ergibt:

$$\begin{aligned} J_1(u) &\approx \alpha/2 \int_\Omega ||\nabla u||^2 \, d\Omega \\ J_2(u) &= -\int_\Omega f u \, d\Omega \end{aligned}$$

$\int_\Omega f \, d\Omega$ bedeutet hierbei, dass über das gesamte Gebiet Ω integriert wird. $J_1(u)$ und $J_2(u)$ sind also die Resultate mehrdimensionaler Integration.

Eine Funktion $u = u(\mathbf{x})$ ist die gesuchte Auslenkung, falls sie das Prinzip der minimalen Energie erfüllt:

$$\begin{aligned} J(u) \leq \quad & J(v) \quad && \text{für alle zulässigen Auslenkungen } v \text{ bzw.} \\ J(u) \leq \quad & J(u+\varepsilon v) \quad && \text{für alle } \varepsilon > 0 \text{ und alle zulässigen Auslenkungen } v \end{aligned}$$

Das Auslenkungsproblem ist also auf das folgende Minimierungsproblem zurückgeführt worden:

$$\partial_\varepsilon J(u+\varepsilon v)|_{\varepsilon=0} = 0 \text{ für alle zulässigen Funktionen } v. \tag{2.1}$$

Die in Gleichung 2.1 gewählte Notation soll verdeutlichen, dass $J(u+\varepsilon v)$ gerade für $\varepsilon = 0$ sein Minimum annimmt und somit das Prinzip der minimalen Energie wie gewünscht erfüllt wird. Man erhält nach weiteren Umformungen die folgende äquivalente Integralschreibweise:

$$\alpha \int_\Omega \nabla u \nabla v d\Omega - \int_\Omega f d\Omega\, v = 0 \text{ für alle zulässigen } v.$$

Mit Hilfe der Greenschen Formel kann man nun das Integral aufteilen in ein *Gebietsintegral* und ein *Randintegral*. Da die Einspannbedingung Nullrandwerte vorschreibt, fällt das Randintegral weg. Es bleibt:

$$\int_\Omega (-\alpha \Delta u - f) v \, d\Omega = 0 \text{ für alle zulässigen } v$$

Da dies für alle Funktionen v gilt, muss der Ausdruck in der Klammer nach dem DuBois-Raymond-Lemma bereits die Nullfunktion sein. Man erhält die Gleichung

$$-\alpha \Delta u = f \quad \text{in } \Omega \text{ mit Nullrandbedingungen.} \tag{2.2}$$

Dies ist das prototypische *Poissonproblem*, allgemein werden Gleichungen diesen Typs auch als *elliptische* PDEs bezeichnet.

2.1.3 Diskretisierung

Nachdem ein mathematisches Modell in Form von Differentialgleichungen vorliegt, muss aus ihm eine Lösung gewonnen werden, die eine Vorhersage für den zugrundeliegenden Naturvorgang erlaubt. Bei einfachen Gleichungen ist es vielleicht mit viel Erfahrung, Geduld und Intuition möglich, direkt durch scharfes „Daraufschauen" eine analytische Lösung zu bestimmen. Bei komplexeren Systemen ist dies nicht mehr möglich, daher ist man auf ein numerisches Lösungsverfahren angewiesen.
Ein Beispiel ist das Differenzenverfahren, bei dem die Differentialoperatoren durch Differenzenquotienten ersetzt werden. Das Verfahren soll hier am Beispiel der Herleitung einer Diskretisierung des dreidimensionalen Laplace-Operators verdeutlicht werden:

Finite-Differenzen-Approximation des Laplace-Operators

Aus der Definition der Differenzierbarkeit von u folgt:

$$\partial_x u = \lim_{h \to 0} \frac{u(x+h,y,z) - u(x,y,z)}{h} \tag{2.3}$$

aber auch:

$$\partial_x u = \lim_{h \to 0} \frac{u(x,y,z) - u(x-h,y,z)}{h} \tag{2.4}$$

Aus (2.3) und (2.4) folgt nun:

$$\partial_x u = \lim_{h \to 0} \frac{u(x+h,y,z) - u(x-h,y,z)}{2h} \tag{2.5}$$

Dies führt zu einer besseren Diskretisierung als (2.3) oder (2.4) (vgl. [55]). Indem der Differentialquotient (2.5) durch den Differenzenquotienten ersetzt wird, erhält man eine Approximation der partiellen Ableitung von u nach x:

$$u_x = \frac{u(x+h,y,z) - u(x-h,y,z)}{2h}, \text{ für kleines } h.$$

Für die zweite partielle Ableitung nach x gilt somit:

$$u_{xx} = \frac{u(x+h,y,z) - 2u(x,y,z) + 2u(x-h,y,z)}{h^2}$$

Dies motiviert $u_{xx} + u_{yy} + u_{zz}$ als Diskretisierung des Laplace-Operators.

Zerlegung des Gebiets Ω

Das kontinuierliche Gebiet Ω wird in eine endliche Anzahl von geometrisch einfachen Teilgebieten zerlegt. Dies sind z.B. Dreiecke und Rechtecke in der Ebene bzw. Tetraeder und Hexaeder im Raum. Diese Teilgebiete werden **Zellen** genannt. Die Gesamtheit wird als **Gitter** bezeichnet. Hierbei werden die folgenden Typen von Gittern unterschieden (vgl. Abb. 2.1):

1. Uniform strukturierte (kartesische) Gitter bestehen aus achsenparallelen Zellen gleichen Typs und gleicher Größe.

2. Rechteckige strukturierte Gitter bestehen aus Rechteckzellen (Hexaederzellen), die aber in Breite und Höhe (und Tiefe) variieren können.

3. Strukturierte Gitter bestehen aus Zellen gleichen Typs und gleicher Konnektivität, die aber in Größe und Ausrichtung variabel sind. Konnektivität bezeichnet hierbei, dass alle Gitterzellen außer denen am Rand gleich viele Nachbarzellen haben.

4. Allgemein strukturierte Gitter sind Gitter, die auf irgendeine Art und Weise strukturiert sind, beispielsweise durch Polygone und Kreise.

5. Unstrukturierte Gitter verwenden verschiedene Zelltypen in einem Gitter oder haben variable Konnektivität.

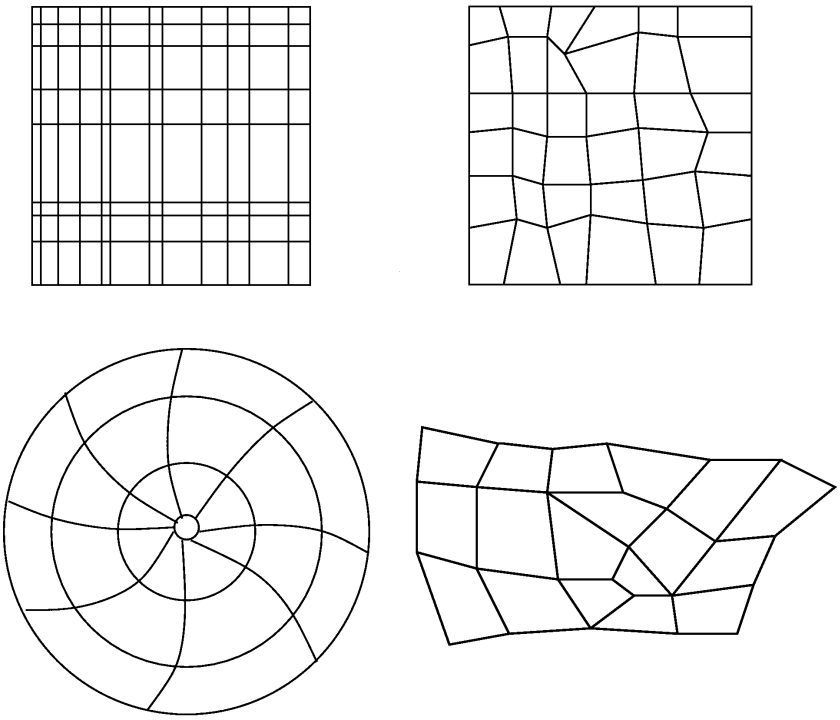


Abbildung 2.1: *Beispiele für die verschiedenen Gittertypen (von links oben nach rechts unten): Rechteckig strukturiertes Gitter, strukturiertes Gitter, allgemein strukturiertes und unstrukturiertes Gitter.*

2.1.4 Behandlung von Randbedingungen

Randbedingungen dienen der Kontrolle des Verhaltens der zu simulierenden Funktionen am Rand des Simulationsgebiets: Werden die Werte einer Variablen v auf $\partial\Omega$ explizit durch die Angabe eines Wertes gesetzt, $v(\mathbf{x}) := f(\mathbf{x})$ für $\mathbf{x} \in \partial\Omega \subseteq \Omega$, so spricht man von **Dirichlet–Randbedingungen**. Wird hingegen lediglich der Wert der Ableitung der Variablen vorgeschrieben, $\nabla v(\mathbf{x}) := f(\mathbf{x})$ für $\mathbf{x} \in \partial\Omega \subseteq \Omega$, so spricht man von **Neumann-Randbedingungen**. Bei **natürlichen Randbedingungen** (engl. *do-nothing-condition*) werden gar keine Werte gesetzt. Man beachte, dass die Randbedingungen durchaus für verschiedene Variablen innerhalb einer Simulation unabhängig gesetzt werden können. Als Beispiel kann die Behandlung der Geschwindigkeitsvariablen bei der Simulation der Strömung in einem virtuellen Windkanal mit einem zu umströmenden inneren Objekt dienen. Hier liegen zwei Randkomponenten vor, der äußere Kanal und das innere Objekt. Eine Seite des Kanals dient als Einströmrand, der Wert der Geschwindigkeit wird hier als Dirichlet-Randbedingung vorgegeben. Ein typischer Fall ist die Vorgabe eines parabolischen Einströmprofils. Für die festen Wände des Kanals und für das innere Objekt, die die Strömung nicht durchdringen darf, wird der Wert der Geschwindigkeit explizit auf Null gesetzt. Der Kanal wird idealisiert als unendlich lang angenommen, d.h. die Strömung kann am Ausströmrand den Kanal verlassen und andererseits nie über diesen Rand wieder einströmen. Für den Ausströmrand werden also keine Randbedingungen gesetzt.

Neumann-Ränder treten beispielsweise bei der Simulation der Verformung unter Druck von mechanischen Bauteilen auf, bei denen sich die Geometrie während der Simulation verändert.
Für die Integration der Randbedingungen in das durch die Diskretisierung aufgestellte Gleichungssystem stehen drei Techniken zur Verfügung. Für eine ausführliche Beschreibung wird auf die Arbeit von Turek [61] verwiesen. Für das Gleichungssystem wird vorausgesetzt, dass bislang nur natürlichen Randbedingungen gesetzt wurden. Es bezeichne X_Ω die Menge der Komponenten des Vektors $\mathbf{x}$, die mit dem Rand assoziiert sind.

Bei der **voll expliziten** Randbehandlung werden aus der Matrix A die Zeilen und Spalten gestrichen, die zu X_Ω gehören. Die Einträge des Vektors $\mathbf{b}$ werden entsprechend modifiziert. Aus dem Vektor der Unbekannten $\mathbf{x}$ werden die Komponenten aus X_Ω ebenfalls gestrichen, weil für sie ja gerade die Dirichlet-Werte schon bekannt sind und sie somit keine Unbekannten mehr sind. Dieses Verfahren findet sich oft bei direkten Lösern, da die Dimension der Matrix reduziert wird.

Bei der **halb impliziten** Randbehandlung wird folgendermaßen vorgegangen: In der Matrix A werden lediglich die Zeilen durch die entsprechenden Zeilen der Einheitsmatrix ersetzt, die mit X_Ω assoziiert sind. Beim Aufbau des Vektors der Unbekannten und der rechten Seite werden die Dirichlet-Werte mit einbezogen. Diese Technik passt hervorragend zur Verwendung von iterativen Lösern (vgl. folgendes Kapitel), weil die Lösungsvektoren immer die Randbedingungen erfüllen.

Bei der **voll impliziten** Randbehandlung wird die Matrix A nicht modifiziert, sondern nur die Komponenten von $\mathbf{x}$ und $\mathbf{b}$ entsprechend der Zugehörigkeit zu X_Ω aktualisiert. Randbedingungen werden vor und nach jedem Iterationsschritt gesetzt. Dies kann durch herausfiltern der von den Randbedingungen beeinflussten Werte geschehen. Selbst wenn verschiedene Randbedingungen gesetzt werden, verändert sich die Matrix nie, es sind lediglich verschiedene Filterungsoperatoren nötig.

2.1.5 Lösung des Gleichungssystems

Es gibt zwei große Klassen an Verfahren zur Lösung von linearen Gleichungssystemen, die direkten und die iterativen Verfahren. Die direkten Verfahren liefern die Lösung nach einem Schritt, sind dafür aber sehr rechenaufwändig. Die iterativen Verfahren sind pro Schritt weniger rechenintensiv, benötigen aber mehrere, unter Umständen sehr viele Schritte, um die Lösung zu liefern. Es tritt jedoch das Problem auf, dass nicht alle Verfahren in allen Situationen auch konvergieren, d.h. im Grenzwert die Lösung bestimmen. Bieten diese Verfahren aber eine Möglichkeit zur Fehlerschätzung, so kann in jedem Schritt beurteilt werden, wie gut die Approximation bereits ist, was in praktischen Szenarien vollkommen ausreicht. Die Darstellung in diesem Kapitel basiert größtenteils auf den Büchern von Meister [36] und Hackbusch [20], beide stellen neben den Grundlagen und Konvergenzbeweisen auch ausführliche spezialisierte Verfahren für den Fall dünn besetzter Matrizen einer festen Struktur vor. Darüber hinaus enthält das Lehrbuch von Meister auch die Ergebnisse experimenteller Analysen des Konvergenzverhaltens iterativer Verfahren für typische Modellprobleme. Für Details wird auf diese Quellen verwiesen.

Direkte Verfahren

In der Praxis ist die Programmbibliothek UMFPACK[1] für kleine Probleme (ca. 10000 Unbekannte) hochperformant. Sie bietet hochoptimierte Implementierungen von verschiedenen Lösungsverfahren. Hauptgrund für die Effizienzsteigerung ist die Verwendung intelligenter Umordnungsstrategien, so dass die Rechnung mit möglichst wenig Transfers zwischen Cache und Speicher auskommt. Experimente zeigen, dass UMFPACK für kleine Problemgrößen auf handelsüblichen PCs fast unschlagbar schnell ist.

Iterative Verfahren

Die zweite Klasse bilden die iterativen Verfahren, die das endgültige Resultat nicht schon nach einem Schritt ausgeben, sondern erst nach mehreren. Dafür sind die Einzelschritte deutlich weniger aufwändig.

CG-Verfahren Das CG-Verfahren setzt symmetrische und positiv definite Koeffizientenmatrizen A voraus, d.h.

$$\begin{aligned} A &= A^T \\ \mathbf{x} \cdot A\mathbf{x} &> 0 \quad \forall \mathbf{x} \in \mathbb{R}^N \setminus \{\mathbf{0}\} \\ \mathbf{x} \cdot A\mathbf{x} &= 0 \quad \Leftrightarrow \mathbf{x} = \mathbf{0}. \end{aligned}$$

Der zur Auflösung des Gleichungssystems erforderliche Aufwand lässt sich durch die Verwendung einer der Aufgabe angepassten Basis $\{\mathbf{p}_j\}$ des $\mathbb{R}^N$ gezielt reduzieren. Basisvektoren mit der Eigenschaft

$$A\mathbf{p}_i \cdot \mathbf{p}_j = \delta_{ij},$$

mit dem Kronecker-Symbol δ_{ij} ($\delta_{ij} = 1$, für $i = j$ und 0 sonst) heißen konjugiert oder A-orthogonal. Stellt man die gesuchte Lösung $\mathbf{x}$ des Gleichungssystem über der Basis $\{\mathbf{p}_j\}$ dar, d.h.

$$\mathbf{x} = \sum_{j=1}^{N} \eta_j \mathbf{p}_j,$$

dann lassen sich die zugehörigen Koeffizienten η_j, für $j = 1, \ldots, N$ wegen der A-Orthogonalität von $\{\mathbf{p}_j\}$ explizit darstellen durch

$$\eta_j = \frac{\mathbf{b} \cdot \mathbf{p}_j}{A\mathbf{p}_j \cdot \mathbf{p}_j}, \quad j = 1, \ldots, N.$$

[1] Zu finden unter http://www.cise.ufl.edu/research/sparse/umfpack/.

Man erhält als Iterationsvorschrift:

$$\begin{aligned}
\mathbf{q}^{(k)} &= A\mathbf{p}^{(k)} \\
\alpha^{(k)} &= \frac{\rho^{(k)}}{\mathbf{p}^{(k)} \cdot \mathbf{q}^{(k)}} \\
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)} \\
\mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{q}^{(k)} \\
\rho^{(k+1)} &= \mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)} \\
\beta^{(k)} &= \frac{\rho^{(k+1)}}{\rho^{(k)}} \\
\mathbf{p}^{(k+1)} &= \mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{p}^{(k)}
\end{aligned} \tag{2.6}$$

Dabei ist $\mathbf{x}^{(0)}$ eine geeignete Startlösung und die Iteration wird beendet, sobald $\rho^{(k)}$ unter eine vorgegebene Schranke fällt. Für Details wird auf Hackbusch [20] und Meister [36] verweisen.

Defektkorrekturverfahren Diese Klasse von Verfahren arbeitet so, dass zum Iterationsvektor ein Korrekturwert hinzuaddiert wird, um so näher an die Lösung zu gelangen. Das zu lösende lineare Gleichungssystem wird folgendermaßen umgeschrieben:

$$\begin{aligned}
\mathbf{x} &= A^{-1}\mathbf{b} \\
&= \mathbf{x} - \mathbf{x} + A^{-1}\mathbf{b} \\
&= \mathbf{x} - A^{-1}A\mathbf{x} + A^{-1}\mathbf{b} \\
&= \mathbf{x} + A^{-1}(\mathbf{b} - A\mathbf{x})
\end{aligned}$$

In dieser Form ist dies noch ein direktes Verfahren, die Lösung steht nach einem Schritt zur Verfügung. Zur Vorbereitung der weiteren Schritte wird obige Gleichung formuliert als Iterationsverfahren:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + A^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$$

Der Ausdruck in der Klammer ist gerade der Defekt im k-ten Schritt. Die Idee bei Defektkorrekturverfahren besteht nun darin, nicht die vollständige Matrix A zu invertieren, sondern nur Teile davon. Nimmt man z.B. nur die Diagonaleinträge von A, sind zur Invertierung nur deren Kehrwerte zu bilden, und man spricht vom *Jacobi-Verfahren*. Nimmt man die untere Dreiecksmatrix hinzu, die sich durch Rückwärtseinsetzen relativ leicht invertieren lässt, erhält man das so genannte *Gauß-Seidel-Verfahren*. Allerdings konvergieren diese Verfahren so langsam, dass sie als eigenständige Löser in der Praxis nicht verwendet werden.
Ferner ist ein so genannter Relaxationsparameter $\omega < 1$ gebräuchlich, um den Defektkorrekturwert zu dämpfen und somit bessere Konvergenzraten zu erzielen. Die vollständige Iterationsvorschrift schreibt sich schließlich als:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega C^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$$

Die Matrix C wird hierbei auch als *Vorkonditionierungsmatrix* bezeichnet. Ferner ist es möglich, den Parameter ω nicht nur konstant, sondern auch in Abhängigkeit von der Iterationszahl k zu wählen.

2.1.6 Anwendung auf das Poissonproblem

Beispielhaft soll am Poissonproblem (s. Gleichung 2.2) die oben beschriebene Vorgehensweise veranschaulicht werden. Das entstehende Diskretisierungsschema wird in Kapitel 5.8 wieder aufgegriffen und als Grundlage zur Simulation eines Flusses verwendet. Sei Ω das betrachtete Gebiet (hier $\Omega \subseteq \mathbb{R}^3$). Folgende Gleichung gilt es zu erfüllen:

$$-\triangle \mathbf{u} = \mathbf{f} \text{ auf } \Omega \text{ mit } \mathbf{f} = \mathbf{0} \text{ auf } \partial\Omega$$

Dies ist die in Kapitel 2.1.2 dargestellte Poissonproblem. Die numerische Lösung dieser partiellen Differentialgleichung (durch die oben dargestellte Finite-Differenzen-Diskretisierung des Laplace-Operators) wird nun, durch den Aufbau eines linearen Gleichungssystems, vorbereitet.

Die Finite-Differenzen-Matrix

Im weiteren Verlauf wird davon ausgegangen, dass die Punkte eines kartesischen Gitters im zweidimensionalen Fall von „unten links" nach „oben rechts" bzw. im dreidimensionalen Fall schichtenweise so verfahrend durchnummeriert sind. Es soll schrittweise, beginnend mit dem Fall $\Omega \subseteq \mathbb{R}$, die so genannte *Finite-Differenzen-Matrix* aufgebaut werden. Seien $\mathbf{f}$ und $\mathbf{u}$ als Spaltenvektoren definiert, also

$$\mathbf{u} = (u_1, ..., u_{N-1})^T$$
$$\mathbf{f} = (f_1, ..., f_{N-1})^T.$$

Der Fall $\Omega \subseteq \mathbb{R}$: Es gilt:

$$\frac{-u_{j+1} - 2u_j + u_{j-1}}{h^2} = f(x_j), \tag{2.7}$$

für $j = 1, ..., N-1$ (für $j \in \{0, N\}$ befindet man sich auf dem Rand und ist fertig). Eine kompaktere Schreibweise für (2.7) ist:

$$A_{\text{fd}}\mathbf{u} = \mathbf{f}, \tag{2.8}$$

hierbei ist A_{fd} eine symmetrische $((N-1) \times (N-1))$-Matrix, die **Finite-Differenzen-Matrix** genannt wird. Gleichung 2.8 ist das lineare Gleichungssystem, dessen Lösung der numerischen Lösung des Poisson-Problems entspricht. Hier gilt:

$$A_{\text{fd}} = h^2 \cdot \text{tridiag}_{N-1}(-1, 2, -1),$$

also ist A_{fd} eine tridiagonale Matrix, auf deren Hauptdiagonalen jeder Eintrag $2h^2$ ist. Auf den beiden Nebendiagonalen steht stets $-h^2$, alle restlichen Einträge der Matrix für den zweidimensionalen Fall sind 0; h ist hierbei der Abstand der einzelnen Gitterzellen in x-Richtung (vgl. Abb. 2.2).

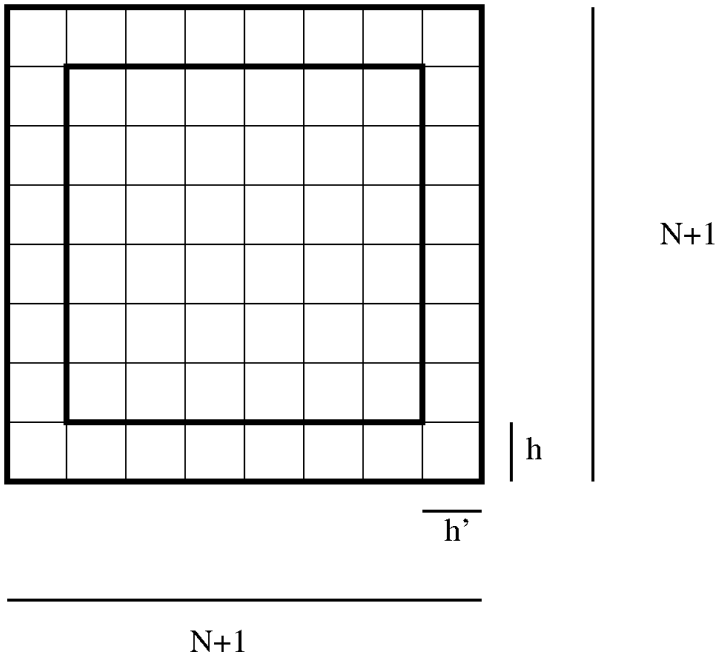


Abbildung 2.2: *Beispiel eines zweidimensionalen strukturierten Gitters.*

Der Fall $\Omega \subseteq \mathbb{R}^2$: Aus Gründen der Übersichtlichkeit wird im weiteren Verlauf angenommen, dass die Abstände innerhalb des Gitters in x- sowie in y-Richtung identisch ($= h$) sind. A_{fd} ist eine symmetrische $((N-1)^2 \times (N-1)^2)$-Matrix. Für die Zeile j gilt:

$$(a_{\text{fd}})_{i,j} = h^2 \cdot \begin{cases} 4 & \text{falls } j = i \\ -1 & \text{falls } j = i-N+1, i-1, i+1, i+N-1 \\ 0 & \text{sonst} \end{cases}$$

Es handelt sich also um eine Matrix, die nur auf fünf Bändern Einträge besitzt, die ungleich 0 sind. Eine solche Matrix wird auch **pentadiagonal** genannt.

Der Fall $\Omega \subseteq \mathbb{R}^3$: Wie schon im zweidimensionalen Fall, werden hier äquidistante Abstände angenommen. Durch Hinzunahme der z-Achse entsteht eine symmetrische $((N-1)^3 \times (N-1)^3)$-Matrix. Sie hat sieben Bänder:

$$(a_{\text{fd}})_{i,j} = h^2 \cdot \begin{cases} 6 & \text{falls } j = i \\ -1 & \text{falls } j = i-(N-1)^2, i-N+1, i-1, i+1, i+N-1, i+(N-1)^2 \\ 0 & \text{sonst} \end{cases}$$

Das Lösen des entstandenen linearen Gleichungssystems mit einer der in Kapitel 2.1.5 dargestellten Verfahren führt zu einer numerischen Lösung der Poisson-Gleichung.

2.1.7 Potential Flow

Da ein ideales inkompressibles Fluid mit konstanter Dichte ohne anfängliche Verwirbelungen auch im weiteren Verlauf keine Verwirbelungen generieren wird, ist es mathematisch sinnvoll, ein Fluid unter der folgenden Bedingung zu betrachten. Zu jedem Zeitpunkt sei

$$\nabla \times \mathbf{u} = \mathbf{0}, \tag{2.9}$$

hiebei ist $\nabla \times \mathbf{u}$ die so genannte **Rotation** des Vektorfeldes $\mathbf{u}$. Sei $\mathbf{u} = (u_1, u_2, u_3)^T$, dann ist $\nabla \times \mathbf{u}$ definiert durch

$$\nabla \times \mathbf{u} = \begin{pmatrix} \partial_2 u_3 - \partial_3 u_2 \\ \partial_3 u_1 - \partial_1 u_3 \\ \partial_1 u_2 - \partial_2 u_1 \end{pmatrix}$$

Ein Vektorfeld, für das Gleichung 2.9 gilt, wird als **wirbelfrei** bezeichnet. Solche Strömungen werden *irrotational flow* genannt und können als Gradient eines skalaren Potentials aufgefasst werden:

$$\mathbf{u} = -\nabla\phi,$$

wobei ϕ Geschwindigkeitspotential genannt wird. Die Inkompressibilitätsbedingung $\nabla \cdot \mathbf{u} = \mathbf{0}$ impliziert, dass

$$\triangle\phi = \mathbf{0}.$$

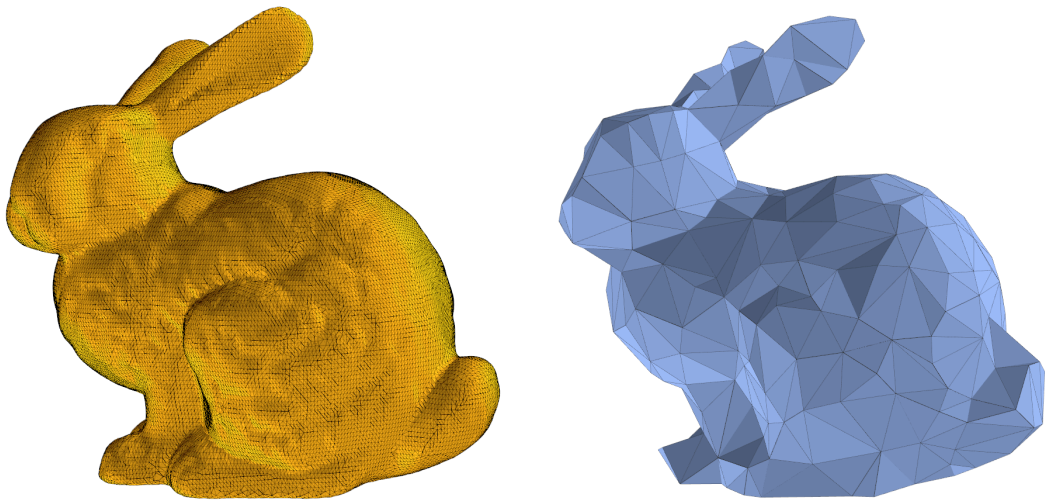
Dies ist wieder die so genannte Laplacegleichung.
Wenn eine Strömung also wirbelfrei ist, wird das Geschwindigkeitsvektorfeld $\mathbf{u}$ durch den Gradienten eines unbekannten Potentials beschrieben, d.h. $\mathbf{u} = -\nabla\phi$. Bei konstanter Dichte sagt das Massenerhaltungsgesetz aus, dass die Divergenz des Vektorfelds gleich $\mathbf{0}$ ist. Also muss folgende Gleichung gelöst werden:

$$\nabla \times (\nabla\phi) = \triangle\phi = \mathbf{0}.$$

2.2 MAPS: Multiresolution Adaptive Parameterization of Surfaces

Bei der Anpassung des Gitters an das Netz des Objektes ist es nicht trivial zu entscheiden, auf welchem Punkt der Oberfläche des Objektes neu entstandene Gitterknoten ihre Position erhalten. Besonders Einbuchtungen in das Objekt stellen schwierige Situationen dar. Eine einfache Projektion des neuen Punktes auf die Oberfläche des Netzes kann zu schlechten Ergebnissen [14] führen. Aus diesem Grund hat sich die PG dafür entschieden, initial eine Parametrisierung der Oberfläche zu berechnen, um mit deren Hilfe erst im folgenden Schritt die Randanpassung des Gitters an die Oberfläche vorzunehmen. Die PG entschied sich für das MAPS-Verfahren [30]. Es soll im Folgenden in seiner Funktionsweise erklärt werden.
MAPS ist eine Methode, um eine glatte (d.h. hier differenzierbare) Parametrisierung einer *Mannigfaltigkeitstriangulierung* über einem *Basisnetz* zu erhalten. Dieses *Basisnetz* kann als Parameterraum angesehen werden. In den Vereinfachungsschritten wird das Dreiecksnetz als Graph angesehen und bestimmte Knoten dieses Graphen werden entfernt, bis die Vereinfachungsprozedur terminiert und das *Basisnetz* bereitstellt (s. Abb. 2.3). Das gesamte Verfahren

hat eine Zeit- und Speicherkomplexität von $O(N \log N)$, wobei N die Anzahl der Knoten des Netzes der feinsten Auflösungsstufe, also des Netzes ist, mit dem begonnen wurde [30]. Während die Vereinfachungsschritte durchgeführt werden erhält man eine Hierarchie von $O(\log N)$ Stufen des Netzes. Die Basis für die Auswahl der zu entfernenden Knoten bildet der Dobkin-Kirkpatrick Algorithmus [9], der eine Hierarchie von $O(\log N)$ Vergröberungsstufen garantiert.



(a) *Mannigfaltigkeitstriangulierung* (b) *Basisnetz*

Abbildung 2.3: *Mannigfaltigkeitstriangulierung und berechnetes Basisnetz für das so genannte „Stanford Bunny" [56].*

2.2.1 Definitionen und Notationen

Es werden zunächst die grundlegenden Begriffe, die zum Aufbau einer Parametrisierung nach dem MAPS-Verfahren benötigt werden, eingeführt. Die Oberfläche des gegebenen Objektes, die parametrisiert werden soll, liegt in Form einer *Mannigfaltigkeitstriangulierung* vor. Eine **Triangulierung** einer Punktmenge sei definiert als Menge planarer Dreiecke, deren Eckpunkte gleich der Punktmenge sind. Die Seiten der Dreiecke werden als Kanten bezeichnet. Ferner bilden die Kanten graphentheoretisch betrachtet eine Zusammenhangskomponente. Ein **Dreieck** mit den Eckpunkten a, b, c wird durch $T = \{a, b, c\}$ beschrieben. Eine **Kante** zwischen a und b wird durch $e = \{a, b\}$ beschrieben. Die Eckpunkte a, b, c eines Dreiecks bzw. die Endpunkte a, b einer Kante werden **Knoten** genannt. Wenn vom geometrischen Ort gesprochen wird, so ist das Analogon zu einem Knoten ein **Punkt** $\mathbf{x} \in \mathbb{R}^d$ und zu einer Kante ein **Vektor** $\mathbf{e} \in \mathbb{R}^d$, $d \in \{2, 3\}$.
Eine *geschlossene Triangulierung* ist eine **Mannigfaltigkeitstriangulierung** wenn gilt: Jede Kante ist zu genau zwei Dreiecken inzident [62].
Eine **Parametrisierung** ist in Bezug auf das MAPS-Verfahren als eine bijektive Abbildung definiert, die Punkte des *Basisnetzes* auf die Punkte der ursprünglichen Oberfläche abbildet. Die zu parametrisierende Oberfläche liegt in Form eines so genannten *Dreiecksnetzes* vor. Ein

Dreiecksnetz wird als Paar (P, K) repräsentiert, hierbei ist P eine Menge von N Punkten des $\mathbb{R}^3$, mit $\mathbf{p}_i = (x_i, y_i, z_i)^T, 1 \leq i \leq N$. K ist die Menge aller Knoten, Kanten und Dreiecke.
Zwei **Knoten** a und b sind **benachbart**, wenn $\{a, b\} \in K$ gilt. Eine Menge von Knoten ist **unabhängig**, wenn die Menge aller Knoten kein benachbartes Knotenpaar enthält. Eine solche Menge ist **maximal unabhängig** wenn sie in keiner größeren unabhängigen Menge enthalten ist.
Die **Eins-Ring-Nachbarschaft** eines Knotens a ist die Menge $N(a) = \{b|\{a, b\} \in K\}$, also die Menge der Knoten b, so dass es eine Kante zwischen a und b gibt. Der **Stern** eines Knotens a, $stern(a)$, ist die Menge der Kanten und Dreiecke, die a enthalten. Während des Aufbaus der Parametrisierung wird es im weiteren Verlauf notwendig sein ein *Polygon* zu triangulieren. Seien $v_0, v_1, \ldots, v_{n-1}$ n Knoten, $\{v_0, v_1\}, \ldots, \{v_i, v_{i+1}\}, \ldots, \{v_{n-1}, v_0\}$ n Kanten, die diese Knoten miteinander verbinden. Diese bilden ein **Polygon** genau dann, wenn die Schnittmenge zweier Kanten nur aus dem Knoten, den sie beide teilen, besteht und nichtinzidente Kanten sich nicht schneiden.
Ein weiterer wichtiger Begriff in diesem Zusammenhang ist der des *Ohres*: Drei im Polygon aufeinander folgende Knoten a, b, c eines Polygons P bilden ein **Ohr**, wenn die Kante $\{a, c\}$ eine *Diagonale* in P bildet. Hierbei bildet eine Kante, $\{a, c\}$ eine **Diagonale**, wenn für alle Kanten e von P, die keinen der Knoten a, c enthalten, gilt: $\{a, c\} \cap e = \emptyset$ und $\{a, c\}$ innerhalb von P liegt.

2.2.2 Aufbau der Hierarchiestufen

Im MAPS-Algorithmus wird eine Hierarchie von Netzen aufgebaut. Das ursprüngliche Netz $(P, K) = (P^0, K^0), \ldots, (P^L, K^L)$ wird sukzessiv durch einen Knotenentfernungsschritt vereinfacht und es ergibt sich eine Folge von Netzen (P^l, K^l) mit $0 \leq l \leq L$, hierbei wird (P^0, K^0) als das **Basisnetz** bezeichnet, es ist das gröbste Netz in der Hierarchie. Die in MAPS verwendete Vereinfachungsprozedur ist angelehnt an das Verfahren von Dobkin und Kirkpatrick [9] und garantiert, dass die Anzahl der Hierarchiestufen L von der Größenordnung O($\log N$) ist. Zusammenfassend kann man sagen, dass eine Netzvereinfachung durch eine Knotenentfernung gefolgt von einer Retriangulierung erfolgt. Diese ist notwendig, da durch die Knotenentfernung sonst Löcher entstehen würden. Die Strategie zur Knotenentfernung wird im Folgenden genauer beschrieben.

2.2.3 Knotenentfernung

Das Vorgehen im MAPS-Algorithmus sieht wie folgt aus: Im Schritt von $P^l \mapsto P^{l-1}$ wird eine maximal unabhängige Menge von Knoten mit einer niedrigen Anzahl ausgehender Kanten entfernt. Zu Beginn des Algorithmus ist keiner der Knoten als nicht entfernbar markiert und die Menge der Knoten, die entfernt werden sollen, ist demzufolge leer. Dann werden zufällig nicht-markierte Knoten k mit $|N(k)| < 12$ ausgewählt und der Stern $stern(k)$ aus P^l entfernt. Die Nachbarn von k werden als nicht-entfernbar gekennzeichnet. Dieses Verfahren wird fortgesetzt bis keine Knoten mehr entfernt werden können. In Netzen mit einem durchschnittlichen Knotengrad von 6 sind weniger als die Hälfte der Knoten vom Grad 12 oder höher [9]. Dobkin und Kirkpatrick zeigten, dass in diesen Netzen zumindest 1/24 der Knoten auf jeder Stufe entfernt wird. Somit ergeben sich die zu Beginn dieses Abschnitts erwähnten O($\log N$) Hierarchiestufen.

Das zuvor beschriebene Vorgehen wird nun noch leicht abgeändert, indem die zufällige Auswahl durch eine Klassifizierung der Knoten in einer Prioritätswarteschlange Q ersetzt wird, wobei die Priorität von geometrischer Information des Knotens abhängt: Entscheidend für die Priorität einer Eins-Ring-Nachbarschaft in der Geometrie sind ihre Oberfläche und Krümmung. Das bedeutet: Je spitzer eine Eins-Ring-Nachbarschaft zuläuft und je grösser ihre Oberfläche ist, desto niedriger sollte die Priorität sein diese zu entfernen. Die Oberfläche einer Eins-Ring-Nachbarschaft ist offensichtlich die Summe der Flächen der Dreiecke, aus denen sie besteht. Die *diskrete Krümmung* wird nach Kim und Levin [23] berechnet aus der *Gausskrümmung* K und der *mittleren Krümmung* H:

$$K = \frac{2\pi - \sum_{i=1}^{n} \alpha_i}{\frac{1}{3}A}$$

und

$$H = \frac{\frac{1}{4}\sum_{i=1}^{n} ||\mathbf{e}_i||\beta_i}{\frac{1}{3}A}$$

Die Bedeutung der Bezeichner kann Abbildung 2.4 entnommen werden, A bezeichnet den Flächeninhalt der Eins-Ring-Nachbarschaft. Abschließend kann die *diskrete Krümmung* K_d berechnet werden:

$$K_\mathrm{d} = \begin{cases} 2|H| & \text{falls } K \geq 0, \\ 2\sqrt{H^2 - K} & \text{sonst} \end{cases}$$

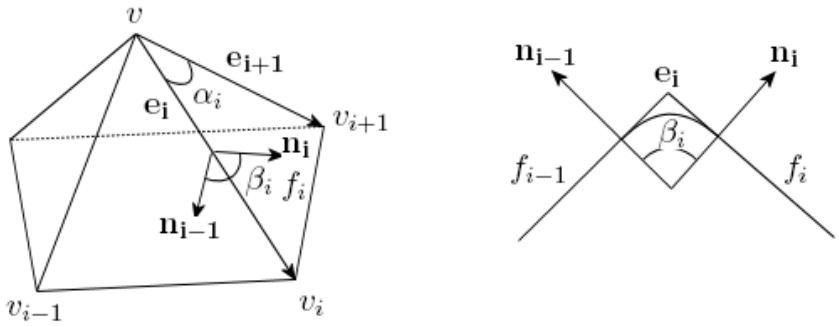


Abbildung 2.4: *Zur Berechnung von $K(i)$ verwendete Notation. $\mathbf{e}_i$ bezeichnet die Kante $\{v_i, v_j\}$, f_i bezeichnet das Dreieck $\{v, v_i, v_{i+1}\}$, α_i ist der Winkel zwischen zwei aufeinander folgenden Kanten $\mathbf{e}_i$ und $\mathbf{e}_{i+1}$ und β_i ist der Winkel an einer Kante $\mathbf{e}_i$, d.h. der Winkel zwischen den Normalen der zu $\mathbf{e}_i$ adjazenten Dreiecke (die Abbildung rechts ist ein Blick von der Seite). Die Zeichnung wurde adaptiert von [23].*

Zusammenfassend berechnet man die Priorität eines Knotens v wie folgt:

$$w = \lambda \frac{a(v)}{\max_{p \in stern(v)} a(p)} + (1 - \lambda) \frac{K(v)}{\max_{p \in stern(v)} K(p)}$$

Der Wert λ ist eine Konstante, mit deren Hilfe gewichtet werden kann, in welchem Maß das Verhältnis von Flächeninhalt und Krümmung zueinander stehen. In MAPS wurde $\lambda = 1/2$

vorgeschlagen. Durch Sortieren der Knoten nach berechneter Priorität erhält man eine Komplexität von $O(N \log N)$.

2.2.4 Projektion und Retriangulierung

Um K^{l-1} aus K^l zu erhalten muss nach der Entfernung der unabhängigen Menge retrianguliert werden. Die Eins-Ring-Nachbarschaft $stern(k)$ eines entfernten Knotens k wird durch die so genannte *konforme*, also winkeltreue, Abbildung z^a in die Ebene projiziert. Hierzu nummeriert man zyklisch alle n Knoten der Eins-Ring-Nachbarschaft $N(k) = \{k_m | 1 \leq m \leq n\}$ so, dass $\{k_{m-1}, k, k_m\} \in K^l$ mit $k_0 = k_n$. Wir verwenden eine stückweise lineare Approximation von z^a, im weiteren Verlauf mit μ_k bezeichnet. Die Abbildung μ_k ist definiert durch die Werte am Mittelpunkt und am Rand von $stern(k)$ also: $\mu_k(\mathbf{p}_k) = \mathbf{0}$ und $\mu_k(\mathbf{p}_{k_m}) = r_m \exp(i\theta_m a)$, wobei

$$r_m = ||\mathbf{p}_k - \mathbf{p}_{k_m}|| \text{ und } \theta_m = \sum_{o=1}^{m} \angle(\mathbf{p}_{k_{o-1}}, \mathbf{p}_k, \mathbf{p}_{k_o}); i = \sqrt{-1}$$

mit $a = 2\pi/\theta_n$ gilt, $\mathbf{p}_k$ ist der zum Knoten k assoziierte Punkt. Analoges gilt für die übrigen Punkte.

Das Polygon, das nach Projektion der Eins-Ring-Nachbarschaft von k in der Ebene entsteht, wird nach Entfernen des inneren Knotens k neu trianguliert. Hierzu wird ein Verfahren, welches mit dem so genannten *Ohrenabschneiden* arbeitet, angewendet [45]. Laut *Meisters Theorem* gibt es eine Diagonale, die ein Ohr vom übrigen Polygon separiert. Das Ohr ist bereits korrekt trianguliert. Somit kann es von der weiteren Betrachtung ausgeschlossen werden. Es wird abgeschnitten und mit dem verbleibenden Polygon rekursiv fortgefahren.

2.2.5 Parametrisierung

Die angestrebte Parametrisierung ist eine Abbildung von einem Dreieck der gröbsten Auflösungsstufe auf die zu parametrisierende Oberfläche. Die Vorgehensweise ist adaptiv, d.h. während die oben angeführten Schritte zur Vergröberung des Netzes für eine Vergröberungsstufe durchgeführt werden (s. Abb. 2.5), werden die entfernten Punkte parametrisiert.

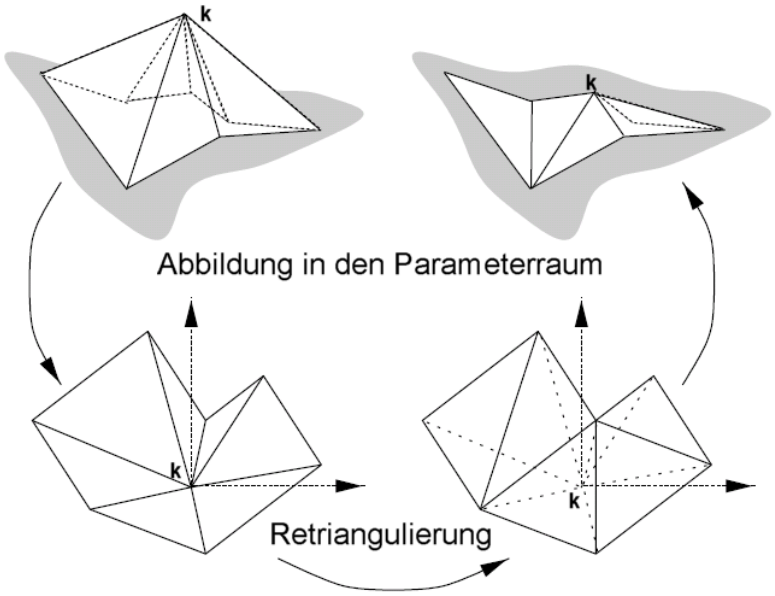


Abbildung 2.5: *Um einen Knoten k zu entfernen, wird der zugehörige Stern stern(k) vom dreidimensionalen Raum durch die konforme Abbildung in die Ebene abgebildet. In der Ebene wird der innere Knoten des Sterns entfernt und das entstandene Polygon neu trianguliert [30]. Die Beschriftungen wurden aus dem Englischen übersetzt.*

Hierzu wird eine Abbildung $\mathbf{\Pi} : K^L \rightarrow K^0$ während der Vereinfachung aufgebaut. Die gesuchte Parametrisierung ist die Umkehrabbildung:

$$\mathbf{\Pi}^{-1} : K^0 \rightarrow K^L$$

Der zu einem Knoten k assoziierte Punkt $\mathbf{p}_k \in K^L$ wird durch $\mathbf{\Pi}$ auf einen Punkt $\mathbf{p}_k^0 \in K^0$ abgebildet. Dieser Punkt befindet sich auf dem Netz der momentan gröbsten Auflösungsstufe, dem **Grobnetz**. Es existiert ein Dreieck auf dem Grobnetz, etwa $T = \{a, b, c\}$, in dem dieser Punkt liegt. Also kann dieser Punkt in baryzentrischen Koordinaten bzgl. dieses Dreiecks dargestellt werden:

$$\mathbf{p}_k^0 = \alpha \mathbf{p}_a + \beta \mathbf{p}_b + \gamma \mathbf{p}_c,$$

wobei $\alpha + \beta + \gamma = 1; \alpha, \beta, \gamma \geq 0; \{a, b, c\} \in K^0$.

Bei der Berechnung der Parametrisierung wird auf Hierarchiestufe l eine stückweise lineare Bijektion

$$\mathbf{\Pi}^l : K^L \rightarrow K^l$$

konstruiert. Beginnend mit $l = L$ erhält man $\mathbf{\Pi}^L = id$ und endet schließlich mit der gesuchten Abbildung $\mathbf{\Pi}^0 = \mathbf{\Pi}$. Hierbei wird $\mathbf{\Pi}^l$ nur an den Knoten in K^L berechnet. Die Werte für alle weiteren Knoten ergeben sich aus der stückweisen Linearität.
Um Π^l aus Π^{l-1} zu berechnen, werden folgende Fälle unterschieden:

1. $k \in K^{l-1}$: Der Knoten wird beim Übergang von Hierarchiestufe l zu $(l-1)$ nicht entfernt. Setze also $\mathbf{\Pi}^{l-1}(\mathbf{p}_k) = \mathbf{\Pi}^l(\mathbf{p}_k) = \mathbf{p}_k$.

2. $k \in K^l \backslash K^{l-1}$: Der Knoten wird beim Übergang $l \rightarrow (l-1)$ entfernt. Die Eins-Ring-Nachbarschaft von $\mathbf{p}_k$ wird in die Ebene projiziert (s.o.). Nach der Retriangulierung können die Koordinaten des zuvor entfernten Punktes $\mathbf{p}$ einem Dreieck, welches $T = \{a, b, c\} \in K^{l-1}$ entspräche, zugeordnet werden. Der Punkt kann folglich in baryzentrischen Koordinaten bezüglich dieses Dreiecks geschrieben werden:

$$\alpha\mu_k(\mathbf{p}_a) + \beta\mu_k(\mathbf{p}_b) + \gamma\mu_k(\mathbf{p}_c).$$

In diesem Fall setze: $\mathbf{\Pi}^{l-1}(\mathbf{p}_k) = \alpha\mathbf{p}_a + \beta\mathbf{p}_b + \gamma\mathbf{p}_c$. Die baryzentrischen Koordinaten werden für die Einbettung im Zweidimensionalen berechnet und dann auf die Punkte des Netzes der feinsten Auflösungsstufe im Dreidimensionalen angewendet.

3. $k \in K^L \backslash K^l$: Der Knoten wurde schon beim Übergang von Π^{l-1} zu Π^l entfernt. Es gilt also:

$$\mathbf{\Pi}^l(\mathbf{p}_k) = \alpha'\mathbf{p}_{a'} + \beta'\mathbf{p}_{b'} + \gamma'\mathbf{p}_{c'}$$

für ein Dreieck $T' = \{a', b', c'\} \in K^l$. Für $T' \in K^{l-1}$ muss keine weitere Unterscheidung getroffen werden, andernfalls gilt: Da die zu entfernenden Knoten in einer Vergröberungsstufe eine maximal unabhängige Menge (s.o.) bilden, ist sichergestellt, dass beim Übergang von l auf $(l-1)$ genau ein Knoten dieses Dreiecks entfernt wurde. Dies sei o.B.d.A. k'. Sei $\mu_{k'}$ die zugehörige Einbettung im Zweidimensionalen (s. Abb. 2.6). Nach der Retriangulierung liegt $\mu_{k'}$ in einem Dreieck $T = \{h, i, j\} \in K^{l-1}$ mit baryzentrischen Koordinaten (α, β, γ) (s. Abb. 2.6). Setze in diesem Fall: $\mathbf{\Pi}^{l-1}(\mathbf{p}_k) = \alpha\mathbf{p}_h + \beta\mathbf{p}_i + \gamma\mathbf{p}_j$.

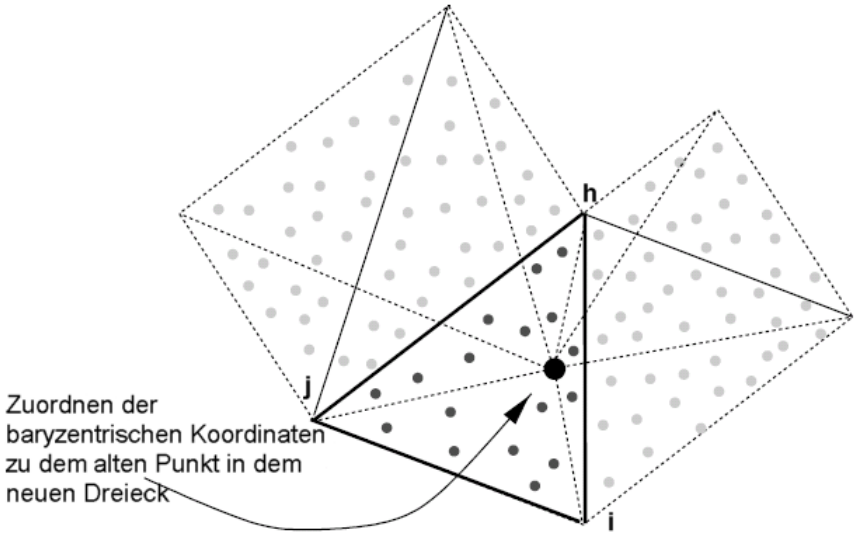


Abbildung 2.6: *Nach Retriangulierung in der Ebene (s. Abb. 2.5) werden dem soeben entfernten Knoten baryzentrische Koordinaten bzgl. des ihn enthaltenen Dreiecks in der gröberen Auflösungsstufe zugeordnet. Analog müssen nun auch allen Knoten der feinsten Auflösungsstufe, die auf ein Dreieck des entstandenen Loches abgebildet wurden, einem der neu entstandenen Dreiecke der gröberen Auflösungsstufe zugeordnet werden [30]. Die Beschriftungen wurden aus dem Englischen übersetzt.*

Diese Fallunterscheidung ist vollständig: Angenommen, es gäbe einen Knoten d, der bei obiger Fallunterscheidung nicht berücksichtigt würde. Dann würde gelten:

- $d \notin K^{l-1}$, da sonst Fall 1 greift,
- $d \notin K^l \backslash K^{l-1}$, da sonst Fall 2 greift und
- $d \notin K^L \backslash K^l$, da sonst Fall 3 greift.

Es folgt: $d \notin K^{l-1} \cup (K^l \backslash K^{l-1}) \cup (K^L \backslash K^l) = K^L$, d ist also kein Knoten des Netzes der feinsten Auflösungsstufe und muss somit nicht betrachtet werden.

Verfahren zur Berechnung der Koordinaten eines beliebigen Punktes auf der Oberfläche

Im weiteren Verlauf werden Dreiecke des Basisnetzes als **Grobdreiecke** und Dreiecke des Feinnetzes als **Feindreiecke** bezeichnet.
Gegeben sei ein Punkt $\mathbf{p}_k \in K^0$ auf dem Netz der gröbsten Auflösungsstufe. Zunächst muss mit Hilfe eines Such-Algorithmus zur Punkt-Lokalisierung das Dreieck T auf dem Feinnetz gefunden werden, das $\mathbf{p}_a$ enthält. Dies ist äquivalent dazu, das Dreieck des Feinnetzes zu finden, welches $\mathbf{k}$ nach Projektion des Feinnetzes auf das Grobnetz enthält. Der Suchalgorithmus geht wie folgt vor: Alle auf ein Grobdreieck projizierten Dreiecke werden durchlaufen und es wird getestet, ob es unter ihnen eines gibt, so dass der projizierte Punkt von allen Kanten dieses Dreiecks „links" liegt. Zur Parametrisierung des Punktes unterscheiden wir vier Fälle.

1. T liegt in einem Grobdreieck.
2. Zwei Knoten aus T liegen in einem Dreieck, der dritte in einem benachbarten Dreieck.
3. Die Knoten liegen innerhalb eines Dreiecksfächers des Grobnetzes und keiner der ersten beiden Fälle trifft zu.
4. Keiner der obigen Fälle trifft zu.

Im ersten Fall werden die baryzentrischen Koordinaten des Punktes $\mathbf{p}_k$ mit Hilfe von T berechnet.
Im zweiten Fall werden die benachbarten Dreiecke überprüft und das Dreieck gefunden, das den letzten Dreieckseckpunkt enthält. Dieses angrenzende Grobdreieck wird mit dem Dreieck in dem sich die anderen beiden Punkte befinden in die Ebene projiziert.
Im dritten Fall wird die Eins-Ring-Nachbarschaft gesucht, zu der die Grobdreiecke gehören, in denen sich die Punkte des projizierten Feinnetzes befinden. Diese Eins-Ring-Nachbarschaft wird mit der oben eingeführten konformen Abbildung in die Ebene projiziert.
Im vierten Fall muss das Netz lokal vergröbert werden. Danach folgt ein weiteres Prüfen der obigen Fälle.
Auf jeder der $O(\log N)$ Hierarchiestufen werden alle Knoten des Netzes durchlaufen. Die Laufzeit beläuft sich somit auf $O(N \log N)$.

2.3 Gitter

Nach der Darstellung der unterschiedlichen Gittertypen in Kapitel 2.1.3 folgt eine Übersicht über die verschiedenen Qualitätskriterien, denen Gitter in der computergestützten Simulation genügen müssen. Anschließend werden verschiedene Methoden vorgestellt, um die Qualität eines Gitters zu erhöhen. Zunächst soll jedoch die Gitterunterteilung motiviert werden. Das Entfernen von sog. Selbstdurchdringungen wird auf Grund seiner Komplexität in einem eigenständigen Kapitel betrachtet.

2.3.1 Gitterunterteilung

Ausgehend von einem Gitter mit möglichst wenigen Zellen, dem so genannten Grobgitter, soll durch wiederholtes, reguläres Unterteilen einer jeden Gitterzelle sukzessive eine bessere Auflösung bzw. Approximation des zu umströmenden Objektes gewährleistet werden. Dazu müssen die neu entstehenden Gitterzellen gegebenenfalls an die Geometrie angepasst werden, ansonsten würde sich zwar die Anzahl der Gitterzellen erhöhen[2], nicht aber die Auflösung der Geometrie.
Die Anpassung der Elemente erfolgt durch die Bewegung der neu entstandenen Randpunkte auf die Geometrie, dies führt dazu, dass einige Gitterelemente stark verzerrt werden oder im schlimmsten Falle sogar ungültige Elemente entstehen. Die zur Beurteilung der Qualität der Gitterzellen notwendigen Kriterien werden im folgenden Kapitel definiert, ebenso wird erläutert, an Hand welcher Merkmale eine Gitterzelle als ungültig zu betrachten ist.

2.3.2 Qualitätskriterien

Die Qualität eines Gitters hängt im Wesentlichen von der Qualität der einzelnen Hexaederzellen ab und kann durch verschiedene Eigenschaften bestimmt werden. Die folgenden Abbildungen und Erläuterungen wurden im Wesentlichen aus den Arbeiten von Kelly [22] und Göddeke [14] übernommen. Messbare Kriterien sind beispielsweise:

- Längenverhältnis (engl. *aspect ratio*),
- Winkelabweichung (engl. *angle deviation*),
- Innenwinkel (engl. *corner angle*),
- Jacobi-Verhältnis (engl. *jacobian ratio / deviation*) und
- Elementinvertierungen und Zelldurchdringungen.

Alle diese Kriterien sind Funktionen der Elementgeometrie, d.h. sie verknüpfen Punktkoordinaten, Kanten und Flächen eines Hexaeders zu einem Maß für die Elementqualität. Die Schwellwerte, ab denen Elemente nicht mehr benutzt werden sollten, sind stark von der verwendeten Simulationssoftware abhängig. In einer Implementierung sollte ihre Festlegung also einem erfahrenen Anwender überlassen werden bzw. eine variable Einstellmöglichkeit für die Schwellwerte vorhanden sein.
Im Gegensatz zum zweidimensionalen Fall lassen sich Längenverhältnis, Parallelabweichung

[2] Aus der Unterteilung einer Hexaederzelle entstehen beispielsweise acht neue Hexaeder.

und Verzerrung nicht direkt berechnen bzw. die direkte Berechnung liefert unzulässige Vereinfachungen. Für diese Qualitätsmaße wird daher indirekt vorgegangen: Alle sechs Seitenflächen sowie die drei Querschnittsflächen (s. Abb. 2.7) werden wie Vierecke im Dreidimensionalen behandelt. Das Qualitätsmaß für das Element ist dann der schlechteste Wert der einzelnen Testflächen. Auch wenn die hier abgebildeten Flächen planar sind, kann von dieser Eigenschaft im Allgemeinen nicht ausgegangen werden. Die weiter unten vorgestellten Algorithmen basieren dabei teilweise auf planaren Approximationen, welche durch Projektion in die Ebene erhalten werden können. Dies ist im Übrigen auch der Grund, warum die sechs Seitenflächen nicht ausreichen, um die Qualität eines Hexaederelements zu beurteilen.

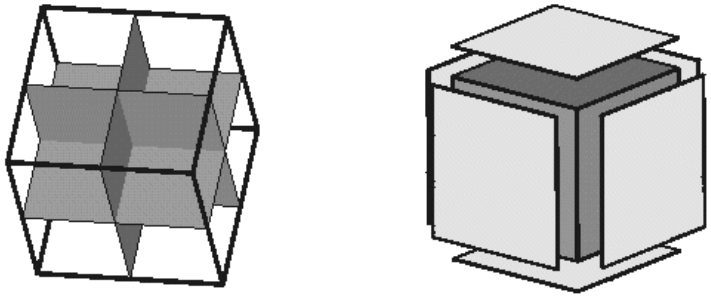


Abbildung 2.7: *Querschnittsflächen und Seitenflächen eines Hexaeders.*

Längenverhältnis

Längenverhältnisse zählen zu den bekanntesten Qualitätskriterien. Sie sind ein Indikator für die zu erwartende numerische Stabilität während der Simulationsberechnung. Hohe Längenverhältnisse können zu starken Fehlern führen, sofern keine geeigneten Stabilisierungsverfahren eingesetzt werden.

