Oberflächenreparametrisierung und Gittererzeugung für numerische Strömungssimulationen mit Hilfe von Catmull-Clark-Methoden

DIPLOMARBEIT

vorgelegt am
Institut für Geometrie und Praktische Mathematik

von
Christian Michael Rom
Matrikelnr.: 250405

Betreuender wissenschaftl. Mitarbeiter:	Dr. Karl-Heinz Brakhage
Betreuender Professor:	Prof. Dr. Siegfried Müller

Aachen, 29. April 2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Stelllen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen und Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, 29. April 2009

_____

Glossar

Bezeichnungen für Gitter(elemente):

Mesh	Gitter
Vertex	Gitterpunkt
Edge	Kante zur Verbindung von zwei Vertices
Face	durch Edges eingeschlossenes Flächenelement
Valenz	Die Anzahl der Edges, die in einem Vertex zusammenlaufen, nennt man Valenz dieses Vertex.
regulärer Vertex	In einem reinen Viereckgitter werden die Vertices mit Valenz $N = 4$ im Inneren und $N = 2$ oder $N = 3$ auf dem Rand als regulär bezeichnet.
reguläres Face	Weisen alle vier Vertices eines Viereckfaces die Valenz $N = 4$ auf, so bezeichnet man dieses Face als regulär.
extraordinary Vertex	Vertex, der nicht regulär ist, auch außergewöhnlich genannt. In dieser Arbeit gilt für einen extraordinary Vertex in einem reinen Viereckgitter: $N \in \{3, 5, 6\}$.
tagged Edge	zur Bildung scharfer oder abgerundeter Kanten im Inneren eines Gitters markiertes Edge
crease Edge	scharfe oder abgerundete Kante
Tag-Summe	Anzahl der in einem Vertex zusammenlaufenden tagged Edges
Patch	Flächenstück; kann aus einem oder mehreren Faces bestehen
$\#V$, $\#E$, $\#F$, $\#LP$	Von links nach rechts: Anzahl an Vertices, Edges, Faces und Limitpunkten
getrimmte Fläche	über eine Trimmkurve zurechtgeschnittene Fläche
rectangular Grid	Gitter mit einer Rechteckstruktur; in jedem inneren Gitterpunkt laufen vier Kanten zusammen
strukturiertes Gitter	Gitter mit geordneter Nummerierung der Vertices
Multiblock-Gitter	Aus mehreren Blöcken zusammengesetztes Gitter, jeder Block selbst ist ein strukturiertes Gitter.

Bezeichnungen aus dem Bereich Subdivision und Approximation:

Subdivision	Unterteilung einer Kurve oder Fläche durch Erzeugung neuer Vertices
Subdivision Surfaces	iterative Methode zur Erzeugung glatter Oberflächen aus einem vorgegebenen groben Anfangsgitter
interpolierende Subdivision	Subdivision unter Beibehaltung der Anfangsvertices
approximierende Subdivision	Subdivision mit Neuberechnung der Anfangsvertices
Vertex-Split-Schema	Subdivision-Methode, bei der jeder Vertex auf zwei neue Vertices aufgeteilt wird, auch Corner Cutting genannt

Edge-Split-Schema	Subdivision-Methode, bei der die neuen Vertices durch die Teilung der Edges entstehen
Face-Split-Schema	Erweiterung des Edge-Split-Schemas: Bei Oberflächen werden zusätzlich zu den Edges auch die Faces geteilt.
Maske	Abbildung oder Matrix, die angibt, welche Vertices bei der Durchführung eines Subdivision-Schrittes mit welcher Gewichtung zur Berechnung eines neuen Vertex verwendet werden
Limitpunkt	Der Limitpunkt $\mathbf{L}_i$ zu einem Punkt $\mathbf{P}_i^j$ ist definiert als $\mathbf{L}_i = \lim_{j\to\infty} \mathbf{P}_i^j$, d.h. als die Stelle, an der der Punkt bei Durchführung von $j \to \infty$ Subdivision-Schritten liegen würde.
Limitkurve/-fläche	aus $j \to \infty$ Subdivision-Schritten entstehende Kurve/Fläche von Limitpunkten
Surfacepunkt	durch die Projektion eines Limitpunkts auf eine gegebene Oberfläche entstehender Punkt

Mathematisches:

Stetigkeit, Glattheit	s. Kapitel 2.1
Spline	Eine Funktion, die stückweise aus Polynomen mit dem maximalen Grad n zusammengesetzt ist, heißt Spline n-ten Grades. Ihre Basisfunktionen sind die Monome $t^0, t^1, t^2, \cdots, t^n$.
B-Spline	Spline mit speziellen Basisfunktionen mit kompaktem Träger (d.h., die Funktionen sind nur auf einem kleinen Intervall von 0 verschieden). Der B-Spline n-ten Grades wird bezeichnet als $B^n(t)$, wobei der Parameter t innerhalb der Grenzen des Knotenvektors liegt.
uniformer B-Spline	B-Spline mit äquidistantem Knotenabstand

1 Einleitung

1.1 Motivation

In der Flugzeugentwicklung lauten die zentralen Ansatzpunkte heutzutage Sicherheit, Wirtschaftlichkeit und Umweltverträglichkeit. Beispielsweise entstehen während der Start- oder Landephase starke Verwirbelungen bei der Umströmung der Flügel, die eine Gefährdung nachfolgender Flugzeuge darstellen können. Beim Reiseflug in einer Höhe von durchschnittlich elf Kilometern dagegen beobachtet man - bedingt durch die umströmende Luft - beträchtliche statische und dynamische Auslenkungen der Tragflächen. Derartige aerodynamische und im zweiten Fall zusätzlich strukturdynamische Effekte wurden im Sonderforschungsbereich 401 „Strömungsbeeinflussung und Strömungs-Struktur-Wechselwirkung an Tragflügeln“ [1] in mehreren Teilprojekten sowohl mit Hilfe von experimentellen als auch numerischen Methoden analysiert, um im Sinne der eingangs erwähnten Entwicklungsschwerpunkte für zukünftige Flugzeuge Fortschritte zu erzielen.

Die numerische Simulation ist hierbei im Laufe der Zeit für die Beschreibung und das Verständnis komplexer Strömungsphänomene, die z.B. im Reiseflug oder während der Hochauftriebsphase auftreten, zu einem unverzichtbaren Mittel geworden. Sie kann Erkenntnisse liefern, die auf experimentelle Weise nur durch hohen zeitlichen und finanziellen Aufwand oder in vielen Fällen überhaupt nicht erreichbar wären.

Für den Sonderforschungsbereich wurde nachträglich das Zusatzprojekt „Transsonische Aerostrukturdynamik bei großen Reynoldszahlen“ genehmigt. Im Rahmen dieses Projekts wurde ein Halbmodell entworfen (s. Abb. 1.1). Alle dort verwendeten Flächen sind ungetrimmte B-Spline-Patches. In einem Transferprojekt wird der Flügel um ein Winglet erweitert und anschließend im Windkanal getestet. Dabei sind umfangreiche und aufwändige Strömungssimulationen erforderlich, die hochqualitative Gitter erfordern. Zur Erzeugung der Oberflächen solcher Gitter eignen sich die sogenannten Subdivision Surfaces, die erstmals 1978 vorgestellt und ursprünglich z.B. für die Produktion von Trickfilmen entwickelt wurden. Sie bilden eine iterative Methode zur Erzeugung glatter Oberflächen aus einem vorgegebenen groben Anfangsgitter. Die bei der Computeranimation in den 1990er Jahren erzielten Ergebnisse lassen sich so adaptieren, dass die Verfahren auch für die hier benötigten Zwecke Modellierung, Reparametrisierung und Gittererzeugung [2] von Interesse sind. Nach einer Modifizierung der Unterteilungsregeln eignet sich eine Variante dieser Subdivision Surfaces für die technische Anwendung besonders gut: die Catmull-Clark-Methode [3], entworfen von Edwin Catmull, heute Präsident der Walt Disney Animation

Studios und Pixar Animation Studios, und James H. Clark, Gründer der Silicon Graphics, Inc. Als Grenzfläche liefert die Methode fast überall reguläre B-Spline-Patches, die mit Ausnahme weniger sogenannter außergewöhnlicher Punkte (extraordinary Vertices) C^2-, d.h. krümmungsstetig sind, und ermöglicht damit die Erzeugung besonders glatter Oberflächen. Von besonderer Bedeutung ist zudem, dass aus einem beliebigen Polyeder-Gitter bereits nach dem ersten Unterteilungsschritt ausschließlich Vierecke vorliegen, welche die einzelnen Abschnitte der Spline-Flächen vorgeben können. Die so gewonnenen Catmull-Clark-Oberflächen lassen sich im Anschluss zur Approximation gegebener Oberflächen einsetzen, indem z.B. die Punkte der Grenzfläche auf die Punkte der Zielfläche projiziert werden.

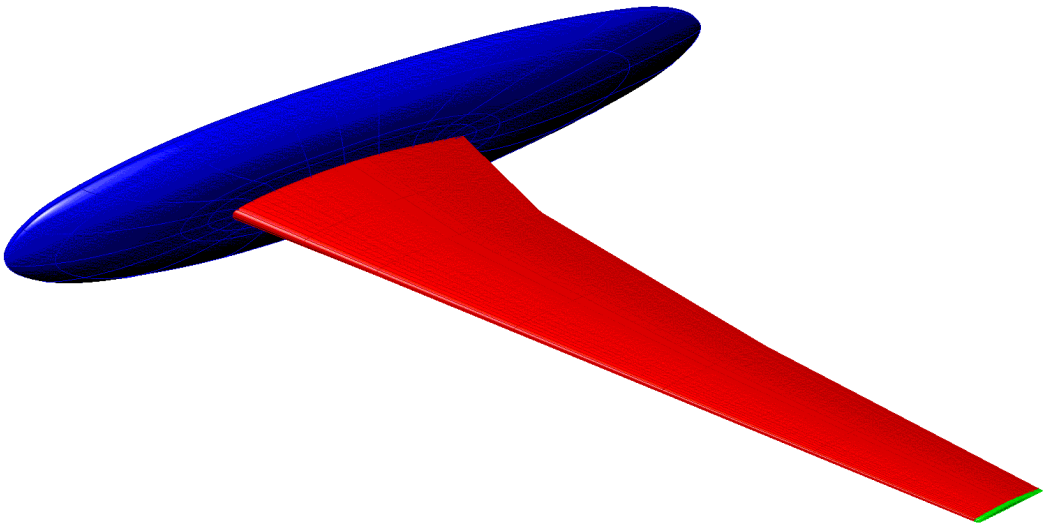

Abbildung 1.1: Halbmodell (Flügel mit halbem, vereinfachtem Rumpf)

1.2 Ziele

Das Hauptziel dieser Arbeit ist die Modifizierung und Erweiterung eines am Institut für Geometrie und Praktische Mathematik durch Dr. K.-H. Brakhage entwickelten C++-Programms (s. [4]), das die Unterteilung von Gittern nach der Catmull-Clark-Methode beherrscht und bereits eine Erweiterung der Unterteilungsregeln zur Modellierung scharfer Kanten und Beibehaltung vorgegebener Kurven enthält. Eine solche unveränderliche Kurve tritt beispielsweise beim Halbmodell aus Abbildung 1.1 am Übergang vom Rumpf zum Flügel auf. Zur grafischen Darstellung des Anfangsgitters und seiner Unterteilungen wird eine OpenGL-Oberfläche verwendet.

Für dieses Programm soll zunächst eine Konvertierung der bestehenden Algorithmen auf ein neues und effizienteres Datenmodell durchgeführt werden. Hierbei werden die für die Bestandteile eines Gitters (Vertices, Edges und Faces) zuvor verwendeten Arrays durch Vektoren aus der Standard Template Library von C++ ersetzt (zur genauen Umsetzung und den Vorteilen s. Kap.

3.3). Aufbauend auf dieser Änderung wird das Programm im Wesentlichen um die folgenden Funktionalitäten erweitert:

- Berechnung der Grenzkontrollpunkte eines Gitters (Kontrollpunkte der Grenzfläche, die nach unendlich vielen Unterteilungsschritten entstehen würde; auch Limitpunkte **L** genannt)
- Aufstellen der für die Approximation einer gegebenen Oberfläche erforderlichen Matrix A und ihrer Transponierten A^T, jeweils bestehend aus den Koeffizienten der Limitpunktberechnung ($A\mathbf{V} = \mathbf{L}$, wobei **V** die Vertices, also Kontrollpunkte des Gitters bezeichnet)
- Projektion der Limitpunkte auf ein Ellipsoid zur Bestimmung der sogenannten Surfacepunkte **S**
- Iterative Lösung der Approximationsaufgabe $\|A\mathbf{V} - \mathbf{S}\|_2 \rightarrow min$ über die CGLS-Methode
- Ausgabe der erzeugten Oberflächengeometrie für den institutseigenen B-Spline-Gittergenerator GNAGG: Nach der Erweiterung des generierten Oberflächengitters zu einem Volumengitter kann GNAGG dieses zu einem B-Spline-Gitter, wie es für adaptive Rechnungen mit dem Strömungslöser QUADFLOW (s. Kap. 5.3) benötigt wird, weiterverarbeiten.

Die Realisierung dieser Ziele wird in Kapitel 3 beschrieben. Zuvor wird die zum Verständnis notwendige Theorie über Stetigkeit, Subdivision, B-Splines, das Catmull-Clark-Schema und die Approximation gegebener Oberflächen in Kapitel 2 behandelt. Kapitel 4 zeigt beispielhaft mögliche Anwendungsweisen des entwickelten Programms. In Kapitel 5 wird anhand des konkreten Falls der Simulation einer Überschallanströmung einer Kugel, deren Oberfläche mit dem hier entwickelten Programm erstellt worden ist, demonstriert, für welche Zwecke das Programm eingesetzt werden kann. Zudem wird überprüft, ob sich bei der Simulation physikalische Ergebnisse einstellen. Kapitel 6 präsentiert schließlich eine Zusammenfassung des Erreichten und gibt einen Ausblick auf mögliche Programmerweiterungen.

2 Theorie

Die Beschreibung der Stetigkeit von Kurven und Flächen in Abschnitt 2.1 basiert auf [5]. Die Ausführungen zu Subdivision-Grundlagen und Splines in den beiden darauf folgenden Abschnitten 2.2 und 2.3 sind in ähnlicher Form in [6] und [7] zu finden. Für die Kapitel 2.4 (Catmull-Clark-Schema) und 2.5 (Approximation gegebener Oberflächen mit Catmull-Clark-Gittern) schließlich bildet [4] die Grundlage. Dort sind auch die Abbildungen 2.3, 2.4, 2.7, 2.8, 2.14 und 2.16 in gleicher oder ähnlicher Form zu finden.

Zum besseren Verständnis der in den Unterkapiteln 2.4 und 2.5 beschriebenen Theorie kann das in Kapitel 4.2 angegebene Beispiel dienen, das anhand des hier entwickelten Programms unter anderem die Durchführung der Catmull-Clark-Subdivision und der anschließenden Approximation demonstriert.

2.1 Stetigkeit und Glattheit von Kurven und Flächen

Da die Begriffe Stetigkeit und Glattheit im Laufe der Arbeit immer wieder auftauchen, werden an dieser Stelle zunächst Definitionen dieser Bezeichnungen gegeben. Im Folgenden seien zwei Kurvenstücke $\mathbf{x}(u)$ mit $u \in [u_0, u_1]$ und $\mathbf{y}(t)$ mit $t \in [t_0, t_1]$ gegeben, die sich im Punkt $\mathbf{P} = \mathbf{x}(u_1) = \mathbf{y}(t_0)$ berühren. Für C^k-Stetigkeit ist nun die Erfüllung der Bedingung

$$\left.\frac{d^i}{dt^i}\mathbf{y}(t)\right|_{t_0} = \left.\frac{d^i}{dt^i}\mathbf{x}(u)\right|_{u_1} \qquad \forall\ i = 0, \ldots, k \tag{2.1}$$

gefordert.

Die geometrische Stetigkeit GC^k dagegen ist eine Abschwächung der C^k-Stetigkeit. Hier lautet die Bedingung nach der Umparametrisierung der Kurve $\mathbf{x}(u)$ durch $u \to u(t)$ und mit $u(t_0) = u_1$

$$\left.\frac{d^i}{dt^i}\mathbf{y}(t)\right|_{t_0} = \left.\frac{d^i}{dt^i}\mathbf{x}(u(t))\right|_{u_1 = u(t_0)} \qquad \forall\ i = 0, \ldots, k. \tag{2.2}$$

Tabelle 2.1 listet für den Kurvenfall die in der Arbeit auftretenden Stetigkeiten und die jeweils an sie geknüpften Bedingungen auf.

Gegeben seien nun zwei Flächen $\mathbf{x}(u, v)$ und $\mathbf{y}(s, t)$, die sich im Punkt $\mathbf{P} = \mathbf{x}(\bar{u}, \bar{v}) = \mathbf{y}(\bar{s}, \bar{t})$ berühren. Nach den Umparametrisierungen $u \to u(s, t)$ und $v \to v(s, t)$ kann die Bedingung für

C^0	$\mathbf{y}(t_0) = \mathbf{x}(u_1)$ (stetig)
C^1	C^0 und $\frac{d}{dt}\mathbf{y}(t)\big\|_{t_0} = \frac{d}{dt}\mathbf{x}(u)\big\|_{u_1}$ (stetig differenzierbar)
C^2	C^1 und $\frac{d^2}{dt^2}\mathbf{y}(t)\Big\|_{t_0} = \frac{d^2}{dt^2}\mathbf{x}(u)\Big\|_{u_1}$ (krümmungsstetig)
$GC^0 (= C^0)$	$\mathbf{y}(t_0) = \mathbf{x}(u_1)$
GC^1	GC^0 und $\frac{d}{dt}\mathbf{y}(t)\big\|_{t_0} = \frac{d}{dt}u(t)\big\|_{t_0} \cdot \frac{d}{du}\mathbf{x}(u)\big\|_{u_1}$
GC^2	GC^1 und $\frac{d^2}{dt^2}\mathbf{y}(t)\Big\|_{t_0} = \left(\frac{d}{dt}u(t)\big\|_{t_0}\right)^2 \cdot \frac{d^2}{du^2}\mathbf{x}(u)\Big\|_{u_1} + \frac{d^2}{dt^2}u(t)\Big\|_{t_0} \cdot \frac{d}{du}\mathbf{x}(u)\big\|_{u_1}$

Tabelle 2.1: Übersicht über Stetigkeitsbedingungen für Kurven

GC^1-Stetigkeit mit Hilfe der Abkürzungen

$$\begin{aligned} a_{mn} &= \frac{\partial^k}{\partial s^m \partial t^n} u(s,t)\Big|_{(s,t)=(\bar{s},\bar{t})} && \text{mit } k = m+n, \\ b_{mn} &= \frac{\partial^k}{\partial s^m \partial t^n} v(s,t)\Big|_{(s,t)=(\bar{s},\bar{t})} && \text{mit } k = m+n \end{aligned} \tag{2.3}$$

geschrieben werden als

$$\begin{pmatrix} \mathbf{y}_s \\ \mathbf{y}_t \end{pmatrix} = \begin{pmatrix} a_{10} & b_{10} \\ a_{01} & b_{01} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_u \\ \mathbf{x}_v \end{pmatrix}. \tag{2.4}$$

Für GC^2-Stetigkeit muss zusätzlich die Bedingung

$$\begin{pmatrix} \mathbf{y}_{ss} \\ \mathbf{y}_{st} \\ \mathbf{y}_{tt} \end{pmatrix} = \begin{pmatrix} a_{20} & b_{20} \\ a_{11} & b_{11} \\ a_{02} & b_{02} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_u \\ \mathbf{x}_v \end{pmatrix} + \begin{pmatrix} a_{10}^2 & 2a_{10}b_{10} & b_{10}^2 \\ a_{10}a_{01} & a_{10}b_{01} + a_{01}b_{10} & b_{10}b_{01} \\ a_{01}^2 & 2a_{01}b_{01} & b_{01}^2 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_{uu} \\ \mathbf{x}_{uv} \\ \mathbf{x}_{vv} \end{pmatrix} \tag{2.5}$$

erfüllt sein.

Der Begriff der Glattheit einer Kurve oder Fläche hängt direkt mit der Stetigkeit zusammen. Vereinfacht ausgedrückt weist eine Kurve bzw. Fläche eine umso größere Glattheit auf, je höher ihre Stetigkeit ist.

2.2 Subdivision-Grundlagen

Die Grundidee der Subdivision könnte wie folgt formuliert werden: „Subdivision liefert eine glatte Kurve oder Oberfläche als Limit einer Sequenz von aufeinander folgenden Verfeinerungen.“ [6] Beispiele hierfür werden in den Abbildungen 2.1 für eine Kurve und 2.2 für eine Oberfläche gezeigt. Im Fall des Beispiels für die Kurve erkennt man, dass die vier Anfangspunkte im Laufe der Unterteilungsschritte beibehalten werden. Dies nennt man interpolierende Subdivision. Bei der Verfeinerung des Würfels dagegen handelt es sich um approximierende Subdivision, da die Anfangspunkte mit jeder Iteration neue Koordinaten erhalten.

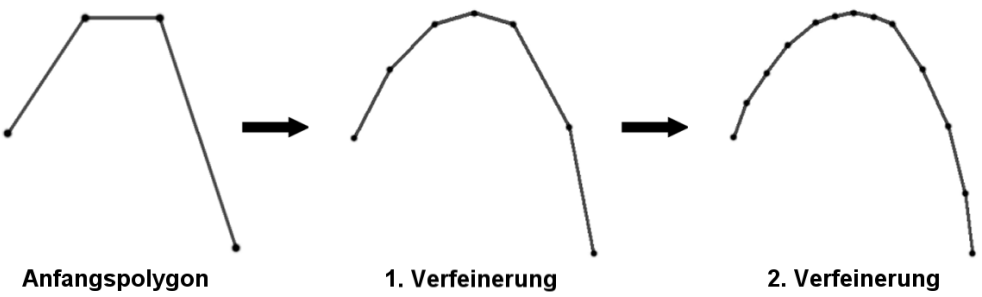

Abbildung 2.1: Beispiel für interpolierende Subdivision bei einer Kurve

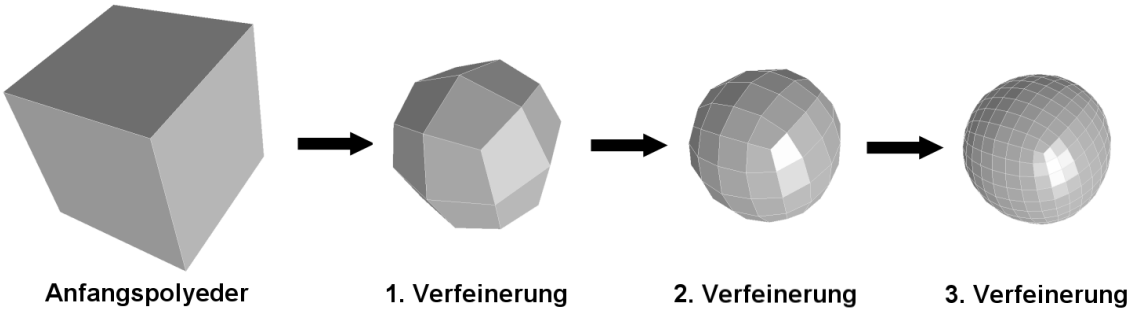

Abbildung 2.2: Beispiel für approximierende Subdivision bei einer Oberfläche

Die Anfänge der Subdivision liegen über 60 Jahre zurück. 1947 veröffentlichte Georges de Rham einen Artikel über Corner Cutting [8] mit einem Verhältnis von 1 : 1 : 1 (s. Abb. 2.3). Aus dem Startpolygon mit den Kontrollpunkten $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{n-1}$ werden die Punkte $\mathbf{P}_i$ nach folgender Vorgehensweise aufgeteilt:

$$\begin{aligned}
\mathbf{P}_0 \quad &\rightarrow \quad \begin{cases} \mathbf{Q}_{2n-1} &= \frac{1}{3}\left(2\mathbf{P}_0 + \mathbf{P}_{n-1}\right) \\ \mathbf{Q}_0 &= \frac{1}{3}\left(2\mathbf{P}_0 + \mathbf{P}_1\right) \end{cases} \\
\mathbf{P}_1 \quad &\rightarrow \quad \begin{cases} \mathbf{Q}_1 &= \frac{1}{3}\left(2\mathbf{P}_1 + \mathbf{P}_0\right) \\ \mathbf{Q}_2 &= \frac{1}{3}\left(2\mathbf{P}_1 + \mathbf{P}_2\right) \end{cases} \\
&\vdots \\
\mathbf{P}_i \quad &\rightarrow \quad \begin{cases} \mathbf{Q}_{2i-1} &= \frac{1}{3}\left(2\mathbf{P}_i + \mathbf{P}_{i-1}\right) \\ \mathbf{Q}_{2i} &= \frac{1}{3}\left(2\mathbf{P}_i + \mathbf{P}_{i+1}\right) \end{cases}
\end{aligned} \tag{2.6}$$

Das resultierende Polygon ist Ausgangspunkt für den nächsten, auf die gleiche Weise ausgeführten Schritt. Da bei dieser Methode alle Kontrollpunkte auf zwei neue Punkte aufgeteilt werden, spricht man von einem Vertex-Split-Schema. Die Limitkurve, die sich aus diesem sich wiederholenden Prozess ergibt, ist nur GC^0 $(= C^0)$, also stetig. Später verallgemeinerte de Rham sein Schema, indem er ein Gewicht ω einführte und als Verhältnis $\omega : 1 - 2\omega : \omega$ verwendete. Er

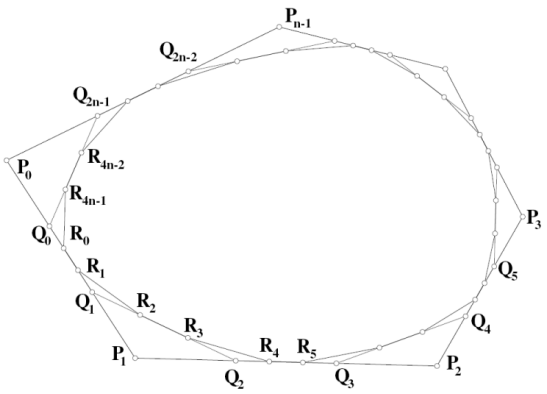

Abbildung 2.3: Corner Cutting nach de Rham von 1947: Das äußere Polygon mit den Punkten $\mathbf{P}_0, \mathbf{P}_1, ..., \mathbf{P}_{n-1}$ wird zwei Mal mit dem Edgeverhältnis 1:1:1 unterteilt. Das Resultat ist die innere Kurve $(\mathbf{R}_0, \mathbf{R}_1, ..., \mathbf{R}_{4n-1})$.

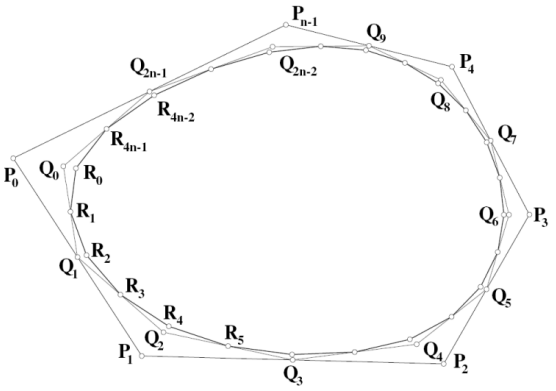

Abbildung 2.4: Edge-Split-Schema mit zwei Schritten: In jedem Schritt werden die Edges in zwei Hälften geteilt. Die neuen Punkte auf den Edges (ungerade Nummern) werden mit dem Vertexverhältnis 1:1 konstruiert, die alten Vertices erhalten ihre neue Position durch das Vertexverhältnis 1:6:1.

zeigte, dass die Limitkurve GC^1 $(= C^1$ nach Reparametrisierung), also stetig differenzierbar, nur für den Fall $\omega = \frac{1}{4}$ ist. 1974 entwickelte George Chaikin einen leistungsfähigen Algorithmus für die Generierung von Kurven [9], der mit de Rhams Methode mit $\omega = \frac{1}{4}$ übereinstimmte. Eine Analyse dieses Algorithmus brachte die Erkenntnis, dass sich als Limitkurve eine uniforme B-Spline-Kurve ergibt, deren Kontrollpunkte die Vertices des Startpolygons sind. Weitere Untersuchungen zeigten, dass jede uniforme B-Spline-Kurve mit beliebigem Grad über eine solche Subdivision-Methode konstruiert werden kann. Für den kubischen Fall ist dies in Abbildung 2.4 aufgezeigt. Dort werden die Edges in zwei Hälften geteilt. Aus diesem Grund nennt sich ein solches Vorgehen Edge-Split-Schema. Die Formeln hierbei lauten:

$$\begin{aligned} \mathbf{Q}_0 &= \frac{1}{8}(\mathbf{P}_{n-1} + 6\mathbf{P}_0 + \mathbf{P}_1) \\ \mathbf{Q}_1 &= \frac{1}{2}(\mathbf{P}_0 + \mathbf{P}_1) \\ &\vdots \\ \mathbf{Q}_{2i} &= \frac{1}{8}(\mathbf{P}_{i-1} + 6\mathbf{P}_i + \mathbf{P}_{i+1}) \\ \mathbf{Q}_{2i+1} &= \frac{1}{2}(\mathbf{P}_i + \mathbf{P}_{i+1}) \end{aligned} \tag{2.7}$$

Die ersten Schemata für die Subdivision bei Flächen wurden 1978 vorgestellt. Eines davon ist das Catmull-Clark-Schema, das in Kapitel 2.4 ausführlich vorgestellt wird. Eine Klassifizierung der verschiedenen Subdivision-Schemata für Flächen kann über folgende Aspekte erzielt werden:

- Art der Unterteilungsregel (Vertex-Split oder Edge-/Face-Split)
- interpolierendes oder approximierendes Verhalten

- Glattheit bzw. Stetigkeit der Grenzfläche
- Typ des zugrundeliegenden Meshes (z.B. Dreiecks- oder Viereckmesh)

Beispiele für Methoden, die sich in diese Klassifizierung einteilen lassen, sind in Tabelle 2.2 aufgelistet. Auf diese verschiedenen Schemata wird mit Ausnahme der Catmull-Clark-Methode im Folgenden nicht weiter eingegangen.

	Edge-/Face-Split		**Vertex-Split**
	Dreiecksmesh	Viereckmesh	
approximierend	Loop (C^2)	Catmull-Clark (C^2)	Doo-Sabin, Midedge (C^1)
interpolierend	Modified Butterfly (C^1)	Kobbelt (C^1)	Biquartic (C^2)

Tabelle 2.2: Klassifizierung verschiedener Subdivision-Schemata für Flächen

Da die meisten heute verwendeten Subdivisiontechniken wie oben beschrieben auf Splines, insbesondere B-Splines, basieren und dies auch für die Catmull-Clark-Methode gilt, folgt in Kapitel 2.3 eine kurze Erläuterung zu Splines, die auch die Verbindung zur Subdivision deutlich machen soll.

2.3 Splines

Spline-Kurven sind stückweise zusammengesetzte Polynome, die in ihren Segmenttrenngrenzen festgelegte Stetigkeitsbedingungen erfüllen sollen. Der Grad n einer Spline-Kurve ergibt sich aus dem maximalen Grad der beteiligten Polynome. In einem Beispiel für kubische Splines kann jedes Polynomsegment in Vektorschreibweise formuliert werden als

$$\mathbf{f}^i(t) = \mathbf{a}_3^i t^3 + \mathbf{a}_2^i t^2 + \mathbf{a}_1^i t + \mathbf{a}_0^i. \tag{2.8}$$

Hierbei sind die $\mathbf{a}^i$ die konstanten Koeffizienten, die die Form der Kurve im zugehörigen Segment i bestimmen. Diese Formulierung verwendet Monome (t^3, t^2, t^1, t^0) als Basisfunktionen. Bei kubischen Splines möchte man üblicherweise erreichen, dass die Übergänge und damit die gesamte Kurve C^2-stetig sind, wodurch die Koeffizienten benachbarter Kurventeilstücke bestimmte Vorschriften erfüllen müssen. Will man die Form der Kurve ändern und dabei die Vorschriften beibehalten, erfordert dies schwierige Umformungen, da sich alle Koeffizienten gegenseitig beeinflussen. Verallgemeinert lautet die Darstellung der Gleichung (2.8)

$$\mathbf{f}(t) = \sum_i b_i(t)\mathbf{P}_i. \tag{2.9}$$

$b_i(t)$ sind die Basisfunktionen und $\mathbf{P}_i$ die Koeffizienten, die Kontrollpunkte genannt werden. Statt der Monome wählt man jetzt die Basisfunktionen $b_i(t)$ so, dass der Einfluss eines Kontrollpunkts lokal ist (kompakter Träger) und dennoch stetige Kurven erzielt werden können. Das bedeutet,

dass bei beliebiger Verschiebung von Kontrollpunkten die Spline-Kurve dennoch ihre Stetigkeit behält, beispielsweise C^2 im kubischen Fall. Diese Glattheitseigenschaft liefern die im folgenden Abschnitt beschriebenen B-Splines.