Abbildung 2.8: *Beispiele für Längenverhältnisse von 1 (1:1) und 20 (1:20).*

Im Zweidimensionalen bezeichnet das Längenverhältnis das Verhältnis von kürzester zu längster Kante eines Viereckselements. Ein Analogon im Dreidimensionalen, nämlich das Verhältnis größter zu kleinster Seitenfläche, ist vorstellbar, erfordert jedoch bei nichtplanaren Seitenflächen einen zu hohen Berechnungsaufwand zur Approximation dieser so genannten Regel-

flächen oder Minimalflächen. Stattdessen wird eine Approximation der neun oben genannten Testflächen vorgenommen und jeweils das Seitenverhältnis eines solchen 3D-Viereckselements berechnet. Als Längenverhältnis des Hexaeders wird dann der schlechteste Wert genommen. Ein ideales Element hat ein Längenverhältnis von 1, alle Kanten sind gleich lang, das Element ist würfelförmig (s. Abb. 2.8).

Winkelabweichung

Die Winkelabweichung misst, wie stark die Innenwinkel zwischen jeweils zwei inzidenten Kanten in einem Hexaeder oder einem Viereckselement von einem rechten Winkel abweichen (s. Abb. 2.9). Ihre Berechnung mittels Skalarprodukt ist trivial. Als Winkelabweichung wird das Maximum aller so berechneten Werte bezeichnet. In einigen Simulationspaketen ist dieser Wert wichtig, wenn beispielsweise für die Berechnung von Ableitungswerten möglichst rechteckige Zellen vorausgesetzt werden.

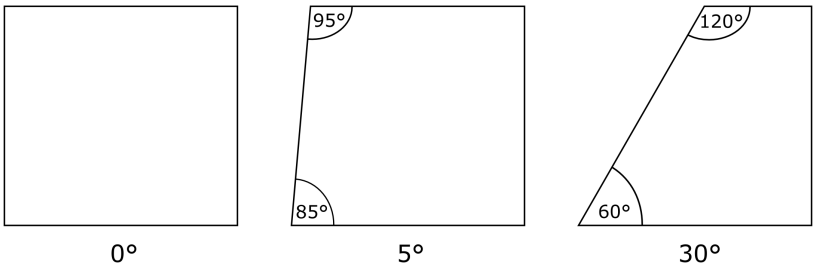


Abbildung 2.9: *Beispiele für Winkelabweichung von 0°, 5° und 30°.*

Innenwinkel

Die Berechnung minimaler und maximaler Innenwinkel wird für alle Paare inzidenter Kanten eines Hexaeders oder Viereckselements durchgeführt. Mit diesem Maß können nicht tolerierbar starke Elementverformungen schnell gefunden werden. Beispielsweise hat ein zu einem Dreieck degradiertes Viereckselement einen maximalen Innenwinkel von 180° (s. Abb. 2.10) .

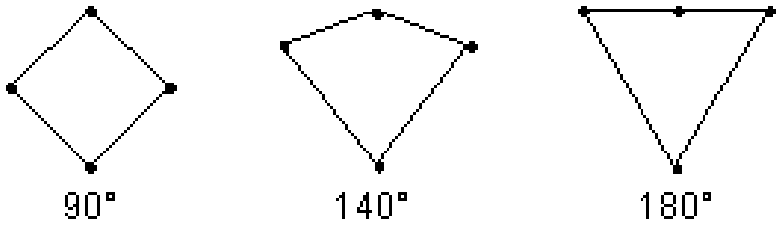


Abbildung 2.10: *Beispiele für den maximalen Innenwinkel von 90°, 140° und 180°.*

Jacobi-Verhältnis

Dieses Maß ist für viele numerische Simulationsprogramme und auch in der Visualisierung von fundamentaler Bedeutung, und zwar immer dann, wenn die verwendeten Algorithmen eine Koordinatentransformation zwischen einem Tensorproduktgitter (*computational space*) und einem verzerrten, randangepasstem Gitter (*physical space*) beinhalten. Man spricht von einer Transformation auf Referenzelemente. Als Jacobi-Verhältnis wird das Verhältnis von größter zu kleinster Determinante der Jacobi-Matrizen in allen Eckknoten des Hexaederelements bezeichnet. Die Determinante der Jacobi-Matrix eines Eckknotens beschreibt das durch die inzidenten Kanten aufgespannte Volumen eines Tetraeders (vgl. Kap. 2.4.1). Ein Jacobi-Verhältnis nahe 1 impliziert somit die Singularität der Matrix, sehr hohe Werte bedeuten, dass die Transformation unzuverlässig wird (s. Abb. 2.11). In einem idealen Element gibt es keinen Vorzeichenwechsel in den getesteten Knoten, und die Werte sind alle ähnlich groß. Deshalb wird häufig nicht nur von Jacobi-Verhältnis, sondern auch von *Jacobi-Abweichung* gesprochen.

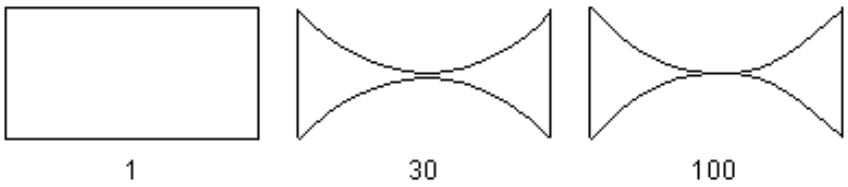


Abbildung 2.11: *Beispiele für die Jacobi-Verhältnisse 1, 30 und 100.*

Für Hexaeder wird das Jacobi-Verhältnis aus allen Eckknoten berechnet: In jedem Knoten wird die Jacobi-Matrix aufgestellt, dabei ist sicherzustellen, dass die drei Vektoren immer gleich gerichtet sind (sie zeigen alle vom Testknoten weg) und immer in der gleichen Reihenfolge als Spaltenvektoren in die Matrix eingetragen werden.

Elementinvertierungen und Zelldurchdringungen

Bei der Verfeinerung und Unterteilung des Grobgitters können unter Umständen gerade bei der Randanpassung Zellen entstehen, die sich gegenseitig durchdringen oder die invertiert, also quasi „nach außen gestülpt" sind (s. Abb. 2.12). In diesem Fall würde die Transformation auf das Referenzelement zu Fehlern in der Simulation führen, die letztendlich das gesamte Simulationsergebnis unbrauchbar machen würden. Die Erkennung solcher Invertierungen oder Durchdringungen erfolgt durch die Berechnung der zuvor erwähnten Jacobi-Determinanten. Möglichkeiten zur Behandlung dieses Problems werden in Abschnitt 2.4 vorgestellt.

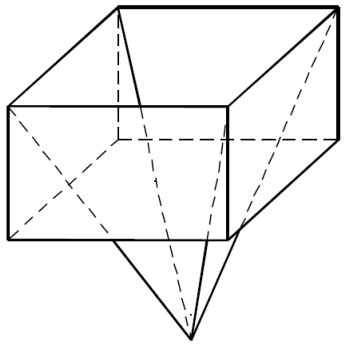


Abbildung 2.12: *Eine invertierte Hexaederzelle.*

2.3.3 Glättungsmethoden

Gitterglättung bezeichnet den Vorgang, ein existierendes Gitter allein durch (lokale) Koordinatenverschiebung von Knoten so zu verändern, dass die einzelnen Gitterzellen im Mittel eine höhere Qualität bzgl. verschiedener Qualitätskriterien (s. Kap. 2.3.2) erreichen. In diesem Abschnitt werden verschiedene Glättungsverfahren vorgestellt.

Laplace-Glättung

Der Laplace-Glätter durchläuft alle Knoten des Gitters und berechnet für einen Knoten $\mathbf{p}$ seine neuen Koordinaten $\mathbf{p}_{\text{neu}}$ aus den umliegenden Knoten wie folgt:

$$\mathbf{p}_{\text{neu}} := (1-\alpha)\cdot\mathbf{p} + \frac{\alpha}{|Adj(\mathbf{p})|}\sum_{\mathbf{q}_j\in Adj(\mathbf{p})}\mathbf{q}_j \tag{2.10}$$

Dabei ist $\alpha \in [0,1]$ ein Gewichtungsfaktor, der die lineare Interpolation zwischen alter und neuer Position steuert: $\alpha = 0.5$ ist ein guter Ausgangswert. Die Menge $Adj(\mathbf{p})$ enthält für jeden Knoten $\mathbf{p}$ die Knoten, die mit $\mathbf{p}$ durch eine Kante verbunden sind.
Da die Koordinaten des neuen Knotens $\mathbf{p}_{\text{neu}}$ das Mittel der umliegenden Knoten $\mathbf{q}_i \in Adj(\mathbf{p})$ sind, kann es an nicht konvexen Rändern zur Entstehung von invertierten Elementen kommen. Ein Lösungsansatz für dieses Problem ist die *smart-Laplace* Technik. Ein Knoten $\mathbf{p}$ wird nur dann verschoben, wenn dies zu einer Verbesserung der Qualitätskriterien führt.

Umbrella-Glättung

In seiner Diplomarbeit hat Göddeke [14] die Eignung des *Umbrella-Operators* [25] zur Glättung von Finite-Elemente-Gittern experimentell nachgewiesen. Für die Herleitung des Operators sei auf [14] verwiesen. Es wird hier nur die diskrete Variante skizziert.
Der Umbrella-Glätter durchläuft alle Knoten des Gitters und berechnet für einen Knoten $\mathbf{p}$ seine neuen Koordinaten $\mathbf{p}_{\text{neu}}$ aus den umliegenden Gitterknoten und deren Nachbarn wie folgt:

$$\mathbf{p}_{\text{neu}} = (1-\alpha)\cdot\mathbf{p} + \frac{\alpha}{\sum\limits_{1\le j\le|Adj(\mathbf{p})|} d_j}\sum_{\mathbf{q}_j\in Adj(\mathbf{p})} d_j\mathbf{q}_j, \tag{2.11}$$

wobei $\alpha \in [0,1]$ ein Gewichtungsfaktor ist, die Menge $Adj(\mathbf{p})$ für jeden Knoten $\mathbf{p}$ dessen Nachbarn $\mathbf{q}_j$ enthält und d_j die Länge der Kante $\{\mathbf{p}, \mathbf{q}_j\}$ bezeichnet .

Optimierungsbasiertes Glätten

Ein weiteres Verfahren zur lokalen Gitterglättung wird von Freitag [13] vorgestellt. Das so genannte optimierungsbasierte Glätten versucht die Gitterpositionen $\mathbf{x}_j$ zu finden, die ein Qualitätsmaß optimiert (s. Kap. 2.4). Die zu minimierende Zielfunktion ist in diesem Fall:

$$f(\mathbf{x}) = \min_{1 \leq i \leq n} q_i(\mathbf{x}) \tag{2.12}$$

Hierbei ist $q_i(\mathbf{x})$ ein Qualitätsmaß des Elementes i wie z.B. der minimale Winkel. Es wird über alle n Elemente minimiert. Typischerweise ist q_i eine nichtlineare, glatte und stetig differenzierbare Funktion. Da die partiellen Ableitungen von $f(\mathbf{x})$ nicht stetig sind, führt dies zu einem nicht glatten Optimierungsproblem, das beispielsweise mit der Methode des steilsten Abstiegs (engl. *steepest descent method*) gelöst werden kann [13].

2.4 Entfernen von Selbstdurchdringungen

2.4.1 Allgemeine Formulierung des Optimierungsproblems

Das Entfernen von Selbstdurchdringungen durch lineare Optimierung (engl. *optimization based mesh untangling*) ist ein Verfahren, das es ermöglicht, invertierte Elemente in verschiedenen Netzen (z.B. Dreiecks-, Vierecks- oder Hexaedernetze) wieder in gültige Elemente zu überführen. Betrachtet man also einen Knoten eines invertierten (ungültigen) Elements und seine Nachbarschaft, so wird das Ziel angestrebt, den Knoten an eine Stelle zu verschieben, so dass nur noch gültige Elemente in der Nachbarschaft (dem lokalen Teilnetz) existieren. Die Menge der Knotenpositionen, die gültige Elemente ergibt, wird Gültigkeitsbereich (engl. *feasible region*) genannt.
Zur Entfernung von Selbstdurchdringungen durch lineare Optimierung wird die Strategie verfolgt, ein Qualitätsmaß für Netze zu wählen und dieses in einer lokalen Umgebung zu optimieren. Für Dreiecksnetze wird als Qualitätsmaß der Flächeninhalt eines Dreiecks gewählt. Dieser Flächeninhalt soll über alle Dreiecke durch Optimierung maximiert und so die Netzqualität verbessert werden. Es wird eine lineare Gleichung zur Berechnung des Flächeninhalts aufgestellt (s. Kap. 2.4.1) und als *Zielfunktion* an ein *lineares Programm* (LP) übergeben. Die Zielfunktion wird optimiert und das Ergebnis der Optimierung ist eine neue Knotenposition. Ein Qualitätsmaß muss gewisse Anforderungen erfüllen, damit es sich zum Entfernen von Durchdringungen eignet. Es sollte zwischen gültigen und ungültigen Elementen unterscheiden können und weiterhin sollte bei der Optimierung stets das globale Extremum gefunden werden. Für das Qualitätsmaß „minimaler Flächeninhalt" ist dies möglich. Hierzu wird zunächst der Begriff der *Level Sets* definiert: Für eine Funktion $f : D \longrightarrow \mathbb{R}^m, D \subset \mathbb{R}^n$ und ein festes $\mu \in \mathbb{R}^m$ ist das **Level Set** von f auf dem so genannten Level μ, definiert als:

$$L_f(\mu) := \{\mathbf{x} | f(\mathbf{x}) = \mu\}$$

Die Konvexität des *Level Sets* spielt eine große Rolle. Falls die *Level Sets* der ausgewählten Funktion konvex sind und man ein Minimum findet, so ist dieses Minimum global. Für Qualitätsmaße wie z.B den minimalen Winkel im lokalen Teilnetz oder den Sinus dieses Winkels

sind die *Level Sets* nicht konvex, wenn der freie, d.h. der zu verschiebende Knoten $\mathbf{p}$, nicht im Gültigkeitsbereich liegt. Also sind diese Maße nicht geeignet für das Entfernung von Selbstdurchdringungen durch lineare Optimierung.
Das Qualitätsmaß, das in der PG verwendet wird, ist die minimale Fläche über alle Elemente im lokalen Teilnetz. Diese Fläche wird maximiert. Es kann gezeigt werden, dass diese Funktion konvexe *Level Sets* hat im Falle, dass der freie Knoten innerhalb oder außerhalb des Gültigkeitsbereichs liegt [13].

Allgemeiner Ansatz

Es wird ein allgemeiner Aufruf für eine Prozedur, die Durchdringungen entfernt vorgestellt, ein solcher Aufruf hätte die Form:

$$\mathbf{p}_{\text{new}} = \text{Untangle}(\mathbf{p}, Adj(\mathbf{p}))$$

Hierbei bezeichnet $Adj(\mathbf{p})$ die Knoten $\mathbf{u}$, so dass eine Kante $\{\mathbf{p}, \mathbf{u}\}$ existiert und $\mathbf{p}$ ist der Knoten, dessen Position verbessert werden soll. Es wird der minimale Flächeninhalt eines Dreiecks im lokalen Teilnetz maximiert und die Position des freien Knotens dementsprechend angepasst. Die Funktion f, die den Flächeninhalt des Dreiecks angibt, lässt sich allgemein schreiben als:

$$f(\mathbf{p}) = \min A_i(\mathbf{p}), \text{ für } 1 \leq i \leq n.$$

Hierbei ist n die Anzahl der Dreiecke im lokalen Teilnetz und A_i der Flächeninhalt des i-ten Dreiecks im lokalen Teilnetz.

Flächeninhalt und Jacobi-Determinante

Betrachtet man im Zweidimensionalen ein Dreieck mit den Punkten $\mathbf{p} = (p_x, p_y)^T$, $\mathbf{u} = (u_x, u_y)^T$ und $\mathbf{v} = (v_x, v_y)^T$, so besteht die Jacobi-Matrix aus den Spaltenvektoren $\mathbf{e}_1 = \mathbf{u} - \mathbf{p}$ und $\mathbf{e}_2 = \mathbf{v} - \mathbf{p}, J = [\mathbf{e}_1, \mathbf{e}_2]$. Im Dreidimensionalen wird die Matrix analog gebildet. Die Determinante dieser Matrix gibt nun den Flächeninhalt eines Elements an, es gilt also:

$$A = \frac{1}{2}\det(J) = a_x p_x + a_y p_y + c, \text{ wobei } a_x = u_y - v_y, a_y = v_x - u_x, c = u_x v_y - v_x u_y \quad (2.13)$$

Im Dreidimensionalen gilt analog für ein Tetraeder mit freiem Knoten $\mathbf{p} = (p_x, p_y, p_z)^T$ und seine Nachbarn $\mathbf{u} = (u_x, u_y, u_z)^T$, $\mathbf{v} = (v_x, v_y, v_z)^T$ und $\mathbf{w} = (w_x, w_y, w_z)^T$:

$$A = \frac{1}{6}\det(\mathbf{u} - \mathbf{p}, \mathbf{v} - \mathbf{p}, \mathbf{w} - \mathbf{p}) = a_x p_x + a_y p_y + a_z p_z + c, \quad (2.14)$$

hierbei ist

$$\begin{gathered} a_x = -\det\begin{bmatrix} 1 & 1 & 1 \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix}, \quad a_y = -\det\begin{bmatrix} u_x & v_x & w_x \\ 1 & 1 & 1 \\ u_z & v_z & w_z \end{bmatrix}, \\ a_z = -\det\begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ 1 & 1 & 1 \end{bmatrix}, \quad c = -\det\begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \end{gathered} \quad (2.15)$$

Für das Entfernen von Durchdringungen ist folgender Zusammenhang von Bedeutung: Ein Netzelement ist gültig, wenn die Jacobi-Determinante größer Null ist [24]. Durch Berechnen der Jacobi-Determinante kann folglich entschieden werden, ob ein Element gültig oder ungültig ist. Für die Optimierung ist wichtig, dass sich der Flächeninhalt für Dreiecke (s. Gleichung 2.13) und für Tetraeder (s. Gleichung 2.14) als lineare Funktion darstellen lässt und daher als Zielfunktion eines linearen Programms verwendet werden kann.

Grundlagen der linearen Programmierung

Ein **lineares Programm** (LP) in Standardform besteht aus k reellwertigen *Variablen* $x_1, ..., x_k$, einer *Zielfunktion* $z(x_1, ..., x_k)$ und $m + k$ linearen *Nebenbedingungen*. Für $i = 1, ..., m$ und $j = 1, ..., k$ seien c_j, b_i und n_{ij} reelle Zahlen. Gesucht ist eine Belegung der Variablen, so dass die Zielfunktion

$$z(x_1, ..., x_k) = \sum_{j=1}^{k} c_j x_j$$

optimiert wird unter den Nebenbedingungen

$$\sum_{j=1}^{k} n_{ij} x_j \leq b_i \quad \text{für } i = 1, ..., m,$$
$$x_j \geq 0 \quad \text{für } j = 1, ..., k.$$

In Matrix-Vektor-Schreibweise ist ein LP gegeben durch:

$$\text{Optimiere } z(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \text{ unter } N\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0},$$

wobei $\mathbf{x} = (x_j)$, $\mathbf{c} = (c_j)$, $\mathbf{b} = (b_i)$ und $N = (n_{ij})$ für $i = 1, ..., m$ und $j = 1, ..., k$ ist.

Zum Finden der Lösung von linearen Programmen wird in der PG das Programm *lpsolve* (s. Kap. 3.9) verwendet. *Lpsolve* löst ein lineares Optimierungsproblem in mehreren Phasen. So wird in einer Phase eine zulässige Lösung bestimmt und in einer darauf folgenden Phase daraus eine optimale Lösung errechnet. Für die unterschiedlichen Phasen werden unterschiedliche Formulierungen des Problems verwendet, es gibt ein *primales Problem* und ein *duales Problem*. Sei ein LP in Standardform gegeben:

$$\max z(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \text{ unter } N\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}$$

Dann gilt für ein $\mathbf{y} = (y_0, ..., y_m)^T \in \mathbb{R}^m_{\geq 0}$ mit $\mathbf{y}^T N \geq \mathbf{c}^T$ folgende Ungleichung:

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T N\mathbf{x} \leq \mathbf{y}^T\mathbf{b},$$

für $\mathbf{x}, \mathbf{y} \geq \mathbf{0}$. Also liefert jeder Vektor $\mathbf{y}^T \in \mathbb{R}^m_{\geq 0}$ mit $\mathbf{y}^T N \geq \mathbf{c}^T$ eine obere Schranke für $\mathbf{c}^T\mathbf{x}$, und zwar $\mathbf{y}^T\mathbf{b}$. Die beste (kleinste) obere Schranke, die man so konstruieren kann, ist durch folgendes LP gegeben:

$$\min \mathbf{y}^T\mathbf{b} \text{ unter } \mathbf{y}^T N \geq \mathbf{c}^T, \mathbf{y} \geq \mathbf{0}$$

Dieses LP wird *duales LP* zum obigen *primalen LP* genannt. Sei $\hat{\mathbf{x}}$ eine optimale Lösung für das obige primale LP, und sei $\hat{\mathbf{y}}$ eine optimale Lösung für das duale LP, dann dienen diese dazu eine Ungleichung wie $N\mathbf{x} \geq \mathbf{b}$ durch Subtraktion auf der rechten Seite auf Gleichungsform $N\mathbf{x} = \mathbf{b} - \mathbf{s}$ zu bringen. Für das von *lpsolve* angewandte Lösungsverfahren ist es stellenweise nötig, Ungleichungen auf diese Weise in Gleichungen zu überführen.

Aufstellen des linearen Optimierungsproblems

Es wird das Optimierungsproblem für einen freien Knoten $\mathbf{p}$, dessen Position opimiert werden soll, aufgestellt. Sei nun n die Anzahl der zu $\mathbf{p}$ inzidenten Elemente, dann ist A die $3 \times n$ Matrix deren i-te Spalte der Vektor $[\mathbf{a_{xi}}, \mathbf{a_{yi}}, 1]^T$ ist. Diese Vektoren werden wie in Gleichung 2.13 für jedes Element des lokalen Teilnetzes gebildet. Sei weiterhin $\mathbf{m}$ der (d+1)-wertige Vektor, der die Koordinaten von $\mathbf{p}$ und in der letzten Komponente die momentane Schätzung der minimierten Flächen enthält. Nun gilt also:

$$A^T\mathbf{m} = \mathbf{c} - \mathbf{s},$$

hierbei ist $\mathbf{c}$ der Vektor, der die in Gleichung 2.15 definierten c_i enthält, und $\mathbf{s}$ der Vektor der Schlupfvariablen, wobei die i-te Komponente s_i die Differenz zwischen der momentanen Schätzung des minimalen Flächeninhalts und der Fläche des Dreiecks angibt. Das ergibt das duale Problem:

$$\max \mathbf{b}^T\mathbf{m} \quad \text{unter der Bedingung } A^T\mathbf{m} + \mathbf{s} = \mathbf{c}, \mathbf{s} \geq \mathbf{0}$$

Hierbei ist $\mathbf{b}$ ein 3-wertiger Vektor, dessen erste zwei Komponenten Null sind und die letzte Komponente eins. Nach Definition ist dann das primäre Problem:

$$\min \mathbf{c}^T\mathbf{y} \quad \text{unter der Bedingung } A\mathbf{y} = \mathbf{b}, \mathbf{y} \geq \mathbf{0}$$

Das lineare Optimierungsproblem ist gelöst, wenn $\mathbf{s} \geq \mathbf{0}$, also alle Flächeninhalte mindestens so groß sind wie der minimierte Flächeninhalt und gleichzeitig die Bedingung $\mathbf{y}^T\mathbf{s} = 0$ erfüllt ist.

2.4.2 Anwendung auf Hexaedernetze

Die Entfernung von Selbstdurchdringungen durch lineare Optimierung kann nicht garantieren, dass die Netzqualität verbessert wird, laut Freitag [31] ist es sogar nicht wünschenswert, die Prozedur auf Elementen, die nicht ungültig sind, durchzuführen. Der Test, ob ein Element ungültig ist, muss an Hexaedernetze angepasst werden, denn für Rechtecksnetze oder Hexaedernetze ist es möglich, dass Elemente, für die es negative Jacobi-Determinanten gibt, nicht invertiert sind. Abbildung 2.13 zeigt beispielhaft zum Einen ein nicht-invertiertes Viereckselement mit einem negativen Jacobi-Wert am Knoten v_3 und ein nicht-invertiertes Hexaederelement mit einem negativen Jacobi-Wert am Knoten v_6 zum Anderen.

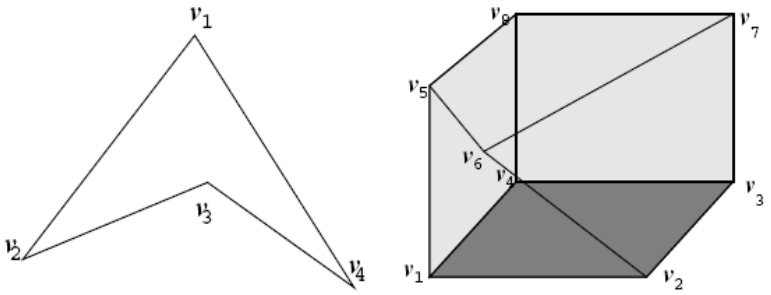


Abbildung 2.13: *Nicht invertierte Elemente mit negativen Jacobi-Werten, bei solchen Elementen genügt die Anwendung herkömmlicher Glättungsmethoden [31].*

Ein Vierecks-oder Hexaederelement ist gültig, wenn die Schnittmengen der Kanten oder Flächen Netzelemente von geringerer Dimension sind, also sollte z.B. der Schnitt von zwei Hexaederflächen entweder leer, eine Kante oder ein Knoten des Netzes sein. Bedingungen für die Gültigkeit, die leichter zu berechnen ist, liefern die folgenden Beobachtungen:

Ein Viereckselement ist gültig genau dann, wenn höchstens eine der vier Jacobi-Determinanten der Knoten negativ ist.

Dies lässt sich folgenderweise auf Hexaederelemente übertragen:

Ein Vierecks- oder Hexaederelement ist gültig genau dann, wenn es keine zwei benachbarten Knoten mit negativen Jacobi-Determinaten enthält.

Ein Beweis hierfür wurde von Li [31] geführt. Der grundlegende Ansatz der Entfernung von Durchdringungen bleibt aber auch bei der Anwendung auf Hexaedernetze bestehen: Es wird für jeden Knoten eines ungültigen Elements das minimale Volumen der Tetraeder, die der Knoten mit seinen Nachbarn bildet, maximiert.

2.5 Aktive Konturen

Das ursprüngliche Konzept der aktiven Konturen (engl. *snakes*) wurde von Kass et al. [21] bereits 1988 vorgestellt, dabei wurde eine Kontur an Hand von bestimmten Bildinformationen verformt. Der Algorithmus wurde auch „Snakes" genannt, da die Bewegung der Konturen der Bewegung von Schlangen ähnelte. Insbesondere in der medizinischen Bildverarbeitung werden Snakes zur Extraktion von relevanten Bestandteilen (Segmentierung) aus zwei- oder dreidimensionalen Bilddaten verwendet. So können beispielsweise bestimmte Organe in computertomographischen Aufnahmen hervorgehoben und ihre Eigenschaften näher untersucht werden. Anschaulich kann eine aktive Kontur aufgefasst werden als der Prozess, um ein gegebenes Objekt eine Plastikmembran zu legen und diese dann zu evakuieren (engl. *shrink wrapping*). Auch der umgekehrte Weg, also das „Aufblähen" einer Membran innerhalb eines Objektes ist möglich (engl. *ballons*). Im Kontext der Randanpassung einer Finite-Differenzen-Diskretisierung ist das Analogon zur Gummimembran die Menge der Hexaederseitenflächen, die mit ihren vier

Knoten auf dem geometrischen Randobjekt zu liegen kommen sollen. Wie bei einer realen Plastikmembran ist es hier wichtig, dass keine Risse, Überlagerungen oder „Falten" in der Kontur entstehen.

Eine aktive Kontur im Zweidimensionalen ist eine Funktion $\nu : [0, 1] \rightarrow \mathbb{R}^2$, die auf einem Bild $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ positioniert wird und sich durch Minimierung ihrer Energie in eine optimale Position bewegen soll. Für den Dreidimensionalen Fall kann dies als ein Gummiband aufgefasst werden, das zwischen zwei beliebigen Punkten auf der Oberfläche eines Objektes gespannt wird. Die Energie dieser Kontur setzt sich aus zwei Bestandteilen zusammen, der Bildenergie und der internen Energie:

$$E(\nu, f) = E_{\text{image}}(\nu, f) + E_{\text{int}}(\nu) \tag{2.16}$$

Die interne Energie ist dabei definiert als

$$E_{\text{int}}(\nu) = \int_0^1 \alpha(s)|\nu'(s)|^2 + \beta(s)|\nu''(s)|^2 ds \tag{2.17}$$

Als Bildenergie kann beispielsweise ein Filter zur Extraktion von Kanten oder ähnliches verwendet werden.

Der erste Summand der Gleichung 2.17, genannt Membran-Energie, nimmt große Werte an, wenn große Abstände zwischen benachbarten Punkten auf der Kontur vorliegen. Der zweite Summand beschreibt die Biegeenergie der Kontur. Die Gewichtsfunktionen α und β beschreiben somit die Elastizität (engl. *elasticity*) und Steifheit (engl. *rigidity*).

2.5.1 Gradient Vector Flow

Die Funktionsweise der aktiven Konturen hat insbesondere zwei Schwachpunkte: Zum Einen muss sich die Kontur in nächster Nähe zu dem zu umschließenden Objekt befinden, zum Anderen konvergieren die aktiven Konturen häufig nicht in Randeinbuchtungen, da z.B. dann die Biegeenergie stärker als die Bildenergie wirkt. Mit der Methode des *Gradient Vector Flow* (GVF) [66] werden diese Probleme zumindest teilweise gelöst, dabei werden mit Hilfe von Diffusionsgleichungen die für die Snake relevanten Bildinformationen so manipuliert, dass bereits aus relativ großer Entfernung die Snake in Richtung der zu erfassenden Details gezogen wird. In Abbildung 2.14 werden die Vorteile von *Gradient Vector Flow* gegenüber herkömmlichen Snakes deutlich.

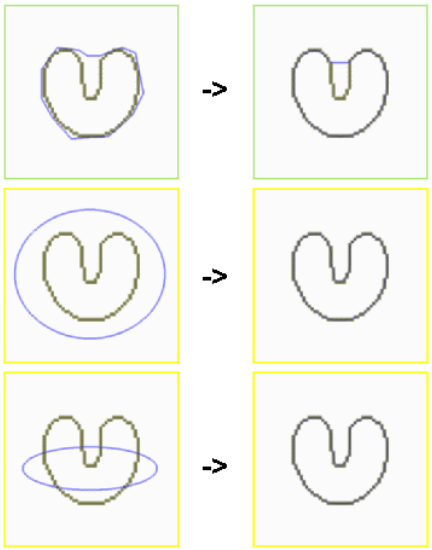


Abbildung 2.14: *Erste Zeile: Eine herkömmliche Snake muss in der Nähe des Objektes initialisiert werden, die Randeinbuchtung wird nicht erfasst. Zweite Zeile: Die GVF-Snake kann in größerem Abstand zum Objekt initialisiert werden und erfasst auch die Randausbuchtung. Dritte Zeile: Die GVF-Snake kann auch quer über das Objekt initialisiert werden, bei herkömmlichen Snakes funktioniert dies häufig nicht.*

2.5.2 Active Net

Eine Erweiterung des Snake-Konzepts auf die dritte Dimension wurde von Takanashi et al. [57] vorgestellt, einen vergleichbaren Ansatz hat Ahlberg [1] entwickelt. Beide verfolgen ein ähnliches Prinzip: Aus einem Volumendatensatz wird eine bestimmte Information extrahiert, indem ein vorgegebenes Objekt, wie z.B. die Oberflächentriangulierung einer Kugel, an das zu extrahierende Objekt „angeschmiegt" wird. Beide Konzepte erwiesen sich letztendlich für die Ziele der PG als ungeeignet, da lediglich auf die Approximation einer Oberflächen eingegangen, die nötigen Qualitätsmerkmale eines adjazenten Gitters zu dieser Oberfläche jedoch nicht betrachtet wurden.
Die Methode des *Gradient Vector Flow* wurde ebenfalls auf die dritte Dimension erweitert [67], der Nachteil der fehlenden Berücksichtigung eines adjazenten Gitters liegt aber auch hier vor.

2.5.3 Snakes auf Oberflächentriangulierungen

Die von der PG benutzte Snakes-Technologie basiert vollständig auf dem Ansatz von Kobbelt et al. [3], die Autoren waren so freundlich, während der Implementierung bei Fragen zur Verfügung zu stehen.

Konsistenzbedingung

Die Snake wird durch einen Polygonzug im dreidimensionalen Raum repräsentiert, für den zwei Bedingungen gelten müssen:

1. Die Knoten des Polygonzuges müssen auf den Kanten der Oberflächentriangulierung liegen.

2. Die einzelnen Streckenabschnitte des Polygonzuges müssen innerhalb eines Dreieckes der Oberflächentriangulierung liegen.

Weiterhin hat jeder Knoten $\mathbf{s} \in \mathbb{R}^3$ des Polygonzuges, der von den Autoren auch *Snaxel* genannt wird, eine Richtung und wird wie folgt beschrieben:

$$\mathbf{s} = (1-d)\mathbf{v}_{\text{from}} + d\mathbf{v}_{\text{to}}, \qquad d \in [0,1)$$

wobei $\mathbf{v}_{\text{from}}$ und $\mathbf{v}_{\text{to}}$ die Endpunkte der Kante auf der Oberflächentriangulierung sind. Die Richtung definiert sich von $\mathbf{v}_{\text{from}}$ nach $\mathbf{v}_{\text{to}}$. Es ist wichtig, dass die Richtung für alle Snaxel konsistent gehalten wird, es müssen also alle Snaxel zur selben Seite der Snake zeigen. Des Weiteren dürfen keine zwei Segmente der Snake im gleichen Dreieck der Oberflächentriangulierung liegen, diese Segmente werden als ungültig bezeichnet. Ungültige Segmente können bei der Überquerung von Knoten entstehen, sie lassen sich aber leicht erkennen und auch entfernen, indem ihre benachbarten Snaxel miteinander verbunden werden (s. Abb. 2.15). Mit dieser stückweise linearen Darstellung lässt sich also jedes Detail des Objektes erfassen, unabhängig von der Auflösung bzw. der Anzahl der verwendeten Dreiecke.

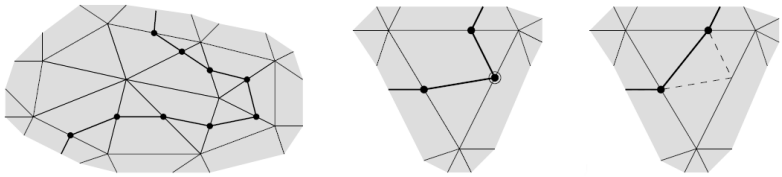


Abbildung 2.15: *Links: Eine gültige Snake. Mitte: Die Konsistenzbedingung wurde verletzt. Rechts: Durch Entfernen des ungültigen Snaxels wird die Konsistenz wieder hergestellt.*

Bewegung

Die Bewegung oder auch *Evolution* einer Snake wird über interne und externe Kräfte bestimmt, welche üblicherweise aus der Biegeenergie der Snake berechnet werden. In unserem Fall wird zunächst jedem Snaxel $\mathbf{s}$ ein Geschwindigkeitswert v_s zugewiesen, der die Energie repräsentiert. Dann wird die Winkelhalbierende zwischen zwei Snake-Segmenten ermittelt. Mit der Kante, auf der der Snaxel $\mathbf{s}$ liegt, ergibt sich nun ein Winkel $\alpha_{\mathbf{s}}$. Die Geschwindigkeit $\hat{v}_{\mathbf{s}}$ entlang der Kante ergibt sich wie folgt:

$$\hat{v}_{\mathbf{s}} = \frac{v_{\mathbf{s}}}{\cos \alpha_{\mathbf{s}}} \tag{2.18}$$

Falls sich mehrere Snaxel auf einem Punkt befinden, wird diese Berechnung nicht funktionieren – man wählt dann einfach die nachfolgenden Segmente der Snake und verwendet lediglich zur Berechnung von α_i dann die entsprechende Kante der Oberfläche, auf der der Snaxel liegt. Nachdem die Geschwindigkeiten für alle Snaxel berechnet sind, wird ein Zeitschritt Δt berechnet, und zwar so, dass kein Snaxel über das Ende einer Kante hinauswandert:

$$\Delta t = \min_{\mathbf{s}}(1 - d_{\mathbf{s}})/\hat{v}_{\mathbf{s}} \tag{2.19}$$

Wenn ein Snaxel nun am Ende einer Kante auf einen Knoten der Valenz n trifft, wird der Snaxel in $(n-1)$ neue Snaxel aufgeteilt, die auf die ausgehenden Kanten gesetzt werden. Durch diese Aufteilung kann die zweite Konsistenzbedingung verletzt werden, dies kann aber, wie bereits erwähnt, relativ einfach behoben werden (s. Abb. 2.15).

2.6 GPGPU

GPGPU bedeutet *general purpose computation using graphics hardware*, d.h. es werden Grafikkarten benutzt um allgemeine Probleme, wie z.B. die Matrix-Vektor-Multiplikation, zu lösen. Durch die in der Pipeline aktueller Grafikkarten frei programmierbaren Vertex- und Pixelshader kann man die Grafikkarte als Stream-Prozessor auffassen. Passende Programmiertechniken vorausgesetzt, erlaubt die Grafikkarte eine Vielzahl von Problemlösungen, welche die Geschwindigkeiten normaler CPU-Lösungen bei weitem übersteigen und somit ein besseres Preis-Leistungs-Verhältnis bieten. Zusätzlich gibt es frei zugängliche Shadersprachen, mit denen man die GPUs effizient auf Hochsprachen-Niveau programmieren kann.
GPGPU wurde schon in vielfältigen Projekten realisiert. Beispielhaft sind hier Ray Tracing [50], Sortieralgorithmen [33], Kollisions- oder FFT-Berechnung [40] zu nennen. Die aktuelle Entwicklung in diesem Bereich lässt sich dem State of the Art Report entnehmen [47]. Da ein Ziel des Projektes in der Realisierung einer Strömungssimulation gesetzt war, wird hier mehr auf den Einsatz von GPGPU zur Berechnung von Strömungen hingewiesen [5, 32]. Insbesondere soll hier auf die Arbeit von Krüger und Westermann eingegangen werden [27]. Sie behandelt generelle Techniken zur Berechnung von algebraischen Gleichungen, wie sie in numerischen Simulationen vorkommen. Zur Lösung von spärlich besetzten linearen Gleichungssystemen werden dort das CG- und das Gauss-Seidel-Verfahren benutzt. Eine interessante Idee ist die Repräsentation von 2D-Matrizen als Menge von diagonal verlaufenden Vektoren, die später ausführlicher behandelt wird. Im von der PG implementierten Löser kommt ein ähnliches Verfahren zum Einsatz, das jedoch für den dreidimensionalen Fall angepasst wurde (s. Kap. 5.8).
Für dünn besetzte Matrix/Vektor-Produkte wurde ein Verfahren erarbeitet, welches es erlaubt den Vertexprozessor in die Berechnungen einzubinden. Dies ist insofern vorteilhaft, als in den meisten Veröffentlichungen für Berechnungen auf der GPU ausschließlich der Fragmentprozessor verwendet wird. Eine interessante Optimierung wurde erzielt, indem Vektoren geeignet aufgeteilt und in die RGBA-Kanäle einer Textur gespeichert wurden. Krüger und Westermann beschreiben verschiedene Verfahren, die in der PG zum Einsatz kommen (s. Kap. 5.8).

Eine dieser grundlegenden Operationen, die für verschiedene Löser benutzt wird, ist SAXPY ($\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$). Eine Multiplikation mit Bandmatrizen kann beispielsweise durch wiederholte Anwendung von SAXPY realisiert werden. Göddeke beschreibt in seinem Tutorial [15] die Realisierung dieser elementaren Operation auf der GPU. Diese Operation wurde von der PG

benutzt, um die verschiedenen Möglichkeiten der GPU-Programmierung zu evaluieren (s. Kap. 4.4).

2.6.1 Hardware

Um die potentielle Rechenkraft der Grafikkarten ausschöpfen zu können, muss der interner Aufbau der Grafikprozessoren ausgenutzt werden. Grafikkarten sind auf die Darstellung von 3D-Szenen optimiert und nicht direkt zur allgemeinen Berechnung gedacht. Die Anbindung der Grafikkarten in das restlichen System erfolgt über einen AGP- oder PCIe-Bus. Da diese Bussysteme nicht die benötigten Geschwindigkeiten zum Hauptspeicher bieten, werden Grafikkarten mit eigenem Speicher ausgestattet. Aktuelle Versionen binden diesen mit einem 256 Bit breiten Bus an und haben zur Zeit eine Speicherbandbreite von bis zu 32 GB/s (z.B. ATI Radeon-X800 [41]).
Die hohe Rechenkraft und die interne Parallelschaltung wird durch die Ausrichtung der Hardware auf *Single Instruction Multiple Data* (SIMD) erreicht. Dieses Konzept besagt, dass ein kleiner Instruktionskernel unabhängig auf verschiedene Daten angewendet wird. Erst nach dem kompletten Abarbeiten des Datensatzes kann auf die neu berechneten Werte zugegriffen werden. Der Instruktionskernel wird im Grafikprozessor von mehreren parallel geschalteten Shadern abgearbeitet. Um Grafikkarten zur allgemeinen Berechnung sinnvoll nutzen zu können, muss dieses Konzept effizient auf die Problemstellung anwendbar sein.
Innerhalb einer Grafikkarte werden Daten in Texturen gespeichert. Texturen sind zweidimensionale Felder von Texeln. Ein Texel kann aus bis zu vier Skalaren bestehen. Diese Organisation der Daten ist durch die eigentliche Aufgabe der Grafikkarte, der Darstellung einer dreidimensionalen Szene, bedingt. Ein Texel, welcher nach vollständiger Berechnung zum Pixel auf dem Bildschirm wird, besteht aus drei Skalaren für die Farbkanäle Rot, Grün und Blau sowie einem weiteren Skalar für Transparenz. Es werden auch 1D- und 3D-Texturen nativ von aktuellen Grafikkarten unterstützt. Die Benutzung führt jedoch zu Geschwindigkeitseinbußen.
Kernel, die auf einer GPU ausgeführt werden, beschreiben wie ein Bild „gerendert" werden soll. Vertexshader manipulieren geometrische Informationen (Knoten, Normalen, usw.), während Fragmentshader auf den Daten von einzelnen Fragmenten (z.B. interpolierten Farb- und Tiefeninformationen) arbeiten.

2.6.2 Shader- und Metasprachen

Shadersprachen

Um die einzelnen Grafikkarten nicht mit Assemblercode programmieren zu müssen, wurden innerhalb der letzten Jahre Shader-Hochsprachen zur Programmierung dieser Karten entwickelt. Im Gegensatz zu so genannten Offline-Shadersprachen, deren verwendete Algorithmen nicht effizient auf Hardware implementierbar sind, wurden *Cg* [34], *HLSL* [38] und *GLSL* [44] in Hinblick auf ihre Implementierbarkeit bezüglich einer Rendering Pipeline entworfen. Ziel bei der Entwicklung war es also, eine Sprache zu entwickeln, die der GPU-Hardware leicht zugänglich ist und dort direkt ausgeführt werden kann. Da sich die Sprachen im Aufbau und Leistung sehr ähnlich sind, bezieht sich das folgende Beispiel auf Cg.

Vorteile einer Shadersprache am Beispiel von Cg

Bei der Entwicklung der Sprache Cg wurde hauptsächlich auf die Kriterien Programmierbarkeit und Portabilität besonderes Augenmerk gelegt. Einfache Programmierbarkeit bedeutet in diesem Fall ein höheres Abstraktionslevel als Assemblersprachen und Wiederverwendbarkeit von Quelltext und die Möglichkeit zur Quelltext-Strukturierung durch Schnittstellen, Structs und Methodenaufrufe. Unter Portabilität versteht man die Unabhängigkeit von Hardwareherstellern, Hardwaregenerationen, Betriebssystemen und den unterschiedlichen 3D-APIs. Weitere wichtige Kriterien waren die vollständige Unterstützung der Hardwarefunktionalität, die Perfomanz, die minimale Beeinflussung der Organisation von Anwendungsdaten, die einfache Adaption in bestehende Systeme, die Erweiterbarkeit für zukünftige Hardware und die Unterstützung für nicht auf Shading ausgerichtete Anwendungen einer GPU.
Eine Shadersprache ersetzt weder die Grafik APIs (wie DirectX[3] oder OpenGL[4]) noch die Applikation. Sie fungiert als Zwischenschicht und wird explizit nur über definierte Programmparameter angesprochen (s. Abb. 2.16). Die Übergabe kann über Assemblersprache oder die Runtime-API erfolgen.

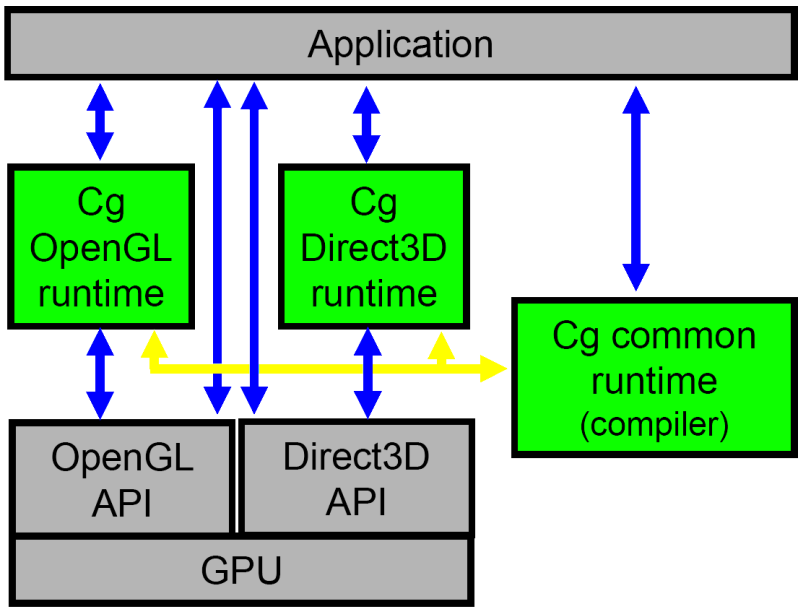


Abbildung 2.16: *Die Cg-Architektur [34].*

Metasprachen

Mit Hilfe von Metasprachen wie *Sh* [35] und *BrookGPU* [6] ist es möglich, eine GPU (soweit sie über programmierbare Shader verfügt) auf abstrakter Ebene zu programmieren. Das heißt, dass man die OpenGL, DirectX oder herstellerspezifischen Erweiterungen nicht beachten muss. Man kann sich ganz auf den Algorithmus und nicht auf die grafikspezifische Implementierung

[3] http://www.microsoft.com/windows/directx/default.aspx
[4] http://www.opengl.org

konzentrieren.
Der Nachteil ist, dass im Vergleich zu handgeschriebene Programmen (zum Beispiel in Cg und OpenGL) Leistungseinbußen in Kauf genommen werden müssen. Das liegt daran, dass *BrookGPU* ein „Übersetzer-Frontend" darstellt und C- bzw. Shader-Programme generiert, und *Sh* mehr als ein „Toolkit" aufgefasst werden soll, das zwar einen Compiler, Optimierer usw. beinhaltet, aber auch nur einen Zwischencode liefert. Auf der einen Seite werden dem Programmierer alle grafikkartenspezifischen Bereiche (die Handhabung von Texturen, Speicher, usw.) abgenommen und die Plattformunabhängigkeit gewahrt, auf der anderen Seite muss man mit einem Leistungsverlust rechnen.

2.6.3 GPU-Konzepte

Im folgenden Abschnitt werden GPU-Analogien für grundlegende Datenstrukturen und Programmierkonzepte vorgestellt. In Programmen für die CPU ist ein Feld von Daten (engl. *array*) eine zentrale Datenstruktur. Eine GPU-Entsprechung hierfür ist das Speichern von Daten in Texturen. Damit die Feldindizes den Indizes der Textur entsprechen ist in den Grafikschnittstelle eine orthographische Projektion zu setzen und die Dimension der Bildebene ist der der Textur anzupassen. Die eigentliche Berechnung wird dadurch durchgeführt, dass die Vertex- oder Fragmentprogramme aktiviert werden, die Texturen oder Texturkoordinaten als Parameter übergeben werden und letztendlich ein Rechteck der zuvor angegebenen Bildschirmdimension gezeichnet wird. Dadurch wird für jedes Texel der Ergebnisstextur (für jeden Eintrag im Feld) das Fragmentprogramm ausgeführt.

Reduktion

Allgemein berechnen Shaderprogramme eine Menge von Ausgabewerten aus einer Menge von Eingabewerten. Bei allgemeinen Berechnungen wird jedoch häufig ein einzelner Wert aus einer Menge von Werten berechnet, wie z.B. das Maximum einer Menge von Werten oder das Skalarprodukt von Vektoren. Das Verfahren, mit dem oben genannte Operationen auf der GPU realisiert werden, heißt Reduktion (engl. *reduce*) [12]. Die Idee ist, in jedem Durchlauf (engl. *rendering pass*) durch die Fragmentpipeline eine partielle Reduktion durchzuführen, bis sich die Anzahl der Pixel auf eins reduziert hat. Das Verfahren, dass in Abbildung 2.17 dargestellt wird, sieht wie folgt aus:

Eine zweidimensionale Textur mit Werten dient als Eingabe. In jedem Durchlauf wird ein Rechteck gezeichnet, dessen Größe einem Viertel der Eingangsgröße entspricht. Jedes Fragment des verkleinerten Rechtecks entspricht dem Maximum eines 2×2 Rechtecks in der Textur.

Wenn die Texturgröße $n \times n$ ist, werden $O(\log n)$ Durchläufe benötigt.

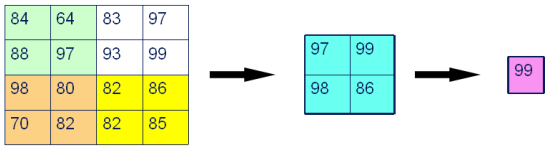


Abbildung 2.17: *Reduktionsverfahren auf der GPU[12].*

Produkt von Bandmatrix und Vektor

Bandmatrizen werden als eine Menge von diagonalen Vektoren aufgefasst und dementsprechend werden diese Diagonalvektoren in einer zweidimensionalen Textur gespeichert. Um Platz einzusparen, kann man an den Diagonalvektor, der in der i-ten Spalte der ersten Zeile beginnt den Diagonalvektor, der in der $(N - i)$-ten Zeile der ersten Spalte beginnt, anhängen (s. Abb. 2.18). Alternativ können die Diagonalvektoren, sofern sie nicht zu weit von der Hauptdiagonalen entfernt liegen, durch ein Auffüllen mit Nulleinträgen auf die Länge N angepasst werden, in der PG wird dieses Verfahren angewendet.

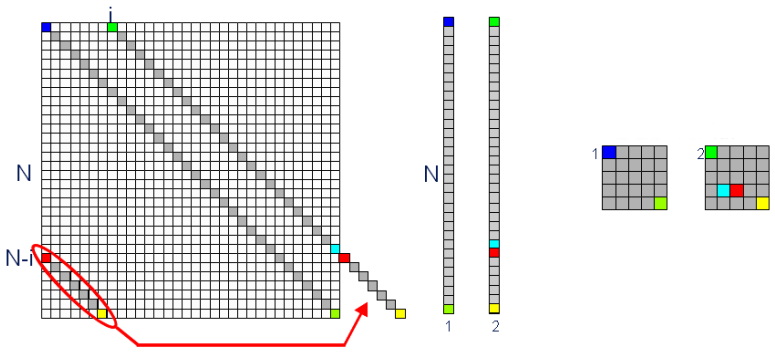


Abbildung 2.18: *Abspeicherung einer Bandmatrix in Texturen [48].*

Zur Durchführung der Matrix-Vektor-Multiplikation liegt die Bandmatrix als Menge von Diagonalvektoren vor. Diese werden in zweidimensionale Texturen überführt, ebenso der Vektor, der mit der Matrix multipliziert werden soll. Nacheinander werden die Diagonalvektoren mit dem Vektor multipliziert und das Resultat in einer Ergebnistextur gespeichert. Bei jedem Durchgang der Diagonalvektor-Vektor-Multiplikation muss darauf geachtet werden, dass die richtigen indizierten Werte miteinander multipliziert werden. Daher ist es nötig einen Hilfsindex auf den aktuellen Index im Vektor aufzuaddieren. Dieser Hilfsindex errechnet sich aus dem Abstand des Bandes zur Hauptdiagonalen (s. Abb. 2.19). Die Ergebnisse der einzelnen Diagonalvektor-Vektor-Multiplikationen werden in den entsprechenden Einträgen der Ergebnistextur aufsummiert.

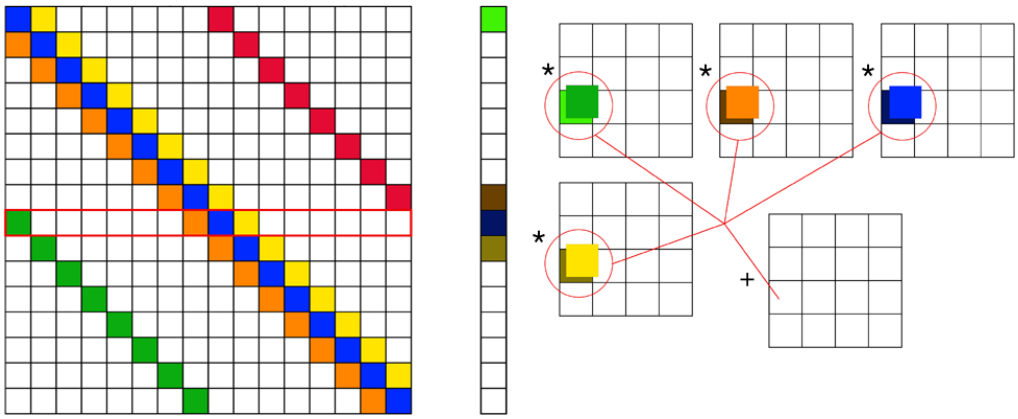


Abbildung 2.19: *Produkt von Bandmatrix und Vektor.*

Kapitel 3

Werkzeuge
Überblick über die benutzte Software

3.1 OpenGL

OpenGL, *Open Graphics Library* [43], ist eine Spezifikation für ein plattform- und programmiersprachenunabhängiges API (*Application Programming Interface*) zur Erzeugung von 3D-Computergrafik. Es ist ein offener, über viele Hardwareplattformen unterstützter Standard. Zudem können andere Organisationen (zumeist Hersteller von Grafikkarten) auch proprietäre Erweiterungen definieren.

3.2 Qt

Qt [60] ist eine Klassenbibliothek für die plattformübergreifende Programmierung graphischer Benutzeroberflächen (GUIs) unter C++. Es stellt viele Funktionen bereit, um Anwendungen mit den modernsten graphischen Benutzerschnittstellen zu versehen. Qt ist objektorientiert und leicht zu erweitern. Es beinhaltet einen einfachen Mechanismus zur Kommunikation zwischen Objekten namens *signals and slots*. Dabei werden bei Ereignissen Signale erzeugt, die von Empfängern aufgenommen werden. Diese Signale werden an korrespondierende Funktionen weitergegeben, die die Ereignisverarbeitung realisieren. Die grafische Benutzeroberfläche der einzelnen Prototypen und des endgültigen Programms wurden mit Hilfe von Qt erstellt. Ein Tutorial ist beispielweise unter [58] verfügbar.

3.3 Visual Studio

Das Programm Visual Studio [37] ist eine von der Firma Microsoft angebotene integrierte Entwicklungsumgebung für Hochsprachen. Neben der Programmierumgebung und einem Debugger sind noch diverse Werkzeuge zur Unterstützung des Entwicklers bzw. zur Erweiterung

von Team-Arbeits-Funktionen integriert. Visual Studio wurde von der „Informatikrechner Betriebsgruppe" (IRB) lizenziert und für die PG zur Verfügung gestellt und diente als Programmierumgebung für C++.

3.4 CVS

CVS, *concurrent versions system* [7], ist ein Programm zur Versionsverwaltung von Dateien, hauptsächlich Softwarequelltexten. CVS vereinfacht die Verwaltung von Quelltext – und allen anderen textbasierten Dateiformaten – dadurch, dass es alle Dateien eines Software-Projektes an einer zentralen Stelle speichert. Dabei können jederzeit einzelne Dateien verändert werden, es bleiben jedoch alle früheren Versionen erhalten, einsehbar und wiederherstellbar. Auch können die Unterschiede zwischen bestimmten Versionen verglichen werden. Dank CVS konnte das Projekt trotz gemeinsamer weitverteilter Bearbeitung stets konsistent gehalten werden.

3.5 LaTeX

LaTeX [29] ist ein frei verfügbares Textsatzprogramm, das speziell auf die Erzeugung von wissenschaftlichen Texten ausgelegt ist. Dieser Endbericht wurde mit LaTeX gesetzt, da man damit die Möglichkeit hatte, auch dieses große Dokument einfach und übersichtlich zu erstellen. So konnten die einzelnen Abschnitte in separaten ASCII-Dateien gespeichert und Querverweise, Abbildungen und Abschnitte komfortabel verwaltet werden. In Verbindung mit CVS konnten alle PG-Mitglieder gleichzeitig am Endbericht arbeiten.

3.6 OpenMesh

OpenMesh [51] ist eine Datenstruktur-Bibliothek zur Repräsentation polygonaler Netze, mit der man Freiform-Flächen im Bereich der geometrischen Modellierung darstellen und verarbeiten kann. OpenMesh basiert auf einer Halbkantendatenstruktur für Netze und stellt vordefinierte Mesh-Kernel zur Repräsentation von besonders effizienten Dreiecks- als auch allgemeinen polygonalen Netzen zur Verfügung. In der PG wird die Parametrisierung der geladenen 3D-Objekte auf der OpenMesh-Datenstruktur durchgeführt.

3.7 Blender

Blender [4] ist ein freies 3D-Modellierungsprogramm, mit dem man dreidimensionale Körper modellieren, rendern, animieren und nachbearbeiten kann. Ebenso ist das Erstellen und Abspielen von interaktiven 3D-Inhalten möglich. Das Programm ist dafür ausgelegt, auf verschiedenen Plattformen lauffähig zu sein. Blender wurde für die Erstellung von 3D-Testobjekten benutzt (s. Kap. 5.1).

3.8 Triangulate

Dieses Werkzeug ist die Implementierung einer Triangulierungsmethode von Joseph O'Rourke [45]. Das Verfahren eignet sich, um Polygone zu triangulieren, innerhalb des Projekts wird es

während der Netzdezimierung angewendet. Die Robustheit der Methode ist der Hauptgrund aus dem das Verfahren im Projekt verwendet wird.

3.9 Lpsolve

Lpsolve [2] ist ein freies Programm zum Lösen von linearen Optimierungsproblemen. *Lpsolve* wurde in ANSI C geschrieben und kann auf verschiedenen Plattformen verwendet werden. Der Löser kann als Bibliothek in ein Projekt eingebunden werden und von verschiedenen Programmiersprachen aufgerufen werden. Im Projekt der PG wird der Löser verwendet, um Optimierungsprobleme zu lösen, die bei der Verbesserung der Gitternetzqualität auftreten.

3.10 Cg

Cg, *C for graphics* [42], ist eine von Nvidia und Microsoft begründete Programmiersprache für Grafikprozessoren. Cg kann als eine Art C-Dialekt mit sehr ähnlicher Struktur und Syntax aufgefasst werden. Sie ist mit dem Ziel der Plattformunabhängigkeit und Unabhängigkeit von bestimmten Herstellern entwickelt worden und bietet zudem eine C-Schnittstelle zu OpenGL und Direct3D (*runtime-environment*) an. Mit Hilfe von Cg wurden GPGPU-Performancetests durchgeführt (s. Kap. 4.4) und der prototypische GPU-Löser implementiert (s. Kap. 5.8).

3.11 BrookGPU

BrookGPU [6] ist eine Metasprache, die entwickelt wurde um GPUs direkt zu programmieren. Das heißt, dass man sich nicht mit OpenGL, DirectX oder ATI/Nvidia-Erweiterungen, also mit der grafikspezifische Implementierung, beschäftigen muss, sondern sich ganz auf den Algorithmus konzentrieren kann. *BrookGPU* wurde bei den GPGPU-Leistungstests benutzt (s. Kap. 4.4).

3.12 LA-Framework

Das „Lineare-Algebra-Framework" (LA-Framework) [26] basiert auf der DirectX-Schnittstelle und wurde entwickelt um die Programmierung numerischer Berechnungen auf der GPU zu vereinfachen. Es bietet eine Vielzahl von fertigen „Bausteinen", die man für diese Berechnungen benutzen kann. Das LA-Framework wurde ebenfalls bei den GPGPU-Performancetests benutzt (s. Kap. 4.4).

3.13 FEASTGPU

FEASTGPU ist eine Implementierung zu aktuellen Forschungsarbeiten am Lehrstuhl für Angewandte Mathematik. Sie wurde der PG als Ausgangspunkt für die Implementierung des prototypischen Lösers von einem Betreuer zur Verfügung gestellt (s. Kap. 5.8).

Kapitel 4

Vorarbeiten

Ergebnisse der vorbereitenden Experimente

Im Verlauf des ersten Semesters wurden verschiedene Ansätze und Programme zum Erreichen der Ziele der PG untersucht. Unter anderem wurde die Gittergenerierungssoftware *Cubit* untersucht, und es wurden Leistungstests auf Grafikkarten durchgeführt. Außerdem sind Veröffentlichungen zur Parametrisierug von Objektoberflächen evaluiert worden. Die Ergebnisse werden in diesem Kapitel vorgestellt.

4.1 Diplomarbeit von Wobker

Die Idee der Diplomarbeit von Wobker [65] zur Parametrisierung von Oberflächennetzen ist folgende: Das zu parametrisierende Netz wird durch das Löschen von Halbkanten und anschließendem Verschmelzen der Knoten (engl. *halfedge collapse*) vergröbert. Das entstandene Grobnetz stellt den Parameterraum für die Parametrisierung dar. In einem weiteren Schritt muss nun das Originalnetz segmentiert werden. Wichtig ist hierbei, dass eine eins-zu-eins Beziehung zwischen der Segmentierung und den Dreiecken des dezimierten Netzes benötigt wird. Um dies zu bewerkstelligen wurden von Wobker die kürzesten geodätischen Distanzen auf der Oberfläche berechnet.
Nach Besprechung dieses Parametrisierungsverfahrens innerhalb der PG fand ein Treffen mit Hilmar Wobker statt, in dem er seine Implementierung vorstellte und auf Probleme und Vorteile dieser Parametrisierung hingewiesen hat. Aufgrund des hohen Implementierungsaufwandes bei der Berechnung der kürzesten geodätischen Wege auf dem Feinnetz riet er eher zu anderen Verfahren, weswegen sich die PG gegen dieses entschieden hat.

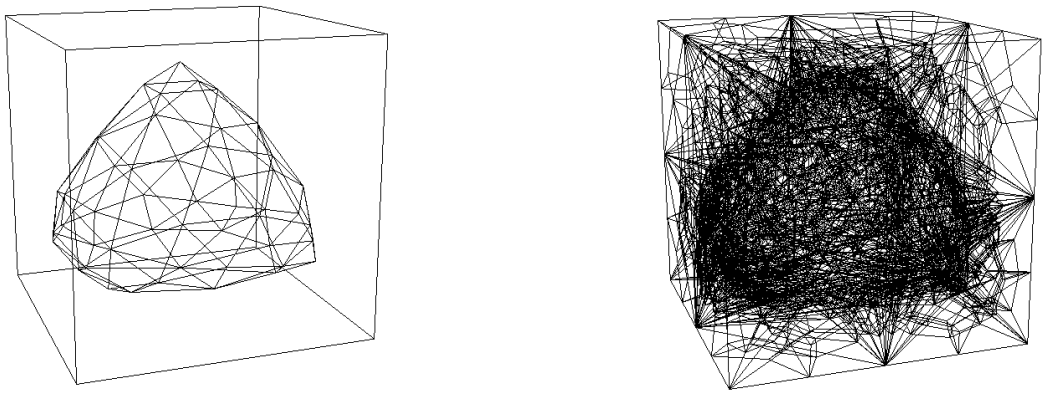
4.2 Normal Meshes

Das *Normal Meshes*-Verfahren dezimiert das vorhandende Dreiecksnetz ebenfalls mit Hilfe von Punktverschmelzungen [19]. Es wird eine Segmentierung berechnet, mit deren Hilfe die einzelnen Punkte des Feinnetzes über den einzelnen Dreiecken des Grobnetzes parametrisiert

werden. Diese Parametrisierung wird mit Hilfe der Länge des Normalenvektors zum Grobdreieck von dem zu parametrisierenden Punkt aus bewerkstelligt.
In der Veröffentlichung wird an dem für die PG relevanten Punkt, nämlich der initialen Parametrisierung, auf MAPS [30] verwiesen. Aus diesem Grund verfolgte die PG auch diese Methode nicht weiter.

4.3 Cubit

Im Rahmen der Projektgruppe wurde Cubit [52] evaluiert, um ein Grobgitter für die Strömungssimulation zu erzeugen. Dazu wurde das zu umströmende Objekt in einem quaderförmigen Kanal positioniert und mittels der Methode *substract* das Differenzvolumen erzeugt (s. Abb. 4.1(a)), welches mit gleichförmigen Hexaederzellen aufgefüllt werden sollte.
Das automatisierte Erstellen eines sinnvollen Grobgitters stellte sich jedoch als unmöglich heraus. Mittels des Schemas *TetMesh* und der Methode *Mesh* kann ein Differenzvolumen mit Tetraedern aufgefüllt werden. Anschließend kann es jedoch nur in ein unstrukturiertes Hexaedergitter überführt werden, indem jedes Tetraeder in vier Heaxeder unterteilt wird (s. Abb. 4.1(b)).
Zwar bietet Cubit einige Operationen an, mit deren Hilfe das Differenzvolumen in reguläre Hexaederzellen unterteilt werden können soll, jedoch befinden sich diese noch im Beta-Stadium und liefern nur für sehr einfache Objekte reguläre Gitter.
Die Methode *Plastering* versucht, die Volumendifferenz mit Quadern aufzufüllen. Dieses setzt ein Objekt voraus, welches einer Blockstruktur ähnelt (s. Abb. 4.2).
Mapping funktioniert nur bei Volumen, die sechs logische Seiten haben und acht logische Ecken. Die Unterteilung erfolgt jeweils bei gegenüberliegenden Seitenflächen.
Die Oberflächentriangulierungen, die in der Projektgruppe Verwendung finden, werden zudem nur indirekt unterstützt. Das Objekt muss erst in ein Volumen konvertiert werden, damit es in *Cubit* verwendet werden kann. Weitere Informationen über die Operationen von Cubit können im Handbuch [52] nachgelesen werden.



(a) *Volumendifferenz mit* Subtract. (b) *Unstrukturiertes Hexaedergitter mit* THex.

Abbildung 4.1: *Testmodell in Cubit.*

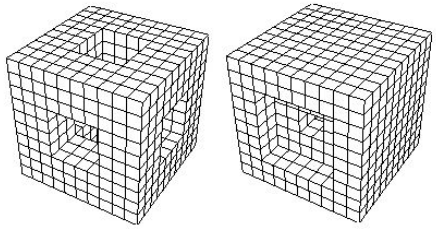


Abbildung 4.2: *Mit der Methode* Plastering *erzeugte reguläre Grobgitter.*

4.4 GPGPU

Da es zur Zeit viele unterschiedliche Standards für Grafikkarten (ATI, NVIDIA), Grafikkarten-APIs (OpenGL, DirectX) und Shadersprachen (Cg, GLSL, HLSL) gibt, einigte sich die PG darauf einen Vergleichstest durchzuführen. Ziel dieses Tests sollte sein, eine Testumgebung zu konstruieren, die es erlaubt, die bestmögliche Technik einfach zu bestimmen, um diese dann anzuwenden. Zusätzlich gibt es noch die Möglichkeit, eine Metasprache wie *BrookGPU* zu benutzen, die es erlaubt Grafikkarten unabhängig von den oben genannten Unterschieden zu programmieren.
Als Grundlage für den Vergleichstest wurden die SAXPY-C und SAXPY-V Operationen gewählt, da diese sehr häufig in numerischen Berechnungen benutzt werden. SAXPY-C ist Teil der BLAS-Bibliothek [49] und damit auf CPUs typischerweise hochoptimiert (SSE, SSE2 usw.)[1]. Die für den Test benutzten Operationen sind wie folgt definiert:

$$\text{saxpy-c: } \mathbf{y}[i] = \mathbf{y}[i] + \alpha * \mathbf{x}[i]$$
$$\text{saxpy-v: } \mathbf{y}[i] = \mathbf{y}[i] + \mathbf{a}[i] * \mathbf{x}[i]$$

Als Rahmen für die DirectX-Tests stand das LA-Framework (s. Kap. 3.12) und für die OpenGL-Tests eine Implementierung aus einem Tutorial zum Thema [15] zur Verfügung. Zusätzlich wurde *BrookGPU* als frei verfügbare Metasprache eingesetzt (s. Kap. 3.11).
Für die Durchführung der geplanten Tests wurde das LA-Framework um eine eigene SAXPY-Klasse ergänzt, welche auch neue Shader beinhaltet. Die OpenGL-Implementierung konnte ohne Änderung benutzt werden, da dort bereits ein SAXPY-Test implementiert war. Für *BrookGPU* musste lediglich ein geeigneter Kernel implementiert werden.
Die Testroutine bestand aus einer bestimmten Anzahl von SAXPY-Durchläufen mit unterschiedlichen Problem- und Iterationsgrößen. Die Anzahl der Iterationen war bei kleinen Problemgrößen höher, um den Einfluss der Datentransferzeit und des sonstigen Overheads möglichst zu minimieren. Bei größeren Problemen führen solche hohen Iterationszahlen aber zu sehr langen Rechenzeiten, daher wurden in diesen Fällen weniger Iterationen durchgeführt.
Die Experimente wurden für verschiedene Einstellungen bzw. Texturformate durchgeführt. Die Tests in OpenGL wurden für alle Kombinationen aus Shadersprache, Texturformat und inter-

[1] SSE *Streaming SIMD Extensions* ist eine x86-Befehlssatzerweiterung.

ner Fließkommarepräsentation durchgeführt. In DirectX und *BrookGPU* wurden exemplarisch verschiedene Texturformate getestet.

Dabei werden folgende Abkürzungen verwendet:

`R32`:	Eine Textur mit einem Farbkanal (*unpacked*) und 32 Bit Farbtiefe
`RGBA32`:	Eine Textur mit vier Farbkanälen (*packed*) und 32 Bit Farbtiefe
`RGBA16`:	Eine Textur mit vier Farbkanälen und 16 Bit Farbtiefe
`2D`:	2D-Texturen[2] (OpenGL)
`RECT`:	RECT-Texturen[3] (OpenGL)
`CG`:	Shader in der Sprache Cg implementiert
`GLSL`:	Shader in der Sprache GLSL implementiert
`BROOK`:	Bei Brook kann man das *Backend* (OpenGL, DirectX) frei wählen

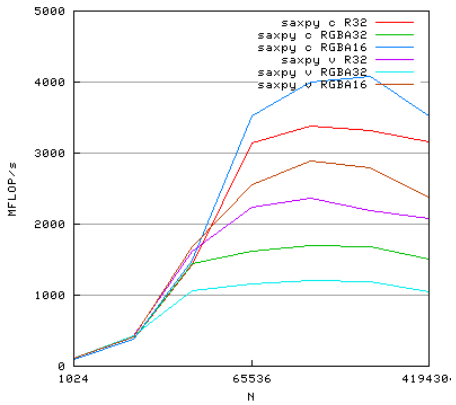
Bei den so genannten „ungepackten" Pixelformaten besteht jeder Pixel nur aus einer Farbkomponente, wohingegen die Pixel in den „gepackten" Formaten aus den üblichen vier Kanälen (RGBA) bestehen.
Es standen verschiedene Grafikkarten der Hersteller ATI und NVIDIA zur Verfügung. Die OpenGL-Tests basierten vollständig auf der Verwendung von so genannten *pBuffers*. Die neuere und einfacher zu handhabende Methode der so genannten *Framebuffer Objects* (FBOs) stand zum Zeitpunkt der Tests in den ATI-Treibern noch nicht zur Verfügung. Beide Methoden zielen darauf ab, eine Textur alternierend als Ergebnisspeicher und Datenquelle zu verwenden. Als exemplarische Testbeispiele sollen hier die Ergebnisse einer „NVIDIA GeForce 6800 (NV40)" sowie einer „ATI Radeon 9800 PRO" dienen. Die *BrookGPU*-Ergebnisse waren um 5 – 10% langsamer als die LA-Framework- und OpenGL-Implementierungen. Zwischen OpenGL und DirectX wurden keine signifikanten Unterschiede festgestellt.

NVIDIA GeForce 6800 (NV40)

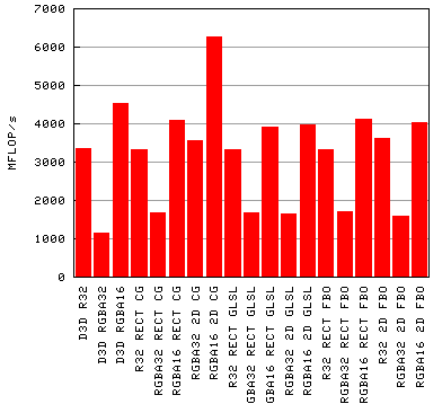
Abbildung 4.3(a) stellt die Rechenleistung für unterschiedliche Problemgrößen, basierend auf *Texture Rectangles* und Cg dar. Es ist deutlich zu erkennen, dass die Geschwindigkeit erst bei größeren Problemen attraktiv wird, bedingt durch Datentransferzeit und Overhead. Erwartungsgemäß sind die 16 Bit Texturformate am schnellsten, allerdings auf Kosten der Genauigkeit, die ohne Nachkorrektur inakzeptabel ist. Ungepackte Pixelformate laufen auf dieser Karte schneller als gepackte.
Abbildung 4.3(b) zeigt die Rechenleistung für unterschiedliche Texturformate bei einer festen Problem- bzw. Texturgröße von $N = 1024 \cdot 1024$. Die FBO-Ergebnisse wurden unter Linux, alle anderen unter Windows XP erstellt.

[2] 2D-Texturen sind das Standardformat für Texturen. Aktuelle Grafikkarten können in 2D-Texturen bis zu $4096 \cdot 4096 = 16.777.216$ Elemente speichern.

[3] RECT-Texturen sind ein weiteres Texturformat, für Details wird auf Göddeke [15] verwiesen.



(a) *MFLOP/s für verschiedene Problemgrößen.*

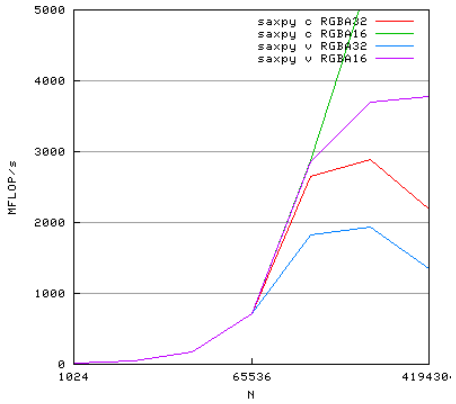


(b) *MFLOP/s für verschiedene Texturformate.*

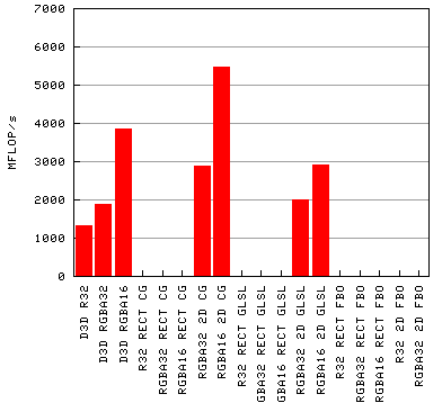
Abbildung 4.3: *Testergebnisse NVIDIA GeForce 6800.*

ATI Radeon 9800 PRO

Abbildung 4.4(a) stellt die Rechenleistung für unterschiedliche Problemgrößen, basierend auf 2D-Texturen und Cg dar. Im Gegensatz zur vorigen NVIDIA Karte erzielt die ATI 9800 Pro ihr Optimum bei der Benutzung eines gepackten Pixelfomates.
Abbildung 4.4(b) zeigt die Rechenleistung für unterschiedliche Implementationen bei einer festen Problemgröße von $N = 1024 \cdot 1024$. Die leeren Felder deuten auf fehlende Eigenschaften der Grafikkarte hin, ATI unterstützte zum Testzeitpunkt das Zeichnen in *Texture Rectangles* nicht.



(a) *MFLOP/s für verschiedene Problemgrößen.*



(b) *MFLOP/s für verschiedene Texturformate.*

Abbildung 4.4: *Testergebnisse ATI Radeon 9800 PRO.*

Fazit

Die PG entschied sich für OpenGL mit der *Framebuffer Object Extension*, die jetzt auch von den ATI Catalyst Treibern unterstützt wird. Die Benchmarks erzielten unter OpenGL auf jeder getesteten Hardware die besten Ergebnisse. Hinzu kommt, dass OpenGL mit der FBO-Erweiterung auch portabel ist, und somit auch Linux bedient werden kann.

Kapitel 5

Systembeschreibung
Überblick und die Module im Detail

Zum besseren Verständnis der Funktionsweise von *InGrid3D* werden nachfolgend die einzelnen Komponenten in ihrer Funktionsweise und Zusammenarbeit erklärt.
Ein trianguliertes 3D-Objekt wird als Grundlage vorausgesetzt. Eine weitere Voraussetzung für eine Gitterunterteilung ist ein geeignetes Gitter, welches die gegebene Geometrie grob approximiert. Dazu müssen einige Gitterknoten auf dem Dreiecksnetz des Objektes liegen. Der eigens für diesen Zweck erstellte Grobgittergenerator *HaGrid3D* liefert die dazu nötigen Manipulationsmöglichkeiten und unterstützt bei der manuellen Erstellung eines solchen Grobgitters.
Nachdem in *InGrid3D* zuerst ein Objekt und das dazu passende, in *HaGrid3D* erstellte, Gitter geladen wurden, folgt als nächster Schritt die Parametrisierung des 3D-Objektes. Sie dient zur schnellen Approximation eines auf dem Dreiecksnetz liegenden Mittelpunktes zwischen zwei ebenfalls auf dem Dreiecksnetz liegenden Punkten. Die Parametrisierung entfernt in einer bestimmten Reihenfolge Punkte der Oberfläche, retrianguliert die dadurch entstandenen Löcher und bestimmt neue Positionen für die bereits entfernten Punkte innerhalb der neuen Triangulierung. Die Positionen werden hierbei mit Hilfe von baryzentrischen Koordinaten innerhalb der neuen Dreiecke gespeichert. Dies funktioniert nach dem in Kapitel 2.2 vorgestellten Verfahren. Das Objekt wird durch diese Methode stark reduziert, ohne dass seine grobe Struktur verloren geht. Das reduzierte Objekt sieht dem ursprünglichen Objekt damit immer noch in gewisser Weise ähnlich[1].
Nach der Parametrisierung folgt die Unterteilung der Hexaeder. Jeder Hexaeder wird hierbei regulär in acht neue Hexaeder unterteilt. Es werden zuerst auf allen Kanten des Hexaeders die Mittelpunkte als neue Knoten hinzugefügt, dann wird in jede der sechs Flächen ihr Mittelpunkt als Knoten gesetzt. Als letzter Knoten wird der Mittelpunkt des Hexaeders eingefügt. Dieses Unterteilungsschema kann auf alle Hexaeder, die mit keinem Knoten auf dem Dreiecksnetz liegen, angewandt werden. Bei den Hexaedern am Objektrand ist diese Unterteilung nicht möglich. Es muss bei diesen Hexaedern darauf geachtet werden, dass neu hinzukommende Knoten eventuell ebenfalls auf dem Dreiecksnetz liegen müssen. Für diese Aufgabe wird ein Suchalgorithmus auf dem reduzierten Objekt gestartet, der aus zwei Teilen besteht. Abhängig

[1] Dies ist eine wichtige Eigenschaft vor allem für die später beschriebenen Snakes.

von der Entfernung der Dreiecke innerhalb derer sich zwei Knoten befinden, werden verschiedene Verfahren angewendet:

- Liegen beide Knoten innerhalb eines Dreiecks, so wird der Mittelpunkt ihrer Verbindungslinie zurückgegeben.
- Liegen beide Knoten in zwei über eine Kante benachbarten Dreiecken, so werden die beiden Dreiecke an ihrer gemeinsamen Kante auf eine Ebene abgebildet, wobei die Abbildungen kongruent zu den Ursprungsdreiecken sind. Danach wird wiederum der Mittelpunkt der Verbindungslinie der beiden Knoten zurückgegeben.
- Haben die zwei Dreiecke einen gemeinsamen Eckpunkt, so wird die Eins-Ring-Nachbarschaft des gemeinsamen Eckpunktes auf eine Ebene abgebildet. Diese Abbildung erfolgt mit Erhaltung der Kantenverhältnisse der Kanten, die vom gemeinsamen Eckpunkt ausgehen. Die Winkel der am gemeinsamen Eckpunkt anliegenden Kanten werden auf 360° verteilt abgebildet. Anschließend wird der Mittelpunkt der Verbindungslinie der beiden Knoten zurückgegeben.
- Haben die zwei Dreiecke keinen gemeinsamen Eckpunkt, so wird die Snake zwischen den beiden Knoten aufrufen und der Mittelpunkt[2] der Snake wird zurückgegeben.

Der von diesem Verfahren erzeugte Mittelpunkt wird auf das Dreiecksnetz projiziert und als Knoten der neuen Hexaeder vermerkt.
Bei komplexen Objekten kann diese Methode manchmal zu unerwünscht verdrehten Hexaederzellen führen, weshalb verschiedene Glätter implementiert wurden, deren Aufgabe darin besteht, die Hexaederzellen möglichst gleichmäßig auszurichten[3]. Am Ende der Systembeschreibung wird ein Löser vorgestellt, der eine Simulation auf GPGPU-Basis auf dem vorher an die Objektoberfläche angepassten Gitter durchführt.

5.1 Geometrieobjekte

Die Software *InGrid3D* wurde erstellt, um für komplexe geometrische Objekte Gitter zur Strömungssimulation zu erzeugen. Zum Import der Objekte ins Programm wird das STL-Format benutzt. Beim STL-Format (*Standard Transformation Language*) handelt es sich um eine (Quasi-)Standardschnittstelle vieler CAD-Systeme (s. Anhang A.3). Sie beinhaltet die Beschreibung der Oberfläche von 3D-Körpern mit Hilfe von Dreiecksnetzen. Jedes Dreieck wird durch seine drei Eckpunkte und die zugehörige Flächennormale charakterisiert.
InGrid3D stellt an das Objekt zusätzliche Anforderungen, so muss die Oberfläche zunächst geschlossen sein, anschaulich könnte man auch von „wasserdicht" sprechen. Konkret bedeutet dies, dass keine Lücken zwischen Dreiecksflächen existieren dürfen. Ebensowenig dürfen doppelte Dreiecksflächen vorliegen, dies würde später bei der Parametrisierung zu Problemen führen. Die Einhaltung dieser Kriterien wird beim Import der Objekte überprüft. In der PG wurden die verwendeten Objekte mit Hilfe der Software Blender (s. Kap. 3.7) erstellt. Ein Beispiel zum Erstellen einer einfachen Geometrie kann Anhang A.2.1 entnommen werden.

[2] Der Mittelpunkt entspricht hierbei dem Punkt auf halber Länge der Snake.

[3] Ein weiteres Kriterium welches zur Entwicklung der Glätter führte, war die Tatsache dass einige Löser mit extrem spitzen oder extrem flachen Winkeln innerhalb der Hexaeder nicht funktionieren.

5.2 Grobgittergenerator

Der Grobgittergenerator *HaGrid3D* dient der möglichst einfachen manuellen Erzeugung eines guten Grobgitters. Die Evaluierung von *Cubit* (s. Kap. 4.3) ergab, dass es für die Aufgaben der PG nicht ausreichend ist. Somit wurde es notwendig, ein eigenes Werkzeug für diese Aufgabe zu implementieren.

5.2.1 Anforderungen

Die wesentlichen Aufgaben dieses Programms sollten hierbei folgende Punkte sein:

- Platzierung des Objektes im Grobgitter,
- Markierung von Knoten, die sich am Rand des Kanals befinden,
- Markierung von Knoten, die sich innerhalb des Objektes befinden,
- Markierung von Knoten, die sich auf dem Objektrand befinden,
- Setzen der Koordinaten einzelner Knoten auf die Koordianten eines Geometrie-Eckpunktes und
- die manuelle Verbesserung von Knotenpositionen.

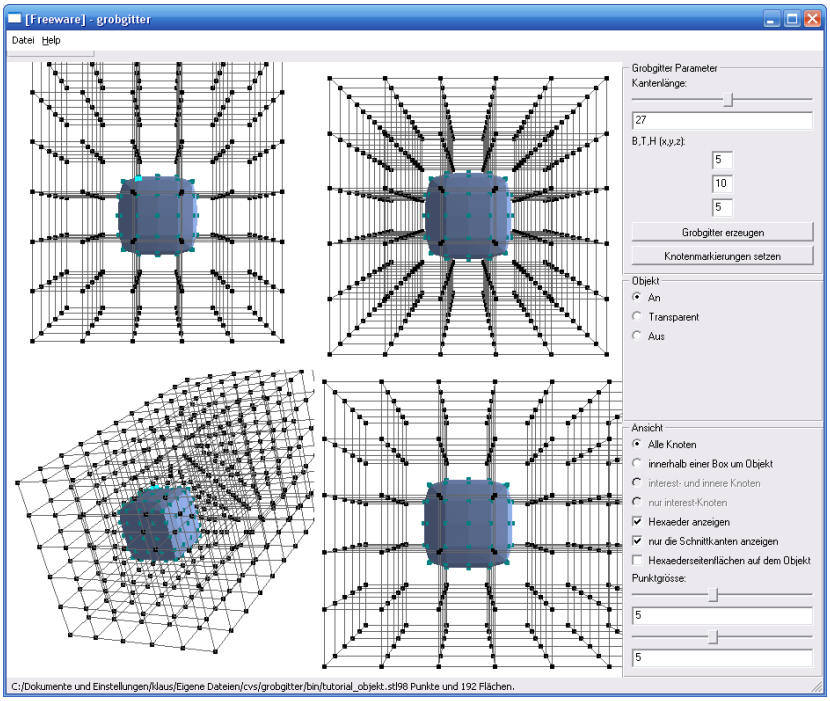


Abbildung 5.1: *Die Benutzeroberfläche des Grobgitter-Werkzeugs* HaGrid3D.

5.2.2 Aufbau des Programms

Die grafische Oberfläche bietet vier verschiedene Ansichten auf das Objekt und das Grobgitter, wobei drei davon 2D-Ansichten sind, die Vorder-, Seiten- und Draufsicht realisieren (s. Abb. 5.1). Die vierte Ansicht ermöglicht eine perspektivische Darstellung. Alle Ansichten können frei konfiguriert werden. Die Knoten werden mit der Maus ausgewählt und in Kombination mit zusätzlichen Tastaturbefehlen werden die jeweiligen Funktionen umgesetzt. Der Import der Geometrie erfolgt mittels einer STL-Schnittstelle. Für das Grobgitter ist das Laden, Speichern und Exportieren nach *InGrid3D* vorgesehen.
Die Hauptfunktionalität besteht darin, die Koordinaten eines Grobgitterknotens auf die Koordinaten eines Geometrie-Eckpunkts zu setzen und gesondert zu markieren, damit sie später bei der Parametrisierung in *InGrid3D* entsprechend behandelt werden können. Alle Grobgitterknoten werden zunächst einem *Inklusionstest* unterzogen, um die Knoten innerhalb des Objektes als *solid* zu markieren. Mit dieser Information kann auch die Menge der Grobgitterknoten, die auf den Objektrand zu bewegen sind, wesentlich eingeschränkt werden. Während des manuellen Verschiebens von Grobgitterknoten ermöglicht eine zuschaltbare Visualisierung der am Objekt anliegenden Hexaederseitenflächen eine Sichtprüfung auf die Wasserdichtigkeit des Grobgitters.

Nach dem Starten des Grobgittergenerators muss zunächst ein Objekt in Form einer Oberflächentriangulierung geladen werden. Anschließend kann das standardmäßig erzeugte Grobgitter angepasst werden, indem die Anzahl der Hexaederzellen in X-, Y- und Z-Richtung und die Kantenlänge der Hexaederzellen verändert werden. Mit diesen Informationen wird ein Grobgitter erzeugt, das aus gleichförmigen Hexaederwürfeln besteht. Das Objekt liegt im Zentrum des Grobgitters und kann mittels Maus in den 2D-Fenstern in alle Richtungen verschoben werden. Befindet sich das Objekt an der gewünschten Position, können im folgenden Schritt mit Hilfe des *Inklusionstest* die Knoten als *solid* markiert werden, die sich im Inneren des Objektes befinden. Die so markierten Knoten beschränken den Objektrand von innen und sind gleichzeitig geeignete Knoten für die Oberflächenapproximation. Ebenso sind natürlich genau die Knoten mögliche Kandidaten, die selber nicht als *solid* markiert sind, aber einen direkten Nachbarn mit *solid*-Markierung haben. Diese werden in *HaGrid3D* auch *interessante* Knoten genannt und können hervorgehoben dargestellt werden.
Zum Abbilden werden einzelne Knotenpaare (jeweils ein Knoten des Grobgitters und einer des Geometrieobjekts) markiert und auf dem Objektrand zusammengeführt, so dass nach und nach immer mehr Hexaederseitenflächen des Grobgitters das Objekt approximieren. Die jeweils vier auf das Objekt verschobenen Knoten einer Hexaederseitenfläche liegen auf dem Objektrand und die Fläche selbst spiegelt eine Vergröberung der Oberfläche des Objektes wider. Werden jetzt mehr und mehr dieser Flächen erzeugt, muss sichergestellt werden, dass die hierbei erzeugte Approximation des Objektes durch Hexaederseitenflächen eine geschlossene Form annimmt. Eine komplett geschlossene Form wird auch wasserdicht genannt und zur manuellen und visuellen Kontrolle werden alle bereits aufliegenden Hexaederseitenflächen rot eingefärbt (s. Abb. 5.2). Der aktuelle Arbeitsfortschritt der Grobgittererzeugung per Hand kann jederzeit abgespeichert und wieder eingelesen werden. Das fertige Grobgitter kann über die Exportfunktion in einer Binärdatei gespeichert werden, die dann von *InGrid3D* eingelesen werden kann.

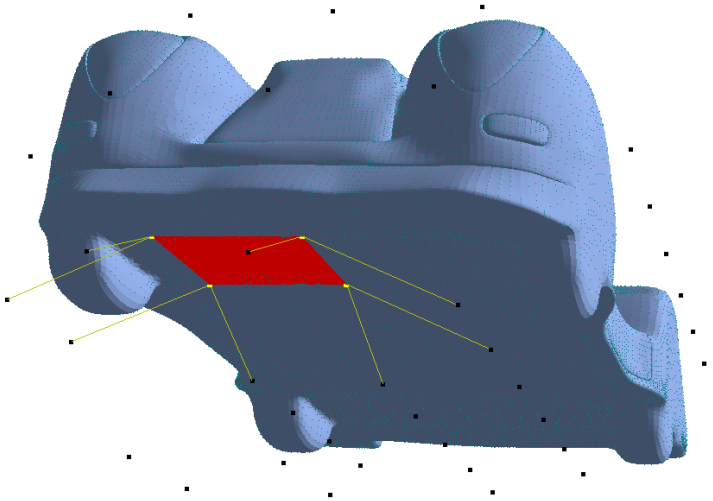


Abbildung 5.2: *Eine bereits auf das Objekt gezogene Hexaederseitenfläche wird rot und alle inzidenten Kanten der beteiligten Knoten werden gelb eingefärbt.*

5.2.3 Verschieben von Grobgitterknoten auf Geometrie-Eckpunkte

Durch das von OpenGL unterstützte *Picking* können Knotenpaare mit der Maus ausgewählt werden. Durch einen Mausklick wird von der Bildschirmoberfläche aus gesehen ein Strahl in den dahinter liegenden dreidimensionalen abgebildeten Raum definiert. OpenGL erfasst nun alle Elemente innerhalb einer festgelegten Umgebung dieses Strahles und gibt sie in Form einer Liste aus. Je nach gedrückter Tastenkombination kann ein Grobgitterknoten oder ein Objektknoten ausgewählt werden. Wurde zuerst ein Grobgitterknoten selektiert, dann wird beim Selektieren eines Objektpunktes derjenige innerhalb der Liste mit der geringsten euklidischen Distanz zum zuvor gewählten Grobgitterknoten selektiert. Dies ist notwendig, da beim *Picking* teilweise sehr viele Objektknoten getroffen werden.

5.2.4 Interne Darstellung des Grobgitters

Die Grobgitterknoten werden in einem dreidimensionalen Feld gespeichert, über das mit drei verschachtelte *for*-Schleifen iteriert werden kann. Die Anzahl kann über die Benutzeroberfläche festgelegt werden, ebenso die Länge der gleichförmigen Hexaederkanten. Zu jedem Knoten werden intern im Programm seine Koordinaten im Raum, der Status der aktuellen Sichtbarkeit, die Information, ob er zu den interessanten Knoten gehört und sein aktueller *solid*-Status gespeichert. Falls der Grobgitterknoten auf dem Objektrand liegt, wird der Index des entsprechenden Objektknotens ebenfalls gespeichert. Zu jedem Grobgitterknoten werden ausserdem die so genannten *Default*-Werte gespeichert. Diese werden bei der Initialisierung gesetzt und repräsentieren die ursprünglichen Koordinaten und den *solid*-Status bei der Erstellung des Grobgitters. Dies ermöglicht ein späteres Zurücksetzen des Knotens auf seine ursprüngliche Position.

Die Hexaederzellen selbst werden ebenfalls in einem dreidimensionalen Feld gespeichert. Die Anzahl ist abhängig von den Grobgitterknoten und muss nicht durch den Benutzer festgelegt werden.

5.2.5 Im- und Export

Alle Knoten des Grobgitters werden in einem dreidimensionalen Feld im Speicher gehalten. Beim Speichern und Laden wird das Feld in drei verschachtelten Schleifen durchlaufen und alle Eigenschaften in einer ASCII-Datei gespeichert. Es ist also möglich die Eigenschaften eines jeden Knotens mit einem Texteditor auszulesen.

Durch das Exportieren wird eine Schnittstelle zwischen dem Grobgittergenerator und *InGrid3D* geschaffen, um das erzeugte Grobgitter dort laden und weiter verwenden zu können. Die Datenstrukturen werden binär gespeichert, um sie zu einem späteren Zeitpunkt in *InGrid3D* einlesen zu können. Der Grobgittergenerator durchläuft die schon im Speicher vorhandenen Knoten-Instanzen und schreibt die Informationen in eine Datei. Die Dateiformate werden im Anhang A.3 im Detail vorgestellt.

5.2.6 Visualisierung

Mit zunehmender Anzahl an Hexaederzellen verliert die Darstellung des Grobgitters stark an Übersichtlichkeit (s. Abb. 5.3). Um dem entgegenzuwirken wurden verschiedene Ansätze diskutiert. Zunächst wurden die vier verschiedenen Ansichten (s. Abb. 5.1) implementiert. In den 2D-Fenstern wird die Ansicht per Tastendruck um 180 Grad gedreht, um damit auf die jeweilige Rückseite zu schalten. Die perspektivische Ansicht wird auf Tastendruck in den Vollbildmodus gebracht.
Ein weiterer Punkt zur Verbesserung der Übersicht ist das Anzeigen nur der interessanten Knoten. Per Auswahlfeld lassen sich in der Bedienungsoberfläche die restlichen Knoten ausblenden, da sie die Sicht auf das Objekt versperren können. Interessante Knoten sind solche, die auf einer Kante liegen, die den Objektrand schneidet. Sie werden über die *solid*-Markierung identifiziert.
Ebenso lässt sich die Übersicht verbessern, indem nicht alle Hexaederzellen angezeigt werden müssen, um ein Grobgitter zu erzeugen. So kann die Anzahl von Hexeaderzellen deutlich reduziert werden, indem nur diejenigen angezeigt werden, die sich in einer definierten Umgebung des Objektes befinden. Hierzu kann in der Bedienungsoberfläche ausgewählt werden, ob alle Zellen zu interessanten Knoten inzident sind (s. Abb. 5.4), oder alle, die sich innerhalb eines fest um das Objekt definierten Quaders befinden, angezeigt werden sollen. Es ist auch möglich, sämtliche Hexeaderzellen zu verstecken, oder das Objekt selbst zu verstecken (s. Abb. 5.5) bzw. transparent darzustellen.

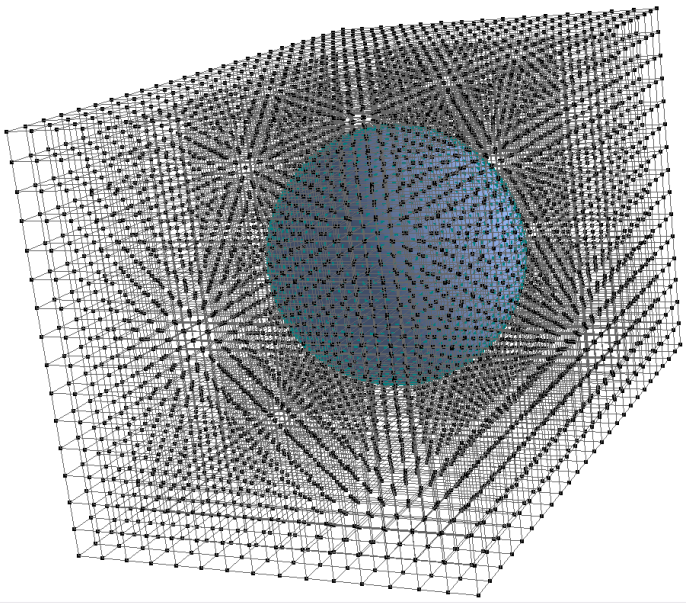


Abbildung 5.3: *Ein Grobgitter mit 15 x 15 x 15 Hexaederzellen.*

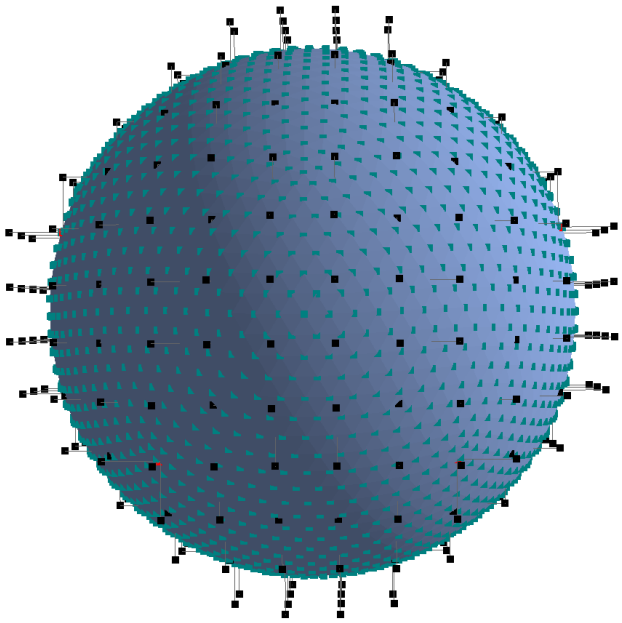


Abbildung 5.4: *Ein Grobgitter mit 25 x 25 x 25 Hexaederzellen, es werden nur die interessanten Knoten angezeigt.*

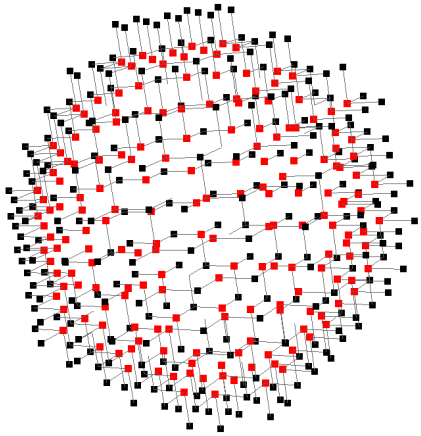


Abbildung 5.5: *Ein Grobgitter mit 25 x 25 x 25 Hexaederzellen, es werden nur interessante Knoten mit den Kanten, die das Objekt schneiden angezeigt. Das Objekt ist ausgeblendet.*

5.2.7 Inklusionstest

Der Inklusionstest soll für jeden Knoten des initialen Grobgitters festlegen, ob er innerhalb oder außerhalb des Geometrieobjekts liegt. Zu diesem Zweck wird ein beliebiger, zufälliger Referenzpunkt außerhalb des Grobgitters und damit auch außerhalb des Geometrieobjekts ausgewählt. Für jeden Punkt des Grobgitters und jedes Dreieck des Objektes wird anschließend geprüft, ob die Strecke zwischen dem Referenzpunkt und dem auf Einschluss zu prüfenden Punkt das jeweilige Dreieck schneidet. Hierbei wird die Anzahl der Schnitte pro Grobgitterpunkt gezählt – über sie kann bestimmt werden, ob der Punkt eingeschlossen ist oder nicht. Bei ungerader Anzahl von Schnitten liegt der Punkt innerhalb des Feinnetzes, bei gerader außerhalb.

Schnitttest

Seien $A = (x_a, y_a, z_a), B = (x_b, y_b, z_b), C = (x_c, y_c, z_c)$ und $D = (x_d, y_d, z_d)$ Punkte im $\mathbb{R}^3$. Das vorzeichenbehaftete Volumen des Tetraeders $DABC$, im Folgenden mit $[DABC]$ bezeichnet, ist wie folgt definiert:

$$[DABC] = \frac{1}{6} \begin{vmatrix} x_a - x_d & y_a - y_d & z_a - z_d \\ x_b - x_d & y_b - y_d & z_b - z_d \\ x_c - x_d & y_c - y_d & z_c - z_d \end{vmatrix}$$

Falls das Volumen $[DABC]$ ein positives Vorzeichen hat, bedeutet dies, dass die Punkte A, B und C vom Punkt D aus betrachtet entgegen des Uhrzeigersinns orientiert sind.
Für zwei Punkte Q und Q' kann zunächst leicht überprüft werden, ob sie auf verschiedenen Seiten des zu prüfenden Dreiecks ABC liegen: Dies ist nämlich genau dann der Fall, wenn genau ein Vorzeichen der Volumina $[QABC]$ und $[Q'ABC]$ positiv und das andere negativ

ist. Ebenso kann mit Hilfe des Volumens überprüft werden, ob ein Schnittpunkt zwischen der Strecke QQ' und dem Dreieck ABC existiert:
Seien ABC ein Dreieck und QQ' eine Strecke im $\mathbb{R}^3$, wobei Q und Q' sich auf verschiedenen Seiten des Dreiecks befinden. Außerdem seien die Punkte des Dreiecks so geordnet, dass das Volumen $[QABC]$ positiv ist. Die Strecke QQ' schneidet das Dreieck genau dann, wenn gilt:

$$[Q', B, A, Q] \geq 0 \wedge [Q, B, C, Q'] \geq 0 \wedge [Q, C, A, Q'] \geq 0$$

Anders ausgedrückt: Die Strecke QQ' schneidet das Dreieck ABC genau dann, wenn die Dreiecke ABQ, CBQ und ACQ von Q' aus betrachtet entgegen des Uhrzeigersinns orientiert sind (s. Abb. 5.6). Ist eines der drei Volumina 0, so bedeutet dies, dass Punkt Q' koplanar zum Dreieck ABC ist. Bei zwei Nullwerten schneidet die Strecke QQ' das Dreieck in einer Dreieckskante – bei drei Nullwerten in einem der Eckpunkte des Dreiecks.

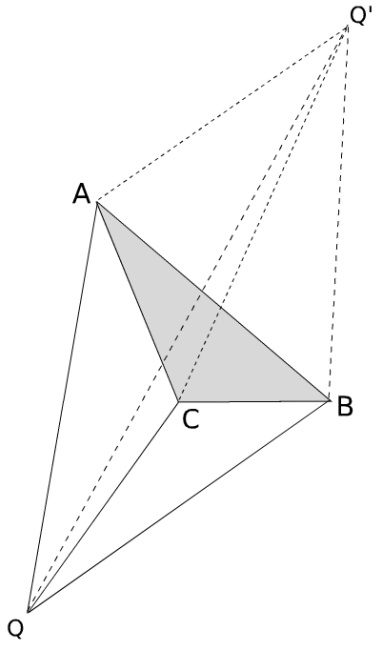


Abbildung 5.6: *Im Schnitttest verwendete Bezeichner.*

Implementierung

Der Inklusionstest ist über drei Funktionen implementiert. Die erste berechnet die Volumina von Tetraedern nach der zuvor beschriebenen Methode. Die zweite überprüft auf bereits beschriebene Art und Weise auf Schnitt zwischen einer Strecke und einem Dreieck. Ebenso überprüft sie im Falle eines Schnittes ob koplanar, in einer Kante oder einem Punkt geschnitten wird. Für diese drei Fälle wird der Schnitttest mit einem anderen Referenzpunkt wiederholt, da er keinem Dreieck eindeutig zuordbar ist. Die letzte Methode erzeugt zunächst einen randomisierten Referenzpunkt Q'. In einer äusseren Schleife werden dann alle Punkte des Grobgitters durchlaufen. In der zugehörigen inneren Schleife wird über alle Dreiecke der Triangulierung

iteriert, ihre Koordinatenwerte werden abgefragt und der Funktion, die auf Schnitt prüft zusammen mit den beiden Punkten, die die Strecke festlegen, übergeben. Je nach Ausgabe wird entweder die Anzahl der Schnitte für diesen Grobgitterpunkt um eins erhöht und beim nächsten Dreieck weitergearbeitet oder ein anderer Referenzpunkt für dasselbe Dreieck erzeugt. Abschliessend wird überprüft, ob die Anzahl der Schnitte gerade (Punkt liegt außerhalb des Feinnetzes) oder ungerade ist, was zu einer Markierung als *solid* führt.

5.3 Parametrisierung

Um eine Reihenfolge der Elemente innerhalb der maximal unabhängigen Menge von Knoten zu bestimmen werden alle Knoten des Objektes durchlaufen. Falls die Anzahl der ausgehenden Knoten zwölf nicht überschreitet, wird die diskrete Krümmung bestimmt und mit einer Referenz zu der Knoten-ID aus OpenMesh (s. Kap. 3.6) in einer Liste gespeichert. Die nach Krümmung und Flächeninhalt sortierte Liste liefert die Reihenfolge, in der die Knoten abgearbeitet werden.

Wird ein Knoten des Objektes entfernt, entsteht ein Loch durch die fehlenden anliegenden Dreiecke. Durch eine Retriangulierung des Loches verändert sich dabei die diskrete Krümmung der an diesem Loch anliegenden Knoten. Die Knoten müssen eine unabhängige Menge (engl. *independent set*) bilden, um die Reihenfolge sinnvoll nutzen zu können. Daher wird abschließend die Liste durchlaufen und jeweils alle Knoten, deren Krümmung sich geändert hat, aus ihr entfernt. So bleibt die Reihenfolge erhalten und eine unabhängige Menge entsteht. Der Vergröberungsprozess beinhaltet gleichzeitig die Parametrisierung und somit zwei Schritte. Zunächst wird die Entfernung des Knotens und Retriangulierung der Eins-Ring-Nachbarschaft im Zweidimensionalen vorgenommen, anschließend wird der Mittelpunkt und alle vorher parametrisierten Knoten in die neue Triangulierung eingebettet. Dabei ist zu beachten, dass es sich bei dem Grad der Vergröberung um eine Heuristik handelt, der vom Anwender variiert werden kann.

5.3.1 Retriangulierung

Für den Vergröberungsprozess wird sequentiell die oben gebildete Liste durchlaufen und jeder Knoten abgearbeitet. Dabei wird zuerst die Eins-Ring-Nachbarschaft um den Knoten herum durch konforme Einbettung in 2D eingebettet. Dabei werden nicht nur die anliegenden Dreiecke eingebettet, sondern auch alle schon vorher parametrisierten Knoten des Netzes. Die konforme Einbettung minimiert dabei die auftretenden Verzerrungen. Nachdem der Mittelpunkt entfernt wurde, wird eine Triangulierung der Eins-Ring-Nachbarschaft des zuvor entfernten Punktes benötigt. Hier wird zuerst die einfache Methode des Ohrenabschneidens (engl. *ear-cutting*) benutzt. Diese liefert immer eine gültige Triangulierung. Als Implementierung wurde auf eine freie Bibliothek von O'Rourke zurückgegriffen (s. Kap. 2.2.4). Dieser Zugang erzeugt jedoch schlechte Triangulierungen bezüglich des minimalen Winkels, so dass eine Nachbearbeitung durch das Kantenkippverfahren (engl. *edge flipping*) vorgenommen wird. Um die Triangulierung im Dreidimensionalen am Objekt anwenden zu können, wird diese nach dem Delauney-Kriterium optimiert.

5.3.2 Einbettung

Nachdem eine Triangulierung gefunden wurde, werden alle vorher eingebetteten Knoten und der neu entfernte Mittelpunkt in den neuen Dreiecken parametrisiert. Dazu werden alle vorher eingebetteten Punkte durchlaufen und das umschließende Dreieck aus der Triangulierung gesucht. Die Einbettung wird für jeden Knoten durch die baryzentrischen Koordinaten bezüglich des umschließenden Dreiecks abgespeichert. Um die Konnektivität der eingebetteten Punkte zu erhalten, wird eine Kopie des Feinnetzes gespeichert.

5.4 Snakes

5.4.1 Initialisierung

Um eine Snake zwischen zwei beliebigen Punkten auf der Oberflächentriangulierung zu initialisieren wird der Single-Source-Shortest-Path-Algorithmus von Dijkstra [8] benutzt. Zusätzlich wurde das Snake-Konzept um die Möglichkeit erweitert, nicht nur Punkte der Oberflächentriangulierung als Start- und Endpunkte zu benutzen, sondern die Punkte auch frei auf einem Dreieck positionieren zu können. Der Dijkstra-Algorithmus liefert eine Menge von Punkten, die miteinander verbunden einen Pfad bzw. kürzesten Weg bilden. Diese Punkte werden gemäß ihrer Reihenfolge durchlaufen, und auf allen ausgehenden Kanten dieser Punkte auf einer Seite des Pfades werden die Snaxel gesetzt. Somit ist sichergestellt, dass alle Snaxel gleichgerichtet sind (s. Abb. 5.7).

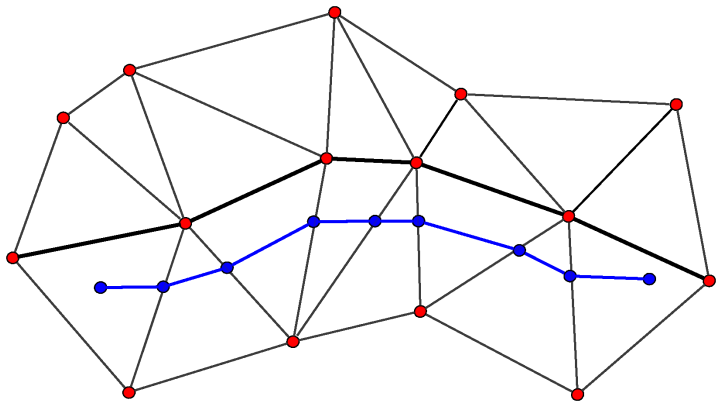


Abbildung 5.7: *Der durch den Dijkstra-Algorithmus ermittelte kürzeste Weg (fett) auf der Oberflächentriangulierung dient der Snake (blau) zur Initialisierung. Alle Snaxel werden auf der gleichen Seite des Pfades erzeugt.*

5.4.2 Bewegung

Die Bewegung der Snake ist ein iterativer Prozess. Zunächst wird jedem Snaxel über die im Grundlagenkapitel 2.5.3 skizzierte Vorgehensweise eine Geschwindigkeit zugewiesen. Dann werden alle Snaxel auf ihren Halbkanten gemäß ihrer errechneten Geschwindigkeit verschoben, wobei darauf geachtet wird, dass maximal ein Snaxel das vordere oder hintere Ende einer Kante erreicht. Sollte dies geschehen, so überquert der Snaxel das Ende der Kante, und auf den

$(n-1)$ inzidenten Halbkanten werden neue Snaxel erzeugt. Hierbei auftretende Verletzungen der Konsistenzbedingungen werden gemäß den Erläuterungen in Kapitel 2.5.3 behandelt.

Diese Schritte werden solange wiederholt, bis eines der Abbruchkriterien greift (s. Abb. 5.8), welche im folgenden Kapitel dargestellt werden.

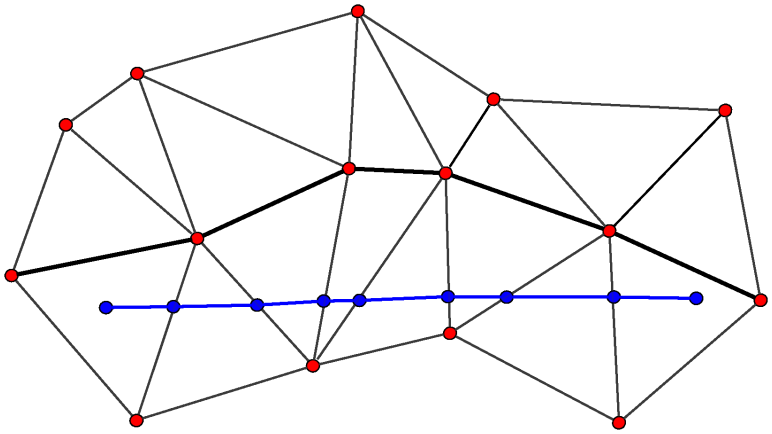


Abbildung 5.8: *Die Snake hat ihre Zielposition erreicht, beide Abbruchkriterien treffen zu.*

5.4.3 Abbruchkriterien

Um zu erkennen, wann die Snake ihr Ziel erreicht hat, wurden zwei Abbruchkriterien implementiert. Das erste Kriterium ist dabei die Geschwindigkeit der Snaxel: Sobald dieser Wert für alle Snaxel so weit gesunken ist, dass kaum noch eine Bewegung stattfindet, werden die Iterationen abgebrochen. Da die Geschwindigkeiten in jeder Iteration neu berechnet werden müssen, wird auch diese Kontrolle in jeder Iteration durchgeführt.
Das zweite Kriterium bezieht sich auf die Länge der Snake – wenn keine weitere Minimierung mehr erfolgt, können die Iterationen ebenfalls beendet werden. Da die Berechnung der Länge rechenintensiver als die Geschwindigkeitsvergleiche ist[4], wird diese nur nach jeweils 1000 Iterationen überprüft. Das Längenkriterium ist unerlässlich, da es in Einzelfällen vorkommen kann, dass die Snake kontinuierlich zwischen zwei Positionen hin- und herspringt, dies ist zum Beispiel dann der Fall, wenn sich die Snake in einer „Zick-Zack"-ähnlichen Form (s. Abb. 5.9) befindet. Das Kriterium der Snaxelgeschwindigkeiten würde in einem solchen Fall nicht greifen.

[4]Die Länge der Snake berechnet sich aus der Summe der Längen der einzelnen Segmente. Die Länge der Segmente wird über die euklidische Distanz zweier benachbarter Snaxel bestimmt.

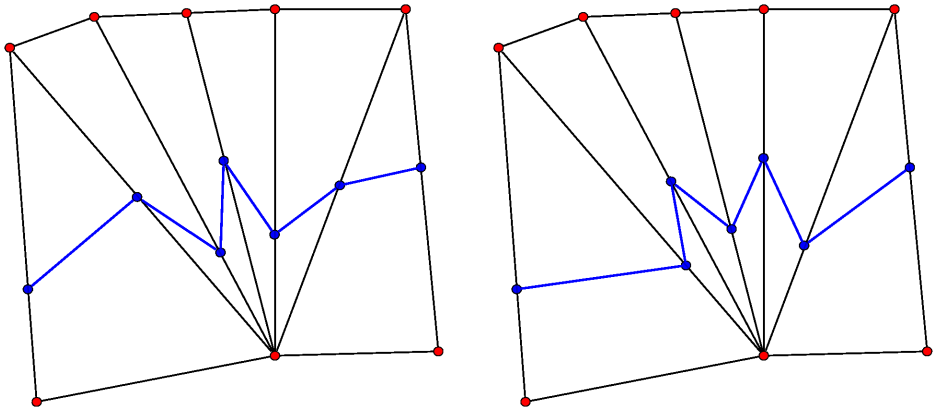


Abbildung 5.9: *Links: Die Snake befindet sich in einer „Zick-Zack"-Konfiguration. Rechts: Im nächsten Iterationsschritt „invertieren" die Snaxelpositionen, da die hohen lokalen Krümmungswerte für hohe Snaxelgeschindigkeiten sorgen. Die Snake schwingt also in der Nähe der Lösung.*

5.4.4 Grenzen des Konzeptes

Auf nahezu planaren Oberflächen arbeitet der Algorithmus sehr zuverlässig und liefert die erwarteten Ergebnisse, dennoch beinhaltet er zwei Einschränkungen. Die Snake bewegt sich nicht in größere Vertiefungen auf der Objektoberfläche, ebensowenig ist sie in der Lage über stärkere Ausbuchtungen zu wandern. Da der kürzeste Weg auf der Oberfläche gesucht wird, ist die erste Einschränkung immanent, so ist es beispielsweise günstiger, am Rand einer Ausbuchtung entlangzulaufen, als in die Ausbuchtung hineinzuwandern. Auch die zweite Einschränkung liegt hierin begründet, der Weg über eine Ausbuchtung würde die Länge der Snake kurzfristig stark vergrößern. Außerdem würde sich die Krümmung der Snake an einer solchen Stelle zu stark erhöhen, so dass das Überqueren letztendlich dadurch verhindert wird, dass die Snaxel dort eine der eigentlichen Bewegung entgegengesetzte Geschwindigkeit annehmen. In der Praxis bleibt die Snake dann am Rand der Ausbuchtung stehen, die Abbruchkriterien greifen, obwohl ein globales Optimum noch nicht erreicht wurde. Diese Einschränkungen treten insbesondere bei zu grob aufgelösten Geometrien auf, u.a. auch dann, wenn der Dijkstra-Pfad zu weit vom globalen Optimum entfernt ist (s. Abb. 5.10).
Die Grenzen des Konzeptes sind für den Einsatz in der Gitterunterteilung jedoch keine praktische Einschränkung, so dass die Implementierung der im Kapitel 2.5.1 vorgestellten Technik des *gradient vector flow* nicht nötig war. So wird die Snake nur auf kurzen, annähernd planaren Teilstücken der Objektoberfläche eingesetzt, die problematischen Aus- und Einbuchtungen wurden zuvor durch die Parametrisierung entfernt.