2.3.1 B-Spline-Kurven

Es gibt verschiedene Möglichkeiten der Konstruktion von B-Splines. Gezeigt werden soll hier für den uniformen Fall (da nur dieser für die Catmull-Clark-Methode benötigt wird) der Ansatz über wiederholte Faltung. Als Startpunkt hierfür wird der einfachste Fall, eine stückweise konstante Funktion (Grad $n = 0$), gewählt, die als

$$\mathbf{f}(t) = \sum_i B_i^0(t)\mathbf{P}_i \tag{2.10}$$

formuliert werden kann, wobei $B^0(t)$ die Rechteckfunktion

$$B^0(t) = \begin{cases} 1, & \text{für } 0 \leq t < 1 \\ 0, & \text{sonst} \end{cases} \tag{2.11}$$

ist und $B_i^0(t) = B^0(t - i)$ die Translationen von $B^0(t)$ sind. Die Faltung zweier Funktionen $f(t)$ und $g(t)$ ist definiert als

$$(f * g)(t) = \int f(s)g(t - s)ds. \tag{2.12}$$

Eine B-Spline-Basisfunktion vom Grad n kann gewonnen werden, indem die Basisfunktion vom Grad $n - 1$ mit $B^0(t)$ gefaltet wird. Als Beispiel sei die Definition des B-Splines ersten Grades angegeben, der sich aus der Faltung der Rechteckfunktion mit sich selbst ergibt zu

$$B^1(t) = \int B^0(s)B^0(t - s)ds. \tag{2.13}$$

Grafisch kann Gleichung (2.13) wie in Abbildung 2.5 interpretiert werden. Eines der beiden Rechtecke wird auf der x-Achse von $-\infty$ bis ∞ verschoben, während das andere seine Position beibehält. Der Wert der Faltung an einer vorgegebenen Stelle ergibt sich dann aus dem Flächeninhalt des Produkts der beiden Rechtecke an dieser Stelle. Das Resultat dieses Vorgangs ist die in Abbildung 2.5 erkennbare lineare Dreiecksfunktion $B^1(t)$.

Faltet man n Mal, ergibt sich der B-Spline vom Grad n zu

$$B^n(t) = \int B^{n-1}(s)B^0(t - s)ds. \tag{2.14}$$

Dessen Stetigkeit folgt aus dem Theorem, dass aus der Faltung einer C^k-stetigen Funktion $f(t)$ mit $B^0(t)$ eine C^{k+1}-stetige Funktion resultiert. Der B-Spline vom Grad n ist also C^{n-1}-stetig, da der lineare B-Spline (Grad 1) C^0-stetig ist.

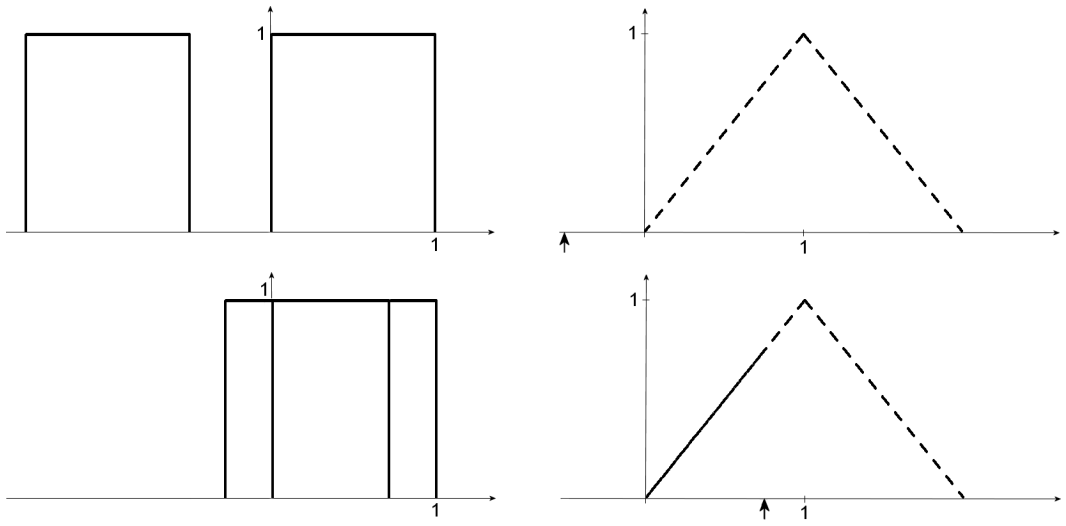

Abbildung 2.5: Definition des linearen B-Splines $B^1(t)$ (rechts) über Faltung von $B^0(t)$ mit sich selbst (links): Die obere und untere Zeile zeigen unterschiedliche Momentanzustände der Faltung, identifizierbar über den Pfeil.

Eine wichtige Eigenschaft von B-Splines ist ihre Verfeinerbarkeit, durch die sie mit Subdivisionalgorithmen eng verbunden werden. Man versteht darunter, dass man neue Kontrollpunkte so einfügen kann, dass sich die Kurve, die durch den B-Spline beschrieben wird, nicht verändert. Die weiter oben konstruierten Basisfunktionen erfüllen die sogenannte Verfeinerungsgleichung

$$B^n(t) = \frac{1}{2^n} \sum_{k=0}^{n+1} \binom{n+1}{k} B^n(2t-k). \tag{2.15}$$

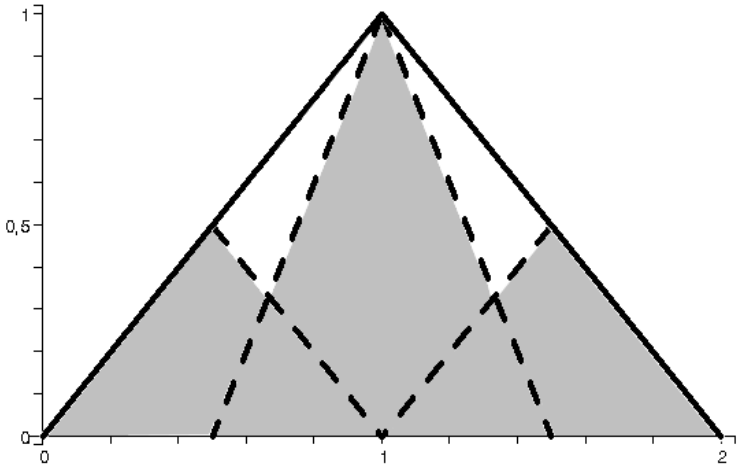

Abbildung 2.6: Die lineare Dreiecksfunktion $B_1(t)$ kann als Linearkombination aus gestauchten und verschobenen Dreiecksfunktionen $\frac{1}{2}B^1(2t) + B^1(2t-1) + \frac{1}{2}B^1(2t-2)$ (gestrichelt und grau hinterlegt) dargestellt werden.

Eine B-Spline-Basisfunktion kann also als Summe über gestauchte und verschobene Kopien von sich selbst formuliert werden. In Abbildung 2.6 ist dies am Beispiel der linearen Dreiecksfunktion dargestellt.

2.3.2 Subdivision für B-Spline-Kurven

Betrachtet werde eine B-Spline-Kurve vom Grad n in der Darstellung

$$\mathbf{f}(t) = \sum_i B_i^n(t)\mathbf{P}_i \tag{2.16}$$

mit dem Vektor von Kontrollpunkten $\mathbf{P} = (..., \mathbf{P}_{-2}, \mathbf{P}_{-1}, \mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, ...)^T$ um einen zentralen Punkt $\mathbf{P}_0$ und dem Vektor der Basisfunktionen $\mathbf{B^n}(t) = (..., B^n(t+2), B^n(t+1), B^n(t), B^n(t-1), B^n(t-2), ...)$. Hiermit lässt sich Gleichung (2.16) auch schreiben als

$$\mathbf{f}(t) = \mathbf{B^n}(t)\mathbf{P}. \tag{2.17}$$

Führt man nun den durch die Verfeinerungsgleichung motivierten Vektor $\mathbf{B^n}(2t) = (..., B^n(2t+2), B^n(2t+1), B^n(2t), B^n(2t-1), B^n(2t-2), ...)$ ein, kann man über eine Subdivision-Matrix S den Zusammenhang

$$\mathbf{B^n}(t) = \mathbf{B^n}(2t)S \tag{2.18}$$

herstellen. Die Einträge der Matrix S sind durch die Verfeinerungsgleichung zu

$$S_{2i+k,i} = \frac{1}{2^n}\binom{n+1}{k} \tag{2.19}$$

gegeben (mit $i = 0, \ldots,$ Anzahl der Kontrollpunkte $- 1$ und $k = 0, \ldots, n+1$) und entsprechen den Koeffizienten aus Gleichung (2.7). Die Kurve $\mathbf{f}(t)$ (s. Gl. (2.17)) lässt sich nun schreiben als

$$\mathbf{f}(t) = \mathbf{B^n}(t)\mathbf{P} = \mathbf{B^n}(2t)S\mathbf{P}. \tag{2.20}$$

Mit der neuen Basis geht man also von den alten Kontrollpunkten $\mathbf{P}$ auf die neuen Kontrollpunkte $S\mathbf{P}$ über, ohne die Kurve zu verändern. Sie wird lediglich von doppelt so vielen Basisfunktionen beschrieben, deren Träger jeweils halb so groß ist. Dieser Unterteilungsschritt lässt sich beliebig oft wiederholen und man erhält

$$\begin{aligned}
\mathbf{f}(t) &= \mathbf{B^n}(t)\mathbf{P}^0 \\
&= \mathbf{B^n}(2t)\mathbf{P}^1 \ = \mathbf{B^n}(2t)S\mathbf{P}^0 \\
&\vdots \\
&= \mathbf{B^n}(2^j t)\mathbf{P}^j = \mathbf{B^n}(2^j t)S^j\mathbf{P}^0,
\end{aligned} \tag{2.21}$$

wobei j das jeweilige Level der Verfeinerung angibt. Für die Beziehung zwischen den Kontrollpunkten zweier oder mehrerer aufeinander folgender Subdivisionlevel ergibt sich

$$\mathbf{P}^{j+1} = S\mathbf{P}^{j} = S^{j+1}\mathbf{P}^{0}. \tag{2.22}$$

Als veranschaulichendes Beispiel zeigt Gleichung (2.23) eine Unterteilung von Kontrollpunkten des Levels j bei Verwendung kubischer B-Splines (Grad $n = 3$). Für das Level $j+1$ sind die Punkte bestimmt über

$$\begin{pmatrix} \vdots \\ \mathbf{P}_{-3}^{j+1} \\ \mathbf{P}_{-2}^{j+1} \\ \mathbf{P}_{-1}^{j+1} \\ \mathbf{P}_{0}^{j+1} \\ \mathbf{P}_{1}^{j+1} \\ \mathbf{P}_{2}^{j+1} \\ \mathbf{P}_{3}^{j+1} \\ \vdots \end{pmatrix} = \frac{1}{8} \begin{pmatrix} \ddots & & & & & & \\ & 4 & 4 & 0 & 0 & 0 & \\ & 1 & 6 & 1 & 0 & 0 & \\ & 0 & 4 & 4 & 0 & 0 & \\ & 0 & 1 & 6 & 1 & 0 & \\ & 0 & 0 & 4 & 4 & 0 & \\ & 0 & 0 & 1 & 6 & 1 & \\ & 0 & 0 & 0 & 4 & 4 & \\ & & & & & & \ddots \end{pmatrix} \begin{pmatrix} \vdots \\ \mathbf{P}_{-2}^{j} \\ \mathbf{P}_{-1}^{j} \\ \mathbf{P}_{0}^{j} \\ \mathbf{P}_{1}^{j} \\ \mathbf{P}_{2}^{j} \\ \vdots \end{pmatrix}. \tag{2.23}$$

2.3.3 B-Spline-Flächen

Der Übergang von B-Spline-Kurven zu B-Spline-Flächen erfolgt über einen Tensorproduktansatz. Analog zu Gleichung (2.16) ergibt sich daraus die Darstellung einer B-Spline-Fläche vom Grad (n, m) zu

$$\mathbf{F}(t, s) = \sum_{i} \sum_{j} B_i^n(t) B_j^m(s) \mathbf{P}_{i,j}. \tag{2.24}$$

Die aus den Tensorprodukten resultierenden Subdivision-Koeffizienten werden im Rahmen des Catmull-Clark-Schemas in Kapitel 2.4, Gleichung (2.25) gezeigt.

2.4 Catmull-Clark-Schema

Die Catmull-Clark-Methode ist ein rekursiver Algorithmus für die Unterteilung bikubischer B-Spline-Patches. Für Gitter mit einer Rechteckstruktur (im Folgenden rectangular Grids genannt), bei denen in jedem inneren Gitterpunkt vier Kanten zusammenlaufen, liefert sie eine Standard-B-Spline-Fläche. Im Gegensatz zu den Tensorprodukt-Splines kann das Schema aber auch auf non-rectangular Grids angewendet werden. In diesem Fall gewinnt man nach Anwendung der Methode bis auf die Ausnahme einiger weniger Kontrollpunkte ebenfalls eine Standard-B-Spline-Fläche. Mit Ausnahme dieser sogenannten extraordinary Vertices (s. auch Begriffserklärung unten), an denen man C^1-Stetigkeit erhält, ergibt sich für diese Fläche schließlich C^2-Stetigkeit. Wie schon in Tabelle 2.2 gesehen, ist das Catmull-Clark-Schema als approximierendes Face-Split-Schema

angelegt. Die hierfür zugrundeliegenden Regeln werden weiter unten angegeben. Zunächst sollen aber wichtige Begriffe geklärt werden, die im Folgenden häufiger auftreten:

- **Valenz**: Die Valenz N eines Vertex ist die Anzahl der Edges, die in diesem Vertex zusammenlaufen. Handelt es sich um einen Vertex, der im Inneren des Gitters liegt, also nicht auf dem Rand, ist die Valenz auch gleich der Anzahl der zum Vertex benachbarten Faces.

- **reguläre** und **extraordinary Vertices**: In einem reinen Viereckgitter werden die Vertices mit Valenz $N = 4$ im Inneren und $N = 2$ oder $N = 3$ auf dem Rand als regulär bezeichnet. Weist ein Vertex eine andere Valenz auf, wird er extraordinary genannt. Für die in dieser Arbeit interessanten Konzepte der Modellierung und Gittergenerierung (s. dafür auch [2]) kann die Valenz der extraordinary Vertices auf $N \in \{3, 5, 6\}$ beschränkt werden.

- **crease/tagged Edges**: Crease Edges sind scharfe Kanten, die z.B. auf der Berandung eines Gitters auftreten. Für die Berandung müssen spezielle Regeln aufgestellt werden. Beim Catmull-Clark-Schema geschieht dies in der Weise, dass bei Durchführung der Subdivision die Berechnung der neuen Vertices der Randkurve nicht von Vertices im Inneren des Gitters abhängt. Um auch im Inneren crease Edges einzuführen, können die gewünschten Edges hierfür markiert werden und die Regeln der Randkurve werden dann auf alle auf diesen sogenannten tagged Edges liegenden Vertices angewendet.

- **Maske**: Subdivision-Regeln werden oft in Form einer Maske angegeben. Hierbei handelt es sich um eine Abbildung, die zeigt, welche Kontrollpunkte bei der Durchführung eines Subdivision-Schrittes mit welcher Gewichtung zur Berechnung eines neuen Kontrollpunkts verwendet werden (s. z.B. Abb. 2.8). Eine Maske kann auch in Matrixform formuliert werden, wie es beispielsweise in Gleichung (2.25) der Fall ist.