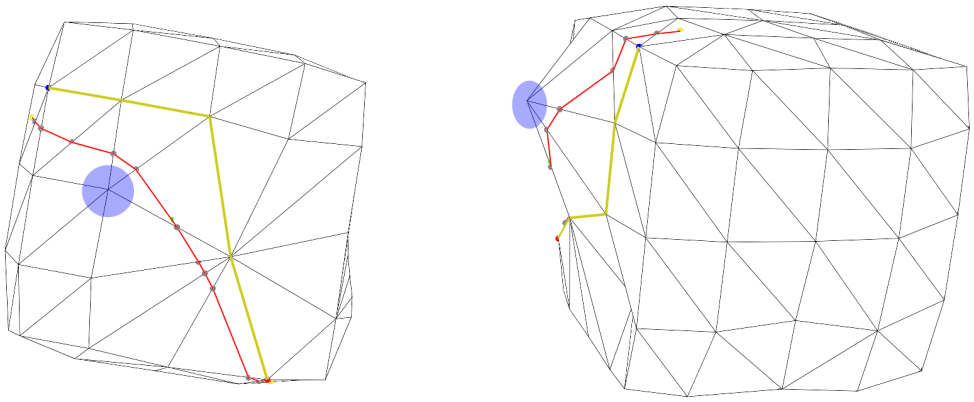


Abbildung 5.10: *Links: Die grob aufgelöste Geometrie führt zu einer ungünstigen Initialisierung der Snake – diese erreicht dadurch die optimale Lage nicht, sie hält vor einer Ausbuchtung (blaue Markierung) bereits frühzeitig an. Rechts: Zur Verdeutlichung die gleiche Situation aus einer anderen Perspektive – die Ausbuchtung ist nun deutlich zu erkennen.*

5.5 Gitterunterteilung

5.5.1 Datenstuktur

Die Gitterstruktur wird als polyedrischer Komplex gespeichert. Dabei bilden Knoten die nulldimensionalen, Kanten die eindimensionalen, Vierecke die zweidimensionalen und Hexaeder die dreidimensionalen Polyeder. Der Knoten besteht aus den Koordinaten im Raum, und der Information, ob er ein Objektrandpunkt ist bzw. eine *solid*-Markierung hat. Die Kanten werden über einen Startpunkt und einen Endpunkt definiert, die viereckigen Flächen werden über vier Kanten definiert, und die Hexaeder bestehen wiederum aus sechs Flächen. Die Kanten und die Flächen speichern, wie die Knoten, eine Information darüber, ob sie am Objektrand oder im Inneren des Objektes liegen, den sogenannten „Randstatus”:

- Standard: Der Polyeder liegt im freien Raum außerhalb des Objektes.
- Objektrand: Der Polyeder liegt auf der Objektoberfläche.
- Objektinneres: Der Polyeder liegt im Inneren des Objektes.
- Rand: Der Polyeder liegt auf dem äußeren Rand bzw. auf dem Windkanalrand. Im Gegensatz zur Objektoberfläche ist die Unterteilung hier trivial, man braucht keine Parametrisierung.

Die Datenstrukturen verwalten Referenzen auf die inzidenten Polyeder der jeweils nächsthöheren Dimension. Ein Knoten hat bis zu sechs Referenzen auf inzidente Kanten, Kanten bis zu vier Referenzen auf inzidente Flächen, und Flächen bis zu zwei Referenzen auf inzidente Hexaeder. Diese Referenzen vereinfachen und beschleunigen viele Algorithmen auf dieser Datenstruktur.

5.5.2 Ablauf der Gitterunterteilung

Kopieren der alten Knoten

Zunächst werden im ersten Schritt alle Knoten des alten Gitters in das neue Gitter kopiert.

Unterteilung der Kanten

Im Mittelpunkt jeder einzelnen Kante aus dem alten Gitter wird ein Knoten erzeugt und der Knotenmenge des neuen Gitters hinzugefügt. Der Randstatus dieses Knotens wird auf den Randstatus der unterteilten Kante gesetzt. Liegt die zu unterteilende Kante auf dem Objektrand, so wird die Mitte der Kante über die Oberflächenparametrisierung bestimmt, ansonsten wird die arithmetisch Mitte bestimmt.

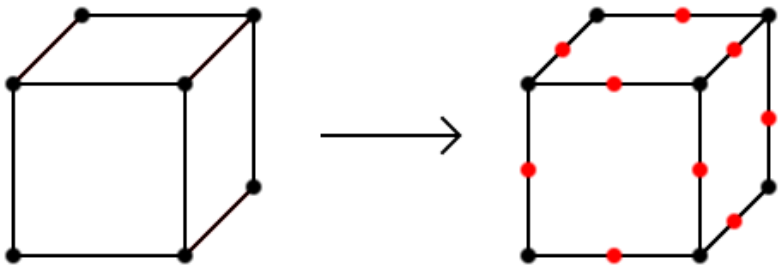


Abbildung 5.11: *Unterteilung der Kanten.*

Dann werden zwei Kanten erzeugt, von den Endknoten der alten Kante zum neu erzeugten Knoten in der Mitte der alten Kante. Die alte Kante wird verworfen, in das neue Gitter werden nur die neuen, kürzeren Kanten übernommen. Diese beiden Kanten erhalten den Randstatus der alten Kante.

Unterteilung der Flächen

In der Mitte jeder Fläche aus dem alten Gitter wird ein neuer Knoten erzeugt und im neuen Gitter gespeichert. Falls die Fläche eine Objektrandfläche ist, so wird die Knotenposition über die Oberflächenparametrisierung bestimmt. Ansonsten wird der Knoten als arithmetisches Mittel der vier Eckknoten der Fläche erzeugt. Alternativ kann auch das arithmetische Mittel der vier neu erzeugten Knoten auf den Kanten berechnet werden, was in diesem Fall zum exakt gleichen Ergebnis führt. Diesen Knoten wird als Randstatus der Status der zu unterteilenden Fläche zugewiesen. Als nächstes werden vier Kanten im Inneren der alten Fläche erzeugt. Die Startknoten dieser Kanten sind die Knoten, die bei der Unterteilung der Kanten neu erzeugt wurden. Diese liegen auf die Mitte der Seitenkanten der aktuell betrachteten Fläche und wurden bereits bei der Unterteilung der Kanten erzeugt. Der Endpunkt der vier neuen Kanten ist jeweils der neu erzeugte Knoten in der Mitte der aktuell betrachteten Fläche. Der Randstatus der alten Fläche wird auf diese vier Kanten übertragen.

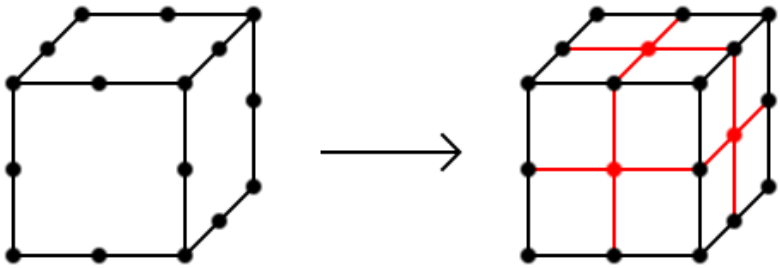


Abbildung 5.12: *Unterteilung der Flächen.*

Insgesamt werden die vier Kanten, die die alte Fläche definieren, in acht Kanten unterteilt, und zusätzlich vier Kanten im Inneren der alten Fläche erzeugt. Aus diesen zwölf Kanten werden nun die vier neuen Flächen definiert, und die alte Fläche wird verworfen. Diese vier Flächen bekommen den gleichen Randstatus wie die ursprüngliche Fläche.

Unterteilung der Hexaeder

Zunächst muss entschieden werden, ob der Knoten, der in der Mitte des alten Hexaeders erzeugt wird, im Objektinneren liegt oder nicht. Hat eine der begrenzenden Flächen des Hexaeders den Status „Objektinneres", befindet sich das Innere des Hexaeders im Inneren des Objektes, und damit auch der im Inneren des Hexaeders erzeugte Knoten. Falls alle sechs Flächen des Hexaeders den Status „Objektrand" haben, ist der innere Knoten ebenfalls im Objektinneren, da dieser Hexaeder dann einen einzelnen unabhängigen Würfel im Raum darstellt. Um nun einen Hexaeder in acht neue, kleinere Hexaeder zu unterteilen, müssen zunächst wie bei den Flächen im Inneren des aktuell betrachteten Hexaeders neue Kanten erzeugt werden. Die Startknoten sind die Knoten, die bereits bei der Unterteilung der Flächen in der Mitte der sechs Randflächen des Hexaeders erzeugt wurden. Als Endpunkt wird diesen Kanten der neue Knoten in der Mitte des Hexaeders zugewiesen.

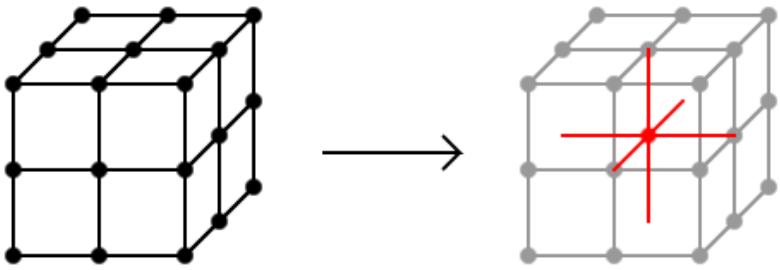


Abbildung 5.13: *Unterteilung der Hexaeder.*

Insgesamt erhält man 24 Kanten auf der Oberfläche des alten Hexaeders – im Inneren der Randflächen – und sechs Kanten im Inneren des alten Hexaeders. Aus diesen werden als nächstes zwölf Flächen im Inneren des alten Hexaeders konstruiert. Bei der Behandlung der Flächen

$(6 \cdot 4 = 24)$ wurden in jeder Randfläche des alten Hexaeders vier neue Flächen erzeugt, und hier nun zusätzlich zwölf innere Flächen im alten Hexaeder, also insgesamt 36 neue Flächen. Aus diesen werden die acht neuen Hexaeder definiert und der alte Hexaeder wird verworfen. Damit ist die Unterteilung abgeschlossen.

5.6 Gitteranpassung an das Objekt

Bei der Unterteilung des Gitters müssen neue Knoten zwischen zwei Objektknoten, im Folgenden Elternknoten genannt, auf die Oberfläche bewegt werden. Diese Aufgabe wird mit Hilfe der Parametrisierung gelöst. Die gesuchte Position dieses neuen Knotens ist dabei die Mitte eines optimalen Verbindungsweges zwischen den zwei Objektknoten der gröberen Unterteilungsstufe auf dem Objekt. Typischerweise kann mit Hilfe einer Parametrisierung ein guter Verbindungsweg gefunden werden. Dabei werden die Elternknoten in den Parameterraum projiziert und die Verbindungslinie zwischen den beiden eingebetteten Punkten im Parameterraum gesucht. Dabei kann es zu den folgenden vier Fällen kommen:

Fall 1: Einfache Einbettung

Die Elternknoten werden im selbem Grobdreieck parametrisiert. Da ein Grobdreieck immer konvex ist, liegt der Mittelpunkt der Verbindungsstrecke der beiden eingebetteten Elternknoten auch in diesem Dreieck (s. Abb. 5.14). Dieser Mittelpunkt wird nun auf die Oberfläche projiziert und als geeignete Mitte angenommen.

Fall 2: Benachbarte Grobdreiecke

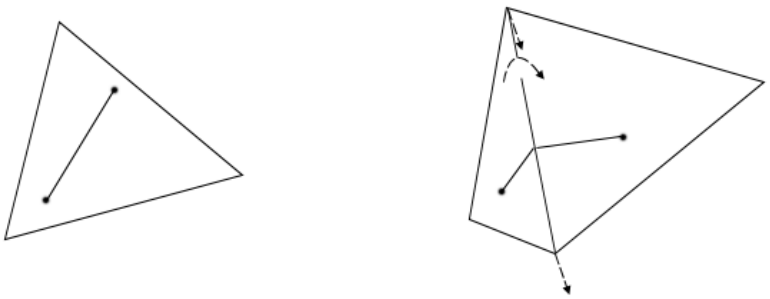
Die Elternknoten werden in benachbarten Grobdreiecken parametrisiert. Hier werden die beiden Dreiecke in eine Ebene abgebildet, wobei die Abbildungen kongruent zu den Ursprungsdreiecken bleiben. Da diese so genannte „Scharnier-Abbildung" (engl. *hingemap*) zweidimensional ist, ist es jetzt wieder möglich, die Verbindungsstrecke zwischen den eingebetteten Elternknoten zu berechnen. Falls die Mitte dieser Strecke in der Einbettung liegt, wird dieser Mittelpunkt wieder auf die Oberfläche projiziert (s. Abb. 5.14). Falls der Mittelpunkt nicht innerhalb eines der beiden Dreiecke liegt (s. Abb. 5.14), wird eine geeignete Eins-Ring-Nachbarschaft gesucht.

Fall 3: Eins-Ring-Nachbarschaft

Die Elternknoten liegen in zwei Grobdreiecken, die nur einen gemeinsamen Eckpunkt haben, oder es konnte im Fall zweier benachbarter Dreiecke kein Grobdreieck gefunden werden. In diesem Fall wird die konforme Einbettung der Eins-Ring-Nachbarschaft, dessen Mittelpunkt der gemeinsame Eckpunkt ist, berechnet (s. Kap. 5). In dieser Einbettung wird die Mitte der Verbindungsstrecke zwischen den eingebetteten Elternknoten berechnet (s. Abb. 5.14). Liegt diese innerhalb dieser Einbettung, wird der Punkt wieder auf die Oberfläche projiziert. Da die Einbettung der Eins-Ring-Nachbarschaft nicht konvex sein muss, kann auch hier der Mittelpunkt ausserhalb liegen. In diesem sehr seltenen Fall wird Fall 4 genutzt.

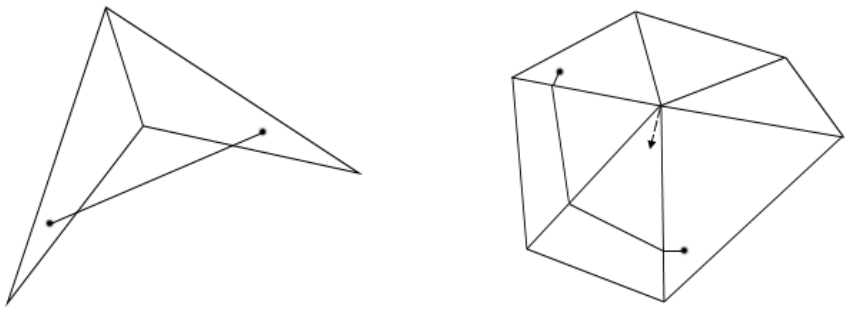
Fall 4: Snake

Die Elternknoten liegen in zwei disjunkten Grobdreiecken oder es wurde kein Grobdreieck in den jeweiligen Einbettungen gefunden. In diesem Fall kann die Parametrisierung keine Verbindungslinie zwischen zwei Knoten auf der Oberfläche berechnen und es wird eine Snake (s. Kap. 2.5) benutzt. Diese berechnet ein lokales Minimum der Verbindungswege auf dem Parameterraum zwischen den eingebetteten Elternknoten. Der Mittelpunkt von diesem Weg und das dazugehörige Dreieck wird als geeigneter Mittelpunkt angesehen und der parametrisierte Punkt auf der Objektoberfläche zum Grobgitter zurückgeliefert.



(a) *Eine einfache Einbettung.*

(b) *Eine* hingemap *kann erzeugt werden.*



(c) *Die Verbindungslinie liegt nicht innerhalb der* hingemap.

(d) *Eine Eins-Ring-Umgebung vor der Einbettung.*

Abbildung 5.14: *Mögliche Fälle bei der Parametrisierung.*

5.6.1 Implementierungsdetails

Ein großes Problem bei der Implementierung des oben beschriebenen Algorithmus bestand darin, den richtigen Fall zu ermitteln. Objektknoten, die bei der Vergröberung (s. Kap. 2.2.3) erhalten bleiben, gehören zu keinem Grobdreieck und können gleichzeitig in mehrere Grobdreiecke

eingebettet werden. Falls ein solcher Knoten als Elternknoten auftritt, müssen alle angrenzenden Grobdreiecke geprüft werden. Zunächst wird überprüft, ob die zwei Punkte adjazent zueinander sind. Ist dies nicht der Fall, wird für jedes mögliche Dreieck des ersten Eckpunktes eine Suche über die anliegenden Dreiecke durchgeführt, um eine mögliche „Scharnier"-Abbildung zu finden. Ist die Suche erfolglos, wird der Snake-Algorithmus benutzt.

5.7 Gitterglättung und Reparatur von Selbstdurchdringungen

Dieses Kapitel beschreibt Verfahren, mit deren Hilfe sich Durchdringungen in Gitterzellen reparieren oder die Gitterzellen im Allgemeinen nach bestimmten Kriterien manipulieren lassen. Die implementierten Glätter lassen sich grob in zwei Bereiche einteilen. Als erstes werden Glätter beschrieben, die alle Gitterknoten manipulieren können, mit Ausnahme von Knoten die auf der Objektoberfläche liegen. Dies ist eine Einschränkung, die zur Entwicklung des zweiten Bereiches geführt hat. Bei diesen Glättern können nur Knoten manipuliert werden, die auf der Objektoberfläche liegen. Zum ersten Bereich gehören die Reparatur von Gitter-Selbstdurchdringungen, die Laplace- und die Umbrella-Glättung. Zum zweiten Bereich gehören die Jacobi- und die Winkel-Glättung. Diese Verfahren werden im Folgenden genauer beschrieben.

5.7.1 Reparatur von Gitter-Selbstdurchdringungen

Das in Kapitel 2.4 vorgestellte Verfahren zur Behebung von Selbstdurchdringungen (engl. *mesh untangling*) wurde implementiert, um invertierte Elemente, die beim Unterteilen des Gitters oder bei der Randanpassung an das Objekt entstehen können, wieder in gültige Elemente zu überführen.

Implementierung

Die Hauptkomponenten des Reparatur-Moduls sind Funktionen zum Auffinden von invertierten Elementen, ein Löser für lineare Programme (*lpsolve*, s. Kap. 3.9) und eine Klasse zum Aufstellen des linearen Programms. Dem Auffinden von invertierten Elementen liegen die Kriterien aus Kapitel 2.4.2 zugrunde. Daher werden zuerst die Jacobi-Determinanten an den acht Eckknoten des Hexaeders berechnet. Falls die Anzahl negativer Jacobi-Determinanten größer oder gleich fünf ist, ist das Element invertiert. Ist die Anzahl negativer Jacobi-Determinanten null oder eins so wird das Element als gültig angesehen. Liegt die Anzahl negativer Jacobi-Determinanten zwischen diesen Werten, so muss getestet werden, ob es benachbarte Eckknoten gibt an denen negative Jacobi-Determinanten vorkommen. Es wird nun das Optimierungsproblem

$$\begin{gathered}\min \mathbf{c}^T\mathbf{y} \\ \text{unter der Bedingung } A\mathbf{y} = \mathbf{b}, \mathbf{y} > \mathbf{0}\end{gathered}$$

aufgestellt und gelöst. Die Komponente zum Aufstellen des linearen Programms erhält sukzessive einen der zuvor berechneten ungültigen Hexaeder als Eingabe. Es werden die Determinanten aus Kapitel (s. Kap. 2.4) berechnet, daraus ergibt sich die Matrix A und der Vektor $\mathbf{c}^T$. Hinzu kommt ein Vektor $\mathbf{b} \in \mathbb{R}^4$, dessen erste drei Komponenten Null sind und die letzte Eins. Damit lässt sich das primale LP aufstellen, welches an an den Löser weitergegeben wird, der

Lösungen für das primale und duale Problem errechnet. Die neuen Punktkoordinaten sind die ersten drei Komponenten des Lösungsvektors des dualen Problems und ersetzen die alten Koordinaten, so dass nachfolgende Berechnungen direkt die neuen Koordinaten verwenden können. Vor dem Durchführen der Reparatur-Prozedur wird das Gitter mit der intelligenten Laplace-Technik (engl. *smart Laplace*) vorgeglättet und nach der Durchdringungsbehebung noch einmal nachgeglättet. In der Praxis hat sich ein Wert von 30 Iterationen intelligenter Laplace-Glättung bewährt [13].

5.7.2 Laplace-Glättung

Zur Verbesserung der Gitterqualität wurde ein Laplace-Glätter in das Projekt integriert. Die Implementierung entspricht der Beschreibung des Verfahrens in Kapitel 2.3.3.

5.7.3 Umbrella-Glättung

Version 1

Der Umbrella-Glätter hat den Vorteil, dass er Längenverhältnisse der Nachbarkanten des betrachteten Knotens beibehält und somit keine Durchdringungen erzeugt. Weiterhin wird eine größere Umgebung betrachtet, da die Kantenlängen der Nachbarn des zu verschiebenden Knotens mit in die Berechnung der neuen Koordinaten eingehen (s. Kap. 2.3.3).

Version 2

Es hat sich als vorteilhaft erwiesen, den Glätter so zu modifizieren, dass Knoten auf der Oberfläche des Objekts nicht in die Berechnung der neuen Koordinaten eingehen, so dass Knoten, die sich in direkter Nachbarschaft zum Objektrand befinden, weiter in die vom Objektrand abweisende Richtung gezogen werden.

5.7.4 Heuristiken zur Gitterglättung auf der Objektoberfläche

Die zuvor beschriebenen Glätter sind nicht in der Lage, Knoten der Hexaederzellen auf der Oberfläche des Objektes zu manipulieren. Tritt dort ein Jacobi-Fehler auf, kann durch diese Glätter keine erfolgreiche Gitterkorrektur durchgeführt werden. Aus diesem Grund entwickelte die PG zwei heurisitische Ansätze, um Elementinvertierungen lokal zu beheben und die Innenwinkel der Randelemente zu optimieren.

Jacobi-Glättung

Verursacht eine Hexaederzelle einen Jacobi-Fehler, so wird versucht, für die Knoten dieser Zelle bessere Positionen auf dem Feinnetz zu finden und somit den Fehler zu beheben. Hierzu wird zu jedem dieser Knoten die Eins-Ring-Umgebung der Punkte auf dem Feinnetz auf bessere Positionen untersucht. Schlägt die Suche fehl, so wird in der Zwei-Ring-Umgebung gesucht, dieses Verfahren wird fortgesetzt bis hin zur Zehn-Ring-Umgebung. Wird für einen Knoten keine bessere Position gefunden, so wird die Routine auf den nächsten Knoten der Hexaederzelle angewandt. Sind alle Knoten der betroffenen Hexaederzelle auf Verbesserungsmöglichkeiten überprüft worden, so bricht der Glätter bei dieser Zelle ohne Erfolg ab – es sei denn, der Jacobi-Fehler wurde bereits durch eine Verschiebungen behoben.

Der Glätter kann lediglich solche Fehler korrigieren, die sich durch das Verschieben eines Knotens beheben lassen. Sollte es nötig sein zwei oder mehr Knotenpositionen einer Hexaederzelle zu ändern, so müssten alle möglichen Kombinationen dieser Positionen untersucht werden, dies würde zu exponentiellem Aufwand führen.

Winkel-Glättung

Ein wesentliches Qualitätsmerkmal eines Gitters sind die Winkel benachbarter Kanten an einem Knoten. Idealerweise haben zwei benachbarte Kanten an einem Knoten in einer Ebene einen Winkel von 90° zueinander. Je größer die Winkelabweichung an den Knoten ist, desto schlechter ist die Qualität des Gitters. Daher ist es sinnvoll, in den Bereichen starker Winkelabweichungen diese durch eine Positionsveränderung der Knoten zu verbessern. Wird durch das Verschieben des Knotens auf dem Feinnetz lokal das Winkelverhältnis verbessert, wird die neue Position übernommen. Für jeden Knoten auf der Objektoberfläche werden zunächst alternative Positionen berechnet und anschliessend bewertet. Zur Bestimmung der alternativen Positionen ist es entscheidend, wo sich der Knoten auf dem Objektrand befindet:

- Liegt er auf einem Punkt des Feinnetzes, so kommen für ihn als neue Positionen Punkte aus seiner Eins-Ring-Nachbarschaft in Frage.
- Liegt er innerhalb eines Feindreieckes, so sind die Eckpunkte des Dreiecks die möglichen Kandidaten für neue Positionen des betroffenen Knotens.

Die Bewertung basiert auf den Winkelabweichungen der Kanten an den jeweiligen Positionen. Ist eine der möglichen Positionen besser bewertet worden als die aktuelle Position, so wird der Gitterknoten an diese verschoben. Knoten an markanten Stellen der Objektoberfläche, wie zum Beispiel an Orten mit starker Oberflächenkrümmung, wurden häufig absichtlich bei der Grobgittererstellung dort positioniert, um Details der Geometrie besser aufzulösen. Da die Winkelglättung diese Knoten für weitere Gitterunterteilungsschritte ungünstig verschob, wurde als weiteres Kriterium die Oberflächenkrümmung in die Heuristik aufgenommen.

5.8 GPU Löser

In diesem Kapitel wird beschrieben, wie auf einem Gitter das Poissonproblem (s. Kap. 2.1) mit Hilfe einer Finite-Differenzen-Approximation realisiert und danach mit dem CG-Verfahren auf der GPU gelöst wird. Aus dieser diskreten Lösung wird der so genannte *potential flow* berechnet.

5.8.1 Datenstruktur

Da eine Finite-Differenzen-Matrix im vorliegenden Fall eine Bandstruktur besitzt (s. Kap. 2.1.6), ist es sinnvoll, anstelle der vollständigen Matrix nur die Inhalte der sieben Bänder zu speichern. Das Analogon zu Feldern auf der CPU sind Texturen auf der GPU. Daher werden die Bänder der Matrix zunächst mit Hilfe der Daten des zugrundeliegenden Gitters auf der CPU erzeugt und anschließend mit den Methoden der FEASTGPU-Bibliothek (s. Kap. 3.13) in Texturen gespeichert.

5.8.2 Implementierung

Die FEASTGPU-Bibliothek enthält bereits Methoden zum Lösen von Gleichungssystemen auf der GPU. Diese Methoden konnten unverändert übernommen werden. Die Komponente zur Matrix-Vektor-Multiplikation ist jedoch auf Struktur der Matrizen ausgelegt, die sich aus speziellen Finite-Elemente-Diskretisierungen ergibt. Die Methoden wurden als Ausgangspunkt zur Implementierung von angepassten Matrix-Vektor-Operationen auf der GPU verwendet. Dies ermöglicht es, die bereits von der Bibliothek bereitgestellte Implementierung eines Lösers für Gleichungssysteme nach dem CG-Verfahren zu nutzen.

Die Anpassung an die gewählte Matrix-Struktur erfolgte durch Implementierung zweier Shader (s. Kap. 2.6). Der Vertexshader berechnet die Ansichtstransformation und leitet die eingehenden Fragmentkoordinaten weiter. Der Fragmentshader berechnet aus diesen Koordinaten die Texturkoordinaten der Nachbarn des jeweils betrachteten Fragments. Um zum Beispiel das links vom aktuellen Fragment liegende zu bestimmen, müssen innerhalb der Textur die Koordinaten des Fragments links vom aktuellen bestimmt werden. Hierbei muss beachtet werden, dass es notwendig sein kann innerhalb der Textur um eine Zeile nach unten und dort zum ersten Fragment zu springen. Dieses gilt analog für die übrigen Nachbarn und wurde durch Verwendung von Modulo-Arithmetik realisiert. Somit ist die Ausführung einer Matrix-Vektor-Multiplikation möglich (s. Kap. 2.6.3).

Zur Berechnung des *potential flows* wurde ein weiterer Fragmentshader implementiert. Als Vertexshader diente der gleiche, der auch schon bei der Realisierung der Matrix-Vektor-Multiplikation verwendet wurde. Der Fragmentshader berechnet zunächst aus den eingehenden Koordinaten die Texturkoordinaten der Nachbarn des jeweiligen Punktes und kann mit diesen die partiellen Ableitungen berechnen.

5.9 Visualisierung

In diesem Kapitel wird die Visualisierung der Ergebnisse des GPU-Lösers beschrieben.
Der Löser berechnet ein Skalarfeld von Druckwerten für ein Gitter. Daraus läßt sich ein Geschwindigkeitsvektorfeld berechnen. Diese Größen sind einzeln oder gemeinsam in *InGrid3D* darstellbar, die Daten werden für einzelne Schichten des Gitters gezeichnet. Es ist möglich die Gitterschichten, parallel zur X-Y-Ebene, parallel zur X-Z-Ebene oder parallel zur Y-Z-Ebene auszuwählen, um aus jeder Richtung die Ergebnisse des Lösers betrachten zu können. Darüber hinaus kann man das Druckfeld, das Geschwindigkeitsvektorfeld oder nur die Gitterzellen der ausgewählten Schicht angezeigt werden.

Zur Darstellung des Druckfeldes wird ein vom Druck abhängiger Farbverlauf verwendet. Hierbei stellen bläuliche Farben geringe, grünliche Farben mittlere und rötliche Farben die hohen Druckwerte dar. In der Standarddarstellung wird eine Gitterzelle in Dreiecke unterteilt (s. Abb. 5.15) und die Farben über diese Dreiecke interpoliert. Bei der verbesserten Interpolation wird eine Gitterzelle in acht Dreiecke unterteilt, um mehr Punkte zur Interpolation verwenden zu können und somit einen genauen und gleichmäßigen Farbverlauf darzustellen (s. Abb. 5.16).

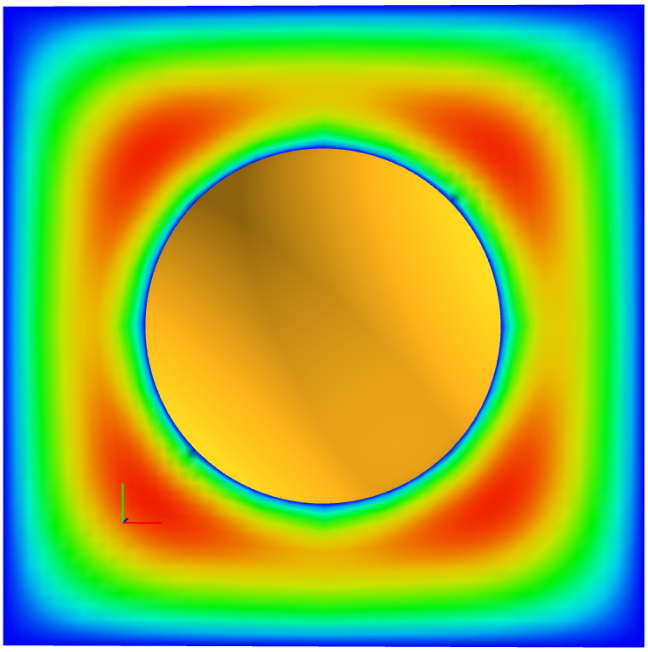


Abbildung 5.15: *Druckfeld um das Kugel-Modell. Sichtbare Artefakte hängen mit der einfachen Interpolation der Farbwerte zusammen.*

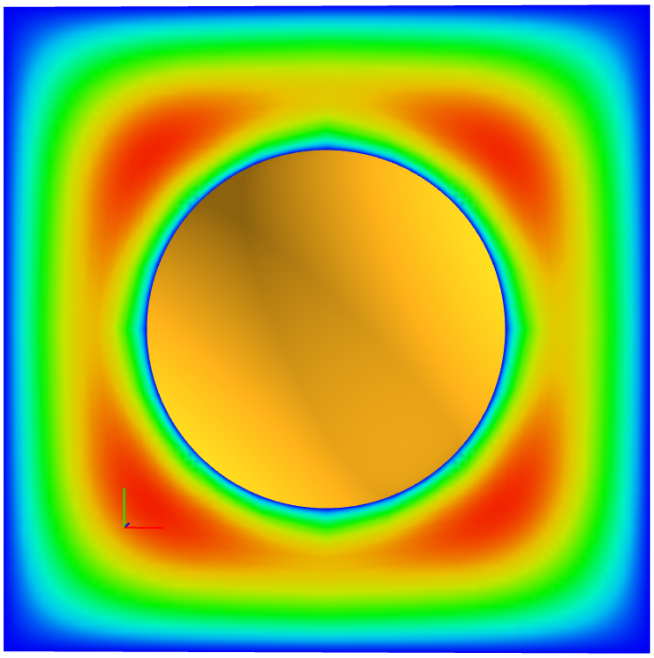


Abbildung 5.16: *Druckfeld um das Kugel-Modell. Bei eingeschalteter verbesserter Interpolation sind keine Artefakte zu erkennen und das Druckfeld wird gleichmäßig dargestellt.*

Zur Darstellung des Vektorfeldes werden die Vektoren als Linien gezeichnet. Sie werden durch ihre Länge sowie durch einen Farbverlauf kenntlich gemacht, der wiederum von der Länge abhängig ist. Es handelt sich hierbei um den gleichen Farbverlauf, den die Druck-Darstellung verwendet (s. Abb. 5.17). Alternativ ist es auch möglich die Vektoren auf gleiche Länge zu skalieren. Dadurch ist ihre Länge nur noch an ihrer Farbe erkennbar, aber die Übersicht verbessert sich.
Es kann jeweils nur eine Gitterschicht einzeln dargestellt werden (s. Abb. 5.18).

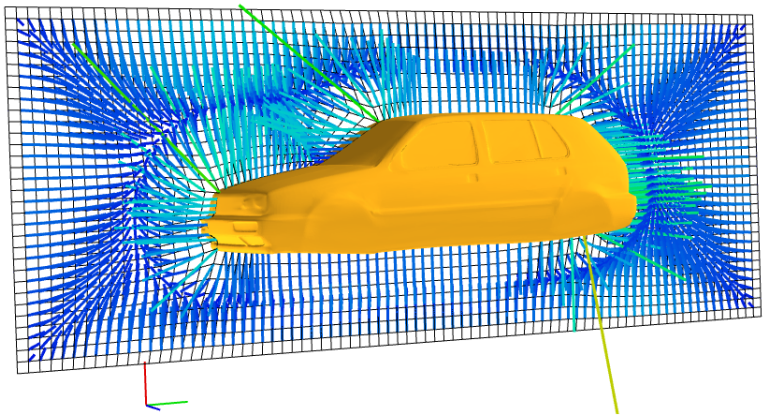


Abbildung 5.17: *Druckfeld um das Kugel-Modell. Bei eingeschalteter verbesserter Interpolation verschwinden die Artefakte und das Druckfeld wird gleichmäßig dargestellt.*

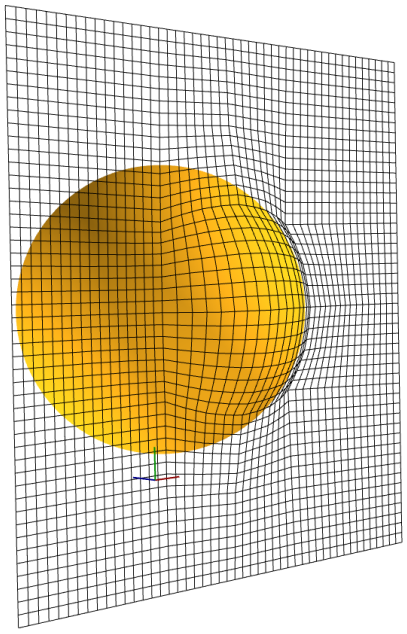


Abbildung 5.18: *Eine Gitterschicht am Kugel-Modell.*

Kapitel 6

Projektvalidierung

In diesem Kapitel wird überprüft, inwiefern die von der PG erstellte Software die zur rechnergestützten Strömungssimulationen notwendigen Arbeitsschritte[1] hinreichend gut bewältigt. Weiterhin wird die Geschwindigkeit und der Speicherplatzverbrauch evaluiert.
Zunächst wird die Validierungstechnik vorgestellt, es folgt eine Auflistung der Testfälle, mit denen die Randanpassung und die implementierten Glättungsalgorithmen untersucht werden. Der Einfluss der initialen Objektreduzierung auf die Parametrisierung wird anschließend geprüft. An einem komplexen Testfall wird die Praxistauglichkeit von *InGrid3D* demonstriert und die Funktionalität von *HaGrid3D* wird an Hand eines Beispiels überprüft. Der darauf folgende Abschnitt untersucht die Leistungsfähigkeit des Lösers. Das Kapitel schließt mit einer zusammenfassenden Diskussion.

6.1 Validierungstechnik

Um die erzeugten Gitter der Projektgruppe objektiv bewerten zu können, müssen geeignete Kriterien definiert werden. Die Messung der Qualitätsmerkmale eines Gitters wurde in Kapitel 2.3.2 beschrieben. In Abhängigkeit des zu verwendenden Lösers gelten für die einzelnen Merkmale bestimmte Toleranzen. In Tabelle 6.1 werden exemplarisch die Toleranzen der Löserimplementierung der PG mit zwei Mehrgitter-Implementierungen (MG1, MG2, vgl. [16]) gegenübergestellt, die sich in der Robustheit ihrer Vorkonditionierer unterscheiden.
Für alle drei Löser ist es also wichtig, dass keine Jacobi-Fehler in den verwendeten Gittern vorhanden sind. Die Einhaltung der beschriebenen Qualitätskriterien wird bei jedem der folgenden Testfälle überprüft. Dazu ist die Implementierung einer speziellen Testumgebung nicht notwendig, da *InGrid3D* alle nötigen Qualitätsinformationen in der Statuszeile zur Verfügung stellt. Der Test erfolgt nach einer aufsteigenden Strategie, so werden zunächst die einzelnen Methoden an Hand von einfachen Testfällen überprüft und anschließend an Hand eines komplexen Testfalls das Gesamtsystem evaluiert.

[1] Die Arbeitsschritte beinhalten die Grobgittergenerierung, die Gitterunterteilung, das Lösen sowie die Visualisierung der berechneten Daten.

Merkmal	MG1	MG2	Eigenimplementierung
Jacobi-Fehler	nicht erlaubt	nicht erlaubt	nicht erlaubt
Innenwinkel	$30° < \alpha < 150°$	$20° < \alpha < 160°$	$20° < \alpha < 160°$
Längenverhältnis	4	10^8	100

Tabelle 6.1: Toleranzen der Qualitätskriterien.

6.2 Testgeometrien und Testgitter

Die den Tests zu Grunde liegenden Geometrien und Gitter sind in Tabelle 6.2 aufgelistet. Die Auflösung des Geometrieobjektes bezeichnet dabei die Anzahl der Dreieckselemente, die die Oberfläche beschreiben. „Fein" bzw. „grob" bezeichnen dabei die relative Auflösung des Geometrieobjektes. Die Qualität des Grobgitters beruht heuristisch auf Erfahrungswerten bei der Benutzung von *InGrid3D* (daher die Bezeichnungen „gut" bzw. „schlecht"). Hierbei muss darauf geachtet werden, dass das Grobgitter möglichst markante Punkte der Geometrie erfasst. Die Anzahl der initialen Hexaederzellen wird in der letzten Spalte angegeben.

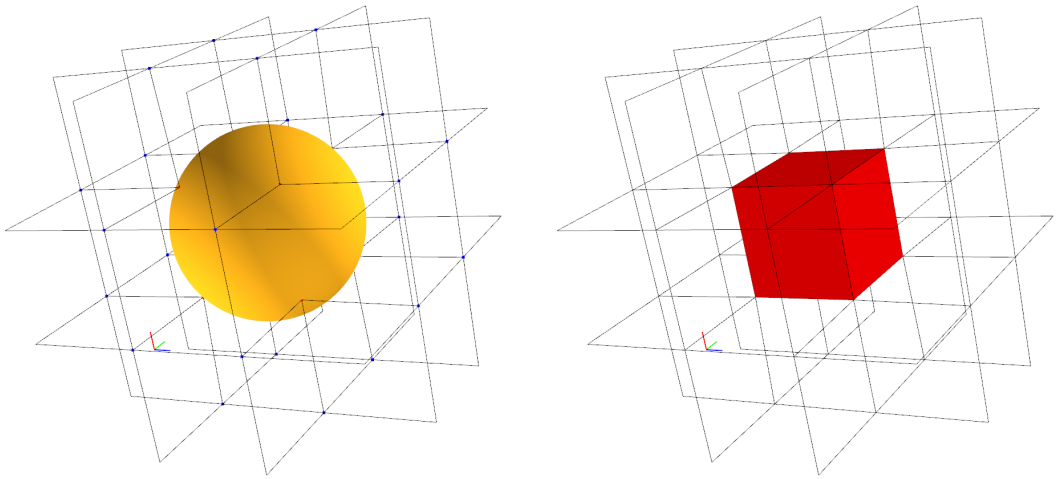
Testfall	Geometrieobjekt		Grobgitter	
	Name	Auflösung	Qualität	Auflösung
1	Kugel	fein	gut	26 Zellen
2	Kugel	grob	gut	26 Zellen
3	Kugel	fein	schlecht	26 Zellen
4	Einbuchtung	fein	gut	1000 Zellen
5	Einbuchtung	fein	schlecht	64 Zellen
6	Klotz	fein	gut	3375 Zellen
7	Klotz	fein	schlecht	125 Zellen
8	Golf III	fein	gut	160 Zellen

Tabelle 6.2: Auflistung der Testfälle und deren Eigenschaften.

Die Testfälle werden in den folgenden Abschnitten zunächst jeweils kurz beschrieben. In den Abbildungspaaren wird auf der linken Seite das Originalobjekt und auf der rechten Seite die Approximation der Objektoberfläche durch die Hexaederseitenflächen angezeigt, aus Gründen der Übersichtlichkeit werden nur die Hexaederzellen auf den Objekträndern angezeigt.

Testfall 1

Als Objekt wird eine hochaufgelöste Kugel (327680 Dreiecke) verwendet. Die Qualität des initialen, aus 26 Hexaederzellen bestehenden Grobgitters ist unter den Einschränkungen, die kartesische Gitter für dieses Objekt implizieren, optimal. In diesem Testfall (s. Abb. 6.1) sollte die Anzahl der Unterteilungsschritte lediglich durch den zur Verfügung stehenden Arbeitsspeicher limitiert sein.

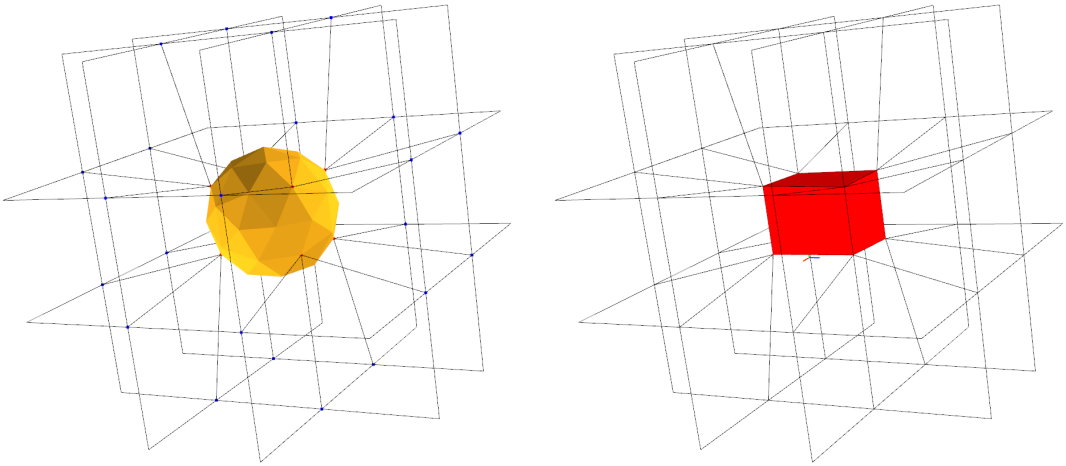


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.1: *Initiales Grobgitter für Testfall 1.*

Testfall 2

Im Gegensatz zu Testfall 1 wird nun eine sehr grob aufgelöste Kugel (80 Dreiecke) verwendet. Das Grobgitter besteht ebenfalls aus 26 Hexaederzellen, die Qualität ist auf Grund von nur 42 Punkten auf der Oberfläche, mit denen die Gitterknoten verbunden werden können, etwas schlechter. Auch in diesem Testfall (s. Abb. 6.2) wird die Anzahl der Unterteilungsschritte lediglich durch den zur Verfügung stehenden Arbeitsspeicher limitiert.

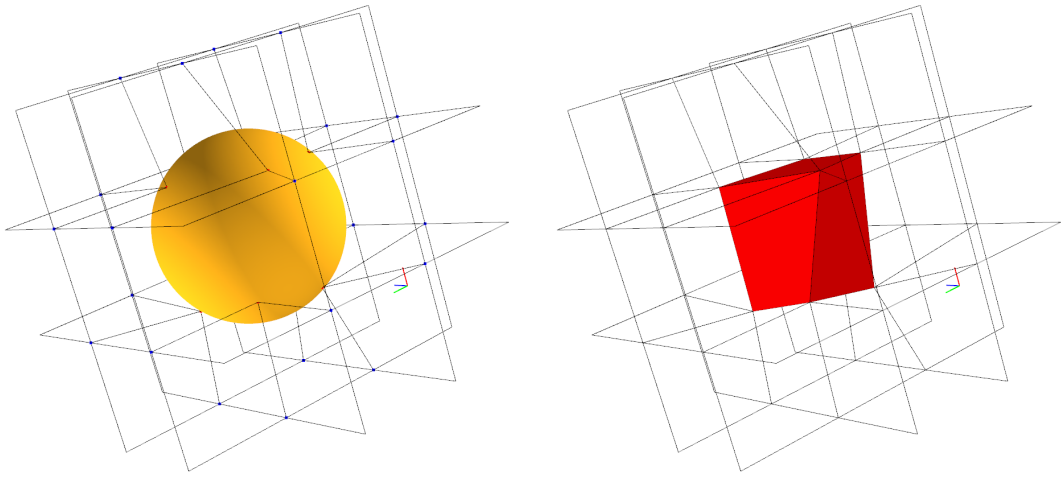


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.2: *Initiales Grobgitter für Testfall 2.*

Testfall 3

Als Objekt wird wieder die hochaufgelöste Kugel aus Testfall 1 verwendet. Die Qualität des initialen, aus 26 Hexaederzellen bestehenden Grobgitters ist in diesem Fall jedoch bewusst schlecht unterhalb der Toleranzen gewählt worden. Da die Anzahl der Hexaederzellen unverändert geblieben ist, ist auch die Anzahl der möglichen Unterteilungsschritte unverändert (vgl. Abb. 6.3).

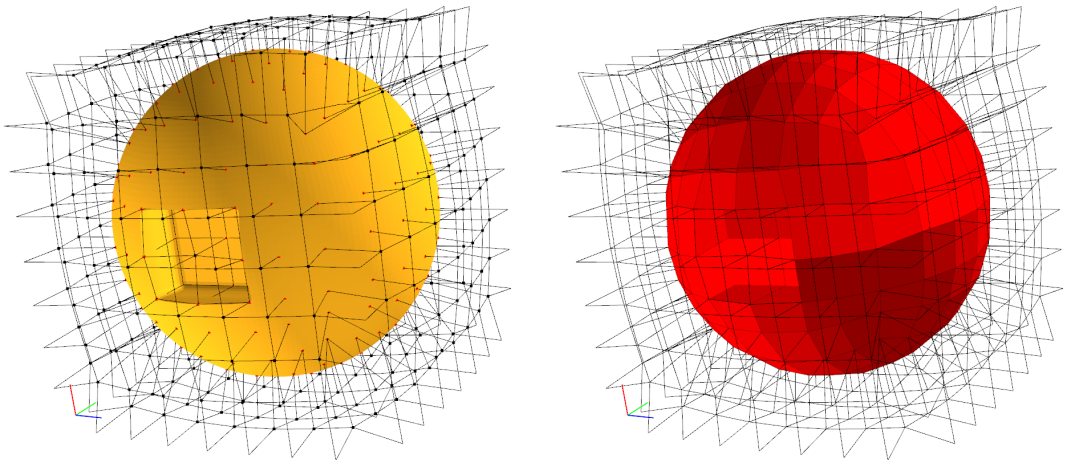


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.3: *Initiales Grobgitter für Testfall 3.*

Testfall 4

Die Geometrie besteht aus einer hochaufgelösten Kugel, in die eine würfelförmige Einbuchtung eingelassen wurde (vgl. Abb. 6.4). Das Grobgitter versucht, diesem Problemfall optimal zu begegnen, indem insbesondere die Einbuchtung mit insgesamt vier Hexaederzellen aufgefüllt wird.

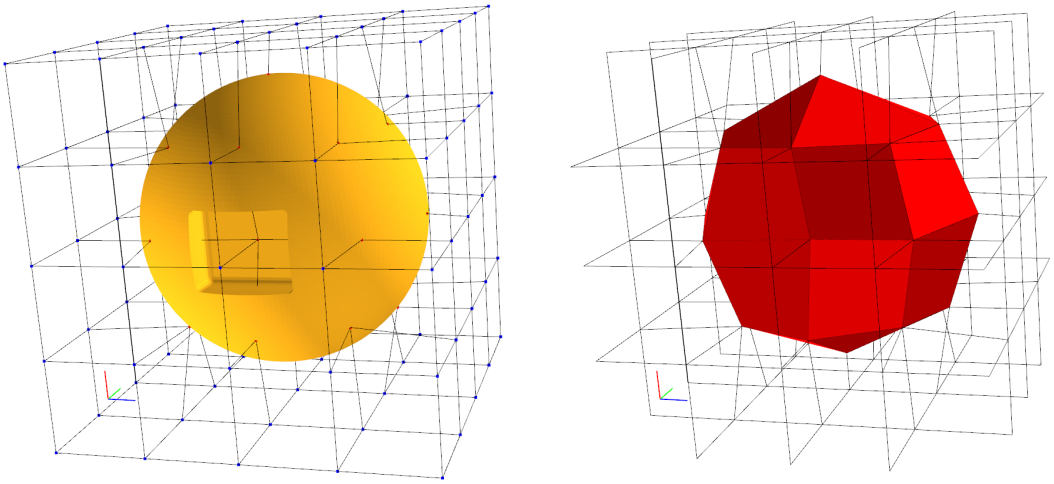


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.4: *Initiales Grobgitter für Testfall 4.*

Testfall 5

Als Objekt wird wieder die Einbuchtung aus Testfall 4 verwendet. Um den Einfluss des Grobgitters auf die resultierende Gitterqualität zu demonstrieren, wurde bei dem hier verwendeten Gitter absichtlich nicht auf die Erfassung der Einbuchtung geachtet (vgl. Abb. 6.5).

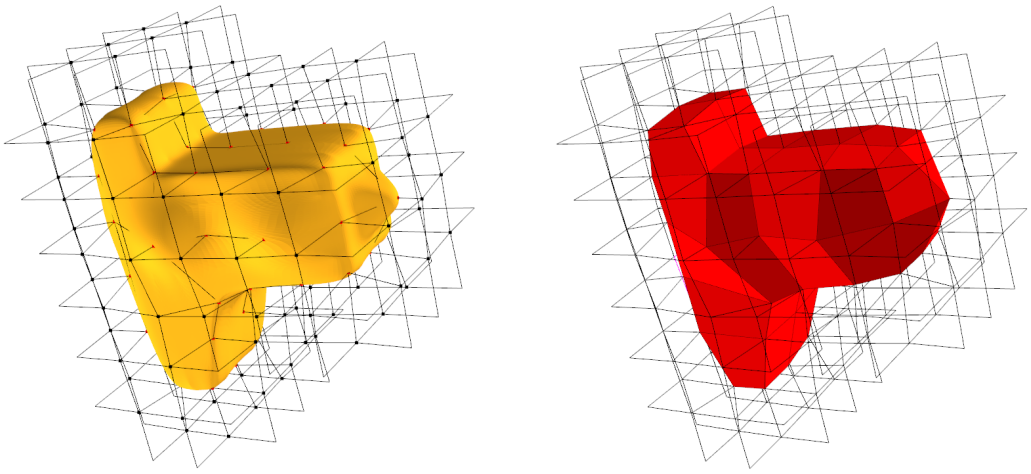


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.5: *Initiales Grobgitter für Testfall 5.*

Testfall 6

Testfall 6 ist ein abstraktes Objekt, das speziell für die Entwicklung von *InGrid3D* entworfen wurde, mit einem von Hand erstellten, möglichst gut an dieses Objekt angepassten Grobgitter (vgl. Abb. 6.6). Mit diesem Grobgitter waren auf dem Testsystem mit 1.5 GB Hauptspeicher drei Unterteilungen möglich, bevor die Speichergrenze erreicht wird, in diesem Augenblick verbraucht *InGrid3D* ca. 900MB Hauptspeicher.

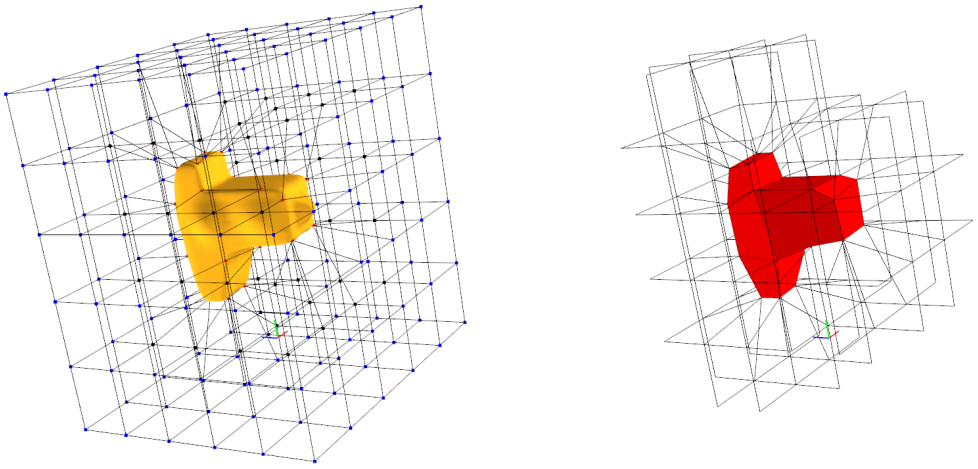


(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.6: *Initiales Grobgitter für Testfall 6.*

Testfall 7

Testfall 7 verwendet das bereits aus Testfall 6 bekannte Testobjekt, welches nun mit einem sehr einfachen, schlecht angepassten Grobgitter approximiert werden soll (vgl. Abb. 6.7). Im Test verbrauchte das Programm nach der vierten Unterteilung bereits annähernd 400MB Hauptspeicher, eine weitere Unterteilung war mit den im Testsystem verfügbaren 1.5 GB Hauptspeicher nicht möglich.

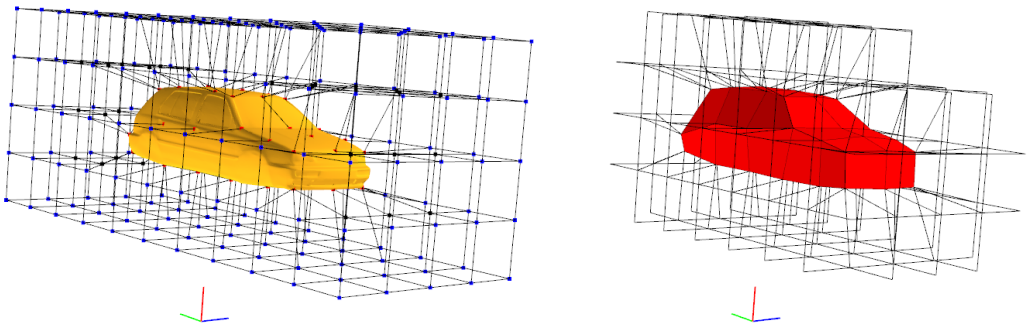


(a) Originalobjekt im Gitter. (b) Durch das Gitter angenäherte Objektoberfläche.

Abbildung 6.7: Initiales Grobgitter für Testfall 7.

Testfall 8

Für Testfall 8 wird das Modell eines Golf III verwendet. Es wird mit einem gut angepassten Grobgitter (s. Abb. 6.8) approximiert. Auch bei diesem Testfall war es aufgrund des beschränkten Arbeitsspeichers nicht möglich, mehr als vier Unterteilungen vorzunehmen.



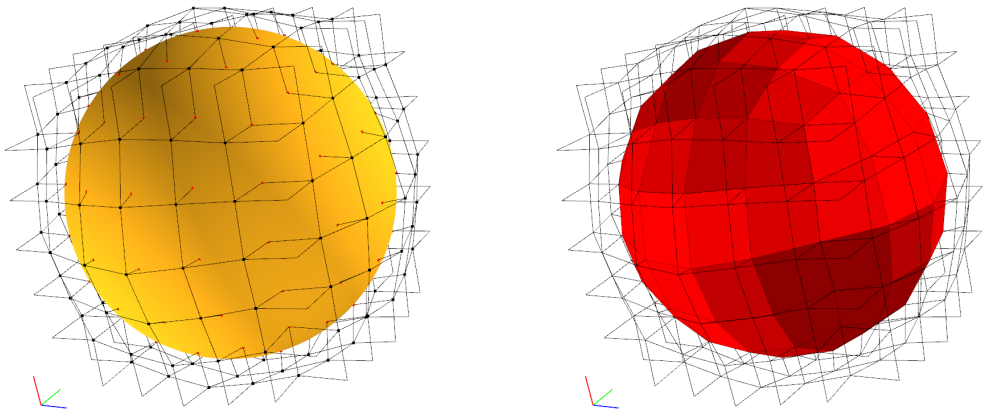
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.8: *Initiales Grobgitter für Testfall 8.*

6.3 Validierung der Randanpassung durch Parametrisierung

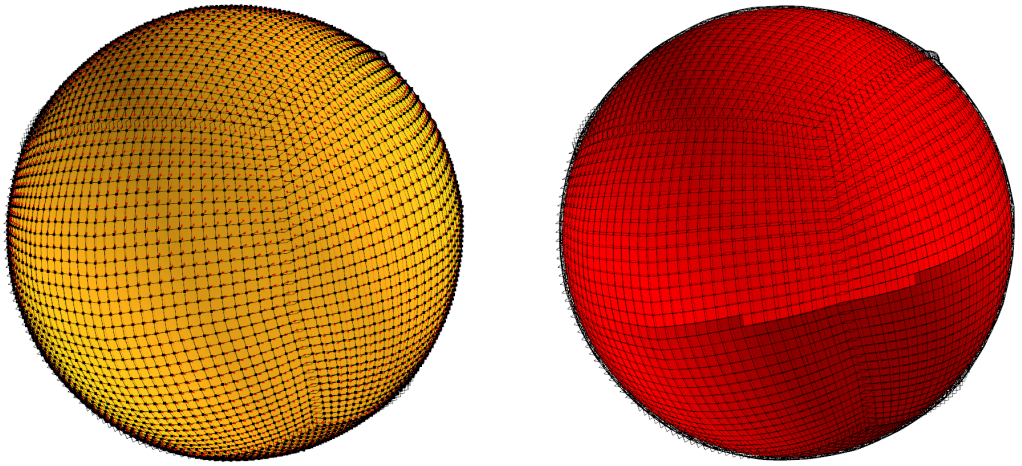
In diesem Abschnitt wird die Randanpassung durch Parametrisierung (s. Kap. 5.3) und Verwendung der Snakes (s. Kap. 5.4) an Hand der vorgestellten Qualitätskriterien für die Testfälle 1 bis 8 überprüft. Es werden keine Glättungsoperatoren oder Korrekturen von Gitter-Selbstdurchdringungen angewendet.

Testfall 1



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.9: *Gitter für Testfall 1 nach zwei Unterteilungen.*



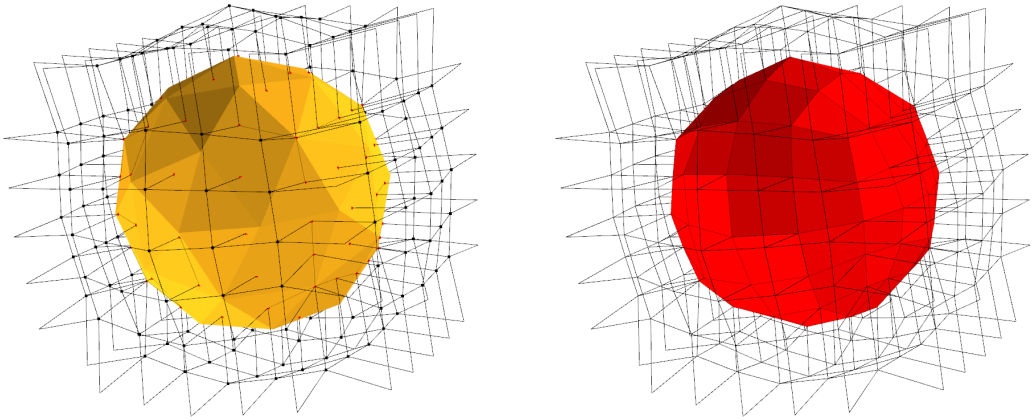
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.10: *Gitter für Testfall 1 nach fünf Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$87° < \alpha < 92°$	1.08	26
1	0	$52° < \alpha < 126°$	2.90	208
2	0	$40° < \alpha < 150°$	3.00	1664
3	0	$31° < \alpha < 160°$	4.68	13312
4	0	$25° < \alpha < 161°$	5.49	106496
5	0	$22° < \alpha < 161°$	6.24	851968

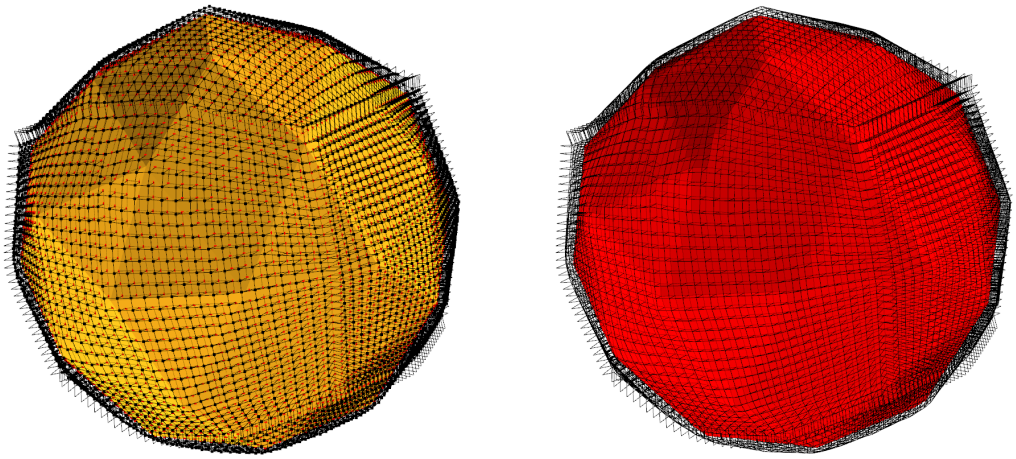
Tabelle 6.3: Werte der Qualitätskriterien für Testfall 1.