2.4.1 Unterteilungsregeln und Eigenschaften des Schemas

Ein Unterteilungsschritt des Catmull-Clark-Schemas läuft nach folgenden Regeln ab, die in Abbildung 2.7 für den nicht-regulären Fall eines N-Ecks mit $N \neq 4$ und in Abbildung 2.8 für den regulären Fall eines Vierecks auch grafisch dargestellt sind:

1. **Face → Vertex**: Einfügen neuer Face-Punkte jeweils als Schwerpunkt aller Vertices eines Faces

2. **Edge → Vertex**: Einfügen neuer Edge-Punkte jeweils als Mittel aus dem Mittelpunkt des Edges und den zwei neuen Face-Punkten der zum Edge benachbarten Faces

3. **Vertex → Vertex**: Neuberechnung der Koordinaten der Vertices jeweils aus dem Mittel der neuen Face-Punkte aller zum Vertex benachbarten Faces, dem Mittel aller Mittelpunkte der im Vertex zusammenlaufenden Edges und dem Vertex selbst

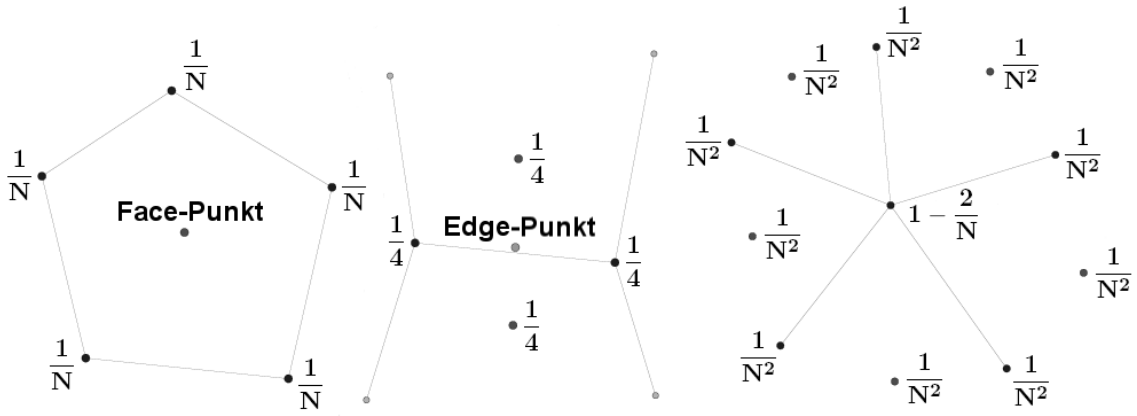

Abbildung 2.7: Subdivision-Regeln für die Catmull-Clark-Methode im nicht regulären Fall (z.B. Fünfeck) mit Angabe der Gewichtung der jeweils beteiligten Punkte: Von links nach rechts die Schritte Face → Vertex (der neue Face-Punkt ist der Schwerpunkt der Vertices des alten Faces), Edge → Vertex (der neue Edge-Punkt berechnet sich aus den neuen benachbarten Face-Punkten und den Vertices des alten Edges) und Vertex → Vertex (die neuen Koordinaten eines Vertex werden aus den neuen Face- und Edge-Punkten in der direkten Nachbarschaft berechnet)

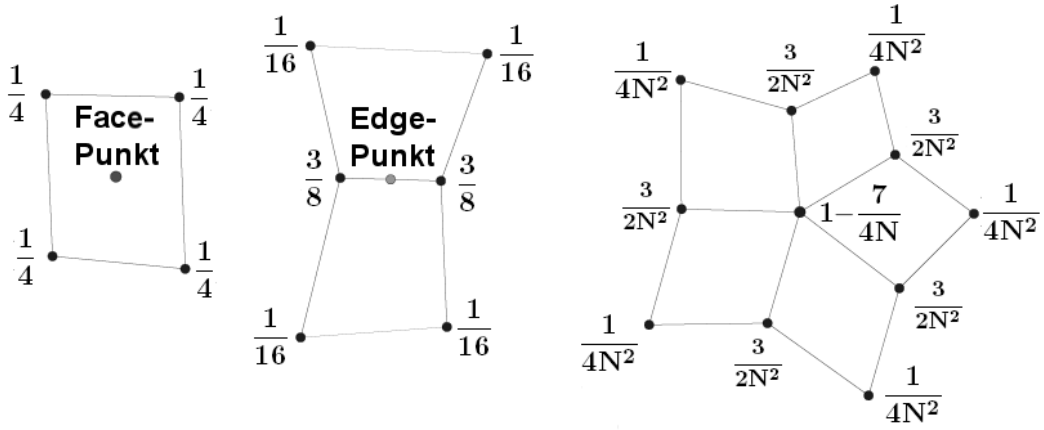

Abbildung 2.8: Subdivision-Regeln für die Catmull-Clark-Methode im regulären Fall des Vierecks mit Angabe der Gewichtung der jeweils beteiligten Punkte: Von links nach rechts die Schritte Face → Vertex, Edge → Vertex und Vertex → Vertex. Im Gegensatz zu Abbildung 2.7 sind die Gewichte hier jeweils nur auf die alten Vertices bezogen und nicht auf die neuen Face-Punkte aus dem ersten bzw. die neuen Edge-Punkte aus dem zweiten Schritt.

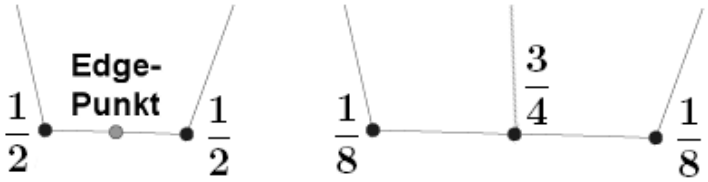

Abbildung 2.9: Subdivision-Regeln für die Catmull-Clark-Methode auf der Randkurve: Links Einfügen neuer Edge-Punkte, rechts Neuberechnung der alten Vertexkoordinaten für den mittleren Vertex. Die Gewichte entsprechen den Koeffizienten für das Knoteneinfügen bei uniformen B-Spline-Kurven.

Auf der Randkurve werden die bereits aus Gleichung (2.7) oder auch (2.19) bekannten Koeffizienten für das Knoteneinfügen bei B-Spline-Kurven verwendet, wie Abbildung 2.9 zeigt. Aus den Tensorprodukten dieser Kurvenkoeffizienten ergeben sich die regulären Koeffizientenmasken entsprechend Abbildung 2.8 (mit $N = 4$) wie folgt:

$$\begin{aligned}
M_F &= \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix} &&= \begin{pmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{pmatrix} && \text{(Maske für Face-Punkte)} \\
M_E &= \begin{pmatrix} \frac{1}{8} \\ \frac{3}{4} \\ \frac{1}{8} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix} &&= \begin{pmatrix} \frac{1}{16} & \frac{1}{16} \\ \frac{3}{8} & \frac{3}{8} \\ \frac{1}{16} & \frac{1}{16} \end{pmatrix} && \text{(Maske für Edge-Punkte)} \\
M_V &= \begin{pmatrix} \frac{1}{8} \\ \frac{3}{4} \\ \frac{1}{8} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \end{pmatrix} &&= \begin{pmatrix} \frac{1}{64} & \frac{3}{32} & \frac{1}{64} \\ \frac{3}{32} & \frac{9}{16} & \frac{3}{32} \\ \frac{1}{64} & \frac{3}{32} & \frac{1}{64} \end{pmatrix} && \text{(Maske für Vertices)}
\end{aligned} \tag{2.25}$$

Nach den Vertex-Schritten 1. bis 3. werden zum Abschluss einer Unterteilung noch zwei weitere Regeln angewendet:

4. **Edge** → **Edge**: Verbindung aller neuberechneten Vertices mit den neuen benachbarten Edge-Punkten

5. **Face** → **Edge**: Verbindung aller neuen Face-Punkte mit den neuen benachbarten Edge-Punkten, so dass innerhalb eines alten N-Eck-Faces N neue Faces entstehen

In Abbildung 2.10 ist ein Unterteilungsschritt, wie er sich durch die Regeln 1. bis 5. ergibt, grafisch für ein Beispiel dargestellt.

Die oben erläuterten Subdivision-Regeln führen zu den folgenden Eigenschaften, die das Catmull-Clark-Schema wie schon erwähnt für die Zwecke der Modellierung und Gittergenerierung so interessant machen:

- Für die Startoberfläche gibt es keine Restriktionen, da die Subdivision-Regeln auf Gitter beliebigen topologischen Typs angewendet werden können.

- Nach einem Unterteilungsschritt sind alle Faces Vierecke.

- Außer in der Nähe von extraordinary Vertices konvergiert die aus der wiederholten Anwendung der Subdivision resultierende Limitfläche gegen uniforme bikubische B-Spline-Patches. Das führt mit Ausnahme der Bereiche, in denen extraordinary Vertices liegen, zu einer C^2-stetigen Oberfläche.

- Die Anzahl der extraordinary Vertices bleibt nach dem ersten Subdivision-Schritt konstant.

- Spätestens nach zwei Unterteilungen sind die extraordinary Vertices isoliert, d.h., jedes Face hat höchstens einen extraordinary Vertex.

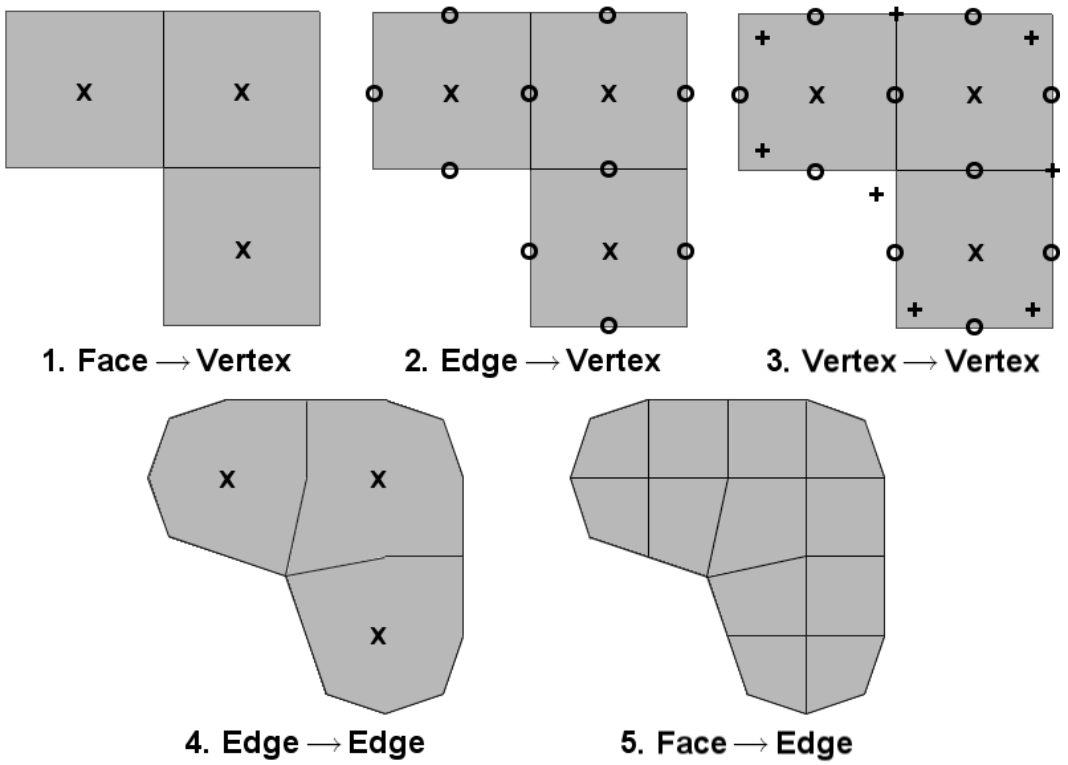

Abbildung 2.10: Veranschaulichung eines Catmull-Clark-Unterteilungsschrittes: 1. Einfügen neuer Face-Punkte (x), 2. Einfügen neuer Edge-Punkte (o), 3. Verschiebung der alten Vertices auf neue Koordinaten (+), 4. Neuverbindung der Vertices, 5. Verbindung der neuen Face-Punkte mit den neuen Edge-Punkten

- Es kann gezeigt werden, dass die Oberfläche in der Nähe eines extraordinary Vertex eine wohldefinierte Tangentenebene aufweist. Die Krümmung ist an dieser Stelle im Allgemeinen jedoch nicht wohldefiniert. Es folgt damit zumeist C^1-Stetigkeit im Bereich der extraordinary Vertices.

- Die Subdivision-Regeln können so angepasst werden, dass sowohl unendlich scharfe als auch abgerundete Kanten, d.h. Kanten, deren „Schärfe“ zwischen null (das bedeutet glatt) und unendlich liegen kann, realisierbar sind. Details hierzu sind in [10] nachzulesen. Diese zusätzliche Modellierungseigenschaft wird, wie in Kapitel 1.2 bereits erwähnt, beispielsweise beim Halbmodell (s. Abb. 1.1) für den Übergang vom Rumpf zum Flügel benötigt.

- Beschreibt man die Anzahl der Vertices mit $\#V$, die der Edges mit $\#E$ und die der Faces mit $\#F$, ergibt sich durch einen Subdivision-Schritt vom Level j zum Level $j+1$, sofern es sich beim Gitter auf Level j um ein reines Viereckgitter handelt:

$$\begin{aligned} \#V_{j+1} &= \#V_j + \#E_j + \#F_j \\ \#E_{j+1} &= 2\#E_j + 4\#F_j \\ \#F_{j+1} &= 4\#F_j \end{aligned} \tag{2.26}$$

2.4.2 Beispiele für die Anwendung der Catmull-Clark-Subdivision

In diesem Abschnitt werden drei Beispiele für die Anwendung der Catmull-Clark-Subdivision aus dem aeronautischen Bereich gezeigt. In allen Abbildungen ist zuerst das jeweilige einfache Anfangspolyeder dargestellt und darunter oder daneben das nach drei Subdivision-Schritten daraus entstandene Polyeder. Abbildung 2.11 zeigt die Modellierung eines Flugzeugtriebwerks.

In Abbildung 2.12 ist ein Teil einer Flügel-Rumpf-Konfiguration zu sehen. Hier werden an zwei Stellen tagged Edges eingesetzt, nämlich am Übergang vom Rumpf zum Flügel und außen am Übergang vom Flügel zur Flügelspitze. Diese tagged Edges ergeben jeweils geschlossene Kurven, die für zwei (im Bild schwarz) bzw. einen Unterteilungsschritt (im Bild grün) wie Randkurven behandelt werden (Verwendung von Kurvenkoeffizienten für die Subdivision wie in Gleichung (2.7)) und für die erst bei weiteren Unterteilungen die Masken für Oberflächen (s. Gl. (2.25)) zum Einsatz kommen. Dieses Vorgehen führt zu abgerundeten Kanten dort, wo sich anfangs die tagged Edges befinden.

Abbildung 2.13 kann als Detailbild des Flügel-Rumpf-Modells verstanden werden, da es die dort verwendete Flügelspitze darstellt. Links im Bild kann man die hohe Gitterqualität auch im Bereich von extraordinary Vertices (hier mit Valenz $N = 3$) erkennen, die die Catmull-Clark-Methode so interessant macht.

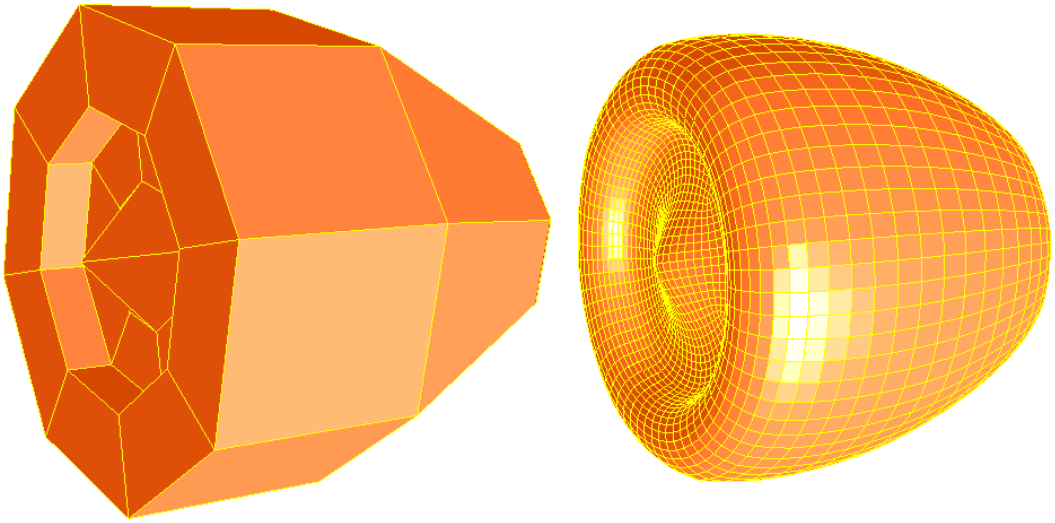

Abbildung 2.11: Flugzeugtriebwerk: Polyeder zu Beginn (50 Vertices) und nach dreimaliger Anwendung der Catmull-Clark-Subdivision (⇒ 3074 Vertices)