Testfall 2



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.11: *Gitter für Testfall 2 nach zwei Unterteilungen.*



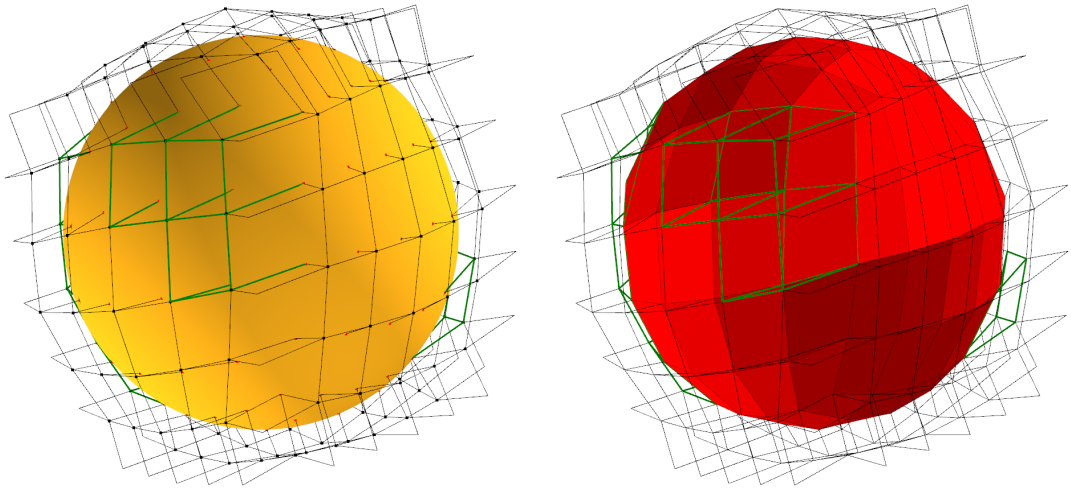
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.12: *Gitter für Testfall 2 nach fünf Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$69° < \alpha < 105°$	2.30	26
1	0	$62° < \alpha < 122°$	2.37	208
2	0	$48° < \alpha < 140°$	2.37	1664
3	0	$38° < \alpha < 152°$	2.44	13312
4	0	$29° < \alpha < 165°$	3.08	106496
5	0	$20° < \alpha < 171°$	8.32	851968

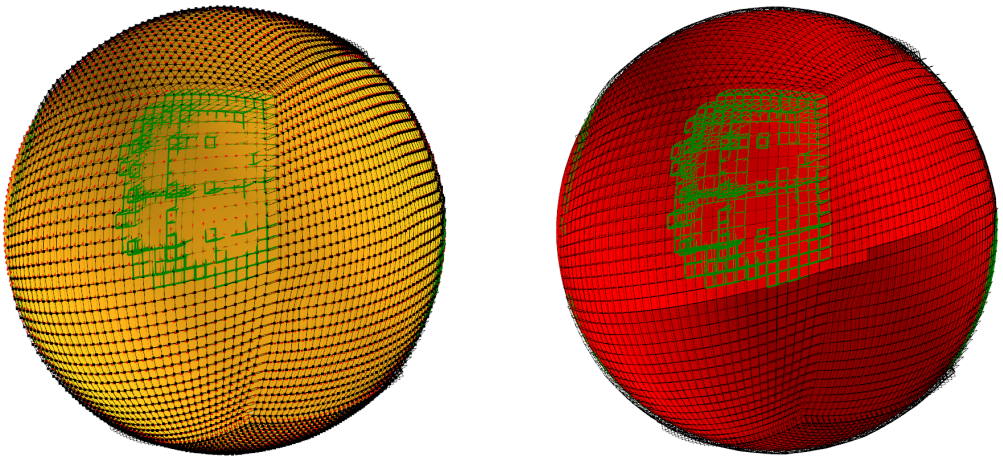
Tabelle 6.4: Werte der Qualitätskriterien für Testfall 2.

Testfall 3



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.13: *Gitter für Testfall 3 nach zwei Unterteilungen.*



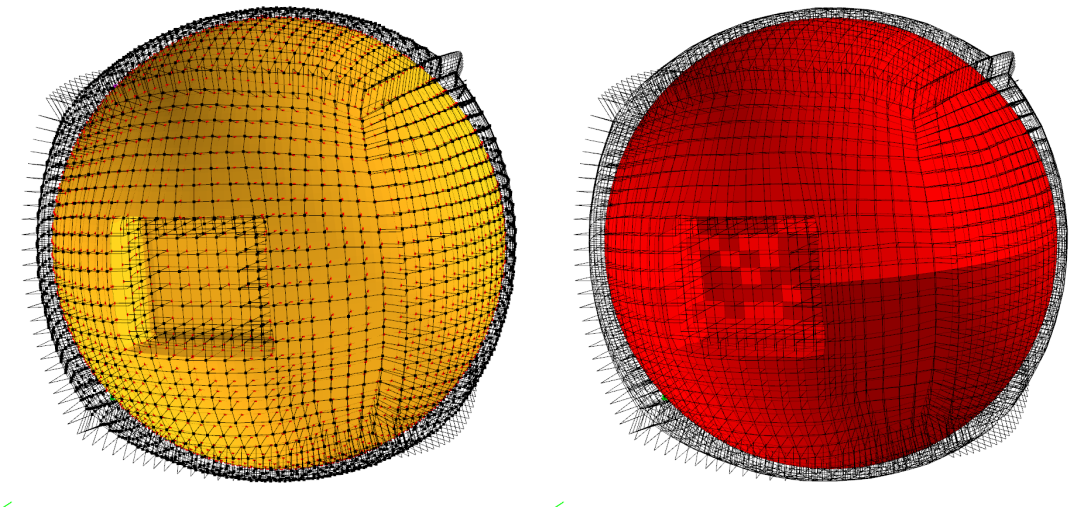
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.14: *Gitter für Testfall 3 nach fünf Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$49° < \alpha < 150°$	2.14	26
1	1	$27° < \alpha < 174°$	3.85	208
2	11	$7° < \alpha < 179°$	5.99	1664
3	80	$2° < \alpha < 178°$	10.15	13312
4	377	$1° < \alpha < 179°$	25.44	106496
5	1692	$0° < \alpha < 180°$	51.40	851968

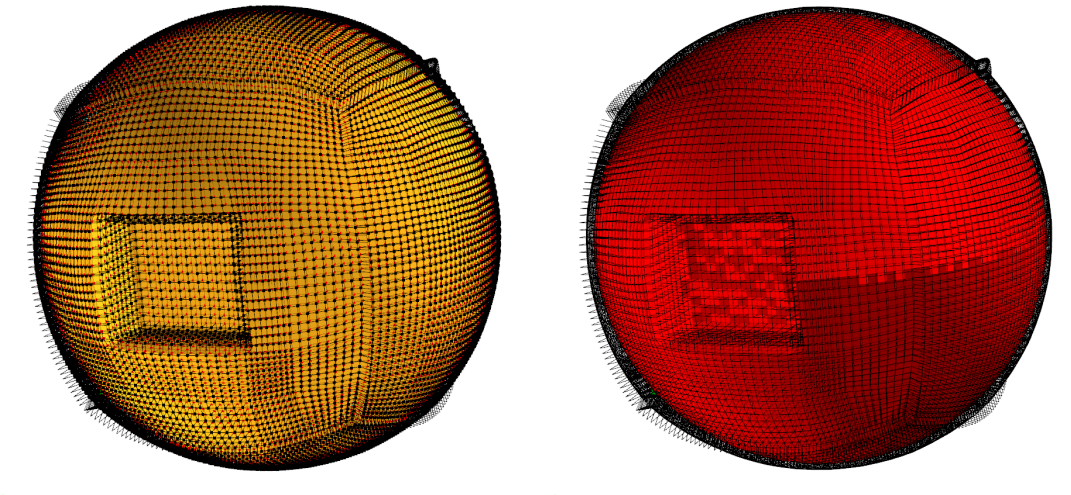
Tabelle 6.5: Werte der Qualitätskriterien für Testfall 3.

Testfall 4



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.15: *Gitter für Testfall 4 nach zwei Unterteilungen.*



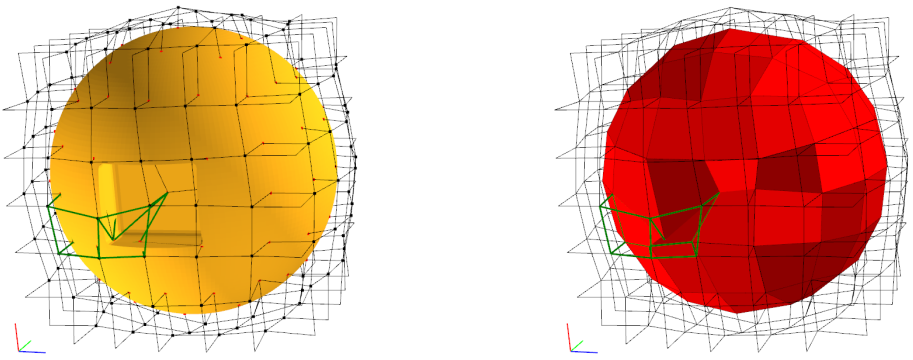
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.16: *Gitter für Testfall 4 nach drei Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$42° < \alpha < 126°$	3.30	1000
1	0	$42° < \alpha < 132°$	3.29	8000
2	0	$42° < \alpha < 141°$	3.29	64000
3	0	$42° < \alpha < 142°$	3.79	512000

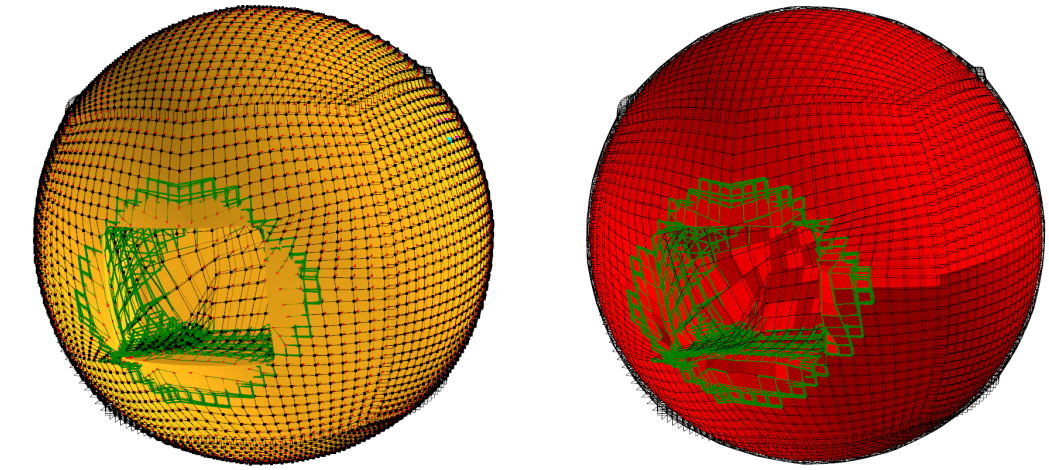
Tabelle 6.6: Werte der Qualitätskriterien für Testfall 4.

Testfall 5



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.17: *Gitter für Testfall 5 nach zwei Unterteilungen.*



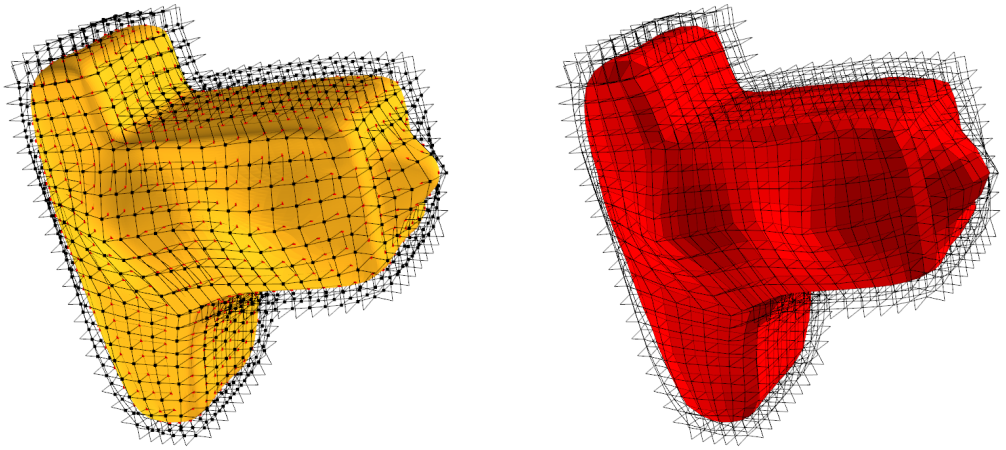
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.18: *Gitter für Testfall 5 nach fünf Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$49° < \alpha < 150°$	2.14	64
1	3	$19° < \alpha < 159°$	4.70	512
2	23	$9° < \alpha < 177°$	5.64	4096
3	80	$0° < \alpha < 180°$	10.88	32768
4	149	$0° < \alpha < 180°$	25.07	262144

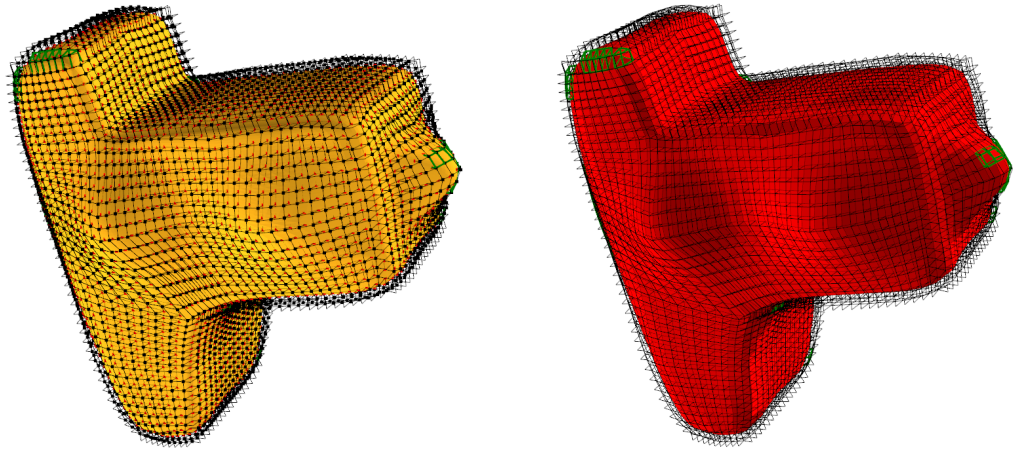
Tabelle 6.7: Werte der Qualitätskriterien für Testfall 5.

Testfall 6



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.19: *Gitter für Testfall 6 nach zwei Unterteilungen.*



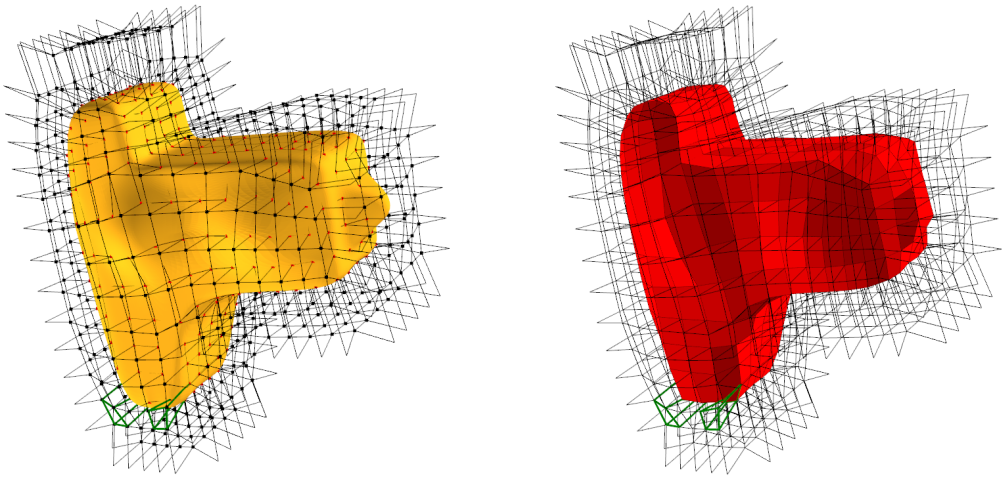
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.20: *Gitter für Testfall 6 nach drei Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$50° < \alpha < 144°$	1.63	3375
1	0	$29° < \alpha < 171°$	2.81	27000
2	0	$18° < \alpha < 173°$	4.91	216000
3	45	$9° < \alpha < 178°$	12.46	1728000

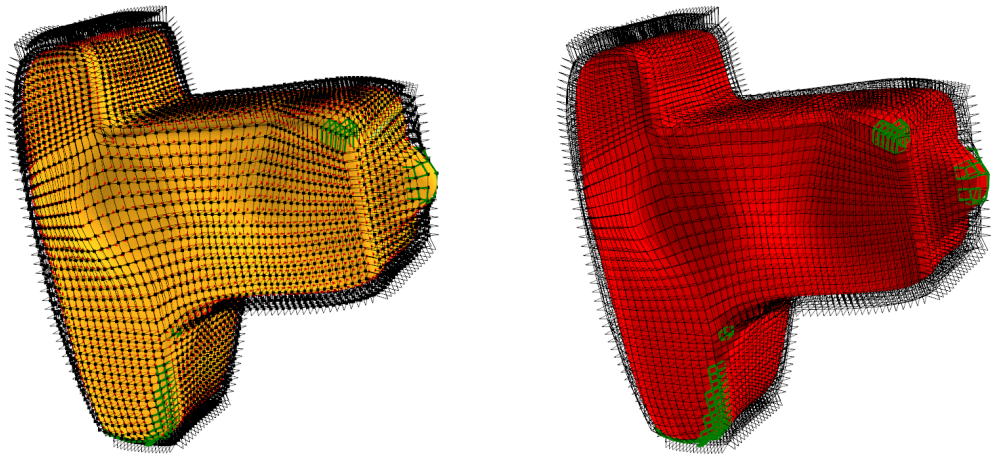
Tabelle 6.8: Werte der Qualitätskriterien für Testfall 6.

Testfall 7



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.21: *Gitter für Testfall 7 nach zwei Unterteilungen.*



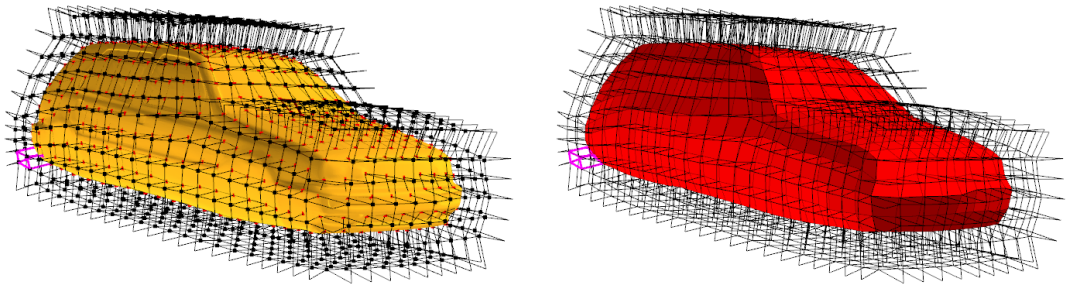
(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.22: *Gitter für Testfall 7 nach vier Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$50° < \alpha < 130°$	2.49	125
1	0	$50° < \alpha < 144°$	2.81	1000
2	2	$30° < \alpha < 160°$	3.20	8000
3	17	$17° < \alpha < 169°$	3.64	64000
4	77	$7° < \alpha < 177°$	8.72	512000

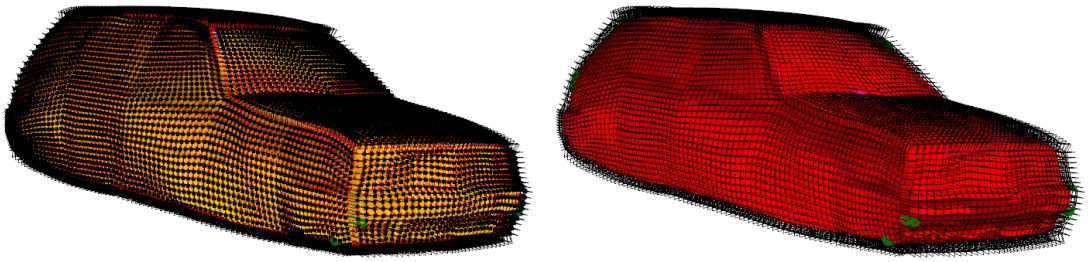
Tabelle 6.9: Werte der Qualitätskriterien für Testfall 7.

Testfall 8



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.23: *Gitter für Testfall 8 nach zwei Unterteilungen.*



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.24: *Gitter für Testfall 8 nach vier Unterteilungen.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$45° < \alpha < 157°$	2.88	160
1	0	$38° < \alpha < 157°$	2.83	1280
2	0	$31° < \alpha < 159°$	3.82	10240
3	1	$25 < \alpha < 167°$	4.99	81920
4	15	$11° < \alpha < 179°$	7.34	655360

Tabelle 6.10: Werte der Qualitätskriterien für Testfall 8.

6.4 Validierung der Glätter

In diesem Abschnitt werden die Fähigkeiten der Glätter, Gitter zu verbessern, untersucht. Es wird zwischen einfachen Glättern und Glättern, die auf der Objektoberfläche arbeiten, unterschieden. Sie werden an Hand von fallspezifischen Beispielen demonstriert. Für detaillierte Funktionsbeschreibungen der Glätter wird auf Kapitel 5.7 verwiesen.

Einfache Glättungsstrategien

Mit Hilfe dieser Testserie wird exemplarisch für Testfall 1 (hochaufgelöstes Kugelobjekt mit optimalem Grobgitter) die Strategie beurteilt, erst bei Erreichen der feinsten Unterteilungsstufe eine Glättung durchzuführen. Nach fünf Unterteilungsschritten werden die in Tabelle 6.11 angegebenen Glätter jeweils so lange iteriert, bis sie keine Änderung des Gitters mehr erzielen konnten.

Glätter	Jacobi-Fehler	Innenwinkel	Längenverhältnis
ohne	0	$29° < \alpha < 154°$	5.62
Laplace	0	$29° < \alpha < 154°$	5.62
Umbrella	0	$25° < \alpha < 157°$	8.22
Umbrella 2	0	$27° < \alpha < 161°$	8.38

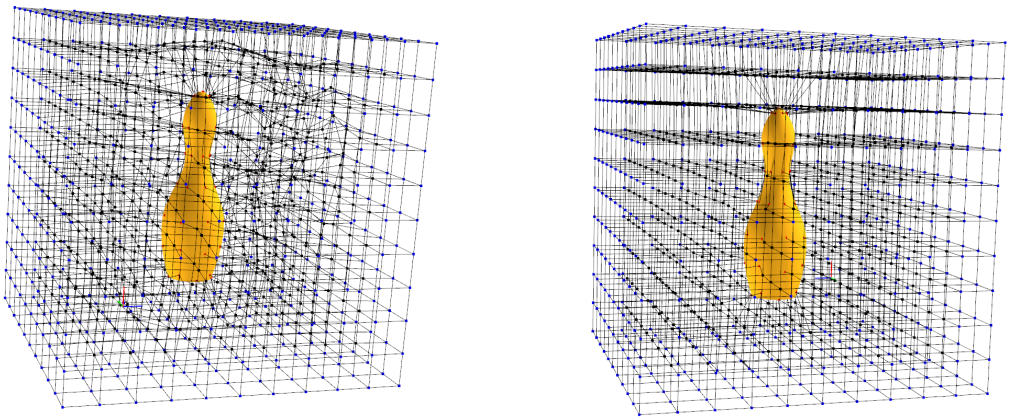
Tabelle 6.11: Werte der Qualitätskriterien nach der jeweiligen Glättung.

Glätter zur Verbesserung der Gitterqualität: Laplace, Laplace - Jacobi, Umbrella und Umbrella 2

Der Laplace-Glätter ist einer der einfachsten Glätter. Das in Abbildung 6.25(a) dargestellte Gitter ist qualitativ schlecht, der minimale Innenwinkel beträgt $27°$, der maximale Innenwinkel $151°$. Nach 14 Iterationen haben sich die Werte auf $42°$ und $136°$ verbessert, weitere Iterationen bleiben ohne Effekt (s. Abb. 6.25(b)). Bei nichtkonvexen Objekten kann der Laplace-Glätter aber auch zu Fehlern führen (s. Abb. 6.26). Aus diesem Grunde wurde eine zusätzliche Kontrolle auf Jacobi-Fehler implementiert, die den Glättungsvorgang abbricht, sobald ein Jacobi-Fehler entstehen würde. Im Beispiel aus Abbildung 6.26(a) wird durch den Laplace-

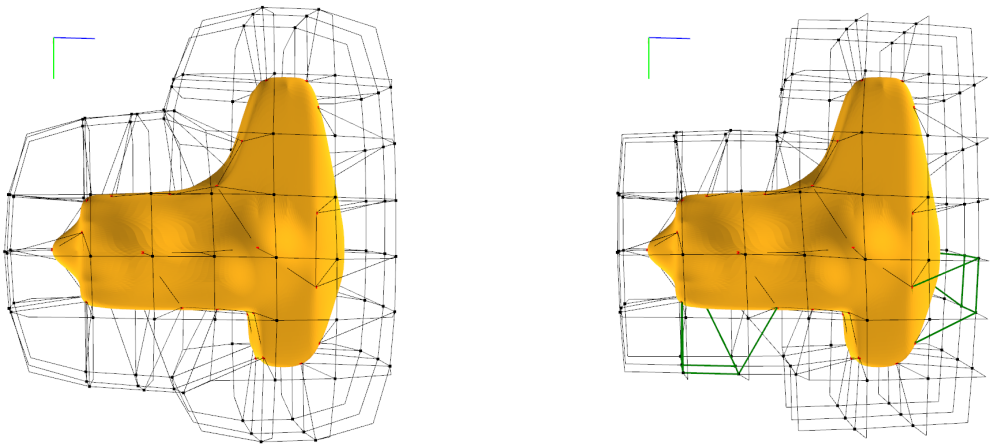
Jacobi-Glätter kein neuer Jacobi-Fehler erzeugt, allerdings wird auch keine Verbesserung des Gitters erziehlt.

Um die Unzulänglichkeiten der beiden vorgestellten Glätter zu kompensieren wurden mit den Umbrella-Glättern zwei weitere Methoden zur Gitterverbesserung implementiert. Der Umbrella-Glätter ist dabei in der Lage, selbst in Situationen, in denen der Laplace-Glätter keine Auswirkungen auf das Gitter mehr hat, noch Qualitätsverbesserungen zu erzielen. So wird bei dem Gitter des Kegelobjektes aus Abbildung 6.25(b) eine weitere Verbesserung der Winkel auf $46°$ und $128°$ erreicht (s. Abb. 6.27(a)). Umbrella 2 kann Jacobi-Fehler in der Nähe des Objektes (s. Abb. 6.26(b)) beheben, indem Knoten, die sich in direkter Nachbarschaft zum Objektrand befinden, weiter in die vom Objektrand abweisende Richtung gezogen werden (s. Abb. 6.27(b)).



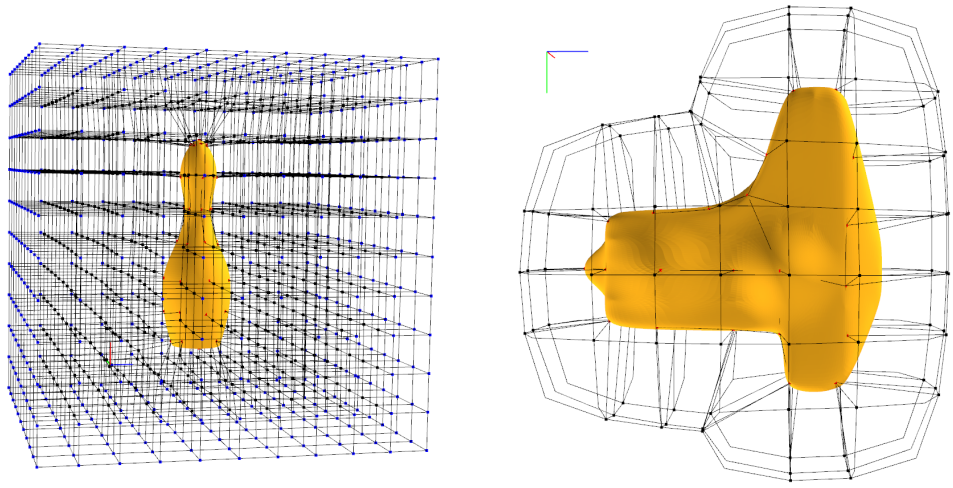
(a) *Qualitativ schlechtes Grobgitter.* (b) *Die Gitterqualität wurde erhöht.*

Abbildung 6.25: *Resultate des Laplace-Glätters.*



(a) *Eine zulässige Gitterkonstellation.* (b) *Der Laplace-Glätter erzeugt Jacobi-Fehler.*

Abbildung 6.26: *Schwachpunkt des Laplace-Glätters.*



(a) *Der Umbrella-Glätter erzielt qualitativ bessere Ergebnisse als der Laplace-Glätter.* (b) *Der Umbrella 2-Glätter „zieht" die Umgebungsknoten vom Objekt weg.*

Abbildung 6.27: *Eigenschaften der Umbrella-Glätter.*

Oberflächenglätter: Oberfläche-Winkel und Oberfläche-Jacobi

Die vorangegangenen Glätter zur Verbesserung der Gitterqualität können auf der Objektoberfläche liegende Knoten nicht verschieben. Dadurch sind sie in ihren Möglichkeiten eingeschränkt, insbesondere weil Jacobi-Fehler häufig an der Objektoberfläche auftreten. Daher

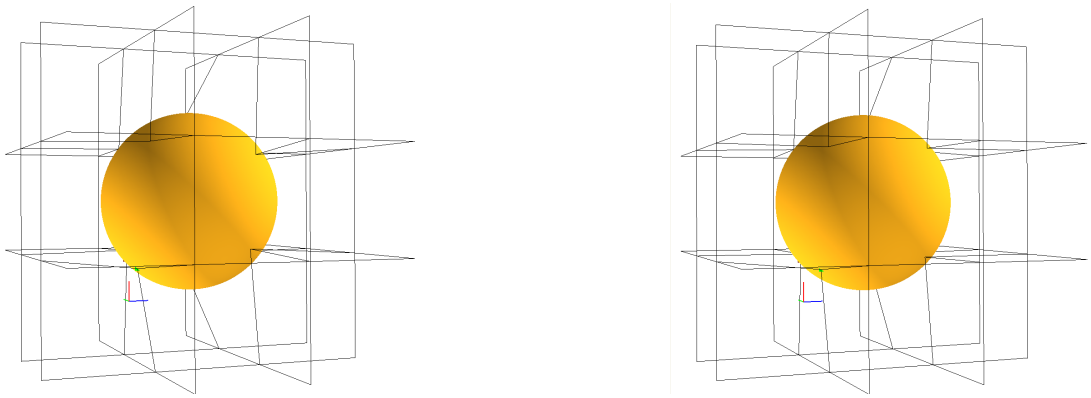
wurden zwei Glätter entwickelt, die Knoten auf der Oberfläche verschieben. Es folgen typische Einsatzfälle und die Resultate dieser Glätter.

Oberfläche-Winkel-Glätter

Der Glätter Oberfläche-Winkel (s. Kap. 5.7.4) versucht die Knoten des Gitters auf dem Objekt so zu verschieben, dass sich die Winkel im Gitter verbessern. Da bei komplexen Modellen eine Verschiebung der Knoten auf dem Objekt nur in Abhängigkeit von Winkelmessungen zu einer Verschlechterung des Gitters führen kann, wird die Krümung der Oberfläche zusätzlich berücksichtigt. Falls ein einstellbarer Schwellwert durch eine Verschiebung überschritten würde, wird diese nicht durchgeführt. Das in Abbildung 6.28 gezeigte, qualitativ schlechte Grobgitter wird nach 70 Iterationen in einen fast optimalen Zustand überführt (s. Abb. 6.29). Die genauen Resultate der durchgeführten Glättung lassen sich Tabelle 6.12 entnehmen.

Iteration	kleinster Winkel	größter Winkel	Längenverhältnis
20	58°	136°	1.71
30	66°	125°	1.51
40	72°	116°	1.35
50	78°	108°	1.23
60	84°	100°	1.16
70	87°	92°	1.08

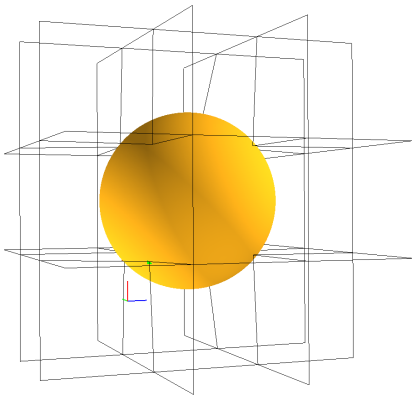
Tabelle 6.12: Werte der Qualitätskriterien für Winkelglätter.



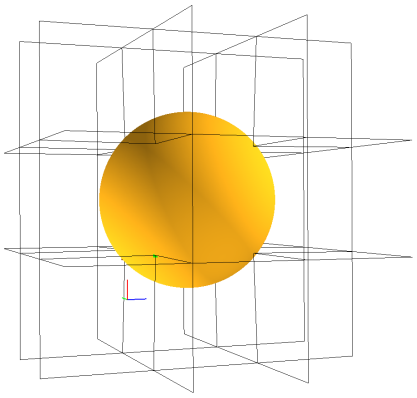
(a) *Oberfläche-Winkel-Glätter 10 mal angewandt.*

(b) *Oberfläche-Winkel-Glätter 30 mal angewandt.*

Abbildung 6.28: *Der Oberfläche-Winkel-Glätter.*



(a) *Oberfläche-Winkel-Glätter 50 mal angewandt.*

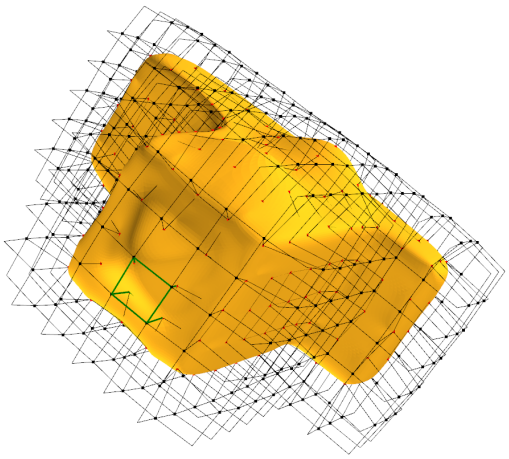


(b) *Oberfläche-Winkel-Glätter 70 mal angewandt.*

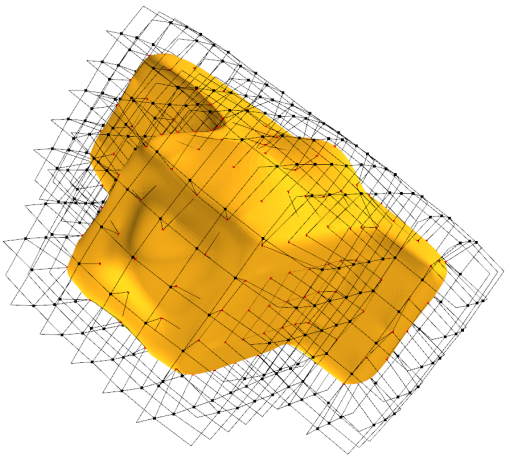
Abbildung 6.29: *Der Oberfläche-Winkel-Glätter.*

Oberfläche-Jacobi-Glätter

Der Oberfläche-Jacobi-Glätter (s. Kap. 5.7.4) versucht durch eine Heuristik, Jacobi-Fehler in den Randelementen zu beheben. Die Abbildungen 6.30(a) und 6.30(b) belegen exemplarisch seine Korrektheit.



(a) *Ein Jacobi-Fehler an der Objektoberfläche.*



(b) *Der Fehler wurde behoben.*

Abbildung 6.30: *Der Oberfläche-Jacobi-Glätter.*

Reparatur-Prozedur

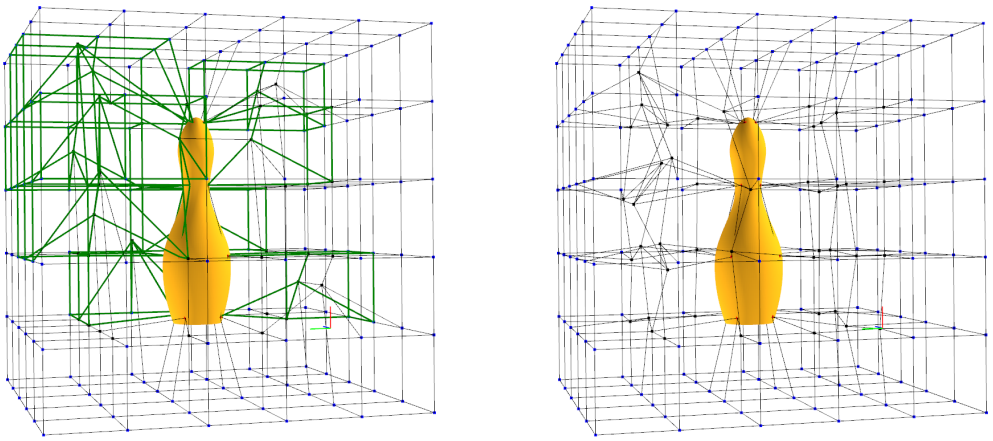
Zunächst werden die Testmethoden für das Reparatur-Modul erläutert, dann die Testdaten präsentiert und mit den Testdaten der Implementierung von Freitag [13] verglichen. Anschlie-

ßend werden einige für das Projekt spezifische Feststellungen und Resultate angeführt. Für den Test des Moduls wurden in einem regulären Gitter mit 1000 Hexaederzellen eine bestimmte Prozentzahl *P* von Knoten verschoben, um ungültige Elemente zu erzeugen. Ein zufällig ausgewählter innerer Knoten $\mathbf{p}$ wurde um eine Distanz h bzw. $2h$ verschoben, wobei h die durchschnittliche Kantenlänge des Testgitters bezeichnet.

Prozentsatz P	*Distanz D*	*Anzahl N*	Zeit (s)
5	h	11	0,048
10	h	34	0,124
25	h	101	0,048
10	2h	164	0,4

Tabelle 6.13: Leistungsdaten der Reparatur-Prozedur

In Tabelle 6.13 stellt *P* den Anteil der verschobenen Knoten in Prozent dar, *D* steht für die Distanz der Verschiebung und *N* für die Anzahl von ungültigen Elementen. Die letzte Spalte gibt die Zeit in Sekunden an.
Es stellt sich heraus, dass die Distanz, um die ein Knoten verschoben wird, die Rechenzeit am stärksten beeinflusst. Ein weiteres Resultat ist, dass die Anwendung der Reparatur-Prozedur auf nicht invertierte Elemente zu einem Gitter mit schlechter Qualität führt. Dies zeigt sich besonders durch das Auftreten von zum Teil sehr spitzen Winkeln. Des Weiteren treten Probleme auf, wenn Knoten der Hexaederzelle, die an die Reparatur-Prozedur übergeben werden, zum Objektrand gehören. Diese Knoten dürfen im Kontext des Projektes nicht von der Oberfläche gezogen, sondern höchstens auf der Oberfläche verschoben werden. Da bei optimierungsbasierter Durchdringungsbehebung aber die neuen Koordinaten nach Maximierung des minimalen Flächeninhalts berechnet werden, kann diese Forderung nicht erfüllt werden. Daher werden diese Knoten von der Betrachtung ausgeschlossen und nur die übrigen Knoten des Hexaeders behandelt. Auf diese Weise lässt sich aber nicht gewährleisten, dass eine Invertierung behoben wird.
Abbildung 6.32 zeigt den Fall eines invertierten Elementes, das Randknoten enthält. Die Randknoten wurden im Unterteilungsschritt erzeugt und an den Objektrand angepasst. Durch die Randanpassung haben diese Knoten die „Nicht-Rand-Knoten" im Hexaeder „überholt" und ein invertiertes Element erzeugt. Eine Durchdringung dieser Art ist mit dem gewählten Ansatz nicht garantiert auflösbar, denn es wird als Nebenbedingung in der Formulierung des linearen Programms gefordert, dass der minimale Flächeninhalt der acht Tetraeder, die um einen Knoten gebildet werden können, vergrößert wird. Diese Forderung kann in einem solchen Fall häufig nicht erfüllt werden.
Anschließend folgt ein weiterer Test der Reparatur-Prozedur. Das in Abbildung 6.31(a) dargestellte Grobgitter enthält 27 Jacobi-Fehler. Die Reparatur-Prozedur erreicht nach drei Iterationen bereits eine gültige Konstellation der Gitterzellen. Wie in Abbildung 6.31(b) ersichtlich werden von diesem Glätter lediglich die Jacobi-Fehler repariert, sonstige Qualitätskriterien bleiben unberücksichtigt.



(a) *27 Jacobi-Fehler im Grobgitter.* (b) *Alle Fehler wurden entfernt.*

Abbildung 6.31: *Die Reparatur-Prozedur im Einsatz.*

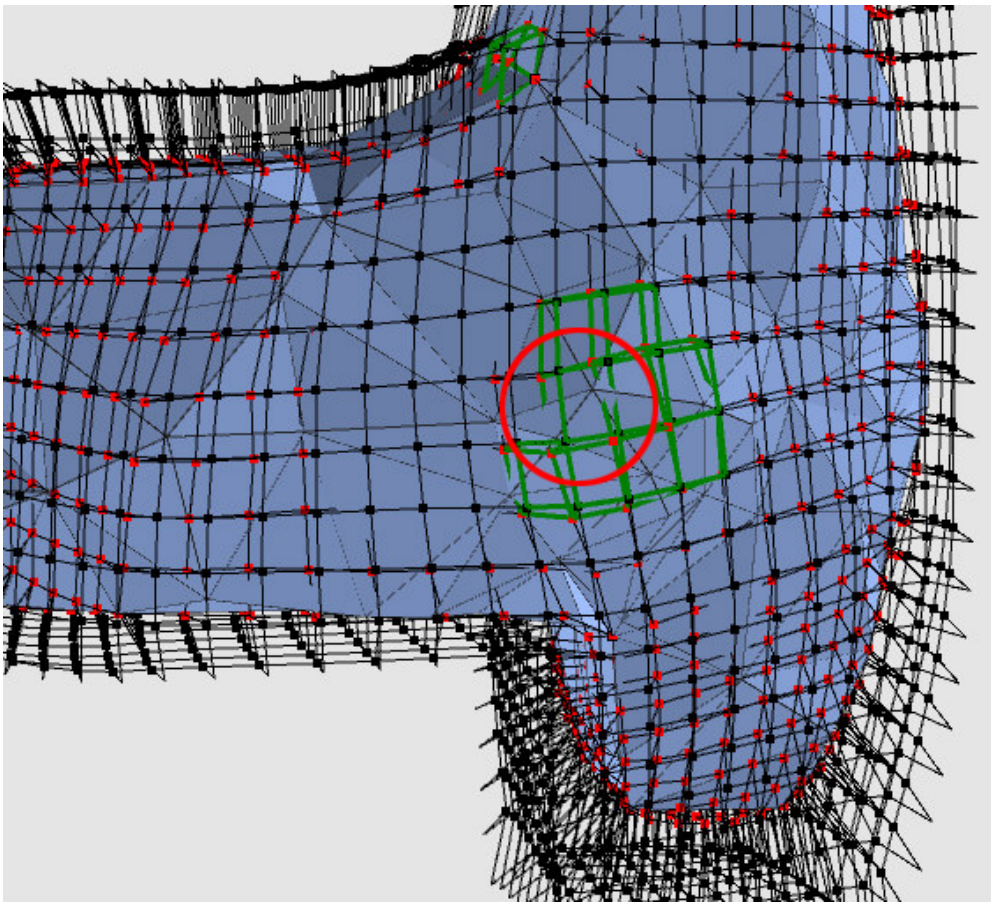


Abbildung 6.32: *Randknoten „überholen" innere Knoten.*

6.5 Einfluss der initialen Objektreduzierung

In diesem Kapitel soll der Einfluß der initialen Objektreduzierung auf die Projektion bei der Unterteilung des Gitters praktisch untersucht werden. Dazu werden drei ausgewählte Objekte mit verschiedenen initialen Objektreduzierungen drei Mal unterteilt, und anschließend die Qualität des Gitters gemessen. Es werden keinerlei Glättungsoperatoren angewendet.
Zunächst wird Testfall 1 betrachtet, der aus der hoch aufgelösten Kugel mit 327680 Dreiecken besteht. Hier ist neben den Qualitätswerten auch die Laufzeit des Programms zu beachten.
Ist die Reduzierung nicht ausreichend, kommen fast ausschließlich die aktiven Konturen zum Einsatz, die dann auf einem sehr feinen Netz kürzeste Wege finden müssen, was zu einer inakzeptabel hohen Laufzeit führt. Darum wurde bei diesem Objekt auf Tests die Evaluierung sehr geringer Reduzierungen verzichtet. Während die dritte Unterteilung auf dem Testsystem ohne Reduzierung 323 Sekunden dauerte, dauerte sie mit 226 Restdreiecken nur 19.5 Sekunden. Reduziert man die Anzahl der Restdreiecke noch weiter, so wird die Laufzeit aber wieder deutlich schlechter.
Als nächstes wird Testfall 2 betrachtet, die grob aufgelöste Kugel. Sie besteht aus 80 Dreiecken. An den Qualitätswerten lässt sich erkennen, daß sich eine Objektreduzierung bei solch groben

Anzahl Restdreiecke	Jacobi-Fehler	Innenwinkel	Längenverhältnis
39086	0	$31° < \alpha < 162°$	4.31
8282	0	$32° < \alpha < 158°$	4.22
1844	0	$37° < \alpha < 152°$	3.95
558	0	$37° < \alpha < 152°$	3.89
226	0	$35° < \alpha < 154°$	4.13
92	0	$33° < \alpha < 157°$	4.54
36	0	$24° < \alpha < 165°$	5.16

Tabelle 6.14: Einfluss der initialen Objektreduzierung für Testfall 1.

Objekten nicht auszahlt und hohe Reduzierungen hier äußerst negative Auswirkungen auf die Qualitätsmaße haben.

Anzahl Restdreiecke	Jacobi-Fehler	Innenwinkel	Längenverhältnis
80	0	$43° < \alpha < 144°$	2.79
62	0	$42° < \alpha < 144°$	2.77
48	0	$43° < \alpha < 144°$	2.85
38	0	$42° < \alpha < 144°$	2.83
28	0	$34° < \alpha < 146°$	3.00
6	7	$3° < \alpha < 177°$	9.65

Tabelle 6.15: Einfluss der initialen Objektreduzierung für Testfall 2.

Abschließend wird Testfall 8 im Hinblick auf die Objektreduzierung betrachtet. Das Objekt besteht aus 91536 Dreiecken. Die vollständigen Ergebnisse sind in den Tabellen 6.14, 6.15 und 6.16 zusammengefasst. Die in Tabelle 6.16 aufgeführten „XXX" bedeuten, dass in diesem Testfall ein Programmfehler auftrat (s. Anhang A.4).

Anzahl Restdreiecke	Jacobi-Fehler	Innenwinkel	Längenverhältnis
25408	XXX	XXX	XXX
18458	XXX	XXX	XXX
7184	XXX	XXX	XXX
1526	XXX	XXX	XXX
584	0	$22° < \alpha < 164°$	3.87
244	0	$25° < \alpha < 162°$	5.11
86	36	$2° < \alpha < 178°$	67.71

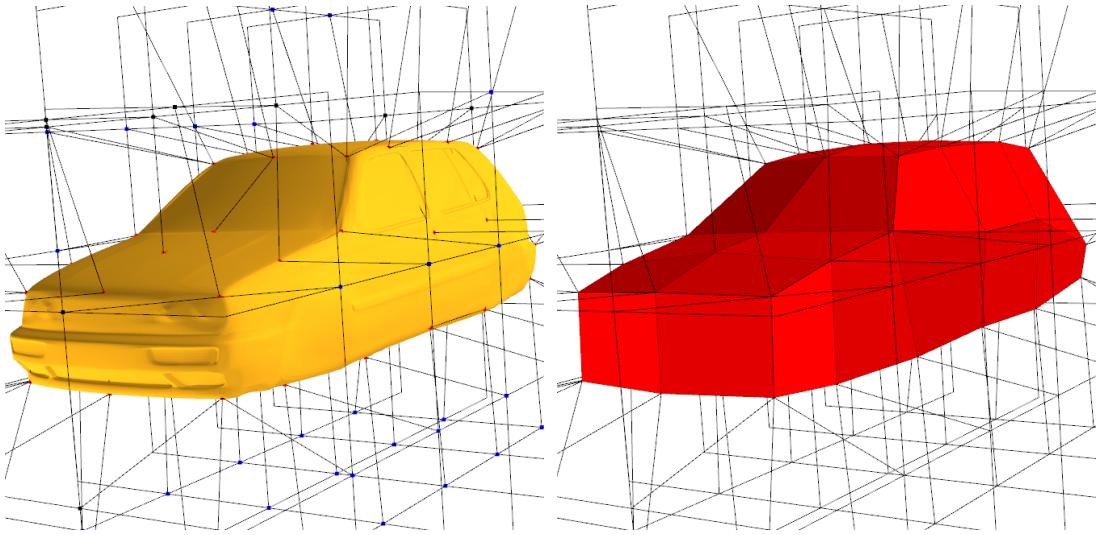
Tabelle 6.16: Einfluss der initialen Objektreduzierung für Testfall 8.

6.6 Komplexer Testfall

Für den komplexen Testfall findet sowohl das Modell des Golf III, als auch das initiale Grobgitter aus Testfall 8 Verwendung. Die Benutzung der Glätter wird durch den Anwender initiiert und beruht auf Erfahrungswerten. Sind in einer Hierarchiestufe Glätter aufgerufen worden, so werden die Qualitätskriterien vor und nach deren Verwendung in Tabelle 6.17 aufgeführt.

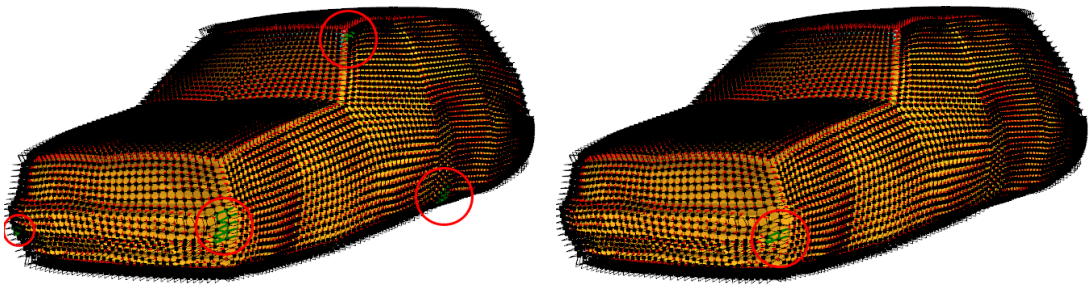
Im Gegensatz zu Unterteilungsschritt 1 und 2, in denen keine Glätter verwendet werden, werden in Iteration 3 und 4 der Oberflächen-Jacobi-Glätter jeweils zehn mal aufgerufen. In Tabelle 6.17 sind die Resultate der Glättung mit „G" markiert.

Wie in Testfall 8 konnte das Grobgitter aufgrund des zu geringen Arbeitsspeichers nicht mehr als vier mal unterteilt werden.



(a) *Originalobjekt im Gitter.* (b) *Durch das Gitter angenäherte Objektoberfläche.*

Abbildung 6.33: *Gitter für den komplexen Testfall.*



(a) *Vor der Benutzung der Glätter.* (b) *Nach der Benutzung der Glätter.*

Abbildung 6.34: *Gitter für den komplexen Testfall nach vier Unterteilungen. Die Jacobi-Fehler sind markiert.*

Unterteilungsstufe	Jacobi-Fehler	Innenwinkel	Längenverhältnis	Gitterzellen
0	0	$45° < \alpha < 157°$	2.88	160
1	0	$38° < \alpha < 157°$	2.83	1280
2	0	$30° < \alpha < 159°$	3.82	10240
3	1	$25° < \alpha < 167°$	4.99	81920
3 G	0	$25° < \alpha < 167°$	4.99	81920
4	14	$10° < \alpha < 178°$	7.34	655360
4 G	2	$12° < \alpha < 179°$	7.25	655360

Tabelle 6.17: Werte der Qualitätskriterien für den komplexen Testfall.

6.7 Grobgittergenerator

HaGrid3D stellt in erster Linie eine Benutzerschnittstelle für die Erzeugung und Bearbeitung eines geeigneten Grobgitters für ein bestimmtes Objekt zur Verfügung. Der einzige Automatismus innerhalb von *HaGrid3D* ist der Inklusionstest zur Klassifizierung von Grobgitterknoten innerhalb des Objektes (s. Kap. 5.2). Im Folgenden wird die Korrektheit des Algorithmus validiert.
Wichtigster Bestandteil des Inklusionstests ist die Berechnung des vorzeichenbehafteten Volumens von Tetraedern. Die korrekte Funktionalität dieser Teilfunktion wurde an Hand verschiedener Beispiele verifiziert. Grundlegend wichtig ist es, beim Zählen der Schnitte zu beachten, dass in verschiedenen Fällen Fehlinformationen zustande kommen könnten. Dies passiert im-

mer dann, wenn ein Schnitt entweder innerhalb einer Kante, eines Eckpunkts oder in einer Ebene auftritt. Bei dem in *HaGrid3D* verwendeten Algorithmus werden diese zweideutigen Schnitte umgangen, indem im Falle eines solchen ein neuer Referenzpunkt für den gerade zu überprüfenden Grobgitterknoten randomisiert erzeugt wird und sämtliche Dreiecke des Objektes erneut auf Schnitt überprüft werden. Da die oben erwähnten, zweideutigen Schnitte extrem selten auftreten, stellt dieses Verfahren kein Problem für die Laufzeit dar.
Nachdem für eine durch einen Grobgitterknoten und einen randomisiert gewählten Referenzpunkt definierten Strecke die Anzahl der Schnitte mit allen Dreiecken des Objektes überprüft wurde, kann an Hand dieser festgestellt werden, ob sich der Grobgitterknoten innerhalb oder ausserhalb des Objektes befindet. Bei ungerader Anzahl befindet sich der Grobgitterknoten innerhalb des Objektes, bei gerader außerhalb: Liegt der Grobgitterknoten außerhalb des Objektes, so schneidet er das Objekt entweder gar nicht oder in mindestens zwei Punkten. Da sowohl Grobgitterpunkt und Referenzpunkt außerhalb des Objektes liegen, muss die Strecke, wenn sie in das Objekt eintritt auch wieder aus diesem austreten. Ansonsten liegt der Grobgitterknoten entweder innerhalb des Objektes oder die Anzahl der Schnitte ist durch zweideutige Schnitte verfälscht. Da der letzte Fall aber durch die Funktionalität des Inklusionstests ausgeschlossen werden kann, ist die Korrektheit des Verfahrens gegeben.

6.8 Löser

Zur Validierung des GPU-Lösers wird eine geeignete Testfunktion definiert:

$$\mathbf{u}_0 := x(X-x) \cdot y(Y-y) \cdot z(Z-z) \rightarrow \mathbb{R}, \mathbf{u}_0 : [0,X] \times [0,Y] \times [0,Z]$$

Die Funktion wird analytisch differenziert und der Vektor $-\Delta\mathbf{u}_0$ als Eingabe für den Löser verwendet: $-\Delta\mathbf{u} = -\Delta\mathbf{u}_0$ (s. Abb. 6.35). Die zu berechnende Lösung $\mathbf{u}$ ist somit analytisch bekannt. Zusätzlich wird eine Referenzlösung auf der CPU berechnet. Zur Validierung werden Visualisierungen der berechneten Skalarfelder $\mathbf{u}$ verglichen.
Abbildung 6.36 zeigt keine sichtbaren Unterschiede zwischen der GPU- und der CPU-Lösung. Weiterhin wurde die Lösung für das Gitter des Testfalls 6 (Klotz) berechnet (s. Abb. 6.37). Als abschließender Test wurde das Gitter des Testfalls 8 (nach drei Unterteilungen) an den GPU-Löser übergeben (s. Abb. 6.38).

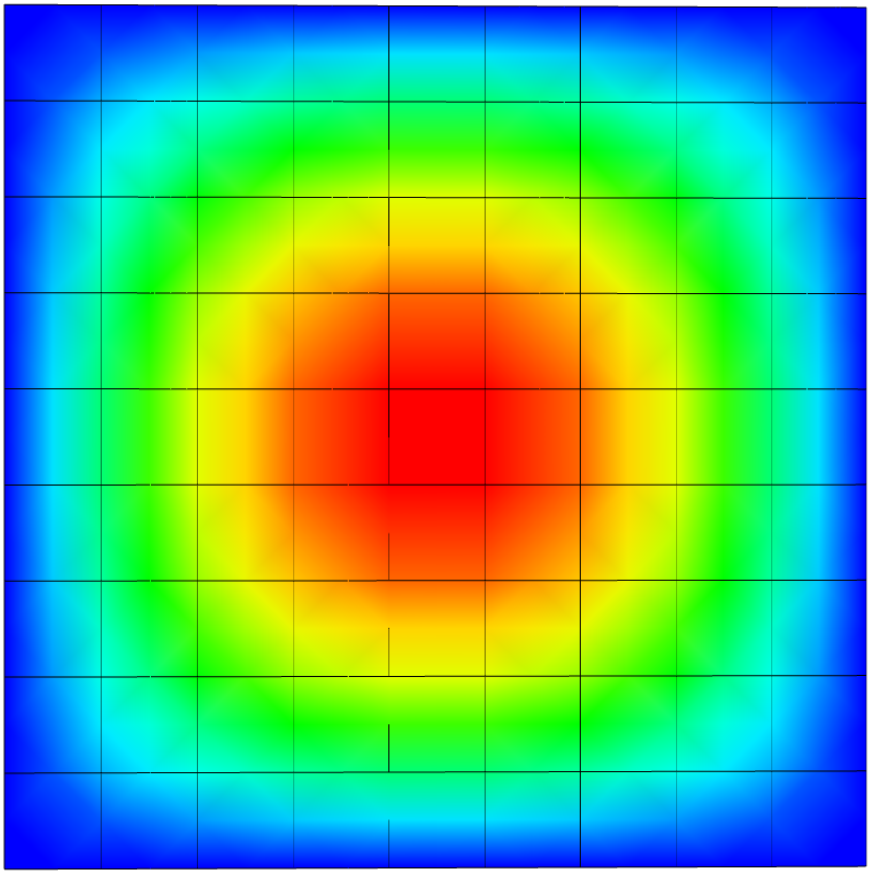


Abbildung 6.35: *Das initiale Skalarfeld wird auf einem regulären Gitter dargestellt.*

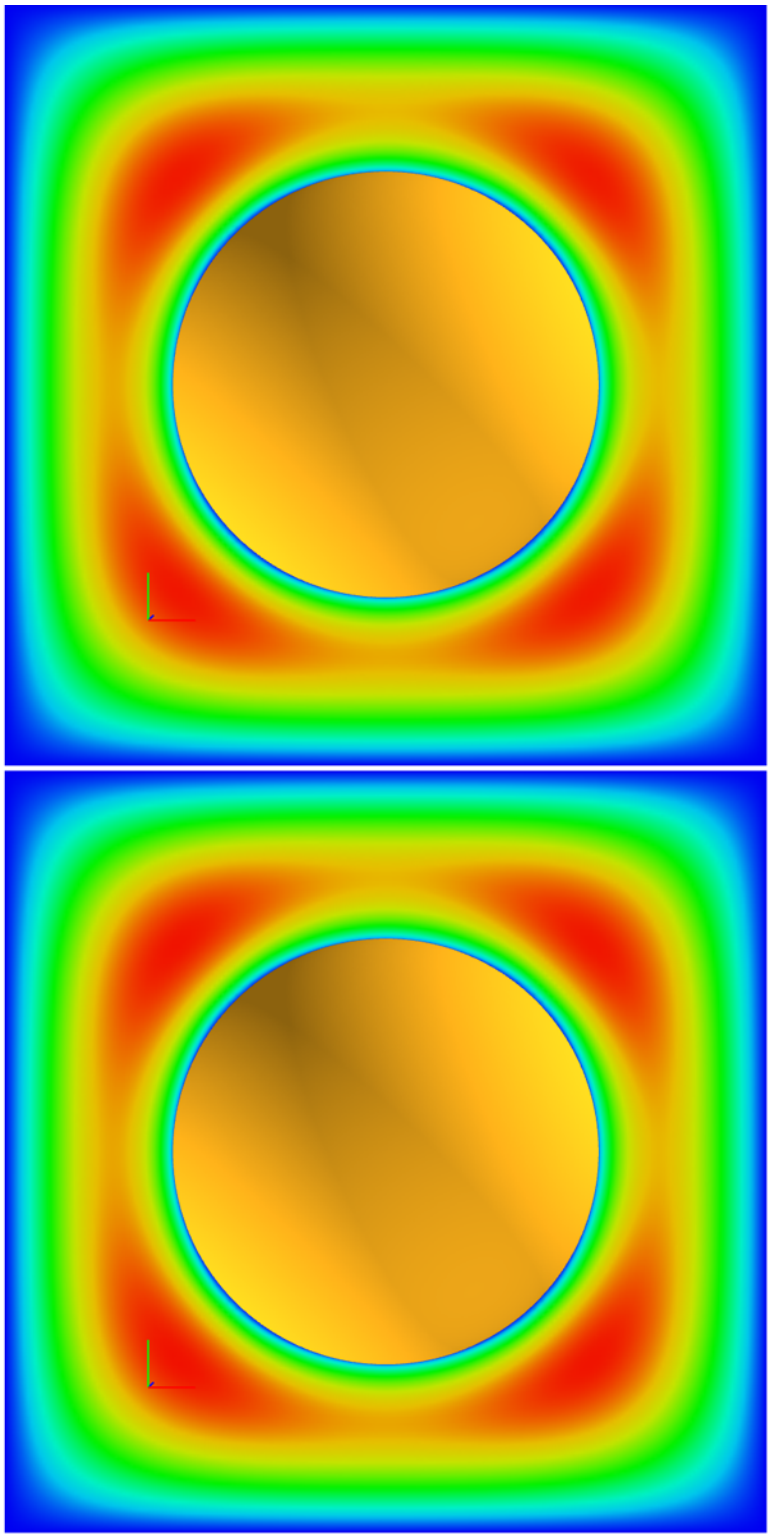


Abbildung 6.36: *Vergleich zwischen der GPU- und der CPU-Lösung. Das obere Bild zeigt das Skalarfeld der Lösung auf der GPU und das untere die Lösung auf der CPU.*

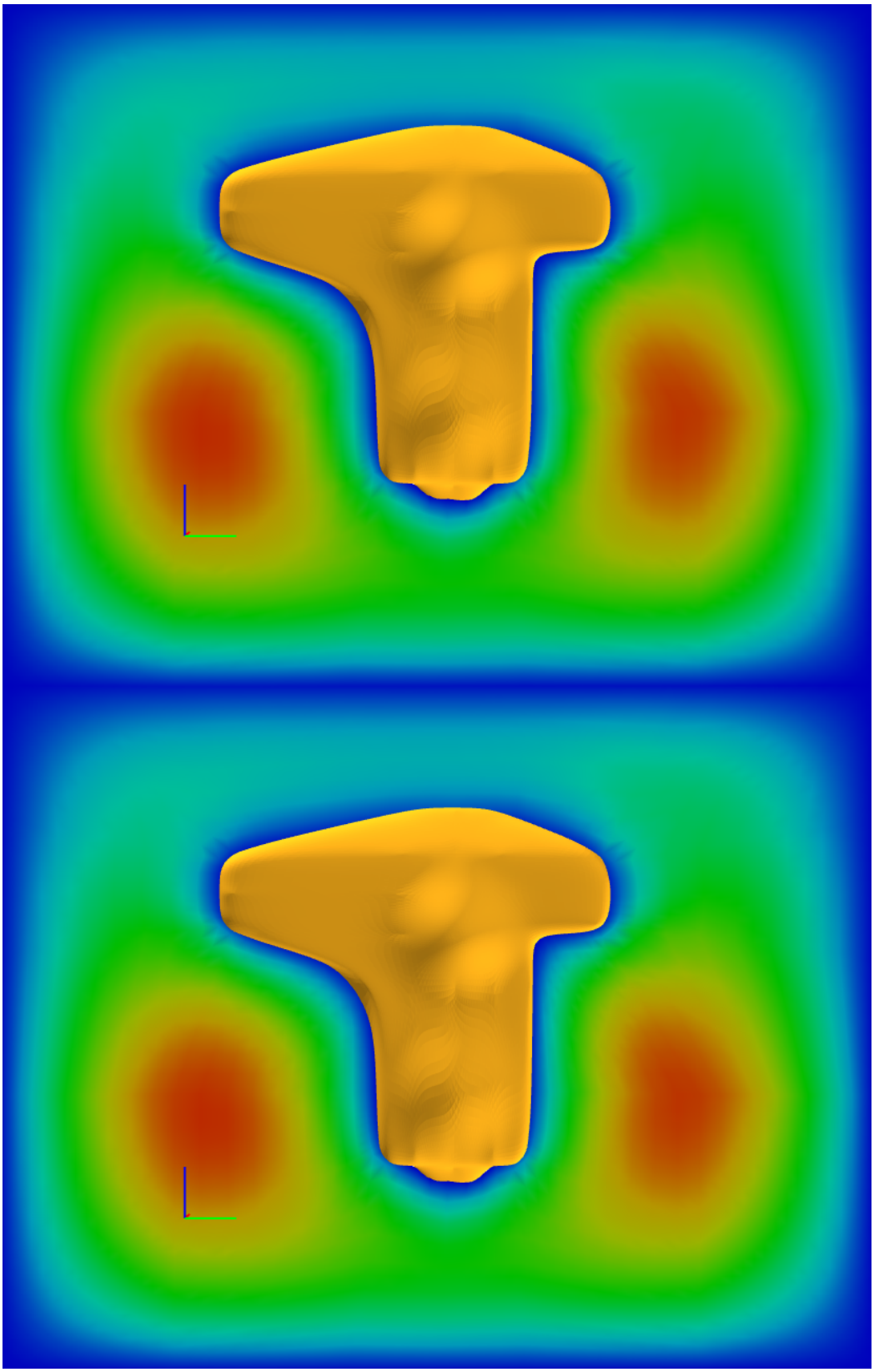


Abbildung 6.37: *Vergleich zwischen der GPU- und der CPU-Lösung. Das obere Bild zeigt das Skalarfeld der Lösung auf der GPU und das untere die Lösung auf der CPU.*

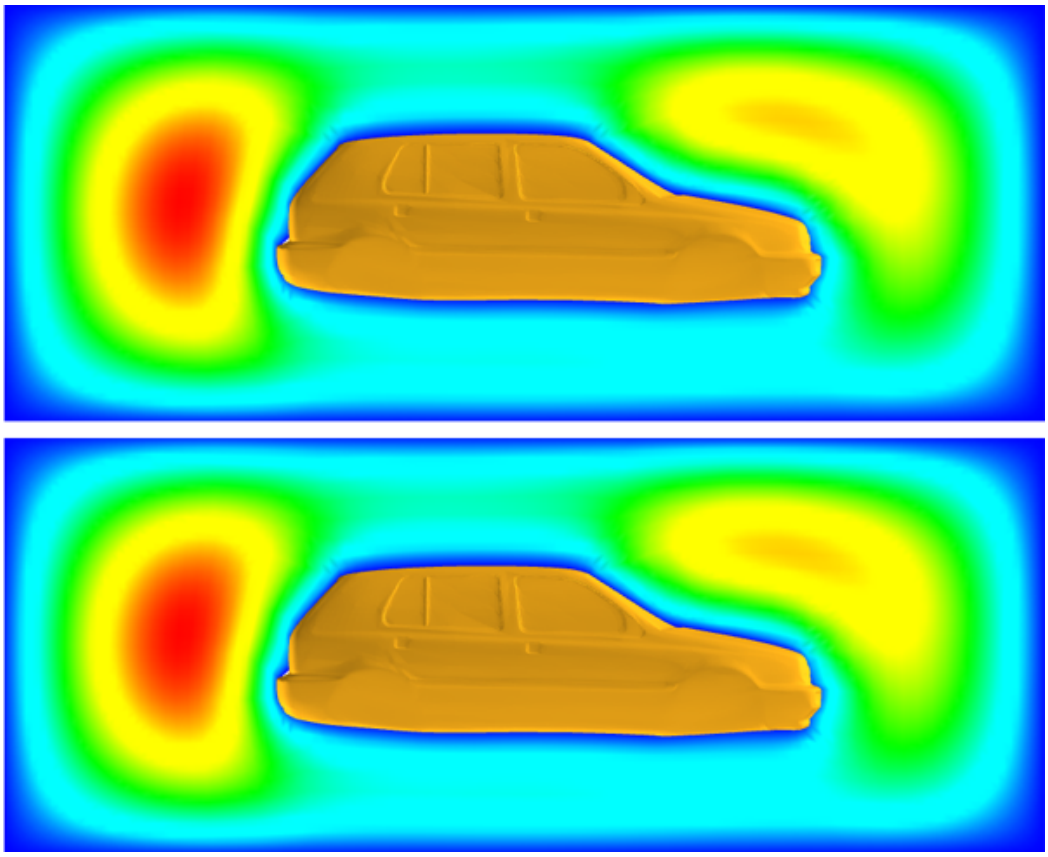


Abbildung 6.38: *Vergleich zwischen der GPU- und der CPU-Lösung. Das obere Bild zeigt das Skalarfeld der Lösung auf der GPU und das untere die Lösung auf der CPU.*

6.9 Leistungstests

In diesem Kapitel soll das Laufzeitverhalten sowie der Speicherverbrauch der Anwendung analysiert werden. Als Testsystem diente ein Athlon 64 3000+ mit 1.5 Gigabyte Hauptspeicher.

Laufzeitverhalten

In Tabelle 6.18 ist die Laufzeit für die Erzeugung des Grobnetzes der Parametrisierung („P") sowie für die ersten fünf Unterteilungsschritte und die jeweils darauffolgenden Qualitätstests des Gitters aufgelistet („Q"). Kein Eintrag in der Tabelle („–") bedeutet, dass diese Unterteilungsstufe nicht mehr erreicht werden konnte, da nicht genügend Arbeitsspeicher zur Verfügung stand.

Speicherverbrauch

In Kapitel 5.5 wurde die Datenstruktur des Gitters beschrieben. Neben den dort beschriebenen Referenzen der Elemente untereinander, den Koordinaten und dem Randstatus werden noch einige weitere Daten gespeichert. Die vier verschiedenen Elemente (Knoten, Kante, Fläche

Testfall	Laufzeit in Sekunden									
	P	1	Q1	2	Q2	3	Q3	4	Q4	5	Q5
Kugel	28,3	4,7	0,0	5,5	0,1	8,6	0,5	24,6	4,1	112,4	34,9
Auto	6,5	5,9	0,0	12,6	0,4	59,1	2,9	247,5	22,9	–	–

Tabelle 6.18: Tabelle zum Laufzeitverhalten.

und Hexaeder) werden in vier Vektoren gespeichert. Jedes Element speichert seinen Index innerhalb des jeweiligen Vektors als Integerzahl. Die Knoten besitzen einen weiteren Index, der jedoch nur für Randknoten relevant ist, er dient der Kommunikation mit dem Modul MAPS. Die Kanten speichern weiterhin eine Richtungsinformation (X-, Y- oder Z-Richtung), um das Gitter gerichtet durchlaufen zu können. In jeder Fläche wird für die Reparatur-Prozedur das Kantenverhältnis festgehalten. Sowohl Kanten als auch Flächen speichern bei der Unterteilung eine Referenz auf den auf ihrem Mittelpunkt erzeugten Punkt (s. Kap. 5.5.2), um die Unterteilung zu beschleunigen. Die Hexaeder speichern ihren Jacobi-Wert, und einen Booleschen Wert, der anzeigt, ob ein Jacobi-Fehler vorliegt. Dieser wird im Folgenden als Integerwert behandelt. In Tabelle 6.19 ist zusammenfassend der Speicherbedarf aller Gitterelemente aufgelistet. Die rechte Spalte zeigt, wie viele Elemente durch eine Unterteilung entstehen. Die Formeln dazu ergeben sich aus dem Ablauf des Unterteilungsprozesses. Dabei steht H für die Anzahl der Hexaeder, F für die Anzahl der Flächen, K für die Anzahl der Kanten und N für die Anzahl der Knoten im zu unterteilenden Netz. Der Speicherverbrauch des Gitters hängt somit nicht nur von der Anzahl der Hexaeder ab, sondern auch davon, wie diese zusammenhängen.

Element	Referenzen	Fließkomma	Integer	Unterteilungsfaktor
Knoten	0 + 6	3	3	$N + E + F + H$
Kanten	2 + 4 + 1	0	3	$2E + 4F + 6H$
Flächen	4 + 2 + 1	1	2	$4F + 12H$
Hexaeder	6 + 0	1	1	$8H$

Tabelle 6.19: Auflistung der Gitterelemente und ihr Speicherverbrauch.

Testfall	Speicherverbrauch in MB						
	0	P	1	2	3	4	5
Kugel	324	590	591	592	598	638	1102
Auto	102	178	180	187	264	769	–

Tabelle 6.20: Tabelle zum Speicherverbrauch exemplarischer Testfälle.

In Tabelle 6.20 ist der Speicherverbrauch für die Testfälle der hochaufgelösten Kugel sowie des Autos für bis zu fünf Unterteilungen aufgelistet. Die Spalte „0” stellt den initalen Speicherver-

brauch[2] nach dem Laden von Netz und Gitter dar, die Spalte „P" zeigt den Speicherverbrauch nach der Vergröberung des Netzes für die Parametrisierung. Die übrigen Spalten zeigen den Speicherverbrauch nach der entsprechenden Anzahl von Unterteilungsschritten. Bei Testfall 8 stand für die fünfte Unterteilung nicht genug Arbeitsspeicher auf dem Testsystem zur Verfügung.

6.10 Bewertung der Testergebnisse

Abschließend sollen nun die Resultate der durchgeführten Tests zunächst bewertet und anschließend diskutiert werden. Dadurch soll die Eignung der von der PG erstellten Programme, die zur rechnergestützten Strömungssimulation notwendigen Arbeitsschritte zu bewältigen, demonstriert werden.

Resultate bei der Randanpassung

Testfall 1 zeigt, dass die Randanpassung an einfache Geometrien für bis zu fünf Unterteilungen ohne die Verwendung von Glättern möglich ist, wobei die Einhaltung der Qualitätskriterien erfüllt ist. Als Tendenz kann jedoch festgestellt werden, dass mit höherer Unterteilungsstufe die maximalen und minimalen Winkel und Längenverhältnisse schlechter werden (s. Tabelle 6.3).
Wird, wie in Testfall 2 zu erkennen, bei einer einfachen Geometrie die Objektoberfläche gröber aufgelöst, werden auch in der fünften Unterteilungsstufe die Qualitätskriterien erfüllt. Diese sind aber schlechter als bei dem gleichen, höher aufgelösten Objekt. Die genauen Werte lassen sich Tabelle 6.4 entnehmen.
Während in den ersten beiden Testfällen ein als gut erachtetes Grobgitter verwendet wird, so kam in Testfall 3 ein schlechtes zur Anwendung. Bei diesem Vorgehen kommt es schon in Unterteilungsstufe eins zu einem Jacobi-Fehler. Bis zu Unterteilung fünf erhöht sich die Zahl der Fehler drastisch auf 1692. Auch die Werte der maximalen und minimalen Winkel erfüllen schon ab Unterteilungsstufe eins nicht die erforderlichen Kriterien (s. Tabelle 6.5).
Im dem Fall, dass diese einfache Geometrie eine Einbuchtung aufweist (Testfall 4), wird mit einem 1000 Gitterzellen enthaltenden, relativ fein aufgelöstem Grobgitter, begonnen. Deshalb kann drei mal unterteilt werden, wobei alle Qualitätskriterien erfüllt werden. Auch die maximalen und minimalen Winkel und das Längenverhältnis befinden sich in dieser Stufe im guten Bereich, wie in Tabelle 6.6 ersichtlich.
Wird bei dieser Geometrie ein schlechtes Grobgitter (Testfall 5) verwendet, so kommt es schon nach der ersten Unterteilung zu drei Jacobi-Fehlern. Die Anzahl erhöht sich bis zur vierten Unterteilungsstufe auf 149. Zusätzlich werden auch ab Unterteilungsstufe zwei die minimalen und maximalen Winkel sowie das Längenverhältnis schnell sehr schlecht. Für die genauen Werte wird auf Tabelle 6.7 verwiesen.
Testfall 6 wird zunächst mit einem guten Grobgitter validiert. Hierbei stellt sich heraus, dass lediglich zwei Unterteilungsstufen die Qualitätskriterien erfüllen. Danach kommt es zu 45 Jacobi-Fehlern sowie zu sehr schlechten Winkeln und einem schlechten Längenverhältnis. Die Testwerte finden sich in Tabelle 6.8.

[2] Darin sind das Betriebssystem und weitere laufende Programme mitgerechnet.

Wird ein schlechtes Grobgitter für die gleiche Geometrie (Testfall 7) verwendet, so stellt sich ein anderes Ergebnis dar: Während es schon in Unterteilungsschritt 2 zu zwei Jacobi-Fehlern kommt, deren Anzahl auf 77 in Stufe 4 steigt, so sind die minimalen und maximalen Winkel bis Unterteilungsstufe 3 deutlich besser. Das gleiche gilt für das Längenverhältnis. Die exakten Werte lassen sich Tabelle 6.9 entnehmen.

Bei Testfall 8 wird ein noch komplexeres und praxisnäheres Objekt gewählt, für den ein gutes Grobgitter entworfen wurde. Ohne die Verwendung von Glättern kann das Grobgitter zwei Mal unterteilt werden, erst in Stufe drei kommt es zu einem Jacobi-Fehler. Die hier auftretenden maximalen und minimalen Winkel sind jedoch noch im Toleranzbereich des von der PG entwickelten Lösers. Ab Stufe vier ist das resultierende Gitter nicht mehr akzeptabel, da sowohl die Winkel schlecht werden als auch die Zahl der Jacobi-Fehler weiter ansteigt. Für die genauen Werte wird auf Tabelle 6.10 verwiesen.

Das Grobgitter hat starken Einfluss auf die Qualität der durch Unterteilung entstehenden Gitter, insbesondere die Entstehung ungültiger Elemente wird durch die Verwendung ungeeigneter Grobgitter begünstigt (Testfall 1 und 3). Testfall 4 und 5 stützen ebenfalls die anfängliche These, insbesondere beim Erfassen der Einbuchtung scheitert das „schlechte" Grobgitter. Aus den Resultaten des ersten Testfalls lässt sich folgern, dass weitere Unterteilungen zu ungültigen Gittern führen würden, da die Qualitätsmaße sich in jedem Unterteilungsschritt verschlechtern. Dies ist darauf zurückzuführen, dass in der Implementierung nur kartesische, reguläre Gitter erzeugt werden, was eine Unterteilung ohne Verschlechterung der Winkel und Längenverhältnisse für diese Testgeometrie ausschließt.

Die Ergebnisse aus Testfall 2 führen zu der Schlussfolgerung, dass hochaufgelöste Geometrien zu besseren Resultaten bei der Gitterunterteilung führen (vgl. Testfall 1 und 2).

Dass die Eignung eines Grobgitters letztlich auf Erfahrungswerten des Anwenders beruht, wird durch die Resultate der Testfälle 6 und 7 deutlich, dass das zu Testbeginn als „gut" bezeichnete Gitter aus Testfall 6 erzeugt zwar weniger Jacobi-Fehler, die Winkel und das Längenverhältnis nehmen aber erheblich früher ungünstige Werte an.

Im Testfall 8 wird deutlich, dass Gitter für komplexe Geometrien zwar prinzipiell aus einem geeignetem Grobgitter erzeugt werden können, der Jacobi-Fehler in Unterteilungsstufe drei aber macht deutlich, dass ohne weitere Maßnahmen eine Gitterunterteilung mit Einhaltung der Qualitätstoleranzen kaum möglich ist.

Resultate der Glätter

Erfahrungen im Umgang mit den verschiedenen Glättern haben ergeben, dass es sinnvoll ist, diese erst zu verwenden, wenn den Qualiätskriterien nicht mehr entsprochen wird. Das bedeutet bei der Validierung anhand der fein aufgelösten Kugel und einem guten Grobgitter, dass die Glätter erst in Unterteilung 5 verwendet werden.

Des Weiteren kann gesagt werden, dass sie bei der Verbesserung der maximalen und minimalen Winkel sowie bei dem Längenverhältnis eher kontraproduktiv sind. Dabei muss erwähnt werden, dass die Grobgitterunterteilung bei einem kartesischen, regulären Gitter ein prinzipielles Problem darstellt. Andere Arbeiten haben gezeigt, dass die Verwendung weniger stark eingeschränkter Hexaedergitter bessere Ergebniss liefern [14]. Im komplexen Testfall soll dargelegt werden, dass die Anwendung von Glättern zur Verhinderung von Zelldurchdringungen von Vorteil ist.

Resultate des komplexen Testfalles

Als Modell kam hier wieder das praxisnahe Beispiel des Golf III zum Einsatz. Bis zur Unterteilungsstufe zwei war der Einsatz von Glättern nicht nötig, da alle Qualitätskriterien erfüllt wurden. Der erste auftretende Jacobi-Fehler in Stufe drei konnte durch Anwendung des Jacobi-Oberflächenglätters behoben werden. Die übrigen Qualitätskriterien wurden jedoch nicht verbessert. In der folgenden Unterteilungsstufe traten 14 Jacobi-Fehler auf, von denen zwölf ebenfalls durch die Anwendung des Jacobi-Oberflächenglätters behoben werden konnten. Trotz verschiedener heuristischer Ansätze, das Grobgitter zu verbessern oder die verbleibenden Zelldurchdringungen durch Glättereinsatz zu beheben, konnte dieses Ziel in diesem Testfall nicht erreicht werden.

Resultate des GPU-Lösers

Für den von der PG realisierten GPU-Löser lässt sich Folgendes festhalten: Der Löser liefert bei einem visuellen Vergleich identische Resultate zu einer CPU-basierten Referenzimplementierung (vgl. Abbildungen 6.36, 6.37 und 6.38). Dies gilt für alle exemplarisch evaluierten Testfälle und deckt sich mit den Untersuchungen von Göddeke, Strzodka und Turek [16]: Eine numerisch höhere Genauigkeit ist ohne zusätzlichen mathematischen Aufwand nicht zu erreichen.
In Kombination mit der schlechten Konditionierung der aufgestellten Finite-Differenzen-Matrix kann dies zu unverhältnismäßig hohen Iterationszahlen führen.
Aufgrund eines bestätigten Fehlers im NVIDIA-Treiber konnten wichtige Optimierungen der Implementierung nicht umgesetzt werden. Deshalb wurde auf einen direkten Leistungsvergleich zwischen CPU und GPU verzichtet.
Abschließend kann gesagt werden, dass der Löser die von der PG gestellten Anforderungen erfüllt. Er demonstriert, dass die mit *InGrid3D* erstellten Gitter prinzipiell zur Strömungssimulation unter Verwendung des Finite-Differenzen-Verfahrens geeignet sind.

6.11 Fazit

Aus den erzielten Ergebnissen der Projektgruppe ergeben sich mehrere Schlußfolgerungen. Die Qualität der durch Unterteilung erzeugten Hexaedergitter steigt und fällt mit der Qualität der initial verwendeten Grobgitter. Leider konnten im Verlauf der PG keine konkreten Kriterien definiert werden, die ein „gutes" Grobgitter beschreiben. Der Erfolg bei der Erstellung eines geeigneten Gitters hängt also von der Erfahrung des Anwenders ab. Bis zu einem gewissen Grad lassen sich bei der Unterteilung auftretende Probleme durch die Verwendung der Glätter beheben, aber auch hier ist die Erfahrung des Anwenders wichtig, um entscheiden zu können, welcher Glätter im konkreten Problemfall eine Verbesserung bringen könnte. Die Komplexität des Geometrieobjektes ist ebenfalls ein wichtiger Faktor: Bei „einfachen" Objekten lassen sich auch relativ einfach die dazugehörigen Grobgitter entwerfen, deren Qualität oft schon so gut ist, dass der Einsatz von Glättern kaum noch Verbesserungen bringt. Bei komplexen Objekten ist sowohl der Entwurf eines geeigneten Grobgitters eine anspruchsvolle Aufgabe, als auch die Entscheidung, zu welchem Zeitpunkt welcher Glätter eingesetzt wird. Ebenfalls spielt es eine Rolle, in welcher Unterteilungsstufe des Gitters ein Glätter angewandt wird. So kann beispielsweise „präventiv" in jeder Unterteilungsstufe geglättet werden, ein Einsatz der Glätter erst bei

auftretenden Problemen stellt ebenso eine geeignete Vorgehensweise dar.
Die Parametrisierung der Objektoberfläche erfordert ebenfalls die Erfahrung seitens des Anwenders. Die Wahl des Basisnetzes, welches durch Vergröberungsschritte erzeugt wird, hat Einfluß auf die Resultate der Unterteilung. Zu starke Vergröberungen haben zur Folge, dass teilweise ungültige Positionen für die neu entstandenen Randpunkte berechnet werden, da die Approximation des ursprünglichen Objektes zu schlecht ist. Eine zu schwache Vergröberung wiederum führt zu langen Laufzeiten, da dann zu häufig der Snake-Algorithmus eingesetzt werden muss.
Die Fähigkeit der von der PG erstellten Software, ein zur Strömungssimulation geeignetes Rechengitter zu erzeugen, basiert also in gewissem Maße auf den Entscheidungen des Anwenders. Bei geschickter Anwendung der von den Programmen *InGrid3D* und *HaGrid3D* zur Verfügung gestellten Werkzeugen ist es also möglich, ein Hexaedergitter zu erzeugen, dessen Größe nur durch den zur Verfügung stehenden Arbeitsspeicher limitiert wird.
Mit Hilfe eines solchen Gitters ist es möglich mittels des Finite-Differenzen-Verfahrens Strömungen um das Objekt zu simulieren. In der PG konnte dies durch den implementierten Löser auf der GPU nachgewiesen werden. Damit wurde auch das ursprüngliche Ziel der PG, die Grafikkarte auch abseits der üblichen Verwendungszwecke zu benutzen, erreicht. Gleichzeitig bewiesen die erfolgreich durchgeführten Simulationen die prinzipielle Praxistauglichkeit der erstellten Gitter.

6.12 Ausblick

Die von der PG erstellte Sammlung von Werkzeugen bietet an vielen Stellen die Möglichkeit, Erweiterungen und Verbesserungen einzubringen. So ist z.B. die Weiterentwicklung und Verbesserung der Glättungsalgorithmen denkbar. Beispielsweise wurde ein von der PG erdachter Ansatz, der die Randelemente an Hand der Oberflächennormalen des Objektes ausrichtet, lediglich aus Zeitgründen nicht mehr implementiert. Des Weiteren besteht die Möglichkeit, zusätzliche Schnittstellen zu anderen Lösern zu implementieren, so könnte zum Beispiel eine komplette Mehrgitterhierarchie exportiert werden.
Da *InGrid3D* die Gitterinformationen auf Basis eines polyedrischen Komplexes speichert, ist auch die Verwendung von unstrukturierten Grobgittern möglich. Die Einschränkung auf reguläre, kartesische Gitter stellt lediglich einen Tribut an den verwendeten Löser dar.

Kapitel 7

Entwicklungsprozess
Ablauf, Prototypen

7.1 Zeitlicher Ablauf

7.1.1 Ablauf der PG im ersten Semester

Die Arbeit der Projektgruppe begann mit einer Seminarphase, in der sich die Teilnehmer in die Thematik einarbeiteten und über Grundlagen des Projektgruppenthemas referierten. Folgende Themen wurden behandelt:

- Mathematische Grundlagen
- Gitter
- Grafikhardware
- DirectX
- OpenGL
- GPGPU
- Qt
- Tools
- Stable Fluids

Nach der Seminarphase wurde eine Aufteilung in Einzelgruppen beschlossen, die sich folgenden Themen widmeten: GPGPU, Gitterdeformationstechniken, Parametrisierung sowie Aktive Konturen. Im Verlauf der PG kam es immer wieder zu Umstrukturierungen, so beschäftigten sich weitere Kleingruppen mit den Themen Entfernen von Selbstdurchdringungen durch lineare Optimierung, Gitterunterteilung und Qt, außerdem wurde existierende, thematisch verwandte Software evaluiert.

Die PG traf sich regelmäßig Donnerstags um 16:00 Uhr, um die Ergebnisse der Teilgruppen zu präsentieren und das weitere Vorgehen abzusprechen. Zu jedem Treffen wurde ein Protokoll angefertigt, das den jeweiligen Stand der Entwicklung, sowie die Aufgaben, die von den PG-Teilnehmern bis zum nächsten Treffen zu erledigen waren, enthielt. Zum besseren Austausch von Informationen wurden ein Online-Forum und ein Webserver eingerichtet, auf dem sämtliche Protokolle, Seminarunterlagen sowie weitere benötigte Dokumente hinterlegt wurden. Zum Ende des ersten Semesters wurden erste Prototypen im Bereich Gitterunterteilung, MAPS und Snakes erstellt. Zusätzlich wurden kurze Texte verfasst, die die bereits geleistete Arbeit zusammenfassen und der PG zu Beginn des zweiten Semesters als Grundlage dienen sollten.

7.1.2 Ablauf der PG im zweiten Semester

Nachdem sich herausgestellt hatte, dass der Gittergenerator *Cubit* den Anforderungen nicht genügen würde, wurde beschlossen, einen eigenen Grobgittergenerator zu implementieren. Die Hauptaufgaben des zweiten Semesters bestanden in der Weiterentwicklung der Prototypen und der anschließenden Integration zu *InGrid3D*. Das offizielle Ende der Projektgruppenarbeit stellt dieser Endbericht sowie die Abschlusspräsentation dar.

7.2 Prototypen

Im Laufe der PG wurden verschiedene Prototypen erstellt, um die eingesetzten Techniken auf ihre Tauglichkeit zu untersuchen und optimieren zu können. Es werden im Folgenden drei Prototypen vorgestellt, deren Techniken auch im Endprodukt zum Einsatz kommen.

7.2.1 Gitterunterteilung-Prototyp

Der Gitterunterteilung-Prototyp bietet eine grafische Benutzeroberfläche, mit der Probleme und Ergebnisse der Gitterunterteilung, Gitterglättung und Gitterkorrektur visualisiert werden können. Beim Start des Prototypen wird ein $3 \times 3 \times 3$-Gitter erzeugt, wobei der innere Hexaeder das zu umströmende Objekt repräsentiert (s. Abb. 7.1). Damit verschiedene Varianten der möglichen Flächenformen ausprobiert werden können, wird an jeder Seite des inneren Hexaeders eine Bézier-Fläche implementiert, deren Verhalten mit Hilfe von Steuervariablen beeinflusst werden kann. Für verschiedene Betrachtungsperspektiven kann das Objekt in alle Richtungen frei gedreht sowie vergrößert werden. Nach der Gitterunterteilung (s. Abb. 7.2) wird das Gitter neu gezeichnet, die entstandene Approximation der Oberfläche kann in der Objektrand-Ansicht betrachtet werden (s. Abb. 7.3). Zur besseren Evaluierung der Ergebnisse wird die Anzahl der Punkte, Kanten, Flächen, Hexaeder und Jacobi-Fehler ebenfalls in der grafischen Benutzeroberfläche angezeigt (s. Abb. 7.4).

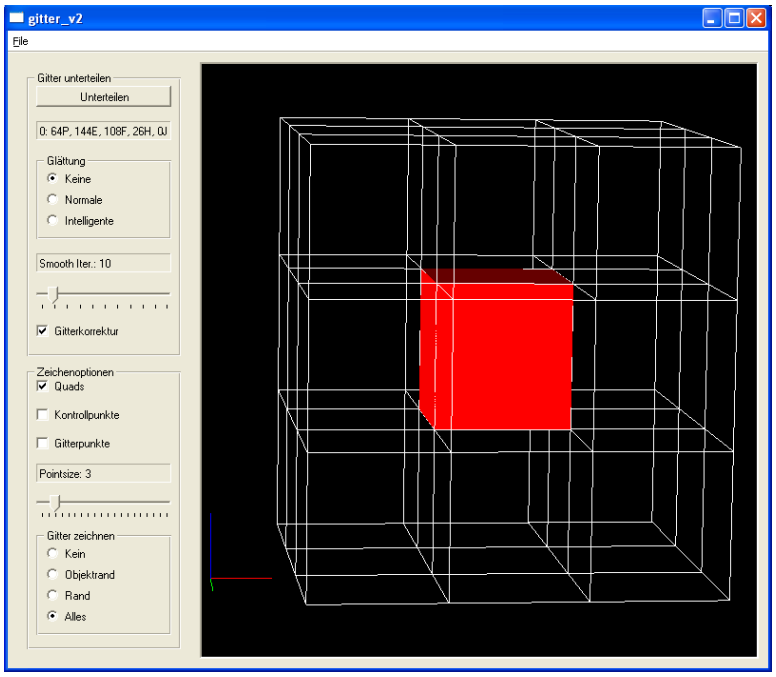


Abbildung 7.1: *Der Prototyp im Ausgangszustand.*

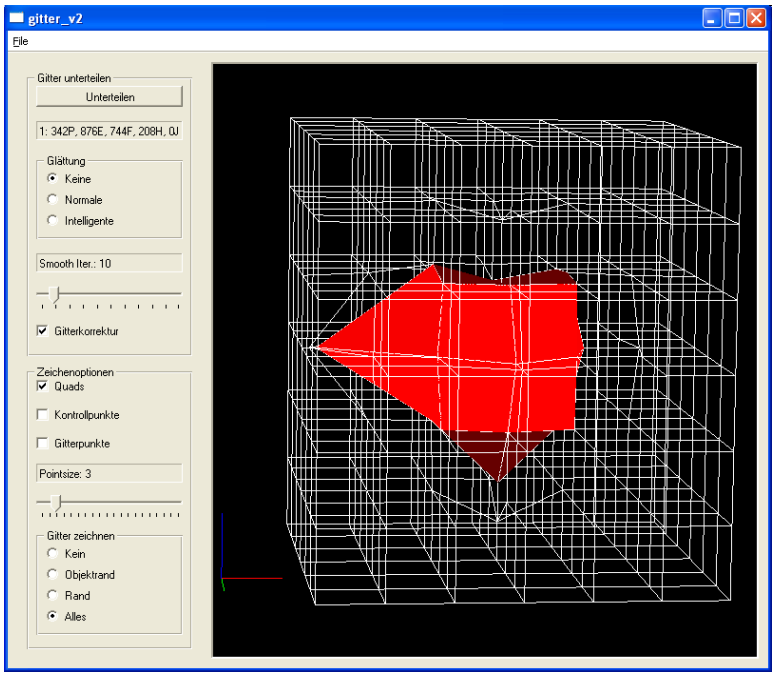


Abbildung 7.2: *Nach Drücken der „Unterteilen”-Schaltfläche wird das ursprüngliche Gitter unterteilt.*

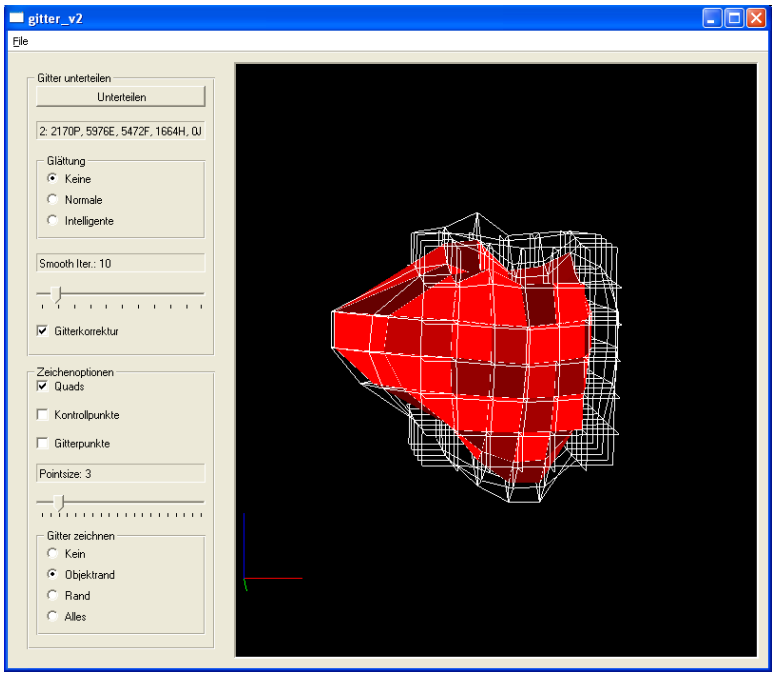


Abbildung 7.3: *Der 2. Unterteilungsschritt: Die Objektrand-Sicht dient zur besseren Betrachtung des entstandenen Gitters.*

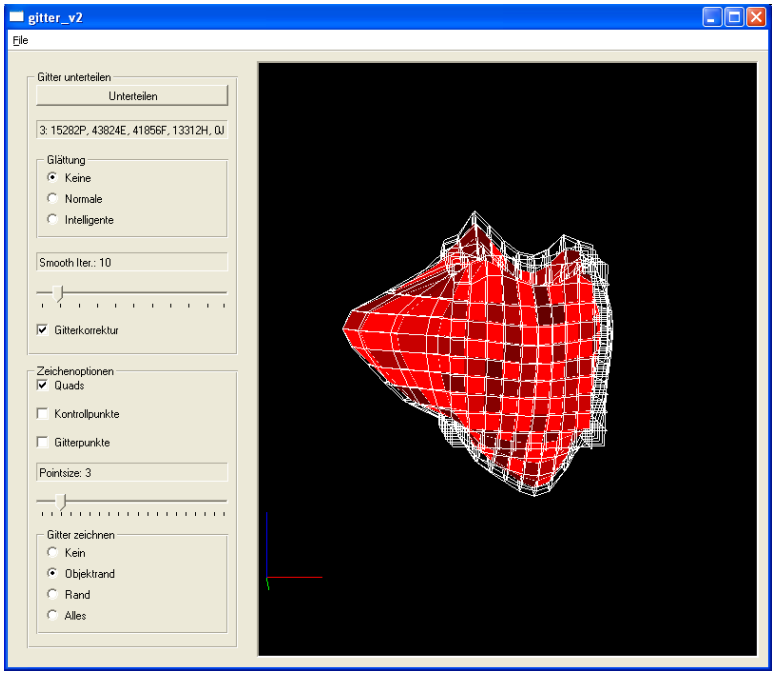


Abbildung 7.4: *Der 3. Unterteilungsschritt.*

7.2.2 MAPS-Prototyp

Der MAPS-Prototyp bietet eine grafische Oberfläche, mit deren Hilfe sich die einzelnen Schritte zur Erzeugung einer MAPS-Parametrisierung visualisieren lassen. Mit dem Start des Programms wird automatisch ein Objekt geladen, alternativ lässt sich über den Menüpunkt „Mesh Laden" ein beliebiges Objekt im STL-Format öffnen. Durch Betätigen der Schaltfläche „Get the queued IS" wird eine maximal unabhängige Menge von Knoten des geladenen Objekts markiert.

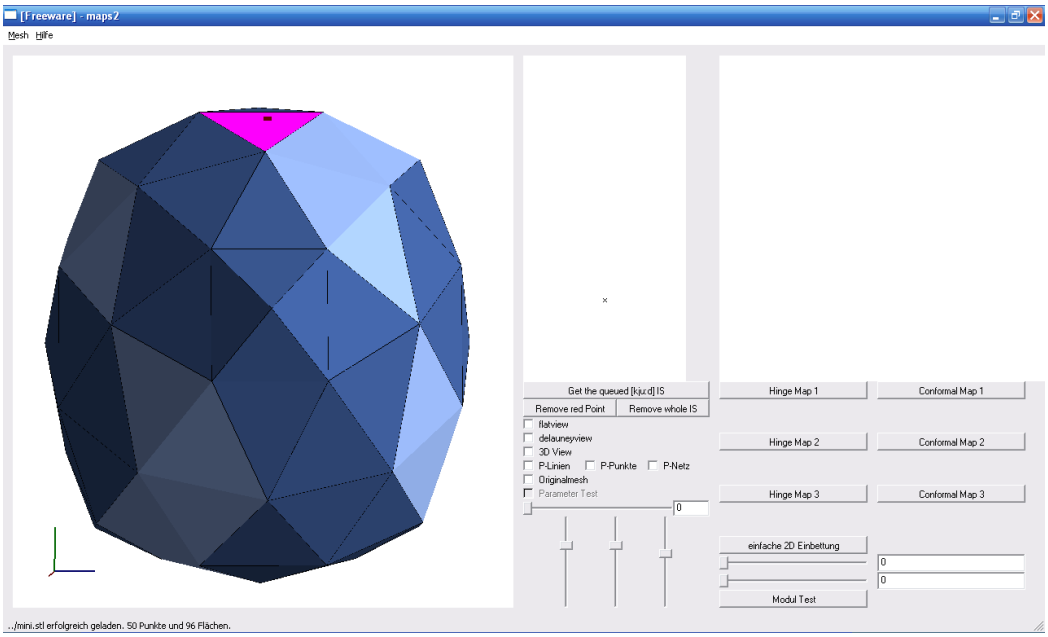


Abbildung 7.5: *Der MAPS-Prototyp.*

Dies wird visualisiert, indem die jeweiligen Eins-Ring-Nachbarschaften der Knoten aus der unabhängigen Menge aufgebaut und die zugehörigen Dreiecke dunkelblau eingefärbt werden. Es bestehen für den Benutzer nun mehrere Möglichkeiten fortzufahren. Durch Betätigen der Schaltfläche „Remove red Point" kann ein einzelner Knoten, dessen Nachbarschaft rot eingefärbt ist, entfernt werden. Durch Setzen von Häkchen in den Auswahlfeldern „flatview" und „delaunayview" werden die 2D-Einbettungen der rot markierten Eins-Ring-Nachbarschaft einmal vor der Entfernung des Knotens und einmal nach Entfernen des Knotens und Retriangulierung dargestellt. Eine andere Möglichkeit ist, die gesamte unabhängige Menge zu entfernen („Remove whole IS"). Ist hierbei das Auswahlfeld „3D View" aktiv, so wird die 3D-Ansicht des Objektes nach jeder Knotenentfernung aktualisiert.
Zur Visualisierung des Fortschritts der Parametrisierung können Parameterlinien zu den parametrisierten Punkten eingeblendet werden, wenn die Häkchen bei „P-Linien" und „P-Punkte" gesetzt sind. Durch Aktivieren des Auswahlfeldes „P-Netz" wird die Projektion der Feindreiecke auf die momentanen Grobdreiecke dargestellt. Ein Häkchen bei „Originalmesh" zeichnet das Originalnetz in der 3D-Ansicht über das Grobnetz.
Mit den drei vertikal angeordneten Schiebereglern kann der rot dargestellte Punkt innerhalb

eines Grobdreiecks durch Änderung seiner baryzentrischen Koordinaten verschoben werden. Mit dem oberen Texteingabefeld (s. Abb. 7.6, rechts unten) lässt sich ein Dreieck des Grobnetzes auswählen. Wählt man nun mit dem unteren Texteingabefeld ein zweites Grobdreieck aus, das entweder eine Kante oder einen Knoten mit dem ersten Dreieck gemeinsam hat, so kann die „Scharnier"-Abbildung oder die konforme Abbildung zur Einbettung dieser Dreiecke angezeigt werden. Die Position des roten Knotens in der 3D-Ansicht entspricht der Position des Kreuzes in der 2D-Ansicht der Einbettung.

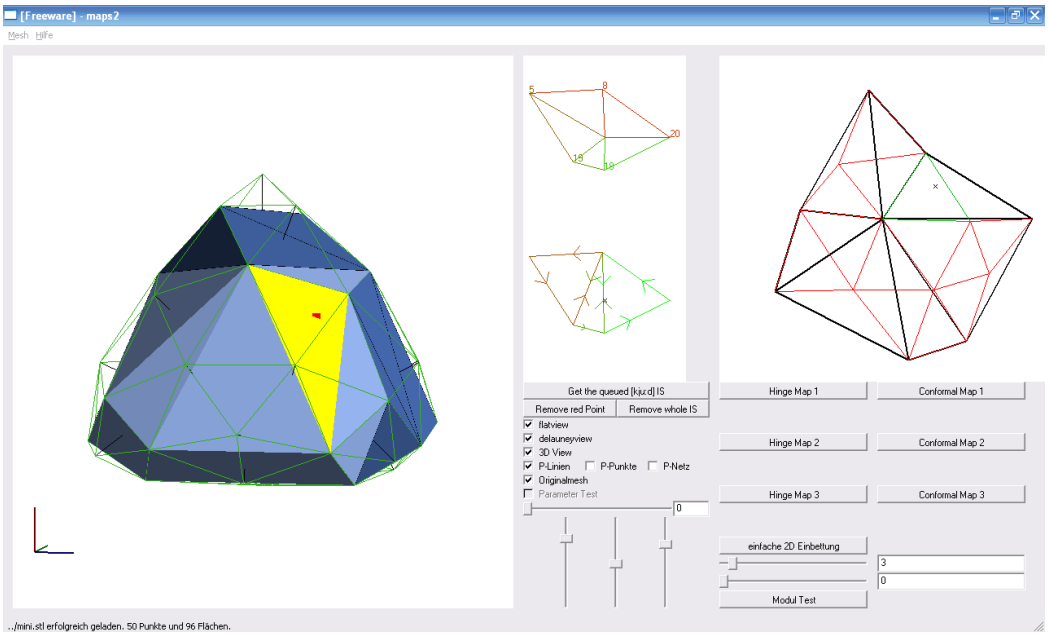


Abbildung 7.6: *Links: Dezimiertes Objekt mit der ursprünglichen Oberflächentriangulierung und Parameterlinien. Mitte: 2D-Einbettung einer Eins-Ring-Nachbarschaft und Retriangulierung. Rechts: Konforme Abbildung einer Eins-Ring-Nachbarschaft, die die beiden farblich markierten Dreiecke enthält.*

7.2.3 Snakes-Prototyp

Der Snakes-Prototyp bietet ebenfalls eine grafische Benutzeroberfläche, mit der das Verhalten der Snake visualisiert werden kann. Beim Start des Prototypen wird automatisch eine Geometrie geladen, alternativ können über das „Datei"-Menü auch andere Objekte ausgewählt werden. Die perspektivische Darstellung des Objektes kann in alle Richtungen frei gedreht werden, eine Zoom-Funktion ist ebenfalls verfügbar. Ein Klick auf die „Dijkstra"-Schaltfläche löst die Berechnung des Dijkstra-Algorithmus' aus (s. Abb. 7.7), nun wird die Snake initialisiert. Durch klicken auf die „Algorithmus starten"-Schaltfläche wird der eigentliche Algorithmus solange ausgeführt, bis eines der Abbruchkriterien (s. Kap. 5.4.3) greift. Um eine akzeptable Geschwindigkeit bei der Darstellung der Snake-Bewegung zu erreichen, wird die Snake lediglich nach jeweils 1000 Iterationen neu gezeichnet. Zur besseren Evaluierung der Ergebnisse und des Verhaltens der Snake wurde eine Transparenzfunktion implementiert (s. Abb. 7.8).

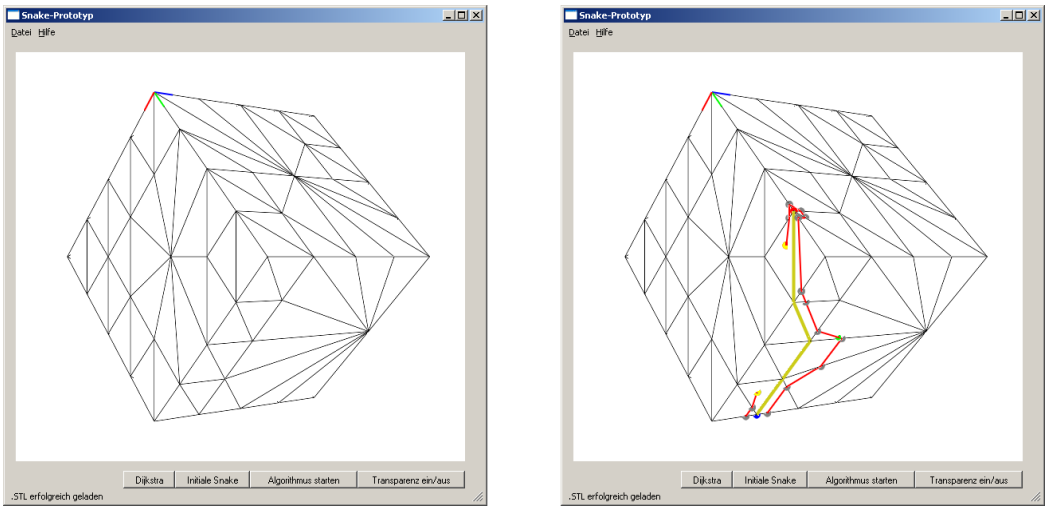


Abbildung 7.7: *Links: Der Prototyp im Ausgangszustand. Rechts: Nach Drücken der „Dijkstra"-Schaltfläche wird die Snake initialisiert.*

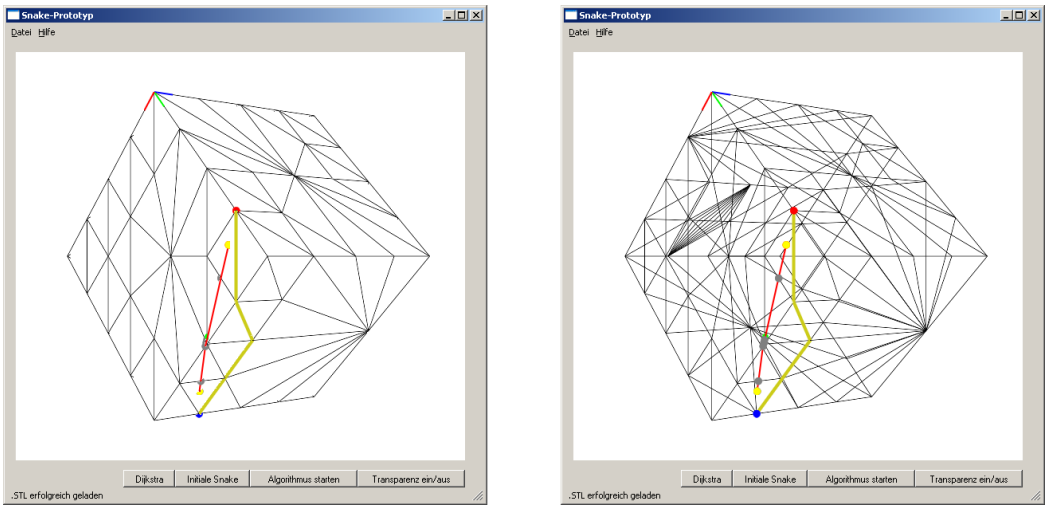


Abbildung 7.8: *Links: Nach dem Start des Algorithmus erreicht die Snake ihre endgültige Position. Rechts: Durch Einschalten der Transparenz wird das Objekt durchsichtig.*

Anhang A

Benutzerhandbuch

A.1 Installation

Systemvoraussetzungen

Betriebssystem: MS Windows ME, NT, 2000 oder Windows XP
Hardware: Prozessor mit min. 1 GHz empfohlen, min. 512 MB Arbeitsspeicher, Nvidia-Grafikkarte ab der GeForce 6-Reihe
Empfohlene Hardware: Prozessor mit 3 GHz, 1-2 GB Arbeitsspeicher, GeForce 6800-Grafikkarte oder besser

Installation der Programme InGrid3D und HaGrid3D

Die Programme *InGrid3D* und *HaGrid3D* finden sich auf der PG-Homepage[1]. Der Name der Datei ist `pg471.zip`. Diese gepackte Datei enthält die Programme *InGrid3D* und *HaGrid3D* und alle notwendigen Bibliotheken. Die ZIP-Datei muss in ein beliebiges Verzeichnis entpackt werden. Zum Starten der Programm muss die Datei `ingrid3d.exe` bzw. `hagrid3d.exe` ausgeführt werden. Die Installation ist damit abgeschlossen.

A.2 Tutorial

Die Bedienung von *InGrid3D* und *HaGrid3D* wird im Folgenden an Hand von Anleitungen (engl. *tutorials*) erläutert. Dabei wird exemplarisch ein Anwendungsfall von der Erstellung eines Objektes, über die Gittergenerierung und Unterteilung, bis hin zur Simulation und Visualisierung der Ergebnisse beschrieben.

[1] http://ls7-www.cs.uni-dortmund.de/students/projectgroups/pg471.shtml

A.2.1 Tutorial: Erzeugung eines 3D-Objektes mit Blender

Die bei der Strömungssimulation verwendeten Objekte können auf unterschiedliche Weise bezogen werden: Entweder durch Herunterladen aus einer Modelldatenbank aus dem Internet [59], durch die Exportfunktion einer CAD-Software oder aber durch die Erzeugung mittels eines 3D-Modellierungsprogrammes. Letzteres soll hier an Hand der Software Blender [4] demonstriert werden. Zunächst müssen noch einmal die wichtigsten Eigenschaften und Voraussetzungen der Objekte für die Verwendung in den von der PG erstellten Programmen erwähnt werden: Es muss als Oberflächentriangulierung vorliegen und es muss eine 2-Mannigfaltigkeit, also ein geschlossenes Volumen, bilden. Diese Eigenschaften werden bereits teilweise durch das Objekt erfüllt, das Blender beim Start lädt – ein würfelförmiges Element (s. Abb. A.1). Dieser Würfel besteht aus Viereckflächen, er kann jedoch leicht in eine Oberflächentriangulierung, also in eine Repräsentation durch Dreiecke, umgewandelt werden. Dies wird jedoch erst im letzten Schritt des Tutorials beschrieben.

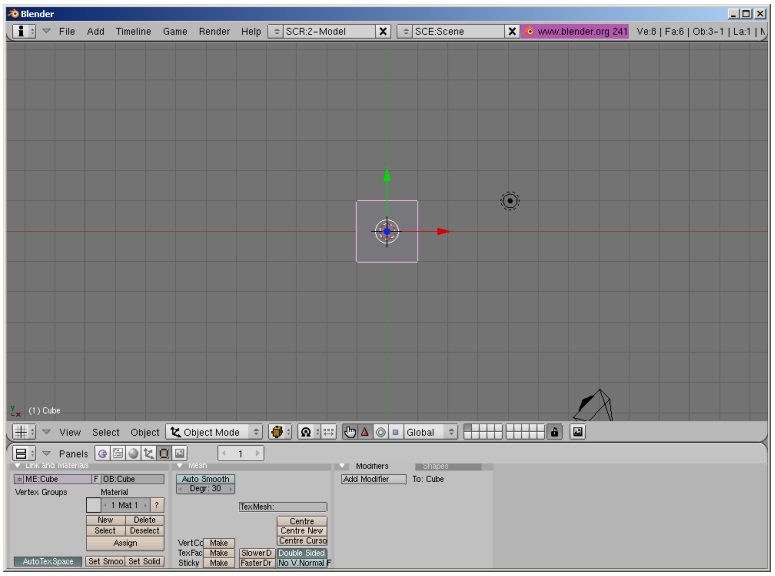


Abbildung A.1: *Blender direkt nach dem Start. Standardmäßig wird eine Szene, bestehend aus einer Lichtquelle, einer Kamera und einem Würfelobjekt geladen.*

Verfeinerung des Objektes

Als erstes soll die gegebene Geometrie verfeinert werden. Dazu muss mit der Tabulator-Taste zunächst in den sog. „Edit-Modus" von Blender gewechselt werden. Mit der Taste [A] können nun alle Punkte des Objektes ausgewählt werden (s. Abb. A.2).

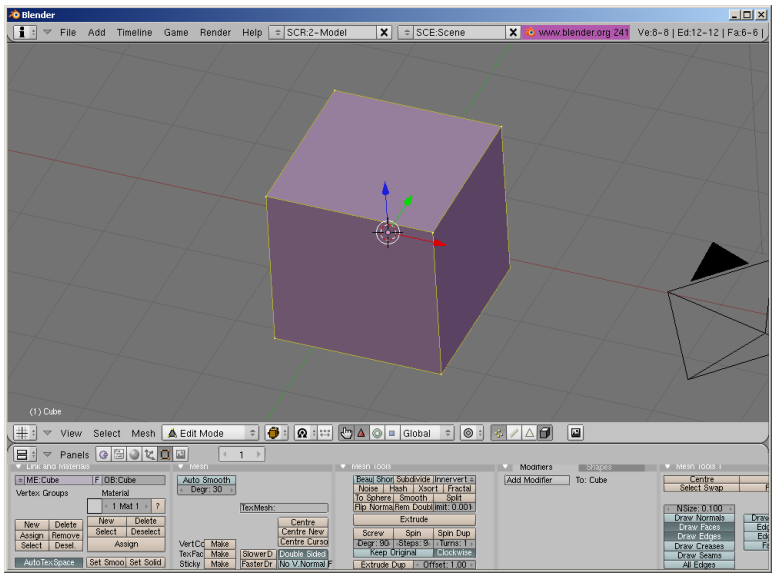


Abbildung A.2: *Alle Punkte des Würfels wurden ausgewählt.*

Nach der Auswahl der Punkte können nun alle Flächen durch Drücken der Schaltfläche „subdivide" unterteilt werden, in diesem Beispiel wurde dieser Schritt zweimal durchgeführt (s. Abb. A.3).

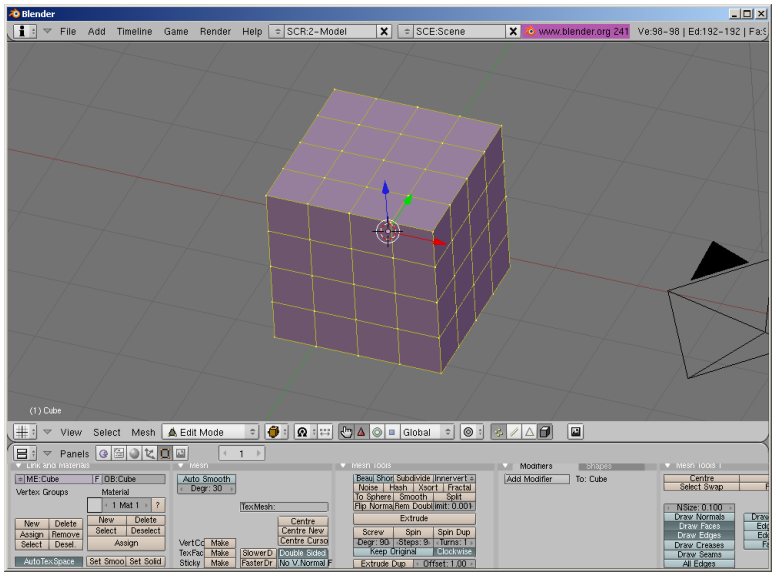


Abbildung A.3: *Der Würfel wurde zweimal unterteilt.*

Durch wiederholtes Benutzen der Schaltfläche „smooth" kann das Objekt nun geglättet werden, dieser Vorgang entspricht der in Kapitel 2.3.3 beschriebenen Laplace-Glättung. Die scharfen Kanten des Würfels werden somit abgerundet (s. Abb. A.4).