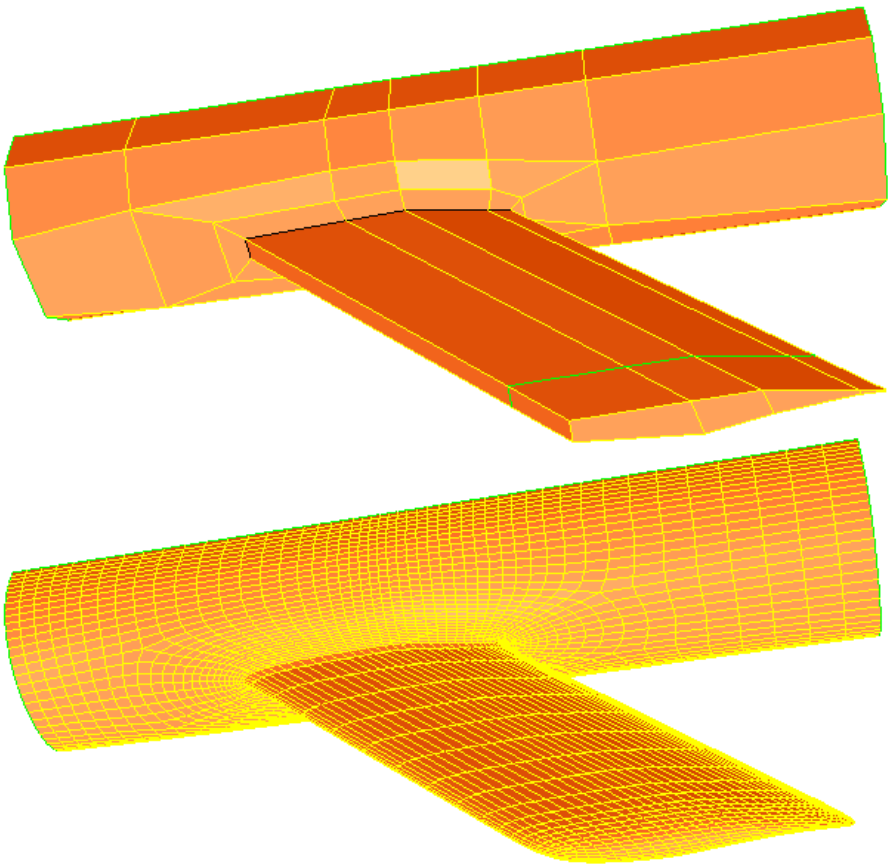

Abbildung 2.12: Flügel-Rumpf-Konfiguration: Polyeder zu Beginn (82 Vertices) und nach dreimaliger Anwendung der Catmull-Clark-Subdivision (⇒ 4569 Vertices)

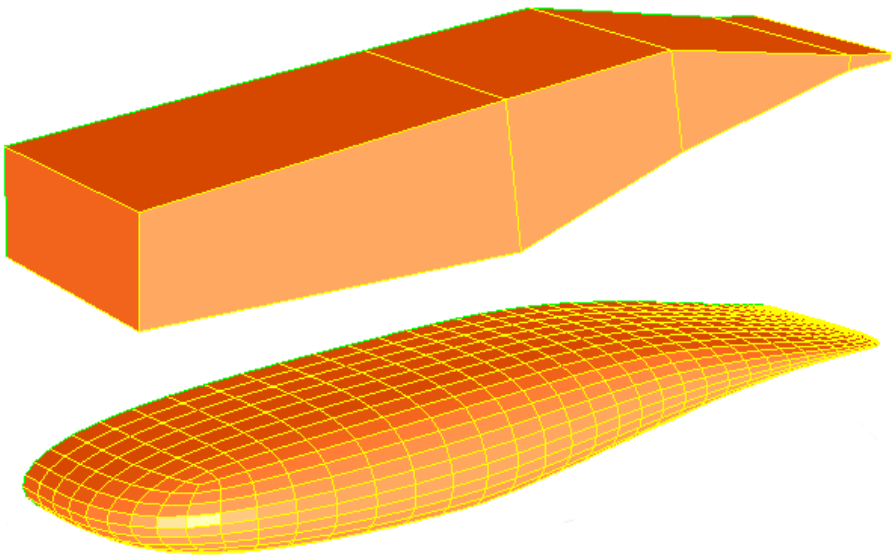

Abbildung 2.13: Flügelspitze: Polyeder zu Beginn (20 Vertices) und nach dreimaliger Anwendung der Catmull-Clark-Subdivision (⇒ 937 Vertices)

2.4.3 Berechnung von Limitpunkten

Ein Limitpunkt zu einem Punkt $\mathbf{P}_i^j$ ist definiert durch

$$\mathbf{L}_i = \lim_{j \to \infty} \mathbf{P}_i^j, \tag{2.27}$$

d.h., er ist definiert als die Stelle, an der der Punkt theoretisch bei Durchführung von $j \to \infty$ Subdivision-Schritten liegen würde.

Bei Oberflächen sind die Subdivision-Matrizen selbst bei Beschränkung der Valenz auf $N \leq 6$, was die Analysis angeht, von hoher Dimension (s. Tabelle 2.3). Aus diesem Grund werden die Ideen für Oberflächen nur erklärt und die Analysis dann für Kurven angegeben. Der Hauptunterschied bei der Subdivision von Kurven und Flächen ist die Existenz der extraordinary Vertices bei Flächen. Während ein Polygon bei der Kurvenunterteilung überall durch bekannte B-Spline-Algorithmen ausgewertet werden kann, ist dies bei Oberflächengittern nicht möglich. Die Kontrollpunktstruktur in der Nähe der extraordinary Vertices entspricht nicht der eines rectangular Grids, also können alle Faces, die extraordinary Vertices enthalten, nicht als uniforme B-Splines ausgewertet werden. Da schon nach einem Subdivision-Schritt alle Faces Vierecke sind und spätestens nach der zweiten Unterteilung in jedem Face höchstens ein extraordinary Vertex auftreten kann, wird für die folgenden Überlegungen von einem Gitter ausgegangen, das diese Voraussetzungen bereits erfüllt. Abbildung 2.14 zeigt, dass Regionen, in denen die Oberfläche nicht mit Standard-B-Spline-Methoden berechnet werden kann, mit jeder Unterteilung schrumpfen.

Das verbleibende Problem besteht in der Auswertung eines Patches, das einem Face mit genau

Valenz	Zeilen	Spalten
3	23	14
4	25	16
5	27	18
6	29	20
N	(2N + 1) + 7 + 9	(2N + 1) + 7

Tabelle 2.3: Dimensionen der Subdivision-Matrizen für verschiedene Valenzen

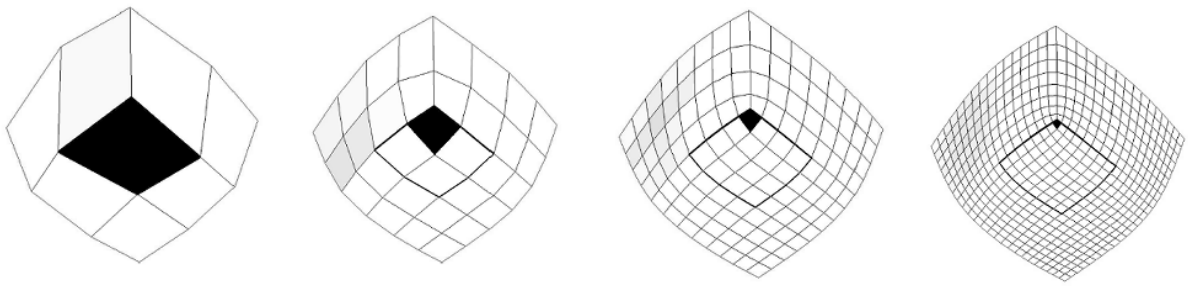

Abbildung 2.14: Verhalten der Catmull-Clark-Methode in der Nähe eines extraordinary Vertex mit der Valenz $N = 3$: Der nicht-reguläre Bereich (schwarz) wird mit jeder Unterteilung kleiner.

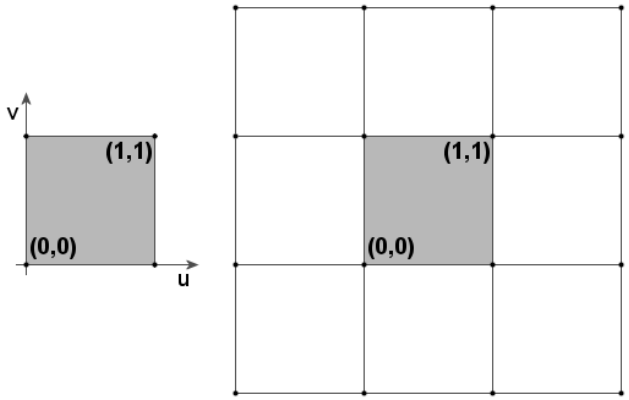

Abbildung 2.15: Ein bikubischer B-Spline ist über 16 Kontrollpunkte für die 16 Basisfunktionen $\mathbf{B^3}(u, v)$ definiert. Darüber lassen sich alle regulären Patches eines Gitters auswerten.

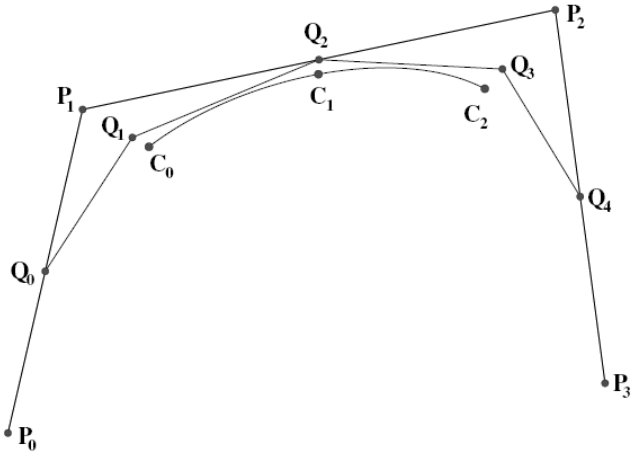

Abbildung 2.16: Kubisches Edge-Split-Schema mit Darstellung des Polygons nach einem Unterteilungsschritt und der Limitkurve $\mathbf{x}(t)$ mit $t \in [0, 1]$: Die Kontrollpunkte $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ bestimmen die Kurve zwischen $\mathbf{C}_0 = \frac{1}{6}(\mathbf{P}_0 + 4\mathbf{P}_1 + \mathbf{P}_2)$ und $\mathbf{C}_2 = \frac{1}{6}(\mathbf{P}_1 + 4\mathbf{P}_2 + \mathbf{P}_3)$. $\mathbf{C}_0 = \mathbf{x}(0)$ gehört dabei zu $\mathbf{P}_1$ und $\mathbf{C}_2 = \mathbf{x}(1)$ zu $\mathbf{P}_2$. Nach einem Subdivision-Schritt kann $\mathbf{C}_1 = \mathbf{x}(0.5)$ berechnet werden über $\mathbf{C}_1 = \frac{1}{6}(\mathbf{Q}_1 + 4\mathbf{Q}_2 + \mathbf{Q}_3) = \frac{1}{48}(\mathbf{P}_0 + 23\mathbf{P}_1 + 23\mathbf{P}_2 + \mathbf{P}_3)$.

einem extraordinary Vertex entspricht, wie z.B. der schwarze Bereich in Abbildung 2.14. Analog zu Kurven (vgl. Abb. 2.16) kann man die Faces parametrisieren und damit ein Patch $\mathbf{x}(u, v)$ über dem Einheitsquadrat $[0, 1] \times [0, 1]$ so definieren, dass der Punkt $\mathbf{x}(0, 0)$ dem extraordinary Vertex entspricht. Auf diesem Punkt $\mathbf{x}(0, 0)$ kann die Oberfläche als Linearkombination der umliegenden Vertices berechnet werden. Zusätzlich möglich ist die Berechnung von $\mathbf{x}(u, 1)$ mit $u \in [0, 1]$ und $\mathbf{x}(1, v)$ mit $v \in [0, 1]$, da es sich hierbei um reguläre B-Spline-Abschnitte handelt. Das Problem der Auswertung von $\mathbf{x}(u, v)$ im übrigen Teil des Einheitsquadrates lässt sich lösen, indem zuerst genügend Subdivision-Schritte ausgeführt werden, so dass (u, v) zu einem regulären Teilgebiet ohne extraordinary Vertices auf der dann erreichten Unterteilungsstufe wird. Danach kann die Auswertung als reguläres B-Spline-Patch erfolgen, veranschaulicht in Abbildung 2.15. Weitere Details hierzu sind in [11] zu finden.

Die zugrundeliegende Analysis wird nun anhand von Kurven erläutert. Stellt man sich $\mathbf{P}_1$ in Abbildung 2.16 als extraordinary Vertex vor, ist die Berechnung von $\mathbf{x}(t)$ mit $t \in [0,1]$ über Standard-B-Spline-Methoden nicht möglich, da hierfür $\mathbf{P}_1$ gekreuzt werden muss. Dieses Gedankenexperiment kann also als Entsprechung zum Fall eines Faces mit einem extraordinary Vertex bei einer Oberfläche angesehen werden und für die Berechnung von $\mathbf{x}(t)$ ist zunächst ein Unterteilungsschritt notwendig. Die zu $\mathbf{P}_1$ gehörende Subdivision-Matrix S_{00} ist dabei gegeben über

$$S_{00} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \\ 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix} \quad \text{mit} \quad \begin{pmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \mathbf{Q}_2 \end{pmatrix} = S_{00} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix}. \tag{2.28}$$

Die Eigenwerte von S_{00} lauten 1, $\frac{1}{2}$ und $\frac{1}{4}$. Für alle Subdivision-Schemata gilt, dass es einen einzelnen, größten Eigenwert 1 gibt, alle Eigenwerte real und die Subdivision-Matrizen diagonalisierbar sind (da alle Eigenwerte gleiche algebraische und geometrische Vielfachheiten haben). Eine Matrix, deren Spalten aus den Eigenvektoren von S_{00} bestehen, lautet

$$V_{00} = \begin{pmatrix} 1 & -1 & -2 \\ 1 & 0 & 1 \\ 1 & 1 & -2 \end{pmatrix} \quad \Rightarrow \quad V_{00}^{-1} = \begin{pmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ -\frac{1}{6} & \frac{1}{3} & -\frac{1}{6} \end{pmatrix} \tag{2.29}$$

und damit folgt

$$V_{00}^{-1} S_{00} V_{00} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix} = \Lambda_0. \tag{2.30}$$

Dieser Ausdruck wiederum ist äquivalent zu

$$S_{00} = V_{00} \Lambda_0 V_{00}^{-1}. \tag{2.31}$$

Die n-te Potenz von S_{00} lässt sich dann berechnen zu

$$S_{00}^n = (V_{00} \Lambda_0 V_{00}^{-1})^n = V_{00} \Lambda_0^n V_{00}^{-1}. \tag{2.32}$$

Bildet man hierfür den Grenzwert $n \to \infty$, ergibt sich

$$\lim_{n \to \infty} \Lambda_0^n = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} =: \Lambda_0^\infty \tag{2.33}$$

und damit

$$S_{00}^{\infty} := \lim_{n\to\infty} S_{00}^{n} = V_{00}\Lambda_0^{\infty}V_{00}^{-1} = \begin{pmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix}. \tag{2.34}$$

Bei der hier betrachteten Anwendung der Subdivision auf $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ liefert dies den Limitkurvenpunkt

$$\mathbf{C}_0 = \mathbf{x}(0) = \frac{1}{6}\mathbf{P}_0 + \frac{2}{3}\mathbf{P}_1 + \frac{1}{6}\mathbf{P}_2, \tag{2.35}$$

was der Anwendung der uniformen kubischen B-Spline-Basisfunktionen

$$\begin{aligned} B_0^3(s) &= \frac{1}{6}(1-s)^3 \\ B_1^3(s) &= \frac{1}{6} + \frac{1}{2}(1-s) + \frac{1}{2}(1-s)^2 s \\ B_2^3(s) &= \frac{1}{6} + \frac{1}{2}s + \frac{1}{2}s^2(1-s) \\ B_3^3(s) &= \frac{1}{6}s^3 \end{aligned} \tag{2.36}$$

für $s = 0$ entspricht und damit die Konsistenz zu uniformen B-Spline-Kurven, bei denen es wie erwähnt eigentlich keine extraordinary Vertices gibt, zeigt. Will man auf die gleiche Weise $\mathbf{C}_1 = \mathbf{x}(0.5)$ berechnen, wird zusätzlich Q_3 benötigt und die Subdivision-Matrix S_{00} muss erweitert werden zu

$$S = \begin{pmatrix} S_{00} & \mathbf{0} \\ S_{10} & S_{11} \end{pmatrix} = \left(\begin{array}{ccc|c} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \hline 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \end{array}\right). \tag{2.37}$$