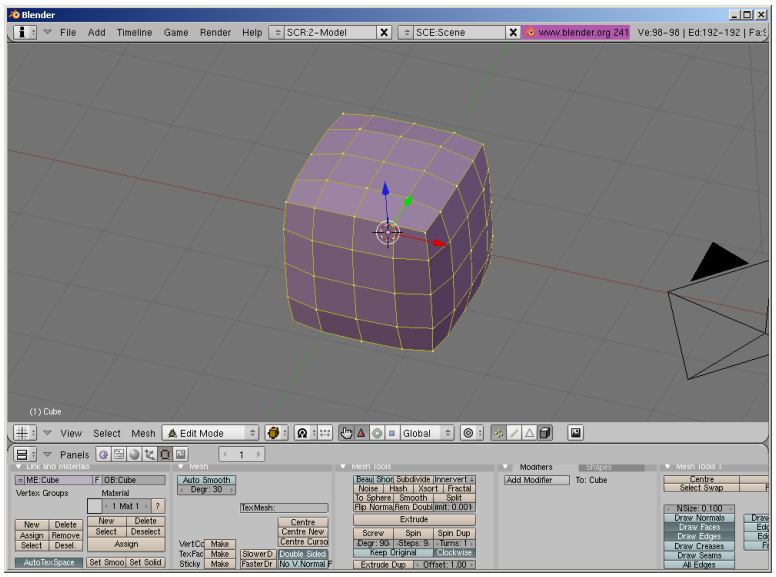


Abbildung A.4: *Das Objekt nach mehrmaliger Glättung.*

Nun wird durch die Tastenkombination [STRG]+[T] eine Triangulierung des Objektes berechnet (s. Abb. A.5).

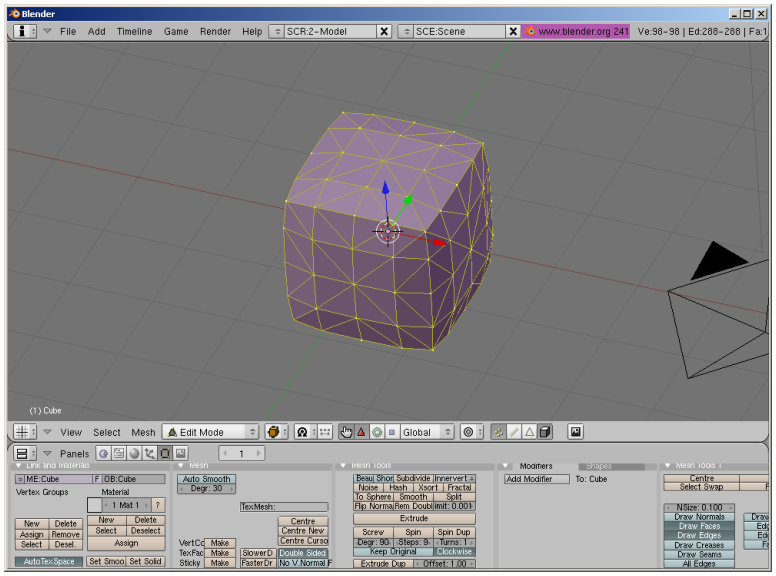


Abbildung A.5: *Der Würfel als Oberflächentriangulierung.*

Der „Edit-Modus" kann nun durch erneutes Benutzen der [Tab]-Taste verlassen werden, Blender wechselt in den „Objekt-Modus".

Exportieren des Objektes

Durch die Tastenkombination [STRG]+[W] kann die komplette Szene im Blenderformat (.blend) abgespeichert werden. Um das Objekt später als STL-Datei zu exportieren, ist das Abbspeichern der Szene unerlässlich, der Export ist grundsätzlich nur aus bereits gespeicherten Szenen möglich. Dazu wird das Objekt mit der rechten Maustaste ausgewählt, eine rosafarbene Umrandung erscheint. Nun kann über das in Abbildung A.6 ersichtliche Menü ein Export in das gewünschte Format, in diesem Falle also STL, vorgenommen werden.

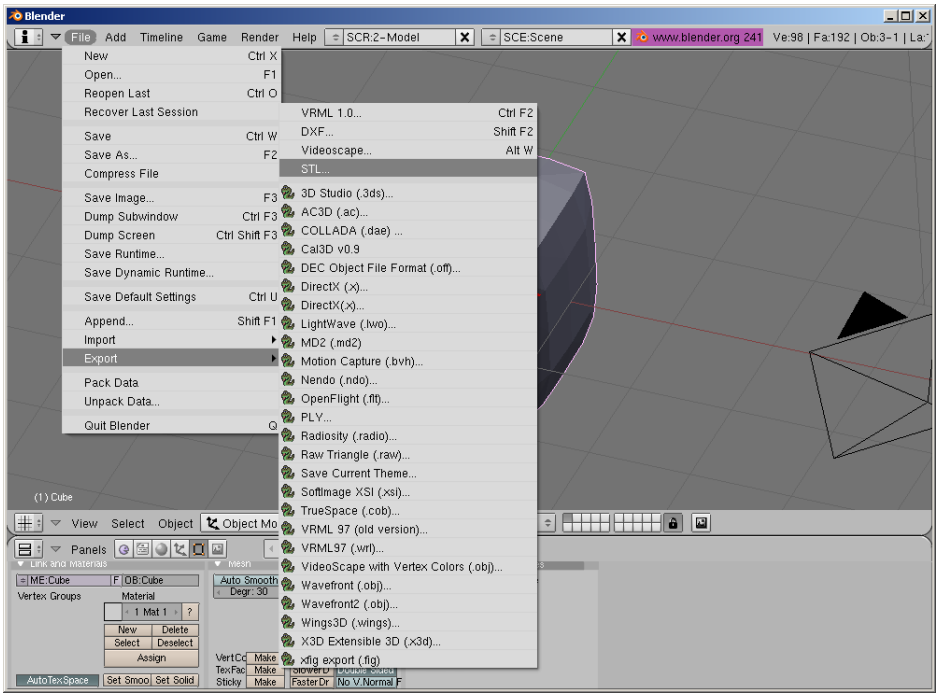


Abbildung A.6: *Blender bietet vielfältige Exportmöglichkeiten für 3D-Objekte.*

A.2.2 Tutorial: Erzeugung eines Grobgitters in HaGrid3D

Nach dem Starten von *HaGrid3D* (s. Abb. A.7) kann zunächst ein Objekt im STL-Format geladen werden. In diesem Tutorial wird das in Kapitel A.2.1 erzeugte Objekt geladen. Um die Perspektive zu verändern, kann die Ansicht bei gedrückter linker Maustaste gedreht und bei gedrückter rechter verschoben werden. Mit Hilfe des Mausrades wird der Bildausschnitt in Stufen, bei gedrücktem Mausrad und vertikaler Bewegungsrichtung der Maus stufenlos vergrößert.

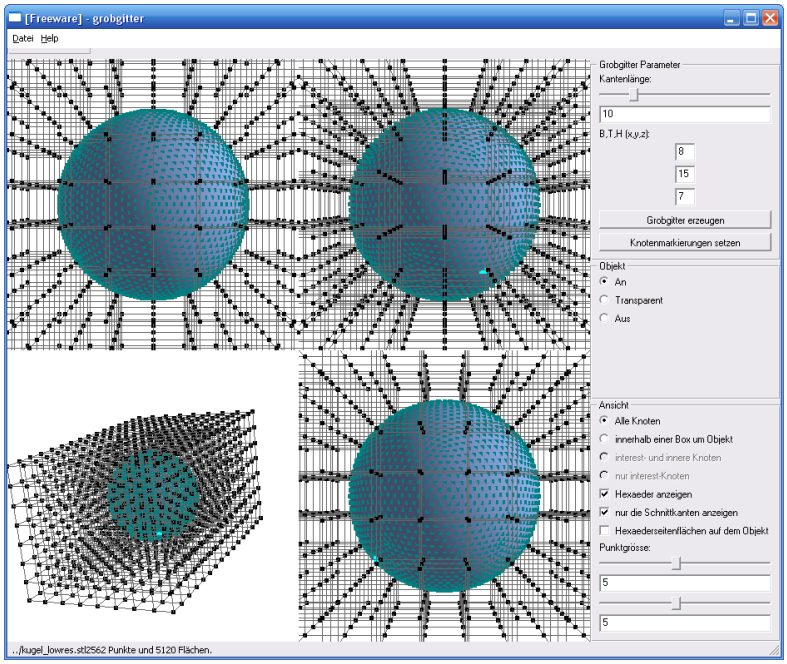


Abbildung A.7: *Die Benutzeroberfläche von* HaGrid3D *unmittelbar nach dem Start.*

Ist das neue Objekt eingelesen worden, so ist es zunächst sinnvoll, die Parameter des Grobgitters an das Objekt anzupassen. Hierzu wird zunächst die Kantenlänge der Hexaederzellen über den gleichnamigen Schieberegler angepasst und dann die Anzahl der Grobgitterknoten in alle Dimensionen bestimmt (s. Abb. A.8). Im nächsten Schritt kann das Objekt im Grobgitter an eine geeignete Position verschoben werden (s. Abb. A.9). Unter Verwendung der [T]-Taste wird das Grobgitter bei gedrückter linker Maustaste in den 2D-Ansichten in die gewünschte Richtung gezogen.

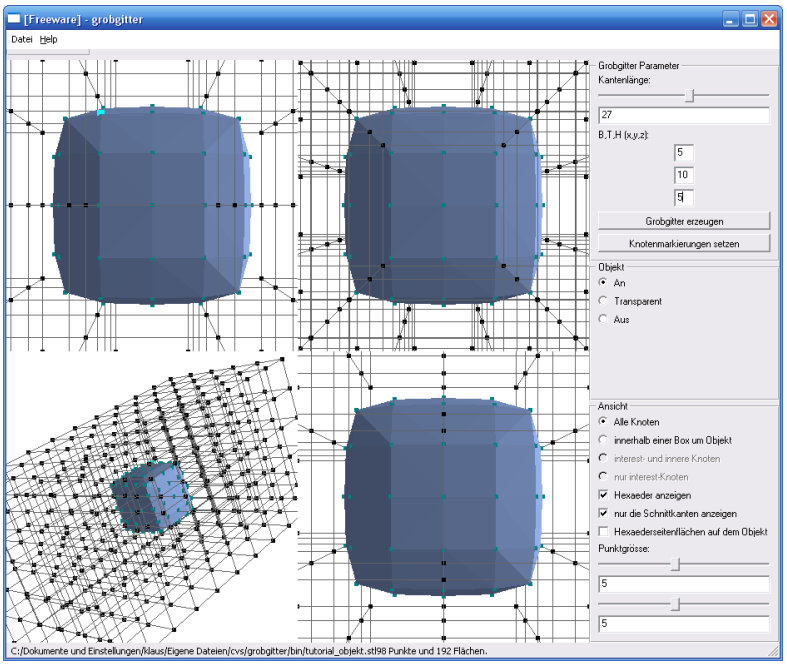


Abbildung A.8: *Anpassen der Kantenlänge der Hexaederzellen an das Objekt und Festlegen der Anzahl an Hexaederzellen in den drei Dimensionen x, y, z.*

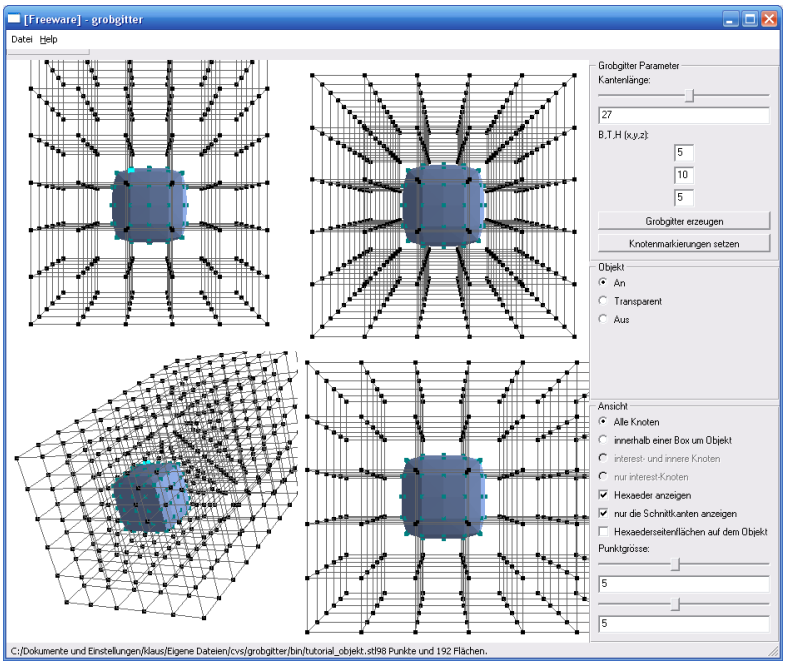


Abbildung A.9: *Positionieren des Objektes im Grobgitter.*

Der nächste Schritt besteht in der automatischen Markierung der Grobgitterknoten, die über die „Knotenmarkierung setzen”-Schaltfläche eingeleitet wird. Ist die Berechnung abgeschlossen stehen von nun an neue Ansichtsoptionen zur Verfügung, die durch die neuen Knotenmarkierungen ermöglicht werden. Die Schaltfläche wird deaktiviert. In Abbildung A.10 ist die Objektoberfläche transparent und somit sieht man die rot eingefärbten inneren Knoten.

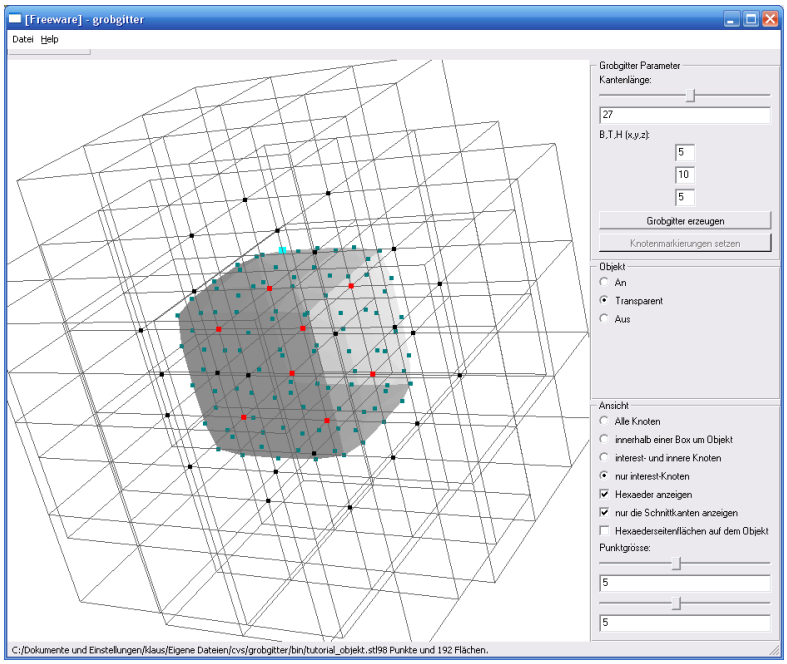


Abbildung A.10: *Nach dem Inklusionstest sind alle inneren Knoten markiert (rot eingefärbt) und es stehen neue Ansichtsoptionen für die interessanten Knoten zur Verfügung.*

Nun kann mit der Manipulation einzelner Grobgitterknoten begonnen werden. Zunächst sollte man in der Optionenleiste die Optionen so auswählen, dass nur die Grobgitterknoten in der Nähe des Objektes angezeigt werden. Sinnvollerweise sollten hier die Hexaederzellen zunächst ganz ausgeblendet werden. Unter den Knoten sollte man sich hier zunächst nur die interessanten Knoten anzeigen lassen. Hat man sich einen Überblick verschafft, so beginnt man nun mit dem Verschieben der Grobgitterknoten auf das Objekt. Die Knoten werden mit dem Mauszeiger selektiert und dann auf Objektknoten verschoben. Um einen Grobgitterknoten zu selektieren, muss zusätzlich zur linken Maustaste noch die Steuerungstaste gedrückt gehalten werden. Möchte man einen Objektknoten selektieren, so muss die Steuerungstaste zusammen mit der [Y]-Taste gedrückt gehalten werden. Abbildung A.11 zeigt ein solches Knotenpaar vor dem Verschieben. Mit der Leertaste kann nun der selektierte Grobgitterknoten auf den Objektknoten verschoben werden. Selektierte Knoten werden etwas dicker als die übrigen dargestellt. Ist ein Grobgitterknoten auf einen Geometrie-Eckpunkt gezogen worden, so wird er fortan gelb dargestellt, wie in Abbildung A.12 zu sehen ist.

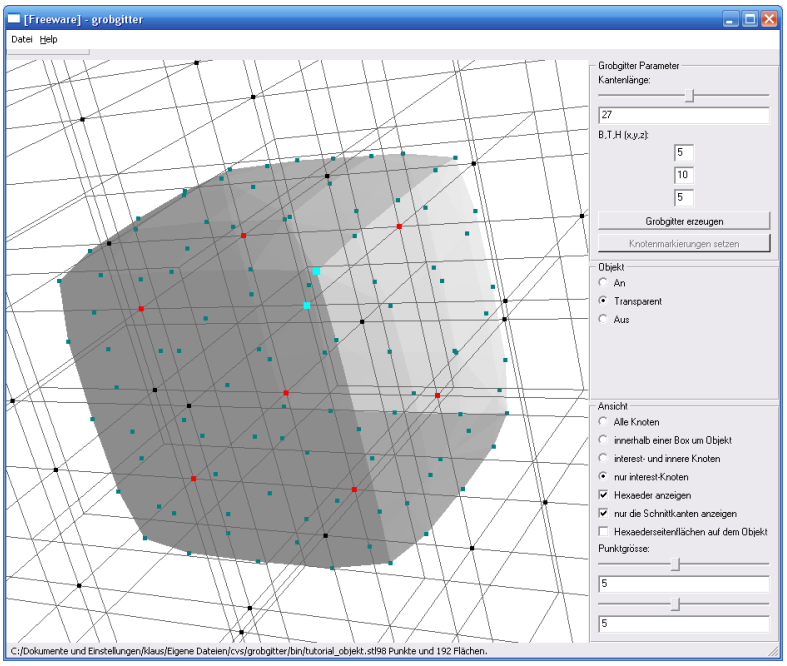


Abbildung A.11: *Ein selektiertes Knotenpaar kann nun mit der Leertaste auf dem Objektrand vereint werden.*

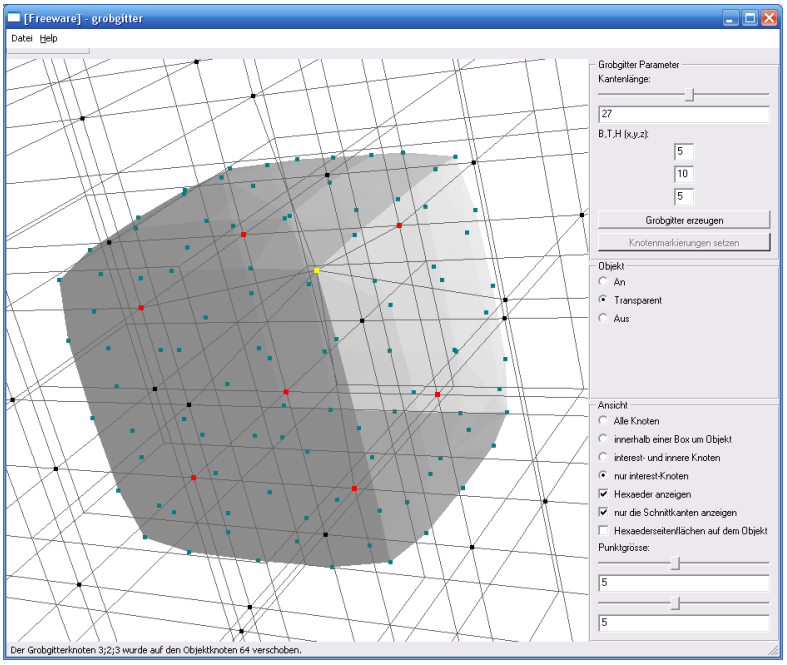


Abbildung A.12: *Nach dem Verschieben des Grobgitterknotens wird dieser fortan gelb dargestellt.*

Auf diese Weise lässt sich gut überblicken, welche Knoten sich schon auf dem Objekt befinden. Sind alle Knoten einer Hexaederseitenfläche auf den Objektrand verschoben worden, so wird die Fläche rot und die inzidenten Kanten der beteiligten Knoten werden gelb eingefärbt. Dies hat den Vorteil, dass zum Einen die Wasserdichtigkeit der Approximation kontrolliert werden kann, und zum Anderen fällt es leichter, mögliche Kandidaten zum Verschieben zu finden. Bilden nun die Hexaederflächen einen vollständigen und wasserdichten Körper, so kann das Grobgitter exportiert und in *InGrid3D* geladen werden. Abbildung A.13 zeigt ein vollständiges Grobgitter für das Testobjekt.

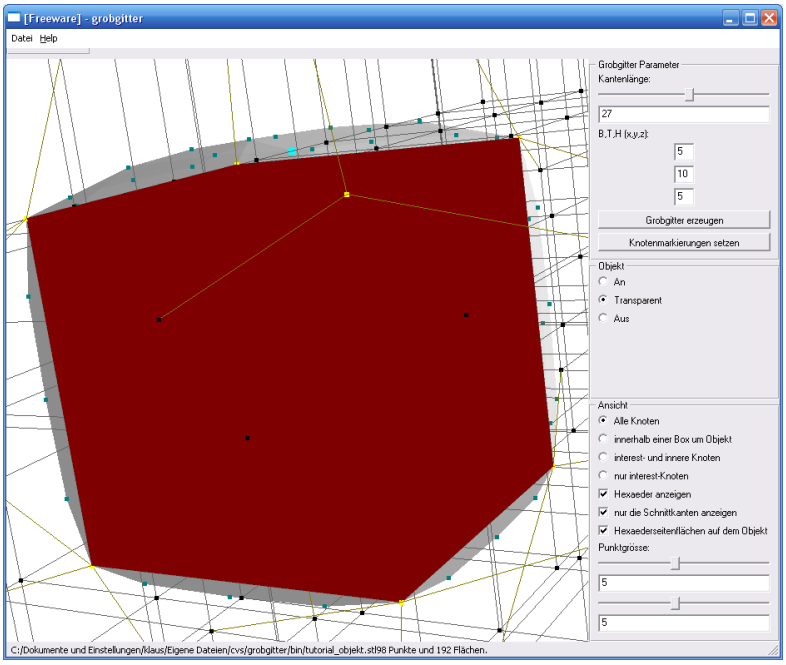


Abbildung A.13: *Ein fertiges Grobgitter für das Testobjekt mit eingefärbten Randflächen der Hexaeder.*

A.2.3 Tutorial: Gitterunterteilung mit InGrid3D

Nach dem Programmstart (s. Abb. A.14) kann zunächst ein Objekt im STL-Format geladen werden. In diesem Tutorial wird die Oberflächentriangulierung aus dem Kapitel A.2.1 geladen. Hierzu wird über das Menü „Mesh" die „Mesh Laden"-Funktion aufgerufen und dann die entsprechende Datei ausgewählt (auch über die Tastenkombination [Strg]+[M] wählbar).

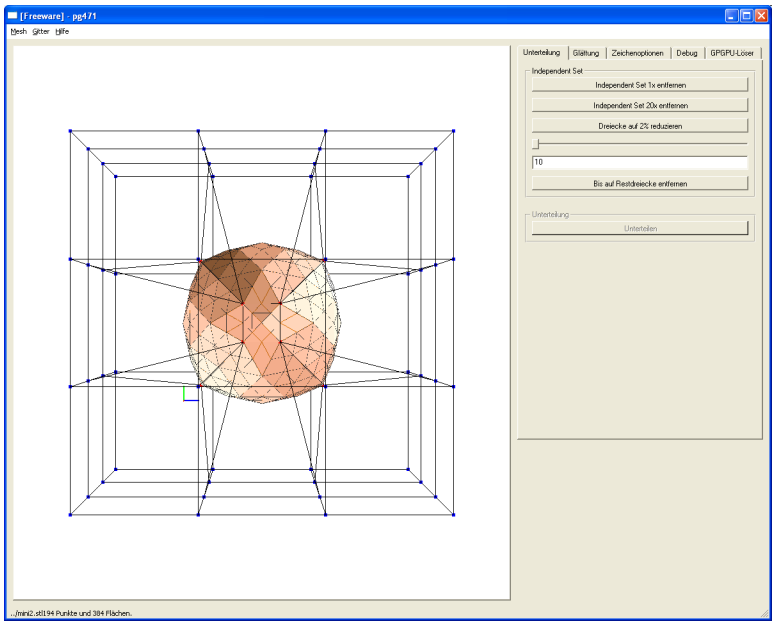


Abbildung A.14: *Die Benutzeroberfläche von* InGrid3D *unmittelbar nach dem Programmstart.*

Ist das neue Objekt eingelesen worden, ist es notwendig, das dazu konstruierte Grobgitter zu laden, das mit Hilfe des Programms *HaGrid3D* erstellt wurde (s. Anhang A.2.2). Hierzu wird über das Menü „Gitter" die „Laden"-Funktion aufgerufen und dann die entsprechende Datei ausgewählt (auch über die Tastenkombination [Strg]+[L] wählbar). Das neue Objekt und das dazu konstruierte Grobgitter sind nun geladen (s. Abb. A.15).

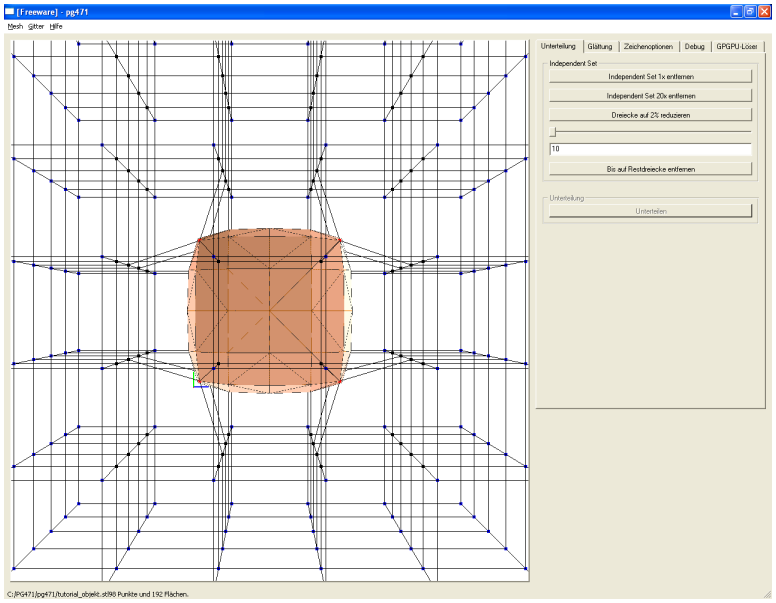


Abbildung A.15: *Das neue Objekt und das dazu konstruierte Grobgitter.*

Im nächsten Schritt muss die Oberflächentriangulierung reduziert werden, indem die „Independent Set 1x entfernen”-, „Independent Set 20x entfernen”-, „Dreiecke auf 2% reduzieren”- oder „Bis auf Restdreiecke entfernen”-Schaltfläche betätigt wird. Im letzten Fall wird zunächst die Anzahl der Restdreiecke über den dazugehörigen Schieberegler angepasst (s. Abb. A.16).

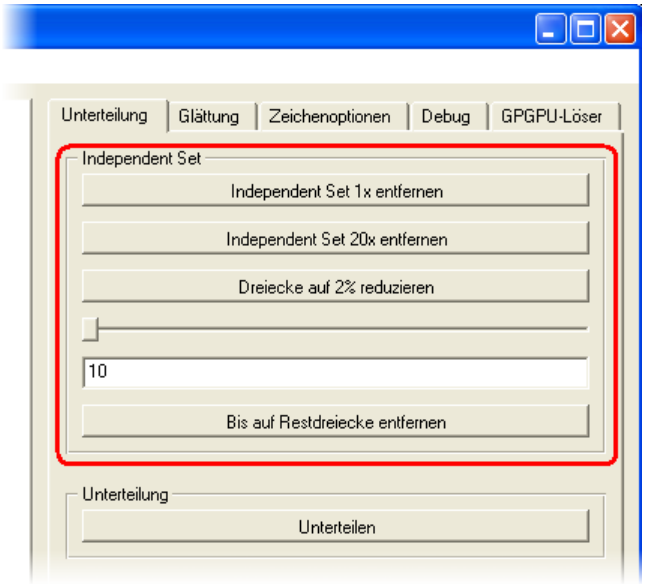


Abbildung A.16: *Die Schaltflächen zur Reduzierung der Oberflächentriangulierung.*

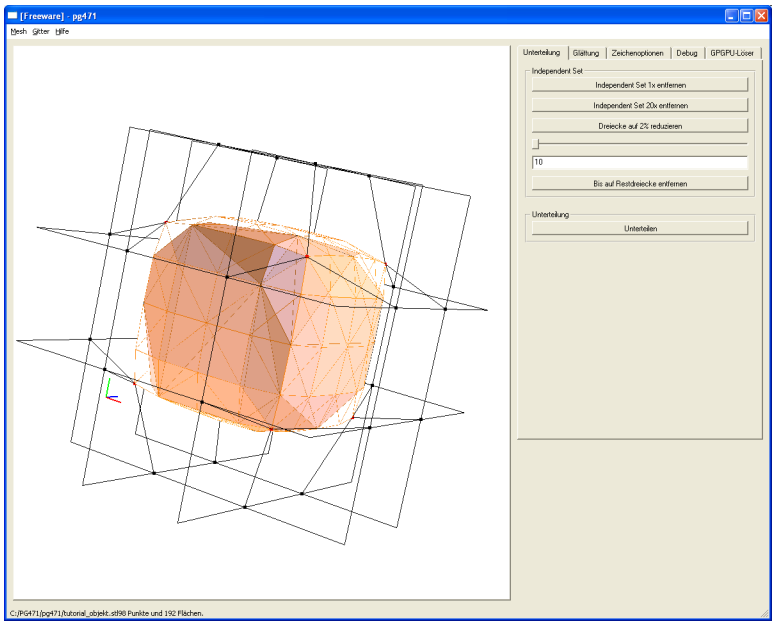


Abbildung A.17: *Die reduzierte Oberflächentriangulierung.*

Ist die Reduzierung abgeschlossen, kann der erste Unterteilungsschritt begonnen werden. In Abbildung A.17 kann man das Ergebniss der Reduzierung sehen (die „Independent Set 1x entfernen”-Schaltfläche wurde vier Mal betätigt). Die Objektoberfläche des Originals ist transparent dargestellt, sodass der Unterschied zwischen der Originalobjektoberfläche und der reduzierten Objektoberfläche gesehen werden kann.

Bevor man mit der Gitterunterteilung beginnt, sollte man zuerst die Zeichenoptionen in der entsprechenden Leiste so auswählen, dass nur die Gitterzellen am Objektrand angezeigt werden (s. Abb. A.18).

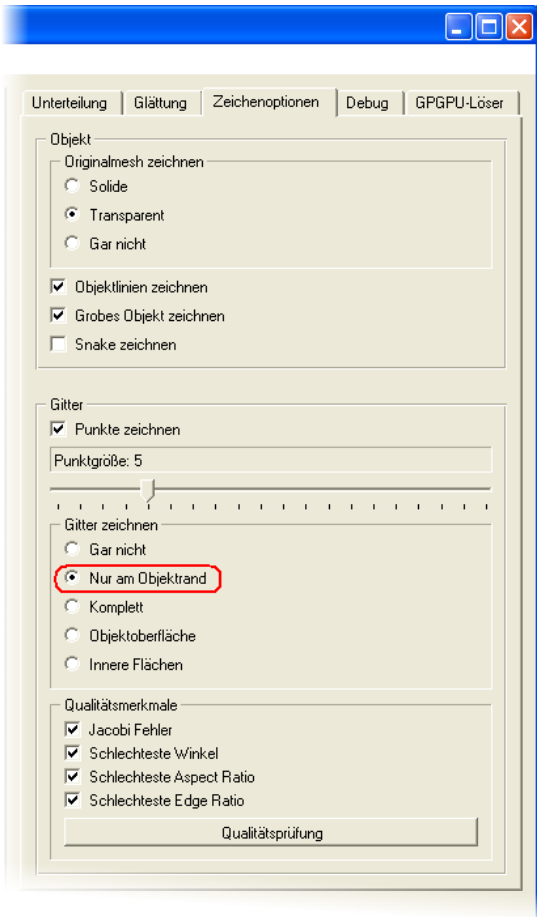


Abbildung A.18: *Die Zeichenoptionen-Leiste.*

Hat man sich einen Überblick verschafft, wird mit der Gitterunterteilung begonnen, indem man die „Unterteilen”-Schaltfläche betätigt. Die Unterteilung kann auch über die Tastenkombination [Strg]+[D] gestartet werden.

Nach jedem Gitterunterteilungsschritt oder nach dem Betätigen der „Qualitätsprüfung”-Schaltfläche werden Gitterqualitätsmerkmale berechnet und in der Statusleiste angezeigt (s. Abb. A.19).

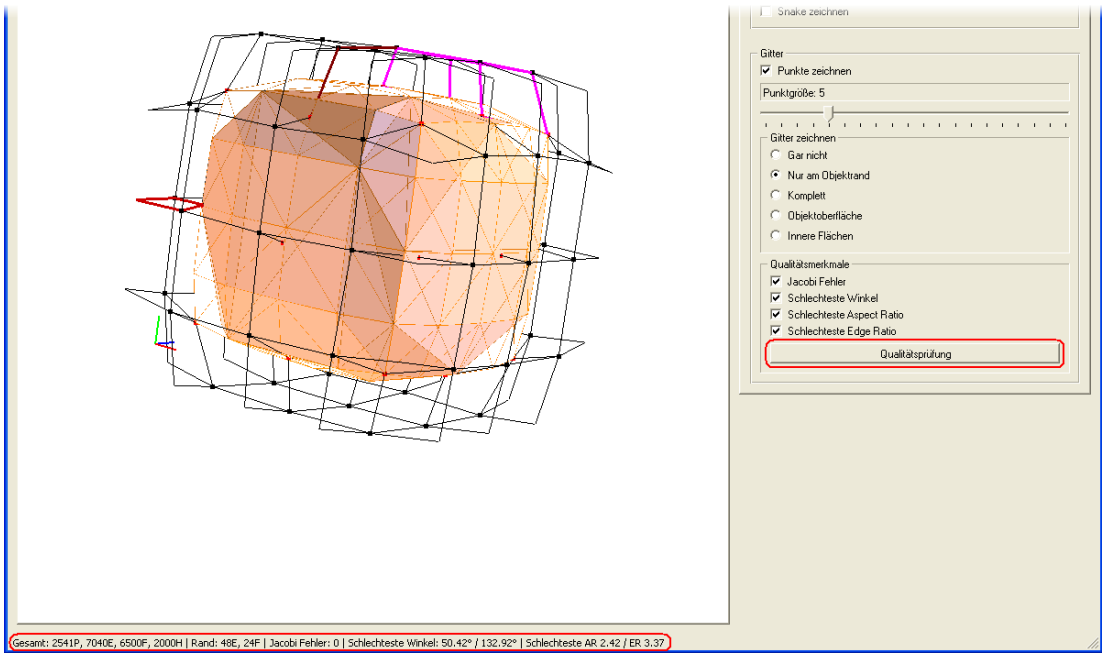


Abbildung A.19: *Die Qualitätsmerkmale des Gitters werden nach Betätigen der „Qualitätsprüfung”-Schaltfläche in der Statuszeile angezeigt.*

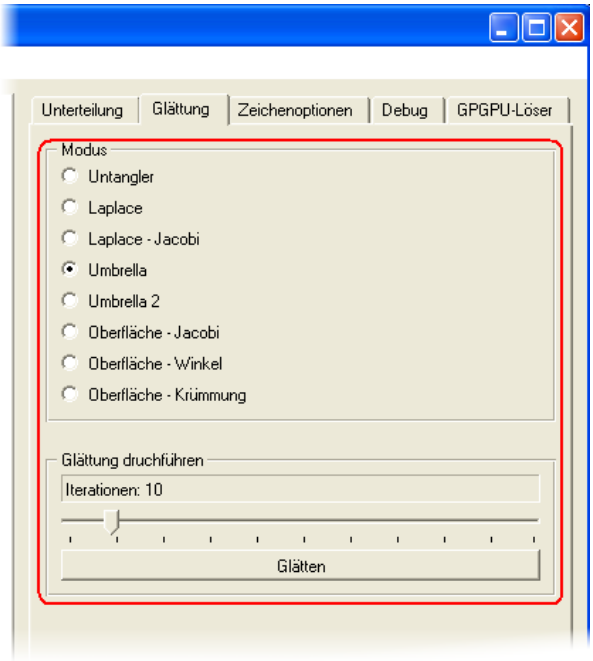


Abbildung A.20: *Die „Glättung”-Leiste.*

Vor oder nach jedem Unterteilungsschritt kann das Gitter geglättet und repariert werden, indem man zuerst die gewünschte Option auf der „Glättung”-Leiste auswählt, die Iterationenanzahl mittels des Schiebereglers anpasst und anschließend die „Glätten”-Schaltfläche betätigt (s. Abb. A.20). Diese Funktion ist auch über die Tastenkombination [Strg]+[G] wählbar.
Zu jedem Zeitpunkt kann das erstellte Gitter gespeichert werden. Hierzu wird über das Menü „Gitter” die „Speichern unter”-Funktion aufgerufen und dann die Ausgabedatei ausgewählt (auch über die Tastenkombination [Strg]+[S] wählbar).

A.2.4 Tutorial: Löser und Visualisierung

Hat man nach einigen Unterteilungsschritten die gewünschte Auflösung des Gitters erreicht, so kann man die Daten für den GPU Löser in eine Datei exportieren. Hierzu wird über das Menü „Gitter” die „Export zum GPU Löser”-Funktion aufgerufen und ein entsprechender Dateiname angegeben (auch über die Tastenkombination [Strg]+[E] wählbar).
Nach erfolgter Berechnung können die von dem GPU Löser gelieferten Daten wieder in *InGrid3D* über das Menü „Gitter” und dann die „Importiere Löserdaten”-Funktion zur Visualisierung importiert werden (auch über die Tastenkombination [Strg]+[I] wählbar).
Der Import der Löserdaten schaltet alle Zeichenoptionen in der GPGPU-Leiste frei (s. Abb. A.21).

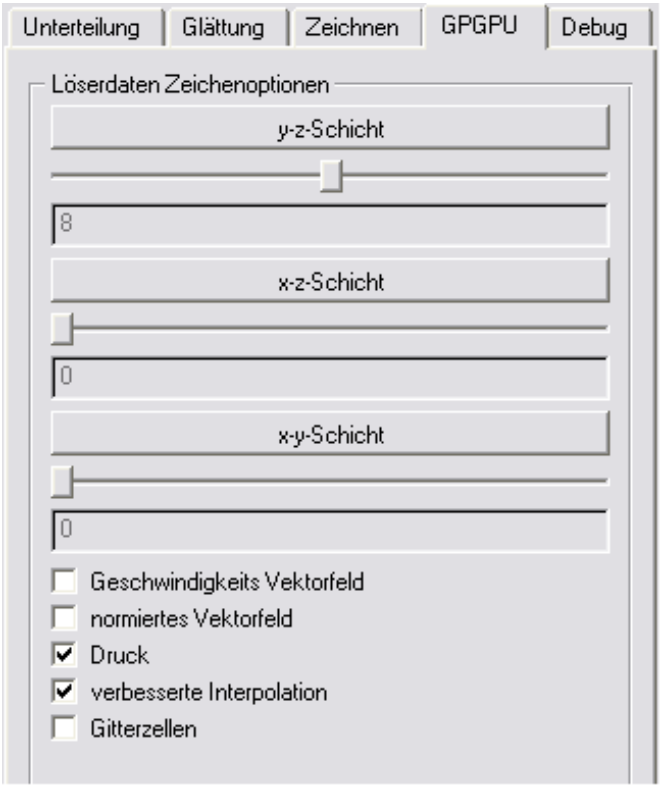


Abbildung A.21: *Das Bedienfeld für die Zeichenoptionen der in der Grafikkarte berechneten Daten.*

Durch das Betätigen der „Schicht”-Schaltflächen oder das Verschieben der jeweils darunter liegenden Schieberegler, ist es möglich durch einzelne Gitterschichten zu wechseln. Zur Visualisierung der Löser-Daten stehen verschiedene Zeichenoptionen bereit. Es ist möglich über die Schaltflächen die Visualisierung des Geschwindigkeitsvektorfeldes einzuschalten. Wird es eingeschaltet, gibt es noch zusätzlich die Option die Vektoren zu normieren, was gelegentlich für die Übersichtlichkeit von Vorteil ist (s. Abb. A.22). Auch das von der GPU berechnete Druckfeld lässt sich einschalten und in verschiedenen Schichten anzeigen. Für gröbere Gitter empfiehlt es sich die „verbesserte Interpolation” hinzu zu schalten. Hierdurch werden Artefakte in der Darstellung des Druckfelds ausgeglichen (s. Abb. A.23). Als letzte Option ist es möglich eine Schicht aus dem Gitter darstellen zu lassen.

Alle drei Datenbereiche können so einzeln oder in beliebiger Kombination miteinander dargestellt werden (s. Abb. A.24).

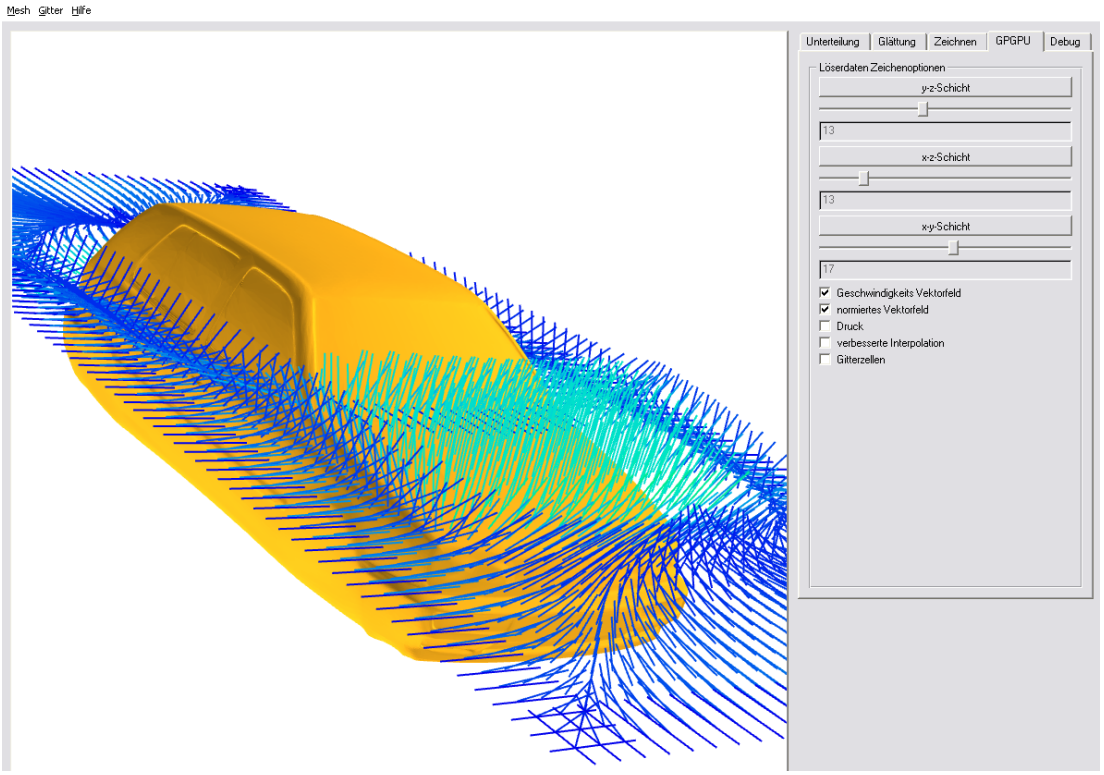


Abbildung A.22: *Das normierte Geschwindigkeitsvektorfeld.*

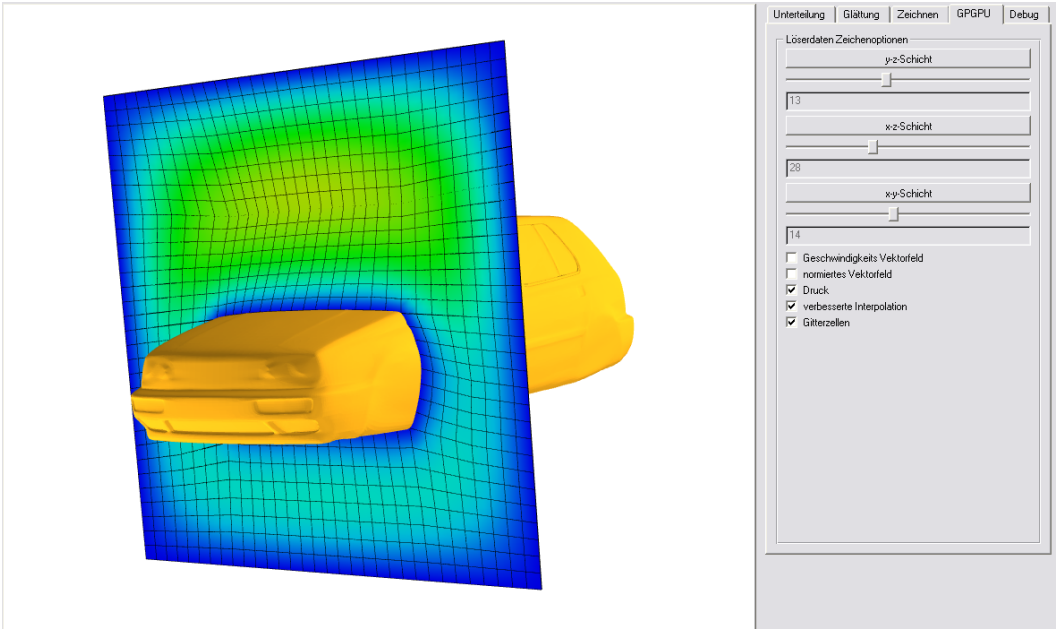


Abbildung A.23: *Das verbessert-interpolierte Druckfeld inklusive Gitterschicht.*

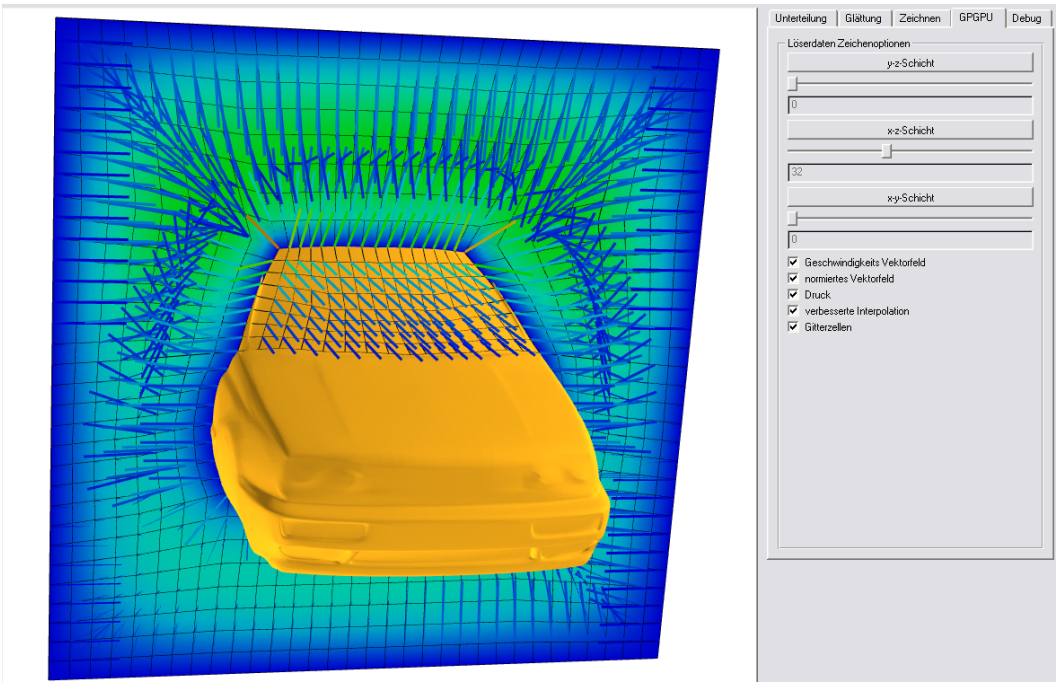


Abbildung A.24: *Alle Zeichenoptionen gleichzeitig eingeschaltet.*

A.3 Dateiformate

A.3.1 STL-Format

Innerhalb von *InGrid3D* und *HaGrid3D* werden die verwendeten Oberflächentriangulierungen in Form von STL-Dateien eingelesen.
Das STL-Dateiformat dient der Abspeicherung von dreidimensionalen Objekten. Die Objekte werden in Form von Punkten und Flächen in einer Datei gespeichert. Es ist ein weit verbreitetes Dateiformat, welches von den meisten 3D-Anwendungen unterstützt wird. Die Punkte werden durch ihre Koordinaten und Flächen durch ihre Punkte beschrieben. Es ist möglich, die Dateien entweder im ASCII-Format oder in binärer Form zu speichern. Wird ein Objekt im ASCII-Format gespeichert, so hat dies den Vorteil, dass die Datei mit einem Texteditor durch den Anwender lesbar ist. Somit hat er die Möglichkeit Debug-Informationen direkt auszulesen oder manuell einzelne Parameter des Objektes zu verändern[2].

A.3.2 CM-Format

Das CM-Format (*coarse mesh*) ist ein eigens entworfenes Dateiformat zur Speicherung von Grobgittern in *HaGrid3D*. Die Dateien werden im ASCII-Format geschrieben und können so ebenfalls mit einem Texteditor gelesen und bearbeitet werden. Die erste Zeile der Datei besteht aus vier Zahlenwerten, die die Anzahl der Hexaederzellen in X-, Y- und Z-Richtung, gefolgt von der Kantenlänge, beschreiben. In den darauf folgenden Zeilen werden die Knoten mit ihren Koordinaten und Eigenschaften beschrieben. Pro Zeile wird ein Knoten gespeichert. Hier werden die Koordinaten des Knotens in X-, Y- und Z-Richtung, der *solid*- und der *interest*-Status und der Referenzknoten des Objektes hintereinander geschrieben. Ein Knoten bekommt den *interest*-Status 1, wenn er einen Nachbarn innerhalb des Objektes hat, selber aber nicht innerhalb des Objektes oder auf dem Rand des Grobgitters liegt. Sonst ist der *interest*-Status 0. Es folgen noch die Koordinatenwerte in X-, Y- und Z-Richtung und der *solid*-Status des Knotens zum Zeitpunkt seiner Erstellung. Dieser beschreibt, ob ein Knoten am Rand des Grobgitters liegt (*solid*=3), sich auf der Objektoberfläche befindet (*solid*=2), oder sich innerhalb des Objektes befindet (*solid*=1). Sonst ist der *solid*-Status 0.

Beispiel Die erste Zeile der Datei

```
8 15 7 0.4
```

beschreibt die Größe des gesamten Gitters mit acht Hexaederzellen in der X-, 15 in Y- und sieben in der Z-Richtung. Die einheitliche Kantenlänge der Hexaederzellen beträgt 0.4.

Die folgenden drei Zeilen

```
-1.6 -3 -1.4 3 0 -1 -1.6 -3 -1.4 3
-1.6 -3 -1.0 3 0 -1 -1.6 -3 -1.0 3
-1.6 -3 -0.6 3 0 -1 -1.6 -3 -0.6 3
```

[2] Für detaillierte Informationen sei auf http://rpdrc.ic.polyu.edu.hk/old_files/stl_introduction.htm verwiesen.

beschreiben die ersten drei Knoten in Z-Richtung. In der ersten Zeile ist der Knoten mit den Indizes 0,0,0 (x,y,z). In den ersten drei Spalten findet man die Koordinaten $(-1.6, -3, -1.4)$ des Knotens, die 3 in der dritten Spalte beschreibt den Status des Knotens und sagt aus, dass es sich hier um einen Randknoten handelt. Der fünfte Wert speichert den *interest*-Status und der sechste Wert gibt den Index des Objektknotens wieder, falls der Knoten auf das Objekt verschoben wurden, ansonsten steht dort wie hier im Beispiel der Wert -1. In den folgenden zwei Zeilen sind die beiden Nachbarn des Knotens aus der ersten Zeile in Z-Richtung. Wie man erkennt, sind die Knoten bis auf die Werte in der dritten Spalte identisch. Dort unterscheiden sie sich um die oben beschriebene Kantenlänge 0.4. Die *default*-Werte unterscheiden sich nicht von den Koordinaten aus den ersten drei Spalten, da die Knoten nicht verschoben wurden.
In den nächsten Zeilen sind vier Knoten beschrieben, die auf das Objekt gezogen worden sind.

```
0.793760 -0.184749 -0.569158 2 1 160  1.2 -0.2 -0.6 0
0.952176 -0.193007 -0.202686 2 1 684  1.2 -0.2 -0.2 0
0.958134 -0.189769  0.187040 2 1 788  1.2 -0.2  0.2 0
0.764738 -0.193005  0.606224 2 1 2082 1.2 -0.2  0.6 0
```

Man erkennt dies leicht an den Werten der Koordinaten, da sie mehr als eine Nachkommastelle haben. Sie haben den *solid*-Status 2 und die jeweiligen Objektknoten haben die Indizes 160, 684, 788 und 2082.

A.3.3 GRID-Format

Das GRID-Format dient zum Speichern und Einlesen von Gittern in *InGrid3D*. Ein Grobgitter in Form einer GRID-Datei kann aus *HaGrid3D* exportiert werden. Im Unterschied zum CM-Format wird das Format binärkodiert gespeichert.

Die folgenden Werte werden in dieser Reihenfolge in die Datei geschrieben: Die Variable `nPoints`, gibt die Anzahl der Gitterknoten an. Es folgen alle Grobgitterknoten in Form von `structs` des Typs `tPointSave`. Es beinhaltet die X-, Y-, und Z-Koordinaten, den *solid*-Status des jeweiligen Knotens und das Attribut `mapsindex`. Dieses dient der Kommunikation zwischen der Parametrisierung und dem Gitter. Als nächstes wird die Breite, Höhe und Tiefe des Gitters gespeichert. Anschließend wird der Wert `mapsIndexCounter` gesichert. Er entspricht der Anzahl der Gitterknoten, die auf dem Objektrand liegen. Es folgen `structs` vom Typ `tParaPoint`, die diese Knoten beschreiben. Sie beinhalten den Index des Punktes auf dessen Position sie sich befinden, das Grob- und Feindreieck in dem sie liegen und die baryzentrischen Koordinaten sowie die Indizes der entsprechenden Eckpunkte des Feindreiecks in dem sie liegen.

A.3.4 GPUGRID-Format

Die Gitterdaten für den GPU-Löser werden im GPUGRID-Format gespeichert. Sie können aus *InGrid3D* exportiert werden, nachdem ein Gitter erstellt wurde. Das GPUGRID-Format wird ebenfalls binärkodiert gespeichert.
Als erstes wird die Variable `nPoints`, die die Anzahl der Gitterknoten angibt, gespeichert. Im Anschluss daran werden alle Distanzinformationen für jeden Knoten als `struct` vom Typ `dist_inf` gespeichert. Darin befinden sich die Abstände zu den jeweiligen rechten, linken, oberen, unteren, vorderen und hinteren Nachbarknoten und der Randstatus des Knotens. Es

folgen die Informationen für die Breite, die Höhe und die Tiefe des Gitters. Als nächstes wird für jeden Punkt die X-, Y- und Z-Koordinate gespeichert. Am Ende der Datei werden erst die maximalen, dann die minimalen X-, Y- und Z-Koordinaten der Knoten gespeichert.

A.3.5 GPU-Format

Dieses Format wird vom GPU-Löser nach der Berechnung erzeugt und kann in *InGrid3D* importiert und dargestellt werden. Es wird ebenfalls im Binärformat gesichert.
Am Anfang der Datei stehen die Informationen für die Breite, die Höhe und die Tiefe des Gitters. Danach folgen die Koordinaten der Punkte. Als nächstes wird das Geschwindigkeitsvektorfeld gesichert in dem zuerst die X-Werte, dann die Y-Werte und danach die Z-Werte geschrieben werden. Im Anschluss daran wird der berechnete Druckwert abgespeichert.

A.3.6 GMV-Format

Das GMV-Format wurde vom GPU-Löser verwendet, um bereits in der Implementierungsphase über eine Visualisierungsmöglichkeit zu verfügen. Die Spezifikation des Dateiformats kann auf der Homepage des Entwicklers eingesehen werden[3].

A.4 Bekannte Fehler

- **Berechnung von Punkten auf der Objektoberfläche scheitert**
  Je nach Grad der Vergröberung kann es bei der Gitterunterteilung vorkommen, dass die Berechnung eines neuen Punktes auf der Objektoberfläche scheitert. Die neu entstehenden Punkte werden in diesem Fall in den Koordinatenursprung gezogen, um diesen Fehler deutlich zu kennzeichnen.
  Dieser Fehler kann umgangen werden, wenn für die Parametrisierung des Geometrieobjekts der Vergröberungsgrad variiert wird.

[3] http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html

Anhang B

Pflichtenheft und Endprodukt

B.1 Pflichtenheft

B.1.1 Zielbestimmung

Ziel ist der Entwurf und die Entwicklung einer Softwareumgebung zur numerischen Strömungssimulation unter Ausnutzung der Leistungsfähigkeit moderner Grafikkarten und entsprechender 3D-Gittermanipulationstechniken. Das System soll objektorientiert und plattformunabhängig entwickelt werden. Beim Entwurf ist ferner auf die Erweiterbarkeit mit weiteren Deformationsansätzen bzw. weiteren numerischen Verfahren zu achten. Die Implementierung soll in der Programmiersprache C++ unter Nutzung von QT, OpenGL und Cg erfolgen.

B.1.2 Produkteinsatz und Produktvoraussetzungen

Hardware

Das Produkt soll auf handelsüblichen Rechnern (32 und 64 Bit) zum Einsatz kommen. Da Berechnungen auf der GPU ausgeführt werden sollen, muss eine Grafikkarte mit programmierbaren *Shadern* vorhanden sein. Zu erwarten ist ein relativ großer Bedarf an Arbeitsspeicher, bedingt durch das Wachstum der Datenmenge bei jedem Unterteilungsschritt.

Software

Als Zielplattform kommt primär Windows 2000/XP zum Einsatz, es soll aber auch die Verwendung anderer Betriebssysteme unterstützt werden.

Interoperabilität

Das Grobgitter zu einem gegebenen 3D-Objekt soll durch eine geeignete externe Software zur Verfügung gestellt werden. Nach durchgeführter Unterteilung und Deformation des Gitters sollen die Ergebnisse (d.h. die erstellte Gitterhierarchie) an einen externen Mehrgitterlöser wie z.B. FEATFLOW [11] übergeben werden können.

B.1.3 Produktfunktionen

Das Endprodukt soll eine Simulationsumgebung mit einer grafischen Benutzerschnittstelle bieten. Es sollen 3D-Objekte und entsprechende Grobgitter geladen werden können. Auch ein Sichern und Wiederherstellen der Unterteilungsschritte soll implementiert werden. Um eine Beurteilung der dreidimensionalen Gitter und Objekte zu gewährleisten, muss die Software über eine frei dreh- und skalierbare Objektansicht verfügen. Eine Auswahl an Glättungs- und Optimierungsalgorithmen soll zur Verfügung gestellt werden. Die visuelle Darstellung der Simulationsergebnisse kann durch ein externes (z.B. Gnuplot [64]) oder selbst implementiertes Programm erfolgen.

B.2 Endprodukt

B.2.1 Erreichte Ziele

Die geplante Ausnutzung der Leistungsfähigkeit moderner Grafikkarten konnte nur im Löser realisiert werden. Die zur Gittermanipulation eingesetzten Algorithmen und Methoden erwiesen sich entweder als nicht auf die GPU übertragbar oder die Probleme waren nicht ausreichend groß bzw. ausreichend parallelisierbar, als dass sich eine Berechnung auf der GPU rentieren würde.
Da der Löser auf GPU-Basis als Konzeptnachweis (engl. *proof of concept*) dienen soll, werden nur reguläre, kartesische Gitter unterstützt, die Gitterunterteilungssoftware *InGrid3D* ist jedoch nicht auf diesen Gittertyp beschränkt.
Dank der objektorientierten Entwicklung des Systems ist eine Erweiterung um zusätzliche Glätter und numerische Lösungsverfahren leicht möglich. Die Implementierung erfolgte in der Programmiersprache C++ unter Nutzung von QT, OpenGL, Cg und weiteren Bibliotheken und Werkzeugen (s. Kap. 3).

B.2.2 Produkteinsatz und Produktvoraussetzungen

Hardware

Die Anforderungen aus dem Pflichtenheft bezüglich der zu verwendenden Hardware wurden erfüllt. So ist sowohl der Gittergenerator *HaGrid3D* als auch die Gitterunterteilungssoftware *InGrid3D* auf handelsüblichen Rechnern einsetzbar, es werden aber aktuelle Systeme empfohlen.
Für den separaten Löser ist eine Grafikkarte der neueren Generation nötig. Es werden alle NVIDIA-Modelle ab der GeForce 6-Reihe unterstützt. Karten des Herstellers ATI werden zur Zeit noch nicht unterstützt.

Wie erwartet wird bei der Gitterunterteilung sehr viel Arbeitsspeicher verbraucht. Es zeigte sich in Versuchen, dass mit einer heute üblichen Arbeitsspeicherausstattung von einem Gigabyte, etwa vier bis sechs Unterteilungsschritte möglich sind, je nach Anzahl der Elemente des Grobgitters.

Software

Die im Rahmen der PG erstellten Programme sind auf Windows 2000/XP lauffähig. Eine Portierung auf das Linux-Betriebssystem wurde aus Zeitgründen nicht vorgenommen, ist aber aufgrund der Verwendung der plattformunabhängigen Bibliotheken Qt (s. Kap. 3.2) und OpenMesh (s. Kap. 3.6) möglich.
Die Verwendung der neuesten Grafikkartentreiber wird empfohlen. Für den Löser ist des Weiteren das Cg-Toolkit in der Version 1.3 oder 1.5 nötig, Version 1.4 ist fehlerbehaftet.

Interoperabilität

Da die Generierung geeigneter Grobgitter durch externe Software nicht gelang (s. Kap. 4.3), implementierte die PG ein eigenes Werkzeug zur Unterstützung bei der manuellen Grobgittererstellung. Nach durchgeführter Unterteilung und Deformation des Gitters kann selbiges an den ebenfalls entwickelten Löser übergeben werden. Eine analoge Exportfunktion zu anderen Lösern ist einfach hinzuzufügen. Die zu umströmenden Objekte müssen als Oberflächentriangulierung vorliegen, konkret unterstützt wird das STL-Format.

B.2.3 Produktfunktionen

Die im Pflichtenheft geforderten Produktfunktionen wurden vollständig umgesetzt. Da nicht alle verwendeten Techniken auf die GPU übertragbar waren, wurde ein GPU-Löser implementiert, um die Korrektheit der generierten Hexaedergitter zu demonstrieren.

Anhang C

API-Dokumentation

C.1 Übersicht über die Pakete und Klassen von InGrid3D

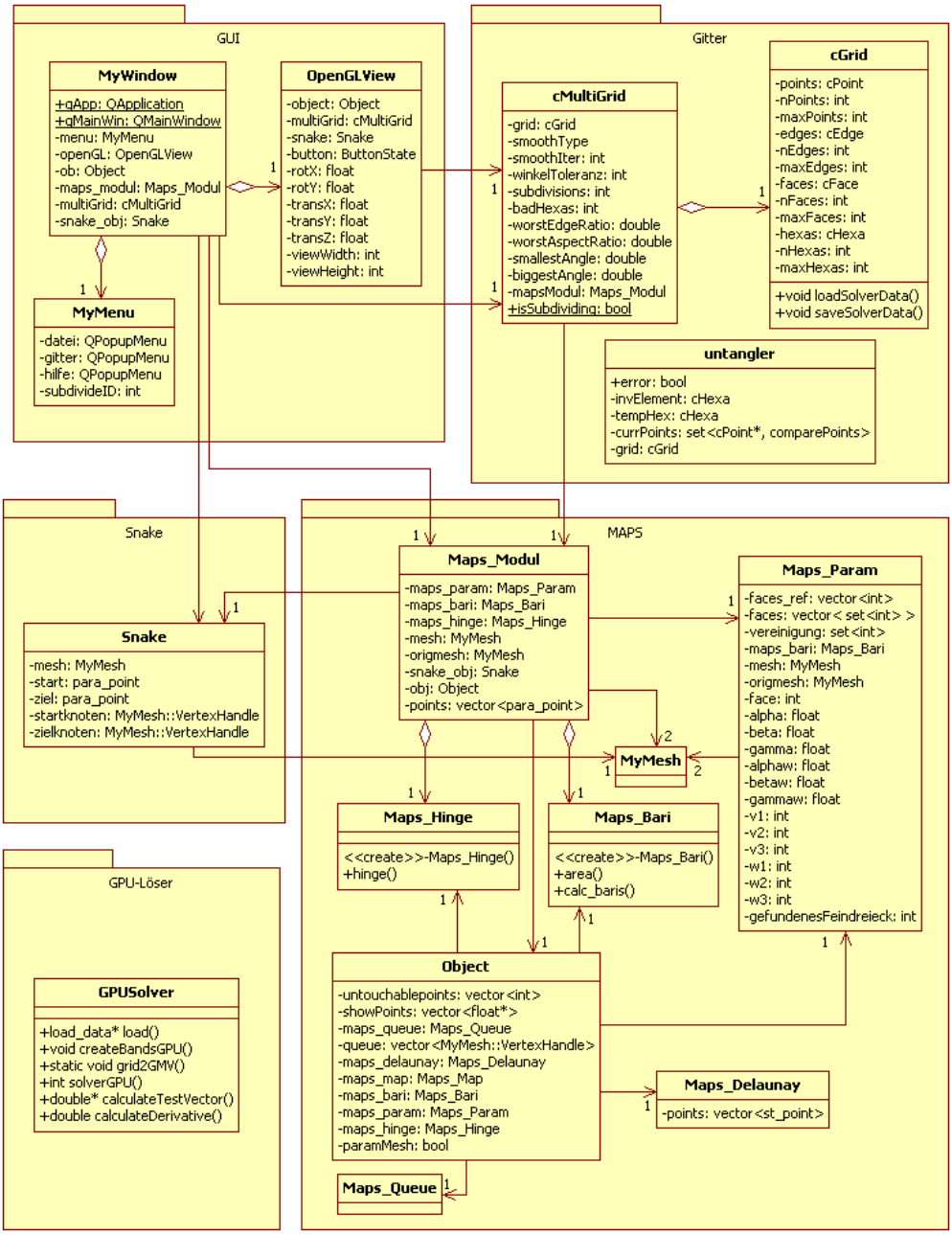


Abbildung C.1: *UML-Klassendiagramm für* InGrid3D.

Die Abbildung C.1 zeigt eine Gesamtübersicht der wichtigsten Klassen in *InGrid3D*. Im Folgenden werden die vier großen Module („Gitter", „Snake", „MAPS", „GPU-Löser") im Detail vorgestellt.

C.1.1 Modul „Gitter”

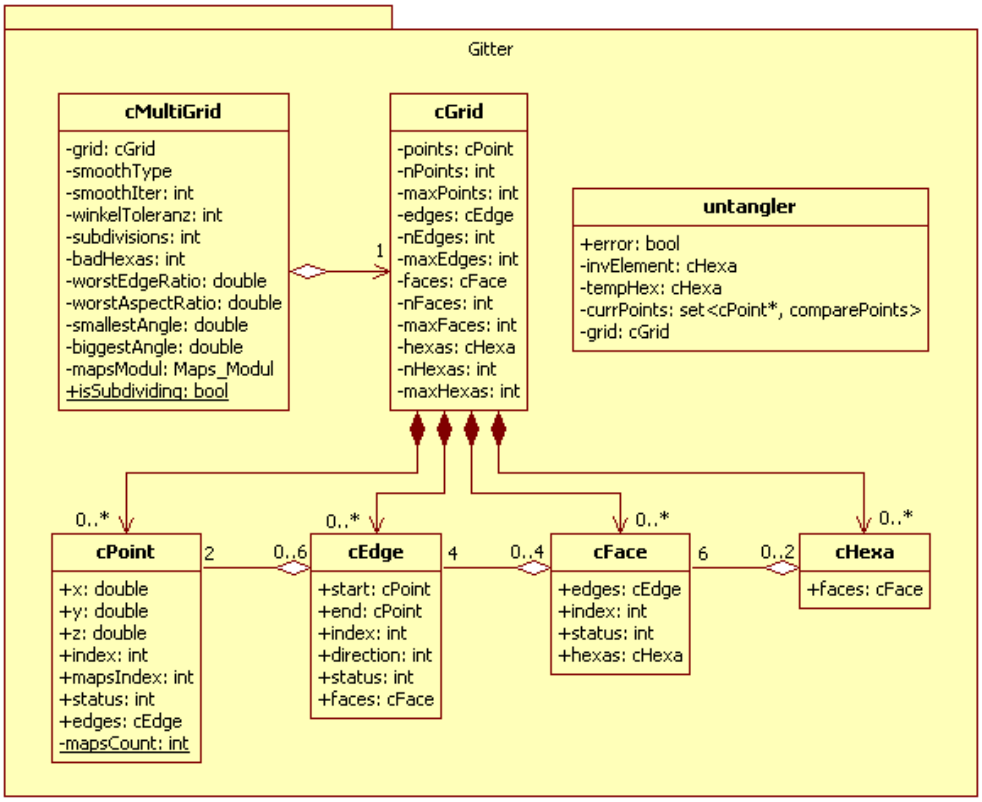


Abbildung C.2: *UML-Klassendiagramm für das Modul „Gitter”.*

Die Abbildung C.2 zeigt eine Übersicht der Klassen im Modul „Gitter”, welche für die Verwaltung der Gitterstruktur zuständig sind.

Klasse cPoint
`cPoint` ist die Entitätenklasse für eine Kante im Gitter

`bool addEdge(cEdge *edge)`

fügt eine von bis zu sechs Kanten in einen Vektor der zum Knoten inzidenten Kanten ein

`void getIncidentHexas(set<cHexa*, compareHexas> *hexaSet)`

liefert alle Hexaeder zurück, die den Knoten als Eckpunkt haben

`int getAdjacentPoints(cPoint**)`

liefert alle Knoten, die mit dem betrachteten Knoten durch Kanten verbunden sind

`bool equals(cPoint *comp)`

testet den Knoten auf Gleichheit mit `comp`, bezogen auf die Koordinaten im Raum

`static void resetMapsCount()`

setzt den Zähler für den MAPS Index zurück

`static void setMapsCount(int count)`

setzt den MAPS Index Zähler auf `count`

`static int getNextMapsIndex()`

liefert den Zählerstand des MAPS Index Zählers und inkrementiert ihn danach

Klasse cEdge
`cEdge` ist die Entitätenklasse für eine Kante im Gitter

`bool addFace(cFace *face)`

fügt eine von bis zu vier Flächen in einen Vektor der zur Kante inzidenten Flächen ein

`cPoint *getOtherPoint(cPoint *point)`

gibt den jeweils anderen Knoten der Kante zurück

`bool checkPoints(int status)`

testet, ob beide Knoten der Kante den im Parameter übergebenen Status haben

`bool hasSolidPoint()`

überprüft, ob einer der beiden Knoten der Kante innerhalb des Objektes liegt

`double calcLength()`

berechnet die Länge einer Kante

`void calcVector(tKoord *koord)`

berechnet den Vektor vom Start- zum Endpunkt der Kante

`static void calcNormal(cEdge *e1, cEdge *e2, tKoord *koord)`

berechnet einen Vektor, der senkrecht auf `e1` und `e2` steht

Klasse cFace
`cFace` ist die Entitätenklasse für eine Fläche einer Gitterzelle

`bool addHexa(cHexa *hexa)`

fügt einen von bis zu zwei Hexaedern in einen Vektor der zur Fläche inzidenten Hexaeder ein

`bool isBorderObj()`

liefert `true`, wenn alle vier Knoten der Fläche den Status „Objektrand" haben.

`bool hasSolidPoint()`

liefert `true`, wenn mindestens einer der vier Knoten der Fläche den Status „Objektinneres" hat

`double calcEdgeRatio()`

berechnet das Verhältnis zwischen der kleinsten und der längsten Kante der Fläche

`double calcFaceSize()`

berechnet die „Größe" einer Fläche, wird benötigt zur Berechnung der Seitenverhältnisse der Hexaeder (s. Kap. 2.3.2)

Klasse cHexa
`cHexa` ist die Entitätenklasse für eine Gitterzelle des Gitters mit erweiterten Abfrageroutinen

`bool isSolid()`

liefert `true`, falls der Hexaeder im Objektinneren liegt.

`double calculateJacobi()`

berechnet das Jacobi-Verhältnis der Gitterzelle (s. Kap. 2.3.2)

Klasse cGrid
`cGrid` verwaltet die einzelnen Elemente der Klassen `cPoint`, `cEgde`, `cFace` und `cHexa` und fügt sie so zum Gitter zusammen

`cGrid(int maxPoints, int maxEdges, int maxFaces, int maxHexas)`

Konstruktor, der ein `cGrid` anlegt, welches die in den Parametern angegebene Anzahl an Elementen aufnehmen kann

```
cPoint* addPoint(double x, double y, double z, int status,
int mapsIndex = 0)
```

Fügt einen Knoten mit den 3D Koordinaten `x`, `y` und `z` zum Gitter hinzu. `status` repräsentiert ob der Knoten Randknoten ist, zur Oberfläche gehört oder innerhalb des Objektes liegt. `MapsIndex` wird für jeden Knoten auf der Objektoberfläche benötigt und dient zur Kommunikation mit dem Modul „Maps".

```
cEdge* addEdge(cPoint *start, cPoint *end, int direction =
DIR_UNDEFINED, int status = NO_BORDER)
```

Fügt zwischen den zwei Knoten `start` und `end` eine Kante mit der Richtung `direction` und dem Status `status` in das Gitter ein. Die Richtung wird für ein gerichtetes Durchlaufen des Gitters benötigt.

```
cFace* addFace(cEdge *edge0, cEdge *edge1, cEdge *edge2,
cEdge *edge3, int status = NO_BORDER)
```

fügt eine Fläche mit den Randkanten `edge0` bis `edge3` mit dem Status `status` in das Gitter ein

```
cHexa* addHexa(cFace *face0, cFace *face1, cFace *face2,
cFace *face3, cFace *face4, cFace *face5)
```

fügt eine Gitterzelle mit den sechs Randflächen `face0` bis `face5` in das Gitter

```
void buildGridFromPoints(int a, int b, int c)
```

Erzeugt ein kartesisches Gitter aus $a \times b \times c$ Knoten. Die Knoten müssen bereits zuvor in einer fest definierten Reihenfolge ins Gitter eingefügt worden sein, damit die Routine korrekt arbeitet. Sie legt dann Kanten, Flächen und Gitterzellen entsprechend an.

```
int getFreeDirection(cPoint* objPoint)
```

Prüft die zu `objPoint` inzidenten Kanten und zählt, wie viele von Ihnen in X-, Y- und Z-Richtung verlaufen. Sobald eine Richtung gefunden wird, in der bisher keine oder nur eine Kante verläuft, so wird diese Richtung als „noch freie Richtung" zurückgegeben.

```
bool smooth(bool smoothSmart = false)
```

Glättet das gesammte Gitter nach Laplace (s. Kap. 2.3.3). Der Parameter `smoothSmart` steuert, ob die Glättung nach einer Verschlechterung im Gitter abbrechen soll.

```
void umbrellaSmooth(bool pre = false)
```

glättet das Gitter mit Hilfe des Umbrellaoperators (s. Kap. 2.3.3)

```
void calcSmallestBiggestAnglesFaces(set<cFace*, compareFaces>
*usedFaces, double *minAngle, double *maxAngle, double *minFace,
double *maxFace)
```

berechnet den kleinsten und größten Winkel sowie die kleinste und größte Fläche innerhalb der Menge `usedFaces` und gibt diese über `minAngle` und `maxAngle`, `minFace` und `maxFace` zurück

```
cEdge* getNextEdge(cPoint *point, cEdge *lastEdge, int direction)
```

Gibt die nächste Kante ausgehend von `lastEdge` über `point` in Richtung `direction` zurück

```
static cPoint* getNextPoint(cPoint *point, cEdge **lastEdge,
int direction)
```

Gibt den nächsten Knoten ausgehend von `lastEdge` über `point` in Richtung `direction` zurück

```
int load(const char *filename, Maps_Modul * maps_modul,
bool faceKorrektur)
```

Lädt die Datei `filename` als Gitter. Dabei werden gleichzeitig in `maps_modul` die entsprechenden Werte gesetzt, damit zu jeden Zeitpunkt die nötige Datenkonsistenz zwischen den beiden Klassen herscht. `faceKorrektur` steuert die Art und Weise, wie der Status der geladenen Punkte auf die darüber konstruierten Kanten und Flächen übertragen wird.

```
int save(const char *filename,Maps_Modul * maps_modul)
```

Speichert das Gitter unter `filename` ab. Dabei werden die nötigen Daten beim Laden für `maps_modul` von `maps_modul` geschrieben.

```
int saveSolverData(const char *filename)
```

Speichert in der Datei `filename` die nötigen Daten des Gitters für einen entsprechenden Löser.

```
int loadSolverData(const char *filename)
```

Lädt vorher auf der GPU berechnete Daten für die Visualisierung.

Klasse cMultiGrid
Die Klasse `cMultiGrid` verwaltet ein `cGrid` und bietet zahlreiche zusätzliche Funktionen, die auch die Oberflächenparametrisierung mit einbeziehen. Ursprünglich sollte die Klasse ein mehrfach aufgelöstes Gitter speichern können (daher der Name), der GPU-Löser benötigt aber nur eine Auflösungsstufe.

```
void buildAllDisplayLists()
```

baut alle OpenGL *Display-Listen* auf, die zur Darstellung des Gitters benötigt werden

```
cGrid *getGrid()
```

liefert das `cGrid` Objekt, auf dem diese `cMultiGrid`-Instanz arbeitet

```
void setSmoothTypeNoSmooth()
void setSmoothTypeSmooth()
void setSmoothTypeSmoothSmart()
void setSmoothTypeUntangler()
void setSmoothTypeUmbrella()
void setSmoothTypeUmbrella2()
void setSmoothTypeOberflaeche()
void setSmoothTypeOberflaecheWinkel()
```

`void setSmoothTypeOberflaecheKruemmung()`

diese Routinen bestimmen, welche Art von Glättung von `doSmooth()` durchgeführt wird

`void doSmooth()`

führt die Glättung aus, die als aktueller Glättungsoperator eingestellt ist

`void untangleHexas()`

führt eine Glättung nach dem Verfahren zur Entfernung von Selbstdurchdringungen durch (s. Kap. 2.4)

`void smoothOberflaecheWinkel()`

führt eine Glättung der Oberflächenrandknoten durch, mit dem Ziel, die Winkel zu verbessern (s. Kap. 5.7.4)

`void smoothOberflaecheJacobi()`

führt eine Glättung der Oberflächenrandknoten mit Jacobi-Fehlern durch, mit dem Ziel, die Jacobi-Fehler zu beseitigen (s. Kap. 5.7)

`void setSmoothIter(int newIter)`

stellt ein, wie viele Iterationen beim Glätten durchgeführt werden sollen

`void setKruemmungsToleranz(int toleranz)`

reguliert, ab wann `smoothOberflaecheWinkel()` Knoten aus einer Position mit hoher Krümmung heraus verschiebt, um den Winkel zu verbessern

`int subdivide()`

führt einen Unterteilungsschritt durch

`int getSubdivisions()`

liefert die Anzahl der bisher durchgeführten Unterteilungsschritte

`void doQualityTests()`

führt folgende Qualitätstests durch: größter/kleinster Winkel, Kantenverhältnis, Seitenverhältnis und Jacobi-Test

`int getBadHexas()`

liefert die Anzahl der Hexaeder, die den Jacobi-Test nicht bestanden haben

`double getWorstEdgeRatio()`

liefert das schlechteste Kantenverhältnis im Gitter

```
double getSmallestAngle()
```

liefert den kleinsten Winkel im Gitter

```
double getBiggestAngle()
```

liefert den größten Winkel im Gitter

```
static cMultiGrid* createHexa(double, double, double, double,
double, double, int, int, int, bool wackeln = false)
```

eine statische Hilfsroutine, die regelmäßige Gitter vordefinierter Größe erzeugen kann, ggf. auch mit vertauschten Koordinaten (Parameter `wackeln = true`)

```
void toggleDrawGPUVec()
void toggleDrawGPUVecNorm()
void toggleDrawGPUPres()
void toggleRefine()
void toggleDrawGPUGrid()
void set_x_Value(int schicht)
void set_y_Value(int schicht)
void set_z_Value(int schicht)
void set_x()
void set_y()
void set_z()
```

Diese Methoden setzen Zeichenoptionen für die Visualisierung. Sie bestimmen welche Gitterschicht und Datenmenge gezeichnet wird, sowie die Art der Darstellung.

Klasse Untangler
Diese Klasse behebt Durchdringungen in Gitternetzen. Durchdringungen können durch Randanpassungen oder Glätten entstehen. Intern wird ein passendes lineares Program aufgestellt und mit Hilfe von *lpsolve* gelöst.

```
set<cPoint*,comparePoints> getNodes()
```

liefert die Knoten des zu optimierenden Hexaeders

```
vector<double*>* getLocalNeighbors(cPoint* point, cHexa* hex)
```

liefert die drei Nachbarn eines Knotens in einem Hexaeder

```
vector<double**>* buildLP(cPoint* fPoint)
```

stellt Zielfunktion und Nebenbedingungen für ein lineares Programm auf

```
double* getCoords(cPoint* point)
```

liefert die Koordinaten eines Gitterknotens

`double detMatrix(double** matrix)`

berechnet die Determinate einer 3×3 Matrix

`cPoint* optimizeLocal(vector<double**>* objective_f)`

Diese Funktion optimiert die Position eines Knotens eines Hexaeders. Dazu wird das lineare Program mit dem Werkzeug *lpsolve* gelöst. Der Rückgabewert ist die neue Position des Knotens.

`void optimizeElement()`

optimiert ein Hexaederelement durch aufeinanderfolgendes Aufrufen von `optimizeLocal()`

C.1.2 Modul „Snake"

Dieses Modul berechnet die lokal kürzeste Verbindungsstrecke (s. Kap. 2.5) auf einer 3D-Oberflächentriangulierung.

Klasse Snake

`float getSnakeLength()`

liefert die euklidische Länge der aktuell berechneten Snake zurück

`para_point bestimme_mitte()`

berechnet die Mitte der aktuell berechneten Snake

`dijkstra()`

berechnet den kürzesten Weg über Knoten mit Hilfe des Dijkstra-Algorithmus'

`snake_evolution()`

berechnet einen Schritt bei der Bewegung der Snake auf der Oberfläche (s. Kap. 2.5.3)

`snake_glaetter()`

berechnet eine längenoptimierte Snake durch sehr kleine Bewegungsschritte

`para_point get_middle(para_point a, para_point b)`

liefert durch Aufrufe von `dijkstra()`, `snake_evolution()`, `snake_glaetter()` und `bestimme_mitte()` die Mitte der Snake zurück

C.1.3 Modul „MAPS”

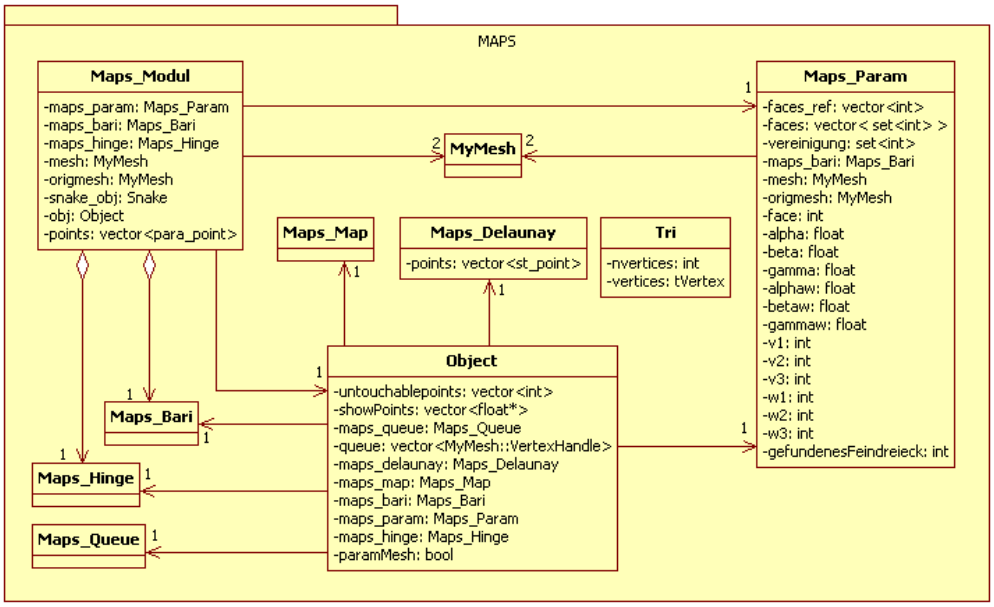


Abbildung C.3: *UML-Klassendiagramm für das Modul „MAPS”.*

Die Abbildung C.3 zeigt eine Übersicht der Klassen im Modul „MAPS”. Dieses Modul beinhaltet die zur Parametrisierung notwendige Funktionen.

Klasse Maps_Bari
stellt Funktionen zur Berechnung von baryzentrische Koordinaten zur Verfügung

`float area(st_point a, st_point b, st_point c)`

> berechnet die Fläche des Dreiecks, das durch die drei zweidimensionalen Knoten, `a`, `b` und `c` aufgespannt wird

`float * calc_baris(st_point a, st_point b, st_point c, st_point x)`

> Berechnet mit Hilfe von `area(...)` die baryzentrischen Koordinaten des Punktes `x` bezüglich des Dreiecks `a b c`. Der Rückgabewert ist eine Liste aus drei Fließkommazahlen. Der erste Wert bezieht sich auf Eckpunkt `a`, danach `b` und der letzte auf `c`.

Klasse Maps_Delaunay
Diese Klasse realisiert, basierend auf der Klasse `Tri`, eine auf das 3D-Gitter anwendbare Triangulierung. Die Triangulierung wird so weit möglich nach dem Delauney-Kriterium optimiert.

```
vector<st_point> delaunay(vector<st_point> punkte, MyMesh * mesh)
```

Liefert eine für das 3D-Netz gültige Triangulierung der Punktmenge `punkte`, die nach dem Delauney-Kriterium optimiert ist. `Mesh` ist eine Referenz auf das 3D-Netz, um die gültige Einbettung zu überprüfen. Der Rückgabewert ist eine Liste der Eingabepunkte, in der immer drei aufeinander folgende Punkte ein Dreieck bilden.

Klasse Maps_Hinge
dient zur Berechnung einer 2D-Einbettung von zwei Dreiecken mit einer gemeinsamen Kante

```
st_point * hinge(MyMesh::HalfedgeHandle heh, bool links_rechts,
MyMesh * myMesh)
```

Liefert eine 2D-Einbettung von allen Dreiecken mit der Halbkante `heh`. Der Parameter `links_rechts` gibt an, ob das, von der Halbkante aus gesehen, linke oder rechte Dreieck eingebettet wird. Bei der Einbettung bleiben alle Innenwinkel und Längenverhältnisse erhalten. Der Rückgabewert ist eine Liste von drei zweidimensionalen Knoten, die das Dreieck bilden.

Klasse Maps_Map
dient zur Berechnung der konformen Abbildung eines Dreieckringes um einen Punkt

```
vector<st_point> map(MyMesh::Point v_point, vector<MyMesh::Point> ring,
bool verbose)
```

Gibt die Einbettung um den Punkt `v_point` zurück. `ring` ist eine Liste der adjazenten Punkte von `v_point`. Die Rückgabe ist die Liste der Nachbarpunkte in der zweidimensionalen Einbettung.

```
double length (MyMesh::Point p)
```

berechnet den Abstand des Punkts `p` zum Nullpunkt

```
double angle (MyMesh::Point x, MyMesh::Point y, MyMesh::Point z)
```

gibt den inneren Winkel des Dreiecks `x y z` am Punkt `y` zurück

```
double tetaK (std::vector<MyMesh::Point> r, MyMesh::Point pi,
int k)
```

gibt die Summe der Winkel der ersten `k` Dreiecke, gegeben durch die Eckpunktliste `r` um den Punkt `pi`, zurück

Klasse Maps_Modul
dient zur Berechnung des mittleren Punktes zwischen zwei bekannten Punkten auf der Oberfläche des Objektes und zur Bestimmung möglicher Ausweichpunkte zur Glättung auf der Oberfläche

```
void setMesh(MyMesh * mesh, MyMesh * origmesh,
Maps_Param * maps_param)
```

setzt die Referenzen zu den Dreiecksnetzen des Objektes, der Vergröberung und der Parametrisierung

```
void setSnake(Snake * snake_obj)
```

setzt die Referenz zur Klasse `Snake`

```
float * find_middle(int index1, int index2, bool addToList = true)
```

Gibt einen Punkt auf der Oberfläche zwischen den zwei bekannten Punkten mit `index1` und `index2` zurück. Diese Mitte wird durch Parametrisierung bestimmt. `AddToList` gibt dabei an, ob der berechnete Punkt den bekannten Punkten hinzugefügt werden soll.

```
int add_point(int grobdeieck, float alpha, float beta, float gamma,
int v1, int v2, int v3, int feindreieck)
```

fügt einen Punkt mit den angegebenen Werten der internen Liste hinzu

```
int add_point(int eckpunkt)
```

fügt einen Eckpunkt des Dreiecknetzes der internen Liste hinzu

```
int add_point(para_point neuerPunkt)
```

fügt einen Punkt `neuerPunkt` mit den angegebenen Werten der internen Liste hinzu

```
vector<para_point> * getPointVector()
```

gibt die interne Liste der bekannten Punkt zurück

```
void setPointVector(vector<para_point> p)
```

überschreibt die interne Liste der bekannten Punkte

```
void setPoint(int maps_id, para_point pp)
```

überschreibt den Punkt an der Stelle `maps_id` in der internen Liste mit `pp`

```
void is_nachbearbeiten()
```

pflegt die Liste der bekannten Punkte, nach einem Vergröberungsschritt

```
set<maps_point*, compare_maps_points>*
getPointsAroundPoint(int maps_point_index)
```

Liefert Eckpunkte der Feintriangulierung zurück, die den Punkt `maps_point_index` `umgeben`. Das sind entweder die Eckpunkte des Feindreiecks, in dem der Punkt liegt, oder alle adjazenten Punkte, falls der Punkt schon Eckpunkt ist.

```
set<maps_point*, compare_maps_points>* Maps_Modul::
getNextRing(set<maps_point*, compare_maps_points>* oldPoints)
```

Berechnet zu jeden Punkt aus `oldPoints` mit Hilfe von `getPointsAroundPoint(...)` die umgebenden Eckpunkte und gibt die Vereinigung zurück. So wird eine größere Umgebung als mit `getPointsAroundPoint(...)` auf der Oberfläche bereit gestellt.

```
void updatePoint(int punkt,int vektorpunktindex)
```

pflegt die Liste der bekannten Punkte nach einer Veränderung durch eine Glättung

```
float getCurvatureMapsIdx(int maps_point_index)
```

Liefert eine Krümmung an dem Punkt mit dem Index `maps_point_index` aus der internen Liste zurück. Falls dieser innerhalb eines Feindreiecks liegt, wird die Krümmung an den drei umliegenden Eckpunkten gemittelt.

```
float getCurvature(int point_index)
```

liefert eine Krümmung an dem Punkt mit dem Index `point_index` aus der Feintriangulierung zurück

```
void setObj(Object* object)
```

setzt die interne Referenz der Klasse Objekt auf `object`

Klasse Tri
Diese Klasse liefert eine gültige Triangulierung einer Punktmenge in 2D. Intern werden dabei sukzessiv konvexe Ecken abgetrennt. Mit Ausnahme der Anpassung der Rückgabetypen ist diese Klasse aus „Computational Geometry in C" , Chapter 1 übernommen worden [45].

```
Tri(vector<st_point> points)
```

Wird die Punktmenge übergeben und eine Instanz des Triangulierungsobjekts wird erzeugt, die Methoden bereitstellt, um die Punktmenge zu triangulieren

```
void ReadVertices(vector<st_point>)
```

Um mit derselben Klasseninstanz unterschiedliche Punktmengen zu triangulieren, kann die Punktemenge mit `ReadVerticies` überschrieben werden.

```
vector<st_point> Triangulate()
```

Liefert die Triangulierung der Punktmenge. Je drei aufeinanderfolgene Punkte im Rückgabewert bilden dabei ein Dreieck.

C.1.4 Modul „GPU-Löser"

stellt einen Löser für das Poissonproblem auf der GPU bereit
Klasse GPUSolver

```
load_data* load(char *filename)
```

lädt eine Datei, die alle für die Lösung des Problems benötigten Gitterdaten enthält

```
void createBandsGPU(int width, int height, int depth, dist_inf* inf,
float *&dataDD, float *&dataDU, float *&dataDL, float *&dataUD,
float *&dataUU, float *&dataUL, float *&dataLD, float *&dataLU,
float *&dataLL, float* vec)
```

weist den Bändern der Finiten-Differenzen-Matrix ihre Werte zu, füllt die Bänder falls nötig mit Nulleinträgen auf und passt die Einträge im Falle von Randpunkten entsprechend an

```
static void grid2GMV(const char *filename, double* data,
gmv_arr* arr, load_data* dat, int schicht)
```

exportiert eine Punktschicht des Rechengitters, das Skalar- und Vektorfeld in das Format des Programms GMV

```
int solverGPU()
```

startet den GPU-Löser auf dem geladenen Rechengitter

```
double* calculateTestVector(int width, int height, int depth,
load_data* data)
```

Diese Funktion berechnet einen Vektor, der auf der rechten Seite der Poissongleichung eingesetzt wird. Die Funktion ruft `calculateDerivative()` auf.

```
double calculateDerivative(double x, double y,double z, int N,
int var, load_data* data)
```

berechnet die partielle Ableitung einer Referenzfunktion nach einer ausgewählten Variablen

C.2 Übersicht über die Klassen von HaGrid3D

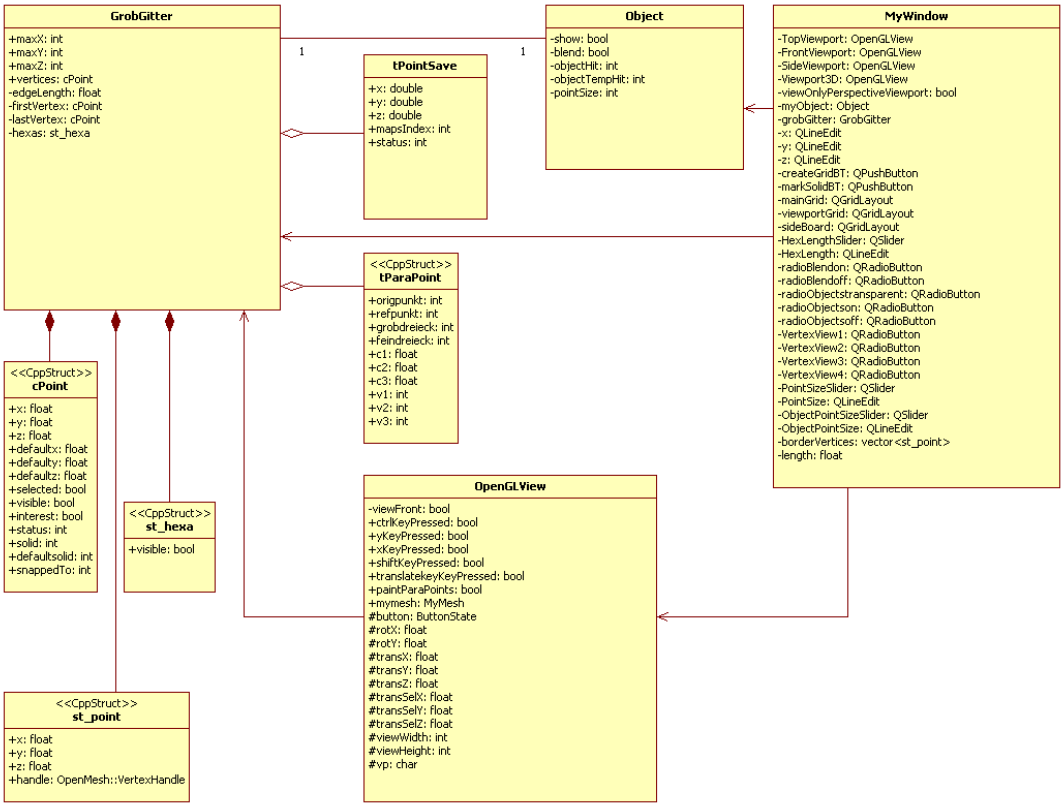


Abbildung C.4: *UML-Klassendiagramm für das Modul* HaGrid3D.

Klasse Grobgitter
erstellt und verwaltet das aktuelle Grobgitter im Editor

`void create(int x, int y, int z)`

Erstellt beim Aufruf ein neues Grobgitter. Als Parameter werden die Anzahl der Knoten in jede der drei Achsen übergeben. Zunächst wird ein eventuell vorhandenes Grobgitter vollständig gelöscht und anschließend neu erstellt. Die Knoten werden in drei verschachtelten Schleifen gleichmäßig hintereinander geschrieben. Der Abstand der Knoten zueinander ist die vordefinierte Kantenlänge der Hexaederzellen. Das Grobgitter wird, genauso wie das Objekt, im Koordinatenursprung zentriert.

`void detect_Interesting_Vertices()`

Diese Methode dient zur Klassifizierung von Knoten, die sich in der direkten Nähe zum Objekt befinden. Die Knoten erhalten beim Aufruf dieser Methode eine ent-

sprechende Markierung, wenn sie zu einer Kante inzident sind, die den Objektrand schneidet.

`void paint_Hexas()`

Sind Hexaederzellen als *sichtbar* markiert, so werden sie mit dieser Methode gemalt. Außerdem wird hier entschieden, ob die Kanten gezeichnet werden, die den Objektrand schneiden, wenn die Hexaederzellen ausgeblendet wurden.

`void paint_Vertices(int i, int o, int p)`

in Abhängigkeit von den Knoteneigenschaften (z.B. *interessant* oder *solid*-Status) wird hier den Knoten die Zeichengröße zugeordnet

`void paint_ParaFaces()`

Beim Aufruf werden die Hexaederseitenflächen, die sich auf dem Objektrand befinden, identifiziert und rot ausgefüllt. Die Kanten, die die Fläche in derselben Ebene verlassen, werden gelb gezeichnet.

`void translate_Vertex(float tx, float ty, float tz, int i, int o, int p)`

Diese Methode wird aufgerufen, wenn ein Knoten des Grobgitters verschoben wird. Mit den Parametern `i`, `o` und `p` wird der zu verschiebende Knoten identifiziert und die drei `float`-Werte `tx`, `ty` und `tz` geben die relativen Koordinatenänderungen in den drei Dimensionen an. Sollte ein Verschieben eine Hexaederzellendurchdringung hervorrufen, so wird die Verschiebung rückgängig gemacht.

`void translate_All_Vertices(float tx, float ty, float tz)`

durch den Aufruf wird das ganze Grobgitter um den `float`-Wert `tx` in X-Richtung, um `ty` in Y-Richtung und um `tz` in Z-Richtung relativ verschoben

`bool snap_Vertex(float* toSnapVertex, int indexOfObjectVertex)`

Der aktuell selektierte Grobgitterknoten erhält die Koordinaten des aktuell ausgewählten Objektknotens. Die Koordinaten werden in Form eines Zeigers auf ein dreidimensionales `float`-Array `toSnapVertex` an diese Methode übergeben und die Koordinaten des aktuellen Grobgitterknotens mit diesen überschrieben. Der Grobgitterknoten erhält den Index des Objektknotens als zusätzliches Attribut. Die Methode liefert als Rückgabewert `true`, falls das Verschieben erfolgreich war, ansonsten `false`.

`bool check_dependency(int i, int o, int p)`

Diese Testmethode liefert den Rückgabewert `true`, falls der Grobgitterknoten mit den Indizes `i`, `o` und `p` noch den vordefinierten Abstand `delta` zu seinen direkten sechs Nachbarn im Grobgitter hat und ansonsten `false`. Der Test ist notwendig, damit sich beim Verschieben von Knoten keine Hexaederzellen durchdringen können.

```
bool check_snap_dependency(int i, int o, int p, float snapx,
float snapy, float snapz)
```

Dieser Test wird aufgerufen, bevor der Grobgitterknoten auf den Objektrand an der Stelle `snapx`, `snapy`, `snapz` gezogen wird. Es wird getestet, ob sich durch das Verziehen Hexaederzellen gegenseitig durchdringen würden.

```
void save_File(const char* file)
```

Das aktuelle Grobgitter wird in einer Datei mit dem übergebenen Namen abgespeichert. Die Knoten werden der Reihe nach zeilenweise zusammen mit ihren Positionen und Attributen im ASCII-Format abgespeichert (s. Anhang A.3).

```
void load_File(const char* file)
```

ein zuvor mit `save_File` gespeichertes Grobgitter wird mit dieser Methode wieder in den Speicher geladen

```
void export_File(const char* file)
```

mit dieser Methode wird ein Objekt vom Typ Grobgitter in eine Datenstruktur überführt und unter `file` gespeichert, das von *InGrid3D* wieder eingelesen werden kann

```
void set_Actual_Vertex_To_Defaultvalues()
```

der aktuelle Grobgitterknoten bekommt durch den Aufruf dieser Methode wieder seine ursprünglichen Koordinaten und seinen *solid*-Status

Klasse OpenGLView
Diese Klasse erstellt und verwaltet alle OpenGL-Objekte, die zur Darstellung benötigt werden. Zudem enthält sie Methoden, die auf die Oberflächentriangulierung und auf das Grobgitter gleichzeitig zugreifen müssen, da sich hier die Schnittstelle aller aktuellen Objekte befindet.

```
reset()
```

die vier Ansichten auf das Objekt werden auf Ihre Startansichten zurückgesetzt

```
change_View_180()
```

in den drei 2D-Fenstern wird die Ansicht auf die jeweilige Rückansicht gesetzt

```
initializeGL()
```

Die Beschreibung der Koordinaten, Richtung, Stärke und Farbe der Lichtquellen, die sich innerhalb der Szene befinden und das Objekt anstrahlen. Die Materialeigenschaften von zu zeichnenden Flächen und Kanten werden ebenfalls definiert.

```
resizeGL( int w, int h )
```

die Grösse der aktuellen Ansicht wird gesetzt

`paintGL()`

die OpenGL Befehle zum Zeichnen des Grobgitters und des Objektes werden hier aufgerufen

`paintSelectedGL()`

Neuzeichnen von Objekten, die durch den Benutzer verändert wurden. Dies beinhaltet das Verschieben von Knoten und das Verändern der Ansicht.

`mousePressEvent(QMouseEvent * evt)`

Interaktion mit der Maus. Bei einer Betätigung der Maustaste werden die aktuellen Bildschirmkoordinaten überschrieben.

`mouseMoveEvent( QMouseEvent * evt)`

Bei einer Bewegung der Maus werden in Kombination mit verschiedenen Tasten folgende Aktionen ausgeführt:

- Bei gedrückter linker Maustaste wird in den 2D-Fenstern die Ansicht auf das Objekt verschoben. Im 3D-Fenster wird die Ansicht in Richtung der Mausbewegung gedreht.
- Bei gedrückt gehaltener rechter Maustaste wird die perspektivische Ansicht verschoben.
- Bei gedrückt gehaltener mittlerer Maustaste bzw. mit dem Mausrad wird die Darstellung vergrößert.

Klasse mywindow
erzeugt die grafische Benutzeroberfläche und stellt verschiedene Dialoge zur Verfügung

`void load_Object()`

öffnet einen Datei-Laden-Dialog, um ein Objekt zu laden

`void save_Grid()`

öffnet einen Speichern-Dialog, um ein Grobgitter mit der Endung `.cm` zu speichern, welches von *HaGrid3D* eingelesen werden kann

`void export_Grid()`

öffnet einen Speichern-Dialog, um ein Grobgitter mit der Endung `.grid` zu exportieren, welches von *InGrid3D* importiert werden kann

`void load_Grid()`

öffnet einen Datei-Laden-Dialog, um eine Grobgitterdatei (`.cm`) zu laden

```
void output_Objectattr_statusbar(const char* name, int vert,
int faces)
```

schreibt Objektinformationen in die Statuszeile

`void status_of_snapping_to_statusbar(bool snapping)`

schreibt den Status beim Verschieben von Knoten in die Statuszeile

`void createGrid()`

erzeugt ein Gitter mit den in der Steuerungsleiste angegebenen Dimensionen

`void updateHexLengthSlider(const QString& text)`

korrigiert die Position des Schiebereglers, wenn der Wert für die Kantenlänge per Hand verändert wurde

`void updateHexLength(int value)`

korrigiert den Wert im Eingabefeld für die Hexaederkantenlänge, wenn dieser über den Schieberegler verändert wurde

`void updatePointSizeSlider(const QString& text)`

korrigiert die Position des Schiebereglers, wenn der Wert für die Punktgröße der Grobgitterknoten per Hand verändert wurde

`void updatePointSize(int)`

korrigiert den Wert im Eingabefeld für die Punktgröße der Grobgitterknoten, wenn dieser über den Schieberegler verändert wurde

`void updateObjectPointSizeSlider(const QString& text)`

korrigiert die Position des Schiebereglers, wenn der Wert für die Punktgröße der Objektknoten per Hand verändert wurde

`void updateObjectPointSize(int)`

korrigiert den Wert im Eingabefeld für die Punktgröße der Objektknoten, wenn dieser über den Schieberegler verändert wurde

`void view_all()`

verändert die Ansicht so, dass alle Grobgitterknoten angezeigt werden

`void view_box()`

verändert die Ansicht so, dass alle Grobgitterknoten innerhalb eines minimalen Quaders, der das Objekt einschliesst, angezeigt werden

`void view_near()`

verändert die Ansicht so, dass alle Grobgitterknoten angezeigt werden, die entweder *interessante* oder *innere* Knoten sind

`void view_interest()`

verändert die Ansicht so, dass alle Grobgitterknoten angezeigt werden, die *interessant* sind

`void hide_Object()`

schaltet die Sichtbarkeit des Objektes aus

`void unhide_Object()`

schaltet die Sichtbarkeit des Objektes ein

`void blend_object_on()`

schaltet die Transparenz des Objektes ein

`void blend_object_off()`

schaltet die Transparenz des Objektes aus

`void refresh_all()`

aktualisiert die Objektansichten

`void drawHexas()`

verändert den Status der Sichbarkeit der Hexaederzellen des Grobgitters

`void mark_solid()`

führt für sämtliche Grobgitterknoten einen Inklusionstest durch und markiert diese entsprechend

`void drawPseudoEdges()`

verändert den Status der Sichtbarkeit so, dass nur Grobgitterkanten angezeigt werden, die den Objektrand schneiden

`void drawPseudoFaces()`

verändert den Status der Sichbarkeit der Hexaederseitenflächen, deren sämtliche Knoten auf den Objektrand verschoben wurden

`void resizeEvent ( QResizeEvent *)`

wird automatisch aufgerufen, wenn die Größe des Fenster verändert wurde und passt die Größe der Ansichtsfenster an

`void keyPressEvent ( QKeyEvent *)`

wird automatisch aufgerufen, wenn die Tastatur betätigt wird, und sorgt für die Ausführung aller Tastaturbefehle

```
void keyReleaseEvent ( QKeyEvent *)
```

wird automatisch aufgerufen, wenn ein Tastendruck ausgelöst wird

Klasse object
verwaltet die Informationen des Geometrieobjekts und stellt einfache Funktionen zur Unterstützung der Visualisierung zur Verfügung

```
void paint()
```

zeichnet das Objekt

```
void paintPoint(const float*, int)
```

zeichnet die Eckpunkte des Geometrieobjekts

```
void load_File(const char* file)
```

liest die Informationen aus einer STL-Datei und erzeugt daraus eine OpenMesh-Datenstruktur

```
vector<st_point> get_min_max()
```

berechnet die kleinsten und größten Koordinaten des Objektes in X-, Y- und Z-Richtung und gibt diese in einem Vektor aus

Anhang D

GNU General Public License

Version 2, June 1991
Copyright © 1989, 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.
Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.
The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice

and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Literaturverzeichnis

[1] AHLBERG, J.: *Active Contours in Three Dimensions*. Diplomarbeit, Linköping University, SE-581 83 Linköping, Sweden, 1996.

[2] BERKELAAR, M., DIRKS, J., EIKLAND, K. und NOTEBAERT, P.: *LPSOLVE*. http://groups.yahoo.com/group/lp_solve/.

[3] BISCHOFF, S., WEYAND, T. und KOBBELT, L.: *Snakes on triangle meshes*. Bildverarbeitung für die Medizin, S. 208–212, 2005. http://www-i8.informatik.rwth-aachen.de/publications/downloads/sotm.pdf.

[4] *Blender*. http://www.blender3d.org.

[5] BOLZ, J., FARMER, I., GRINSPUN, E. und SCHRÖDER, P.: *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. In: *Proceedings of ACM SIGGRAPH 2003* , Bd. 22, S. 917–924, 2003.

[6] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN K., HOUSTON, M. und HANRAHAN, P.: *Brook for GPUs: stream computing on graphics hardware*. ACM Trans. Graph., 23(3):777–786, 2004. http://doi.acm.org/10.1145/1015706.1015800.

[7] *CVS*. http://www.cvshome.org/docs/manual/cvs.html.

[8] DIJKSTRA, E. W.: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1:269–271, 1959.

[9] DOBKIN, D. P. und KIRKPATRICK, D. G.: *A Linear Algorithm for Determining the Separation of Convex Polyhedra.*. J. Algorithms, 6(3):381–392, 1985.

[10] FAN, Z., F. QIU, A. KAUFMAN und S. YOAKUM-STOVER: *GPU Clusters for High Performance Computing* . In: *Proceedings of ACM / IEEE Supercomputing Conference 2004*. ACM Press, 2004. http://www.cs.sunysb.edu/%7Evislab/projects/urbansecurity/GPUcluster_SC2004.pdf.

[11] *FEATFLOW*. http://www.featflow.de/.

[12] FERNANDO, R.: *GPU Gems* , Bd. Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley, 2004.

[13] FREITAG, L. A. und P. E. PLASSMAN: *Local optimization-based simplicial mesh untangling and improvement*. Int. J. Numerical Methods in Engineering, 49(1–2):109–125, 2000.

[14] GÖDDEKE, D.: *Geometrische Projektionstechniken auf Oberflächentriangulierungen zur numerischen Strömungssimulation mit hierarchischen Mehrgitterverfahren*. Diplomarbeit, Universität Dortmund, 2004.

[15] GÖDDEKE, D.: *GPGPU–Basic Math Tutorial*. Ergebnisberichte des Instituts für Angewandte Mathematik, Nr. 300, FB Mathematik, Universität Dortmund, 2005. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html.

[16] GÖDDEKE, D., STRZODKA, R. und TUREK, S.: *Performance and accuracy of mixed precision solvers in FEM simulations on anisotropic grids*. International Journal of Parallel, Emergent and Distributed Systems (IJPEDS), 2006. submitted.

[17] *General Purpose Computations on the GPU*. http://gpgpu.org.

[18] GROSSMANN, CH. und ROOS, H. G.: *Numerik partieller Differentialgleichungen*. Teubner, 2006. 3.Auflage.

[19] GUSKOV, I., VIDIMCE, SWELDENS und SCHRÖDER: *Normal Meshes*. In: *Proceedings of the 27th annual conference on computer graphics and interactive techniques*, Bd. 27, S. 95–102. ACM Press, 2000. ISBN 1-58113-208-5.

[20] HACKBUSCH, W.: *Iterative Lösung großer schwachbesetzer Gleichungssysteme*. Oxford, Clarendon Press, 2002.

[21] KASS, M., WITKIN, A. und TERZOPOULUS, D.: *Snakes: Active contour models*. International Journal of Computer Vision, S. 321–331, 1988.

[22] KELLY, S.: *Element Shape Testing*. AnSyS Inc., 1998. chapter 13 in Ansys Theory Reference; http://www.ansys.com.

[23] KIM, S.-J., KIM, C.-H. und LEVIN, D.: *Surface simplification using a discrete curvature norm.*. Computers & Graphics, 26(5):657–663, 2002.

[24] KNUPP, P. M.: *Matrix Norms and The Condition Number: A General Framework to Improve Mesh Quality Via Node-Movement, Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, U.S.A., pp.13-22, 1999*. http://www.andrew.cmu.edu/user/sowen/abstracts/Kn676.html.

[25] KOBBELT, L. P.: *Iterative Erzeugung glatter Interpolanten*. Dissertation, Universität Karlsruhe, Verlag Shaker, Aachen, 1995, 1994.

[26] KRÜGER, J. und WESTERMANN, R.: *Numerical Simulation on GPUs*. http://wwwcg.in.tum.de/Research/Projects/GPUSim.

[27] KRÜGER, J. und WESTERMANN, R.: *Linear algebra operators for GPU implementation of numerical algorithms*. In: *ACM Transactions on Graphics (TOG)*, Bd. 22, S. 908–916, 2003.

[28] KUZMIN, D.: *Skript zur Vorlesung CFD*, 2003.

[29] LAMPORT, L.: *LaTeX*. http://www.latex-project.org.

[30] LEE, A. W. F., SWELDENS, W., SCHRÖDER, P., COWSAR, L. und DOBKIN, D.: *MAPS: Multiresolution Adaptive Parameterization of Surfaces*. Computer Graphics Proceedings (SIGGRAPH 98), S. 95–104, 1998.

[31] LI, X.-Y. und FREITAG, L. A.: *Optimization-Based Quadrilateral and Hexahedral Mesh Untangling and Smoothing Techniques*. Techn. Ber., Argonne National Lab., 1999.

[32] LIU, Y., LIU, X. und WU, E.: *Real-Time 3D Fluid Simulation on GPU with Complex Obstacles*. In: *Computer Graphics and Applications*, S. 247–256. 12th Pacific Conference on (PG'04), 2004.

[33] LUEBKE, D., M. HARRIS, J. KRÜGER, T. PURCELL, N. GOVINDARAJU, I. BUCK, C. WOOLLEY und A. LEFOHN: *GPGPU: general purpose computation on graphics hardware*. In: *GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, S. 33, New York, NY, USA, 2004. ACM Press.

[34] MARK, W. R., GLANVILLE, R. S., AKELEY, K. und KILGARD, M. J.: *Cg: A System for Programming Graphics Hardware in a C–like Language*. In: *Proceedings of ACM SIGGRAPH 2003*, S. 896–907. ACM Press, 2003.

[35] MCCOOL, M., DU TOIT, S., POPA, T., CHAN, B. und MOULE, K.: *Shader algebra*. ACM Trans. Graph., 23(3):787–795, 2004. http://doi.acm.org/10.1145/1015706.1015801.

[36] MEISTER, A.: *Numerik linearer Gleichungssysteme*. Vieweg, 1999.

[37] MICROSOFT: *Visual Studio*. http://msdn.microsoft.com/vstudio/.

[38] *Microsoft High-Level-Shading-Language (HLSL)*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/HLSL_Shaders.asp.

[39] MÜLLER, H. und ABRAMOWSKI, S.: *Geometrisches Modellieren*. BI–Wissenschaftsverlag, 1992.

[40] MORELAND, K. und E. ANGEL: *The FFT on a GPU*. In: *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, S. 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[41] MORGAN, R.: *REVIEW: ATI Radeon X800 XT*. http://www.barefeats.com/radx800.html.

[42] NVIDIA: *Cg*. http://developer.nvidia.com/page/cg_main.html.

[43] OPENGL ARCHITECTURE REVIEW BOARD: *OpenGL*. http://www.opengl.org.

[44] *OpenGL-Shading-Language (GLSL)*. http://www.opengl.org.

[45] O'ROURKE, J.: *Computational Geometry in C, 2nd edition*. Cambridge University Press, 1998.

[46] OWEN, S.: *A Survey of Unstructured Mesh Generation Technology.*. In: *Proceedings of the 7th International Meshing Roundtable*, S. 239–267, 1998.

[47] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. und PURCELL, T. J.: *A Survey of General-Purpose Computation on Graphics Hardware*. In: *Eurographics 2005, State of the Art Reports*, S. 21–51, 2005.

[48] PHARR, M. (Hrsg.): *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Kap. 44: A GPU Framework for Solving System of Linear Equation, S. 703 – 718. Addison Wesley, 2005.

[49] POZO, R., HEROUX, M. A. und REMINGTON, K. A.: *Sparse BLAS Library: Lite and Toolkit Level Specifications*. Manual, BLAS Technical Forum, 1997.

[50] PURCELL, T. J., I. BUCK, W. R. MARK und P. HANRAHAN: *Ray Tracing on Programmable Graphics Hardware*. ACM Transactions on Graphics, 21(3):703–712, 2002. Proceedings of ACM SIGGRAPH 2002.

[51] RWTH AACHEN: *OpenMesh*. http://www.openmesh.org.

[52] SANDIA CORPORATION: *Cubit*. http://cubit.sandia.gov.

[53] SCHREIBER, P. und S. TUREK: *An Efficient Finite Element Solver for the Nonstationary Incompressible Navier–Stokes Equations in Two and Three Dimensions*. In: *Proc. Workshop 'Numerical Methods for the Navier–Stokes Equations'*, Bd. 47 d. Reihe *Notes on Numerical Fluid Mechanics*, S. 25–28. Vieweg, 1993.

[54] SCHWARZ, H. R.: *Methode der finiten Elemente, 2. Auflage*. Oxford, Clarendon Press, 1984.

[55] SCHWARZ, H. R.: *Numerische Mathematik*. Oxford, Clarendon Press, 1997.

[56] *The Stanford 3D Scanning Repository*. http://graphics.stanford.edu/data/3Dscanrep/.

[57] TAKANASHI, I., MURAKI, S., DOI, A. und KAUFMANN, A.: *3D Active Net for Volume Extraction*. Proc. SPIE Electronic Imaging, 98:184–193, 1998.

[58] THELIN, J.: *The Independent Qt Tutorial*. http://www.digitalfanatics.org/projects/qt_tutorial/.

[59] TIMASHEV, A.: *3D car gallery*. http://www.3dcar-gallery.com/.

[60] TROLLTECH: *Qt*. http://www.trolltech.com/products/qt/index.html.

[61] TUREK, S.: *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*. Springer, Berlin, 1999.

[62] WALTER, R.: *Differentialgeometrie*. Wissenschaftsverlag Bibliographisches Institut, 1978. ISBN 3-411-01543-8.

[63] WESSELING, P.: *Principles of computational fluid dynamics*. Springer, Berlin, 2001.

[64] WILLIAMS, T. und KELLEY, C.: *Gnuplot*. http://www.gnuplot.info/.

[65] WOBKER, H.: *Beschreibung und Implementierung einer Segmentierungs-Methode zur Parametrisierung von Dreiecksnetzen*. Diplomarbeit, Universität Duisburg-Essen, 2003.

[66] XU, C. und PRINCE, J.L.: *Active Contours, Deformable Models and Gradient Vector Flow*. http://iacl.ece.jhu.edu/projects/gvf/.

[67] XU, C. und PRINCE, J.L.: *Snakes, Shapes, and Gradient Vector Flow*. IEEE Transactions on Image Processing, 7(3):359–369, 1998.