Die neue Zeile folgt direkt aus den Regeln für das Knoteneinfügen bei uniformen B-Splines. Mit S lassen sich nun alle Limitkurvenpunkte im Intervall $(0, 0.5]$ berechnen. Für die Berechnung von Werten im Intervall $(0.5, 1)$ jedoch wird Q_4 benötigt, weshalb die Subdivision-Matrix S um eine weitere Zeile, erneut bestimmt durch die Regeln für das Knoteneinfügen bei uniformen B-Splines, vergrößert werden muss. Diese neue Matrix lautet

$$\hat{S} = \begin{pmatrix} S_{00} & \mathbf{0} \\ S_{10} & S_{11} \\ S_{20} & S_{21} \end{pmatrix} = \left(\begin{array}{ccc|c} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \hline 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \\ \hline 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{array}\right). \tag{2.38}$$

Zusammenfassend lassen sich die oben hergeleiteten Ergebnisse wie folgt beschreiben: Mit

$$\mathbf{P}_{00} = \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix}, \quad \mathbf{P} := \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix}, \quad \mathbf{Q}_{00} = \begin{pmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \mathbf{Q}_2 \end{pmatrix}, \quad \mathbf{Q} := \begin{pmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \mathbf{Q}_3 \end{pmatrix} \quad \text{und} \quad \hat{\mathbf{Q}} := \begin{pmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \mathbf{Q}_3 \\ \mathbf{Q}_4 \end{pmatrix} \tag{2.39}$$

lautet der erste Subdivision-Schritt in Matrixschreibweise

$$\mathbf{Q}_{00} = S_{00}\mathbf{P}_{00}, \quad \mathbf{Q} = S\mathbf{P} \quad \text{bzw.} \quad \hat{\mathbf{Q}} = \hat{S}\mathbf{P}. \tag{2.40}$$

Der Wert $\mathbf{C}_0 = \mathbf{x}(0)$ der Limitkurve aus Abbildung 2.16 wird mit Gleichung (2.34) berechnet zu

$$\mathbf{C}_0 = \mathbf{x}(0) = \frac{1}{6}\mathbf{P}_0 + \frac{2}{3}\mathbf{P}_1 + \frac{1}{6}\mathbf{P}_2. \tag{2.41}$$

Um $\mathbf{C}_1 = \mathbf{x}(0.5)$ zu berechnen, kann man folgendermaßen vorgehen: Zunächst wird ein Subdivision-Schritt mit der erweiterten Matrix S ausgeführt, nämlich $\mathbf{Q} = S\mathbf{P}$, so dass die Sequenz $(\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{Q}_3)$ im Gedankenexperiment keinen extraordinary Vertex kreuzt. Dann können die Standard-B-Spline-Gewichte $\frac{1}{6}$, $\frac{2}{3}$ und $\frac{1}{6}$ für die Berechnung von $\mathbf{x}(0.5)$ verwendet werden. Formal ist dies das Produkt der Maske $\mathbf{m} = \begin{pmatrix} 0 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix}$ mit $\mathbf{Q} = S\mathbf{P}$:

$$\mathbf{m}\mathbf{Q} = \mathbf{m}S\mathbf{P} = \begin{pmatrix} \frac{1}{48} & \frac{23}{48} & \frac{23}{48} & \frac{1}{48} \end{pmatrix} \mathbf{P} = \frac{1}{48}\mathbf{P}_0 + \frac{23}{48}\mathbf{P}_1 + \frac{23}{48}\mathbf{P}_2 + \frac{1}{48}\mathbf{P}_3 \tag{2.42}$$

Für die Berechnung von $\mathbf{C}_2 = \mathbf{x}(1)$ wird ebenfalls zuerst ein Subdivision-Schritt durchgeführt, hier mit der nochmals erweiterten Matrix $\hat{S}$, also $\hat{\mathbf{Q}} = \hat{S}\mathbf{P}$. Die Multiplikation der Maske $\mathbf{m} = \begin{pmatrix} 0 & 0 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix}$ mit $\hat{\mathbf{Q}} = \hat{S}\mathbf{P}$ liefert schließlich

$$\mathbf{m}\hat{\mathbf{Q}} = \mathbf{m}\hat{S}\mathbf{P} = \begin{pmatrix} 0 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix} \mathbf{P} = \frac{1}{6}\mathbf{P}_1 + \frac{2}{3}\mathbf{P}_2 + \frac{1}{6}\mathbf{P}_3. \tag{2.43}$$

Da die Sequenz $(\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3)$ im Gedankenexperiment keinen extraordinary Vertex kreuzt, können für die Berechnung von $\mathbf{x}(1)$ aber auch ohne vorherige Subdivision direkt die Standard-B-Spline-Gewichte $\frac{1}{6}$, $\frac{2}{3}$ und $\frac{1}{6}$ eingesetzt werden.

Da bei Kurven eigentlich keine extraordinary Vertices existieren, bekommt man bei direkter Anwendung der Basisfunktionen für uniforme kubische B-Splines aus Gleichung (2.36) mit $s = 0$, $s = \frac{1}{2}$ und $s = 1$ die gleichen Resultate.

Analog zur beschriebenen Berechnung der drei Punkte $\mathbf{x}(\frac{1}{2}i)$ mit $i \in \{0, 1, 2\}$ auf einer Limitkurve lassen sich auch die neun Punkte $\mathbf{x}(\frac{1}{2}i, \frac{1}{2}j)$ mit $i, j \in \{0, 1, 2\}$ eines jeden Faces auf einer Limitfläche des Catmull-Clark-Schemas berechnen. Diese neun Limitpunkte beziehen sich jeweils auf die vier Vertices, die vier Edge-Punkte und den Face-Punkt des betrachteten Faces.

Insgesamt ergibt sich die Anzahl der Limitpunkte zu $\#LP = \#V + \#E + \#F$. Das genaue Vorgehen bei der Bestimmung der Limitpunkte wird jetzt gezeigt. Bei der Berechnung kann faceweise vorgegangen werden, also wird die Theorie auch anhand eines einzelnen Faces demonstriert.

Es gilt weiterhin die Annahme, dass als Startpunkt für die Limitpunktberechnung im Flächenfall ein Gitter vorliegt, das ausschließlich aus Vierecken besteht. Daher können die Limitpunkt-Koeffizienten analog zu den Subdivision-Koeffizienten (s. Gl. (2.25)) über B-Spline-Tensorprodukte berechnet werden, solange ein Face keinen extraordinary Vertex enthält:

$$\begin{aligned}
C_F &= \begin{pmatrix} \frac{1}{48} \\ \frac{23}{48} \\ \frac{23}{48} \\ \frac{1}{48} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{48} & \frac{23}{48} & \frac{23}{48} & \frac{1}{48} \end{pmatrix} = \begin{pmatrix} \frac{1}{2304} & \frac{23}{2304} & \frac{23}{2304} & \frac{1}{2304} \\ \frac{23}{2304} & \frac{529}{2304} & \frac{529}{2304} & \frac{23}{2304} \\ \frac{23}{2304} & \frac{529}{2304} & \frac{529}{2304} & \frac{23}{2304} \\ \frac{1}{2304} & \frac{23}{2304} & \frac{23}{2304} & \frac{1}{2304} \end{pmatrix} \\
C_E &= \begin{pmatrix} \frac{1}{48} \\ \frac{23}{48} \\ \frac{23}{48} \\ \frac{1}{48} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix} = \begin{pmatrix} \frac{1}{288} & \frac{1}{72} & \frac{1}{288} \\ \frac{23}{288} & \frac{23}{72} & \frac{23}{288} \\ \frac{23}{288} & \frac{23}{72} & \frac{23}{288} \\ \frac{1}{288} & \frac{1}{72} & \frac{1}{288} \end{pmatrix} \\
C_V &= \begin{pmatrix} \frac{1}{6} \\ \frac{2}{3} \\ \frac{1}{6} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{pmatrix} = \begin{pmatrix} \frac{1}{36} & \frac{1}{9} & \frac{1}{36} \\ \frac{1}{9} & \frac{4}{9} & \frac{1}{9} \\ \frac{1}{36} & \frac{1}{9} & \frac{1}{36} \end{pmatrix}
\end{aligned} \tag{2.44}$$

Die Berechnung der Limitpunkte eines Faces mit einem extraordinary Vertex wird jetzt am Beispiel der Valenz $N = 3$ gezeigt. Abbildung 2.17 veranschaulicht diesen Fall und zeigt dabei auch die Nummerierung der an der Limitpunktberechnung für das betrachtete Face (grau hinterlegt) $2N + 8 = 14$ beteiligten Vertices. Für Valenzen $N > 3$ liegt die Stelle 7 links von 0 und oberhalb von 6. Ein solcher zusätzlicher Vertex existiert für $N = 3$ nicht. Allgemein lautet die Subdivision-Matrix S_{00} bei der Wahl dieser Nummerierung

$$S_{00} = \begin{pmatrix} a_N & b_N & c_N & b_N & c_N & b_N & \dots & b_N & c_N & b_N & c_N \\ d & d & e & e & 0 & 0 & \dots & 0 & 0 & e & e \\ f & f & f & f & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ d & e & e & d & e & e & \dots & 0 & 0 & 0 & 0 \\ f & 0 & 0 & f & f & f & \dots & 0 & 0 & 0 & 0 \\ \vdots & & & \vdots & & & \ddots & & \vdots & & \\ d & e & 0 & 0 & 0 & 0 & \dots & e & e & d & e \\ f & f & 0 & 0 & 0 & 0 & \dots & 0 & 0 & f & f \end{pmatrix} \tag{2.45}$$

mit $a_N = 1 - \frac{7}{4N}$, $b_N = \frac{3}{2N^2}$, $c_N = \frac{1}{4N^2}$ (vgl. Abb. 2.8, Vertex → Vertex-Schritt), $d = \frac{3}{8}$, $e = \frac{1}{16}$

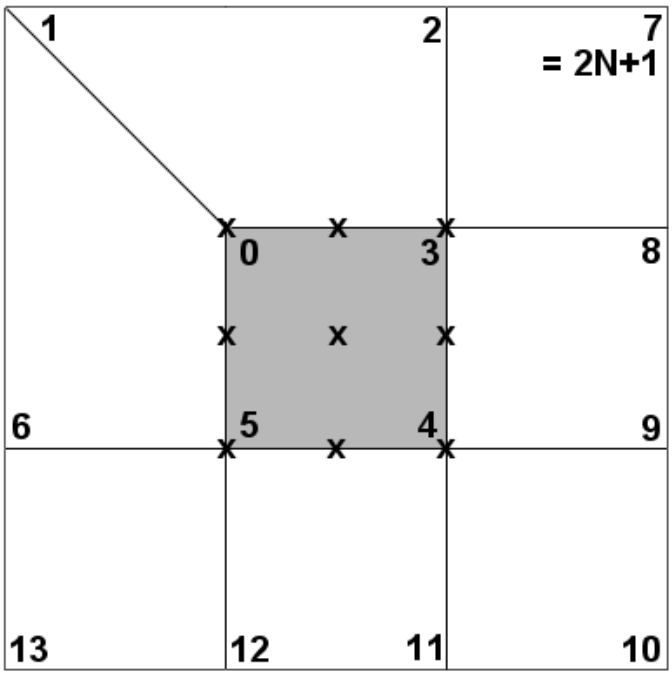

Abbildung 2.17: Ausgangssituation bei der Berechnung der neun mit x gekennzeichneten Limitpunkte für das grau hinterlegte Face mit Angabe der an der Berechnung beteiligten Vertices, einzusetzen in der Reihenfolge der Nummerierung von 0 (extraordinary Vertex mit Valenz $N = 3$) bis $2N + 7 = 13$

und $f = \frac{1}{4}$. Diese Matrix hat die Dimension $(2N + 1) \times (2N + 1)$. Für $N = 3$ ergibt sich damit

$$S_{00} = \begin{pmatrix} a_N & b_N & c_N & b_N & c_N & b_N & c_N \\ d & d & e & e & 0 & e & e \\ f & f & f & f & 0 & 0 & 0 \\ d & e & e & d & e & e & 0 \\ f & 0 & 0 & f & f & f & 0 \\ d & e & 0 & e & e & d & e \\ f & f & 0 & 0 & 0 & f & f \end{pmatrix} = \begin{pmatrix} \frac{5}{12} & \frac{1}{6} & \frac{1}{36} & \frac{1}{6} & \frac{1}{36} & \frac{1}{6} & \frac{1}{36} \\ \frac{3}{8} & \frac{3}{8} & \frac{1}{16} & \frac{1}{16} & 0 & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{3}{8} & \frac{1}{16} & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} & \frac{1}{16} & 0 \\ \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{3}{8} & \frac{1}{16} & 0 & \frac{1}{16} & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} \end{pmatrix}. \tag{2.46}$$

Nur ihre erste Zeile hängt von der Valenz N ab, die anderen sind durch die Koeffizienten für das Knoteneinfügen bei uniformen B-Splines gegeben, die in Gleichung (2.25) berechnet worden sind, wobei für S_{00} nur die Masken für Faces und Edges (M_F und M_E) benutzt werden. Zur Komplettierung der Subdivision-Matrix

$$S = \begin{pmatrix} S_{00} & \mathbf{0} \\ S_{10} & S_{11} \end{pmatrix} \tag{2.47}$$

werden noch die (hier für $N = 3$ angegebenen) Matrizen

$$S_{10} = \begin{pmatrix} \frac{1}{16} & \frac{1}{16} & \frac{3}{8} & \frac{3}{8} & 0 & 0 & 0 \\ \frac{3}{32} & \frac{1}{64} & \frac{3}{32} & \frac{9}{16} & \frac{3}{32} & \frac{1}{64} & 0 \\ \frac{1}{16} & 0 & 0 & \frac{3}{8} & \frac{3}{8} & \frac{1}{16} & 0 \\ \frac{1}{64} & 0 & 0 & \frac{3}{32} & \frac{9}{16} & \frac{3}{32} & 0 \\ \frac{1}{16} & 0 & 0 & \frac{1}{16} & \frac{3}{8} & \frac{3}{8} & 0 \\ \frac{3}{32} & \frac{1}{64} & 0 & \frac{1}{64} & \frac{3}{32} & \frac{9}{16} & \frac{3}{32} \\ \frac{1}{16} & \frac{1}{16} & 0 & 0 & 0 & \frac{3}{8} & \frac{3}{8} \end{pmatrix} \quad \text{und} \quad S_{11} = \begin{pmatrix} \frac{1}{16} & \frac{1}{16} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{64} & \frac{3}{32} & \frac{1}{64} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{16} & \frac{1}{16} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{64} & \frac{3}{32} & \frac{1}{64} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{16} & \frac{1}{16} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{64} & \frac{3}{32} & \frac{1}{64} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{16} & \frac{1}{16} \end{pmatrix} \tag{2.48}$$

benötigt. Die Dimension von S_{10} ist $7\times(2N+1)$, die von S_{11} lautet 7×7 und ist damit unabhängig von der Valenz des extraordinary Vertex.

Abbildung 2.18 zeigt, dass die vier Vertex-Limitpunkte (0, 3, 4 und 5) sowie die beiden Limitpunkte auf den vom extraordinary Vertex aus gesehenen äußeren Edges (3_4 und 4_5) direkt berechnet werden können, ohne vorher einen Subdivision-Schritt durchführen zu müssen. Dies wurde bereits zu Beginn dieses Kapitels anhand der Parametrisierung eines Faces zu einem Patch $\mathbf{x}(u, v)$ erläutert. Für die restlichen drei Limitpunkte (3_1, 4_1 und 5_1) wird dagegen zunächst die Subdivision-Matrix S benötigt (der Index 1 zeigt an, dass ein Unterteilungsschritt vor Berechnung der Limitpunkte notwendig ist).

Wie die neun Limitpunkte eines Faces im Einzelnen berechnet werden, wird jetzt beschrieben. Für den extraordinary Vertex ist der Limitpunkt gegeben über

$$\mathbf{L}_0 = \sum_{i=0}^{2N} c_{0,i}\mathbf{V}_i \tag{2.49}$$

mit den wie in Abbildung 2.17 nummerierten Vertices $\mathbf{V}$ und den Koeffizienten

$$\begin{aligned} c_{0,0} &= \tfrac{N}{N+5} \\ c_{0,2i+1} &= \tfrac{4}{N(N+5)} \\ c_{0,2i+2} &= \tfrac{1}{N(N+5)} \end{aligned} \tag{2.50}$$

für $i = 0, 1, \ldots, N-1$.

Die Limitpunkte $\mathbf{L}_3$, $\mathbf{L}_4$ und $\mathbf{L}_5$ können über Standard-B-Spline-Formeln gewonnen werden: Mit der Koeffizientenmatrix

$$C_V = \begin{pmatrix} \frac{1}{36} & \frac{1}{9} & \frac{1}{36} \\ \frac{1}{9} & \frac{4}{9} & \frac{1}{9} \\ \frac{1}{36} & \frac{1}{9} & \frac{1}{36} \end{pmatrix} \tag{2.51}$$

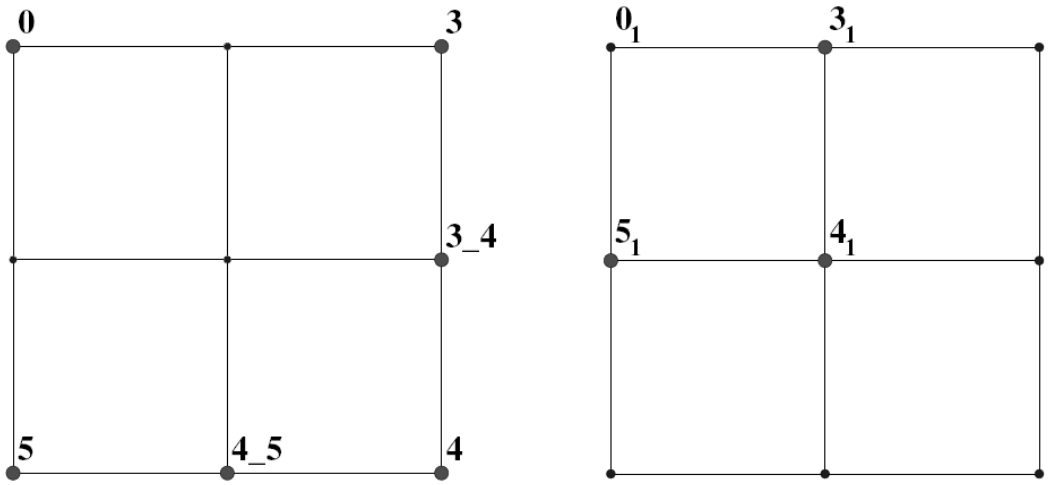

Abbildung 2.18: Berechnung der neun Limitpunkte eines Faces: Links sind die sechs Limitpunkte dargestellt, die direkt (ohne vorherige Subdivision) berechnet werden können. Das sind die Limitpunkte für die vier Vertices und für die zwei vom extraordinary Vertex aus gesehen außen liegenden Edges. Nach einem Subdivision-Schritt entsprechen die übrigen drei Punkte (kleine Kreise links, große rechts) und auch der Vertex mit der Nummer 0, dessen Limitpunkt aber schon bekannt ist, Vertices auf dem nächsten Level und ihre Limitpunkte können ermittelt werden.

aus Gleichung (2.44) und den Index-Matrizen

$$I^3 = \begin{pmatrix} 1 & 2 & 2N+1 \\ 0 & 3 & 2N+2 \\ 5 & 4 & 2N+3 \end{pmatrix}, \tag{2.52}$$

$$I^4 = \begin{pmatrix} 0 & 3 & 2N+2 \\ 5 & 4 & 2N+3 \\ 2N+6 & 2N+5 & 2N+4 \end{pmatrix} \tag{2.53}$$

und

$$I^5 = \begin{pmatrix} 1(7) & 0 & 3 \\ 6 & 5 & 4 \\ 2N+7 & 2N+6 & 2N+5 \end{pmatrix} \tag{2.54}$$

ergeben sich die Limitpunkte $\mathbf{L}_m$ mit $m \in \{3,4,5\}$ zu

$$\mathbf{L}_m = \sum_{i,j=0}^{2} C_{V_{i,j}} \mathbf{V}_{I^m_{i,j}}. \tag{2.55}$$

Der Vertexindex 1 in der Indexmatrix I^5 wird bei Valenzen $N > 3$ durch 7 ersetzt. Wie weiter oben erwähnt, existiert ein solcher zusätzlicher Vertex für $N = 3$ nicht. Mit den Koeffizienten-

matrizen

$$C_E = \begin{pmatrix} \frac{1}{288} & \frac{1}{72} & \frac{1}{288} \\ \frac{23}{288} & \frac{23}{72} & \frac{23}{288} \\ \frac{23}{288} & \frac{23}{72} & \frac{23}{288} \\ \frac{1}{288} & \frac{1}{72} & \frac{1}{288} \end{pmatrix} \tag{2.56}$$

und

$$C_E^T = \begin{pmatrix} \frac{1}{288} & \frac{23}{288} & \frac{23}{288} & \frac{1}{288} \\ \frac{1}{72} & \frac{23}{72} & \frac{23}{72} & \frac{1}{72} \\ \frac{1}{288} & \frac{23}{288} & \frac{23}{288} & \frac{1}{288} \end{pmatrix}, \tag{2.57}$$

die wiederum aus Gleichung (2.44) kommen, und den Indexmatrizen

$$I^{3\_4} = \begin{pmatrix} 1 & 2 & 2N+1 \\ 0 & 3 & 2N+2 \\ 5 & 4 & 2N+3 \\ 2N+6 & 2N+5 & 2N+4 \end{pmatrix} \tag{2.58}$$

und

$$I^{4\_5} = \begin{pmatrix} 1(7) & 0 & 3 & 2N+2 \\ 6 & 5 & 4 & 2N+3 \\ 2N+7 & 2N+6 & 2N+5 & 2N+4 \end{pmatrix} \tag{2.59}$$

lassen sich die beiden Limitpunkte $\mathbf{L}_{3\_4}$ und $\mathbf{L}_{4\_5}$ auf den Edges berechnen über

$$\mathbf{L}_{3\_4} = \sum_{i,j=0}^{3,2} C_{E_{i,j}} \mathbf{V}_{I^{3\_4}_{i,j}} \tag{2.60}$$

bzw.

$$\mathbf{L}_{4\_5} = \sum_{i,j=0}^{2,3} C^T_{E_{i,j}} \mathbf{V}_{I^{4\_5}_{i,j}}. \tag{2.61}$$

Um die Doppelsumme in den Gleichungen (2.55), (2.60) und (2.61) zu vermeiden, kann man neue Koeffizienten c_k^m definieren, die zunächst für $k = 0, \ldots, 2N+7$ den Wert 0 erhalten und dann auf $c^m_{I^m_{i,j}} = C_{V_{i,j}}$ bzw. $c^m_{I^m_{i,j}} = C_{E_{i,j}}$ gesetzt werden mit i und j in den entsprechenden Grenzen der jeweiligen Summe. Dann kann der Limitpunkt $\mathbf{L}_m$ mit $m \in \{3, 4, 5, 3\_4, 4\_5\}$ geschrieben werden als einfache Summe:

$$\mathbf{L}_m = \sum_{k=0}^{2N+7} c_k^m \mathbf{V}_k \tag{2.62}$$

Die übrigen drei Limitpunkte $\mathbf{L}_{3_1}$, $\mathbf{L}_{4_1}$ und $\mathbf{L}_{5_1}$ können wie erwähnt erst nach einem Subdivision-Schritt bestimmt werden. Dafür müssen die alten Vertices mit der Subdivision-Matrix S multi-

pliziert werden und es ergibt sich schließlich für $m \in \{3,4,5\}$

$$\mathbf{L}_{m_1} = \sum_{k=0}^{2N+7} c_k^{m_1} \mathbf{V}_k \tag{2.63}$$

mit den Koeffizienten

$$c_k^{m_1} = \sum_{i,j=0}^{2} C_{V_{i,j}} S_{I_{i,j}^m,k}, \tag{2.64}$$

die die Multiplikation mit S enthalten.

Eine andere Möglichkeit zur Berechnung der Limitpunkte bietet die Benutzung der erweiterten Subdivision-Matrix (vgl. Gl. (2.38))

$$\hat{S} = \begin{pmatrix} S_{00} & \mathbf{0} \\ S_{10} & S_{11} \\ S_{20} & S_{21} \end{pmatrix} \tag{2.65}$$

mit den (wieder für $N = 3$ angegebenen) Teilmatrizen

$$S_{20} = \begin{pmatrix} 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{16} & \frac{3}{8} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{3}{8} & \frac{1}{16} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} \end{pmatrix} \quad \text{und} \quad S_{21} = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{16} & \frac{3}{8} & \frac{1}{16} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} & \frac{1}{16} & 0 & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & \frac{1}{16} & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{16} & \frac{3}{8} & \frac{1}{16} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} \end{pmatrix}. \tag{2.66}$$

Die Dimension von S_{20} lautet $9 \times (2N+1)$, während die von S_{21} mit 9×7 unabhängig von der Valenz des extraordinary Vertex ist. Insgesamt hat $\hat{S}$ dann die Dimension $(2N+1+7+9) \times (2N+1+7)$ (vgl. Tab. 2.3). Angewendet auf die $2N+8$ Vertices ergibt sich ein Unterteilungsschritt, wie er in Abbildung 2.19 für das Beispiel $N = 3$ veranschaulicht ist. Die mit $\mathbf{P}$ bezeichneten neuen Punkte entsprechen Vertices aus dem nächsten Level, es gilt also $\mathbf{P} = \hat{S}\mathbf{V}$. Für die Limitpunktberechnung wird jetzt nur noch die Maske C_V aus Gleichung (2.44) benötigt (und nicht mehr C_E), z.B. berechnet sich $\mathbf{L}_3$ zu

$$\begin{aligned} \mathbf{L}_3 &= \tfrac{1}{36}\,\mathbf{P}_2 + \tfrac{1}{9}\,\mathbf{P}_7 + \tfrac{1}{36}\,\mathbf{P}_{14} \\ &+ \tfrac{1}{9}\,\mathbf{P}_3 + \tfrac{4}{9}\,\mathbf{P}_8 + \tfrac{1}{9}\,\mathbf{P}_{15} \\ &+ \tfrac{1}{36}\,\mathbf{P}_4 + \tfrac{1}{9}\,\mathbf{P}_9 + \tfrac{1}{36}\,\mathbf{P}_{16}. \end{aligned} \tag{2.67}$$

Allgemein formuliert führt dies für $m \in \{3,4,5,3\_4,4\_5,3_1,4_1,5_1\}$ zu

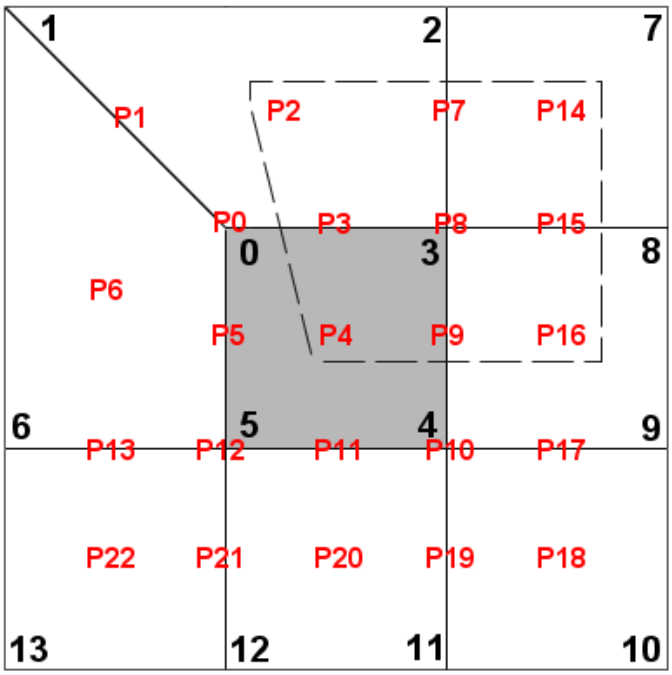

Abbildung 2.19: Limitpunktberechnung mit Hilfe der erweiterten Subdivision-Matrix $\hat{S}$: Die $2N+17 = 23$ Punkte $\mathbf{P}$ entstehen aus den $2N+8 = 14$ Vertices $\mathbf{V}$ (an den Stellen 0 bis 13 einzusetzen) durch den Subdivision-Schritt $\mathbf{P} = \hat{S}\mathbf{V}$ und können dann über Linearkombination mit der Maske C_V für die Berechnung der acht Limitpunkte $\mathbf{L}_3$, $\mathbf{L}_4$, $\mathbf{L}_5$, $\mathbf{L}_{3\_4}$, $\mathbf{L}_{4\_5}$, $\mathbf{L}_{3_1}$, $\mathbf{L}_{4_1}$ und $\mathbf{L}_{5_1}$ (s. Abb. 2.18) des grau hinterlegten Faces verwendet werden. Der gestrichelte Bereich verdeutlicht die für die in Gleichung (2.67) vorgestellte Berechnung von $\mathbf{L}_3$ benötigten Punkte.

$$\mathbf{L}_m = \sum_{k=0}^{2N+7} c_k^m \mathbf{V}_k \tag{2.68}$$

mit den Koeffizienten

$$c_k^m = \sum_{i,j=0}^{2} C_{V_{i,j}} \hat{S}_{I_{i,j}^m,k}. \tag{2.69}$$

Die Unterschiede zu den Gleichungen (2.63) und (2.64) liegen in der Verwendung der erweiterten Subdivision-Matrix $\hat{S}$ und in einem anderen Aufbau der Indexmatrizen I^m. Außerdem wird hier für die Berechnung aller (bis auf den zum extraordinary Vertex gehörenden) Limitpunkte ein Subdivision-Schritt durchgeführt und nicht nur für die Punkte 3_1, 4_1 und 5_1. Für den extraordinary Vertex wird weiterhin Gleichung (2.49) verwendet.

Gezeigt wurde bis jetzt, wie die Limitpunkte für ein inneres Face einer Oberfläche auf zwei verschiedene Arten bestimmt werden können. Enthält die Oberfläche eine Berandung, müssen für die Berechnung der Limitpunkte eines Randfaces angepasste Regeln verwendet werden. Gleiches gilt für Faces, die zwar im Inneren der Oberfläche liegen, aber ein oder zwei tagged Edges enthalten. Solche Sonderfälle werden im Rahmen der Beschreibung der Implementierung in Kapitel 3.5 erläutert.

2.5 Approximation gegebener Oberflächen mit Catmull-Clark-Gittern

Für die Approximation einer gegebenen Oberfläche mit einem Catmull-Clark-Gitter werden zunächst die Limitpunkte $\mathbf{L}_i$ dieses Gitters mit den zugehörigen Koeffizienten c_k aus Gleichung (2.69) benötigt. Fügt man die Koeffizienten c_k zu einem Vektor $\mathbf{c}_i$ zusammen und die zugehörigen Vertices zu $\mathbf{V}^i$, kann ein Limitpunkt über $\mathbf{L}_i = \mathbf{c}_i^T \mathbf{V}^i$ formuliert werden. Die Limitpunkte können dann auf die gegebene Oberfläche projiziert werden: $\mathbf{L}_i \rightarrow \mathbf{S}_i$, wobei die $\mathbf{S}_i$ für die so gewonnenen Punkte auf der Oberfläche stehen und im Folgenden **S**urfacepunkte genannt werden. Hiermit ergibt sich als Ziel

$$\mathbf{L}_i = \mathbf{c}_i^T \mathbf{V}^i \stackrel{!}{=} \mathbf{S}_i. \tag{2.70}$$

Stellt man genügend Surfacepunkte auf, erhält man ein dünnbesetztes lineares Gleichungssystem für Interpolation oder ein entsprechendes überbestimmtes System für Approximation. Hier soll das letztere verwendet werden. Die Approximationsaufgabe lässt sich dann formulieren als

$$\|A\mathbf{V} - \mathbf{S}\|_2 \rightarrow min. \tag{2.71}$$

Die Systemmatrix A wird aufgebaut aus den Koeffizienten c_k. Die Einträge des Vektors $\mathbf{V}$ sind die Vertices, analog dazu enthält der Vektor $\mathbf{S}$ die Surfacepunkte. Es handelt sich also um Vektoren, deren Elemente selbst dreidimensionale Vektoren sind. Dieses lineare Ausgleichsproblem kann über Lösung der Normalengleichungen $A^T A\mathbf{V} = A^T\mathbf{S}$ gelöst werden. Hierfür wird die CGLS-Methode verwendet (Verfahren der konjugierten Gradienten für lineare Ausgleichsprobleme, s.u.), die z.B. in [12] unter dem Namen CGNR (Conjugate Gradient Normal Residual) vorgestellt wird.

Die CGLS-Methode

Die Lösung eines linearen Ausgleichsproblems $\|A\mathbf{x} - \mathbf{b}\|_2 \rightarrow min$ mit einer Matrix $A \in \mathbb{R}^{m \times n}$ mit $m \geq n$ ist gegeben durch die Lösung des Normalengleichungssystems $A^T A\mathbf{x} = A^T\mathbf{b}$ mit der (falls $\text{rang}(A) = n$) symmetrisch positiv definiten Matrix $A^T A$. In den meisten Fällen ist $A^T A$ jedoch schlecht konditioniert und für dünnbesetztes A ist $A^T A$ auch nicht mehr dünnbesetzt. Das Verfahren der konjugierten Gradienten (CG-Methode) kann aber so angepasst werden, dass beide Probleme umgangen werden. Es werden zwei Residuenvektoren $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$ und $\mathbf{s}^{(k)} = A^T\mathbf{b} - A^T A\mathbf{x}^{(k)} = A^T\mathbf{r}^{(k)}$ verwendet. Die Konvergenzgeschwindigkeit wird wie bei der CG-Methode durch die Kondition $\kappa_2(A^T A)$ bestimmt. Der CGLS-Algorithmus lässt sich wie folgt formulieren:

Für $A \in \mathbb{R}^{m \times n}$ mit $\text{rang}(A) = n$ und beliebigen Startvektor $\mathbf{x}^{(0)}$:

$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
$\mathbf{s}^{(0)} = A^T\mathbf{r}^{(0)}$
$\mathbf{d}^{(0)} = \mathbf{s}^{(0)}$
für $k = 0, 1, 2, \ldots$
$\alpha_k = \|\mathbf{s}^{(k)}\|_2^2 \,/\, \|A\mathbf{d}^{(k)}\|_2^2$ // speichere $A\mathbf{d}^{(k)}$
$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}$
$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{d}^{(k)}$
$\mathbf{s}^{(k+1)} = A^T\mathbf{r}^{(k+1)}$
$\beta_k = \|\mathbf{s}^{(k+1)}\|_2^2 \,/\, \|\mathbf{s}^{(k)}\|_2^2$
$\mathbf{d}^{(k+1)} = \mathbf{s}^{(k+1)} + \beta_k\mathbf{d}^{(k)}$
bis Stopp

Gestoppt wird entweder, wenn eine maximale Anzahl K an Iterationen erreicht ist oder das Residuum unter eine vorgegebene Toleranz ϵ fällt, also $\|\mathbf{s}^{(k)}\|_2 \leq \epsilon$ erfüllt ist.

3 Implementierung

An dieser Stelle sei erneut auf das in Kapitel 4.2 angegebene Beispiel hingewiesen, das einen typischen Ablauf bei der Verwendung des in dieser Arbeit entwickelten Programms veranschaulicht und hilfreich für das Verständnis der in den folgenden Abschnitten beschriebenen Implementierungsdetails sein kann.

3.1 Ausgangspunkt

Die Oberfläche des zu Beginn der Weiterentwicklung bestehenden, mit C++ und OpenGL entwickelten Programms ist in Abbildung 3.1 bei geöffnetem Kontextmenü zu sehen. Die Faces eines Gitters sind orange, die Edges gelb oder in einer anderen Farbe, wenn sie tagged sind. Die Farbe hängt dann davon ab, für wie viele Subdivision-Schritte ein Edge tagged bleiben soll. Die Funktio-

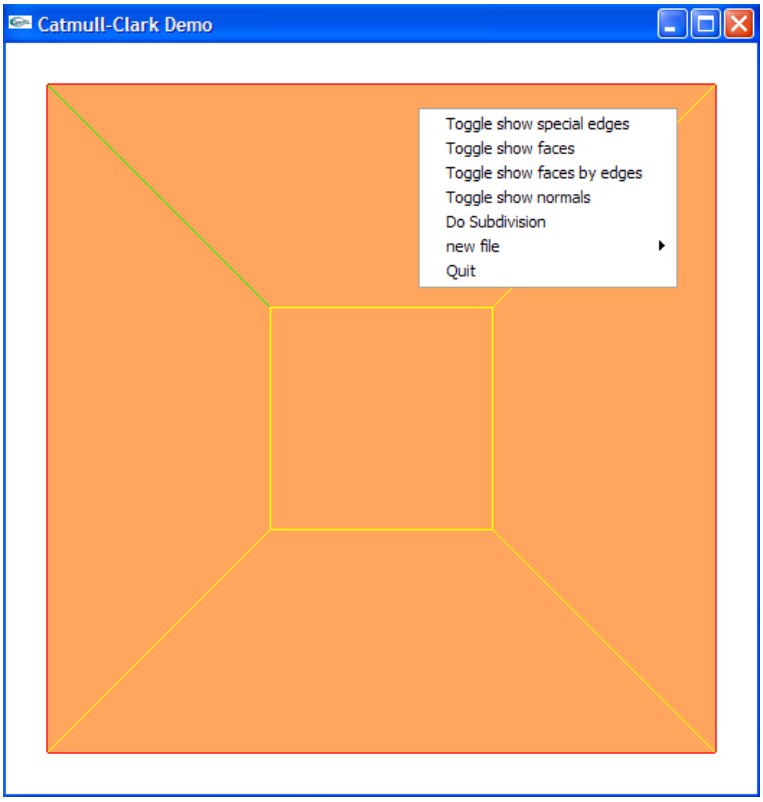

Abbildung 3.1: Oberfläche des C++-Programms zu Beginn der Weiterentwicklung

```
8   5   1

1   -0.6 -0.6  0.0    0
2   -0.6  0.6  0.0    0
3    0.6  0.6  0.0    0
4    0.6 -0.6  0.0    0
5   -0.2 -0.2  0.0    0
6   -0.2  0.2  0.0    0
7    0.2  0.2  0.0    0
8    0.2 -0.2  0.0    0

1    4     1 5 6 2    1
2    4     2 6 7 3    1
3    4     3 7 8 4    1
4    4     4 8 5 1    1
5    4     5 8 7 6    1

1    2 6   2
```

Abbildung 3.2: Vorgabe einer einzulesenden Geometrie: Die erste Zeile weist an, acht Vertices, fünf Faces und ein tagged Edge einzulesen. Es folgen drei Blöcke: 1. Vertices jeweils mit Vertexnummer, x-, y- und z-Koordinate und Markierung als festgelegter Vertex (hier immer 0, also sind alle Vertices veränderlich), 2. Faces (Facenummer, Anzahl der Vertices des Faces, Vertexnummern, aus denen ein Face zusammengesetzt ist und Blocknummer, die z.B. zum Einsatz verschiedener Farben verwendet werden kann), 3. tagged Edges (Edgenummer, Anfangs- und Endvertex eines tagged Edges und Anzahl der Subdivision-Schritte, für die das Edge tagged bleiben soll). Mit diesen Vorgaben erhält man das Gitter aus Abbildung 3.1.

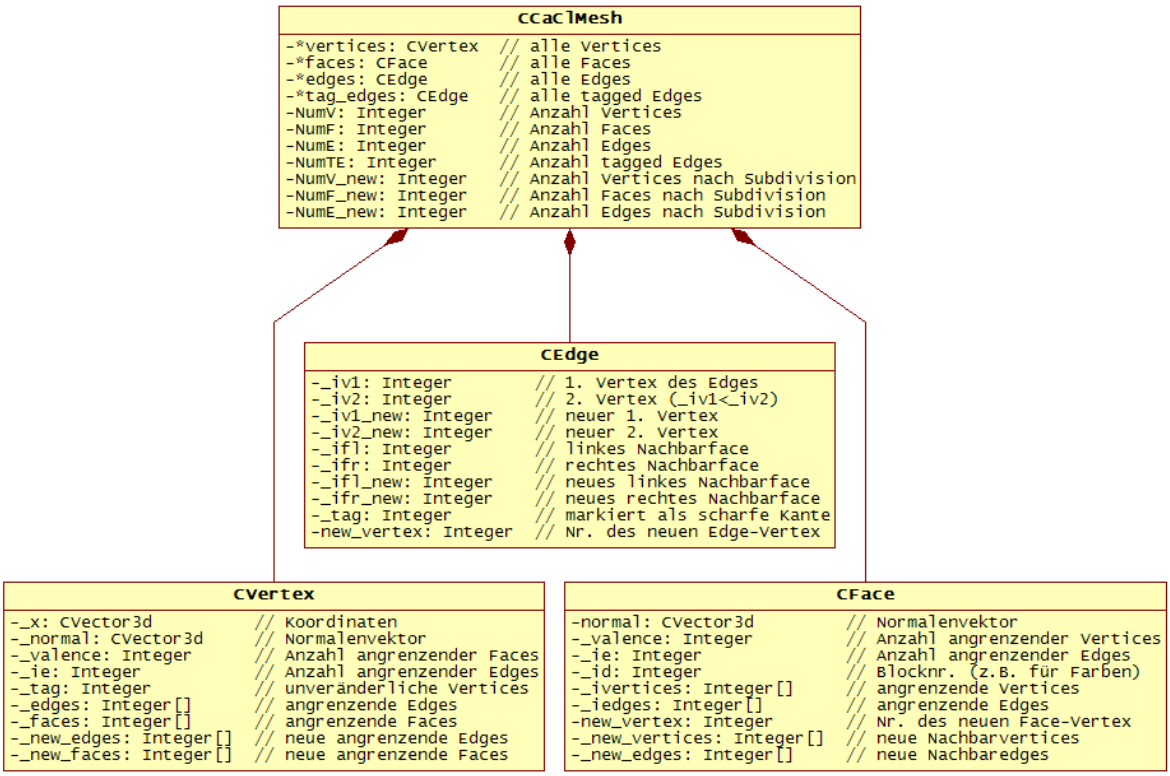

Abbildung 3.3: Klassendiagramm des zu Anfang der Weiterentwicklung verwendeten Datenmodells: Ein Mesh-Objekt des Typs `CCaClMesh` enthält Vertices, Edges und Faces aus den Klassen `CVertex`, `CEdge` und `CFace`. Die Funktion der einzelnen Variablen aus den vier Klassen kann jeweils hinter den Kommentarzeichen nachgelesen werden.

nalität der gezeigten Anfangsversion des Programms beschränkt sich auf die Durchführung von Subdivision-Schritten nach der Catmull-Clark-Methode (mit Erweiterung der Regeln für tagged Edges) für Gittergeometrien, die aus Textdateien wie in Abbildung 3.2 eingelesen werden. Beim Einlesen erhalten die Vertices und Faces ihre Nummerierung nach den Angaben in der Textdatei, während die Edges und ihre Nummerierung automatisch erstellt und dabei mit den tagged Edges begonnen wird. Danach werden alle Faces durchlaufen und noch nicht nummerierte Edges innerhalb der Faces erhalten ihre fortlaufende Nummer.

Das verwendete Datenmodell ist im UML-Klassendiagramm in Abbildung 3.3 dargestellt. Ein Mesh-Objekt der Klasse `CCaClMesh` enthält Vertices, Edges und Faces aus den drei Klassen `CVertex`, `CEdge` und `CFace`. Während der Ausführung des Programms existiert immer nur ein solches Mesh, dessen Variablen bei einer Subdivision überschrieben werden. Für Koordinaten und Normalenvektoren werden Objekte aus der Klasse `CVector3d` angelegt. Diese Klasse enthält Operationen, die den Umgang mit Vektoren ermöglichen. Dabei handelt es sich vor allem um Operatorüberladungen, um z.B. die Addition oder das Kreuzprodukt zweier Vektoren auf einfache Weise verwenden zu können.

Bei Durchführung eines Subdivision-Schrittes muss eine Neunummerierung der Vertices, Edges und Faces durchgeführt werden. Für ein Viereckface mit der Bezeichnung nf, das in vier neue Faces unterteilt wird, ist dies in Abbildung 3.4 dargestellt. Die vier alten Vertices $iv_0, \ldots, iv_3$, die für die Einträge des Arrays `_ivertices` aus der Klasse `CFace` stehen sollen, behalten ihre Nummern. Ähnliches gilt für die vier äußeren Edges $ie_0, \ldots, ie_3$ (Array `_iedges` aus `CFace`), die in zwei Teile getrennt werden und bei denen der erste Teil die alte Nummer behält. Auch eines

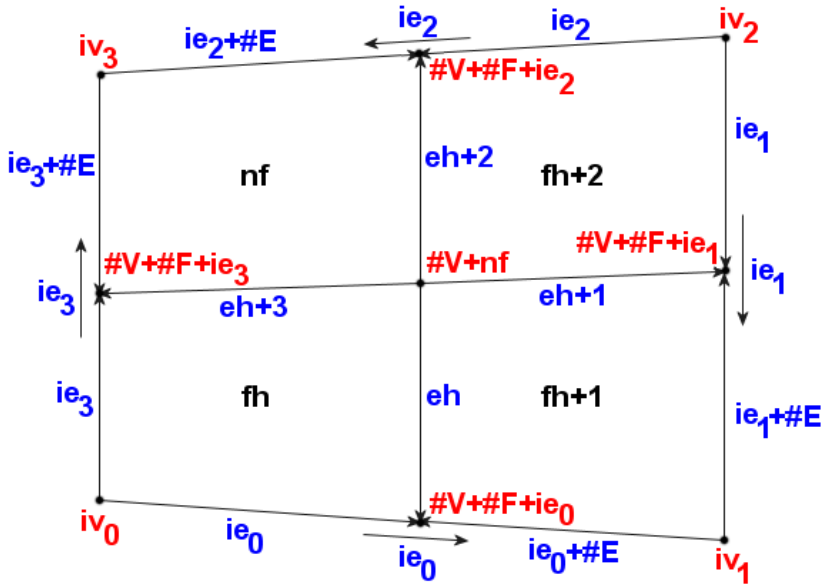

Abbildung 3.4: Neunummerierung der Vertices (rot), Edges (blau) und Faces (schwarz) bei Durchführung eines Subdivision-Schrittes, dargestellt für die vier Faces, die aus der Unterteilung des Gesamtvierecks mit der ursprünglichen Bezeichnung nf hervorgegangen sind.

der neuen vier Faces bekommt die alte Facenummer nf zugeteilt. Alle anderen, neu angelegten Bestandteile erhalten über die Parameter

$$eh = 2\#E + \sum_{i=0}^{nf} N_F(i) \tag{3.1}$$

und

$$fh = \#F - nf + \sum_{i=0}^{nf} N_F(i), \tag{3.2}$$

die in Abbildung 3.4 eingetragen sind, neue Nummern, wobei $N_F(i)$ die Valenz des Faces i (= Anzahl der Vertices des Faces) bezeichnet und außerdem $N_F(0) = 0$ gesetzt wird.

3.2 Aktuelle Version des Programms

Bereits an dieser Stelle soll die aktuelle Version des weiterentwickelten Programms gezeigt werden, um einen Überblick über die in den folgenden Kapiteln beschriebenen neu implementierten Funktionalitäten zu geben. Abbildung 3.5 zeigt die Oberfläche des Programms mit dem gleichen Gitter wie in Abbildung 3.1 bei geöffnetem Kontextmenü. Im Menü sind in Klammern die

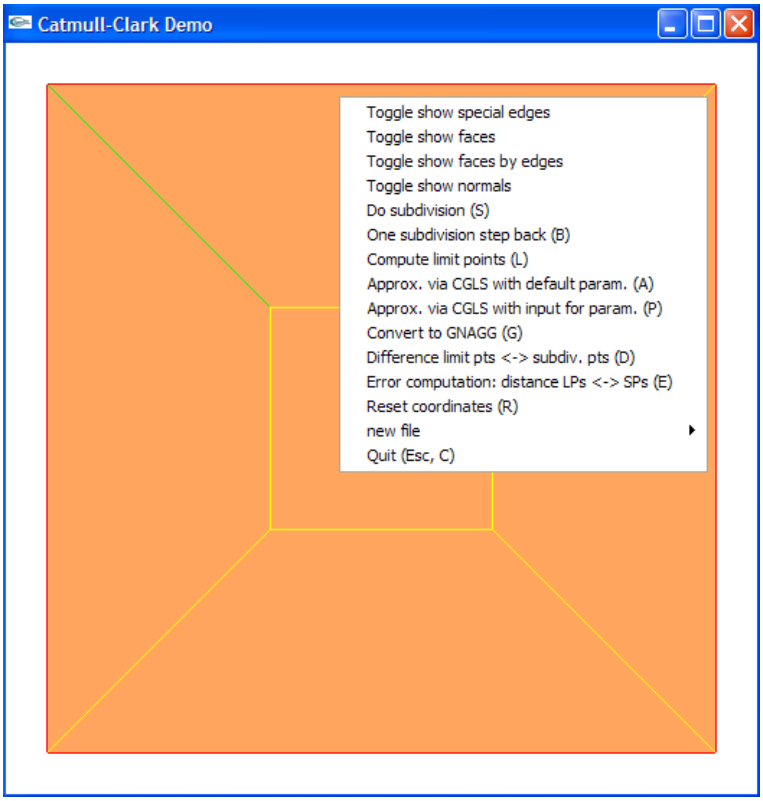

Abbildung 3.5: Oberfläche der finalen Version des C++-Programms

Hotkeys für den Aufruf der jeweiligen Funktionen angegeben. Die folgende Übersicht zeigt, in welchen Kapiteln die Implementierungsdetails der neuen Menüpunkte erläutert werden.

- One subdivision step back (B): Diese Funktion wird durch ein neues Datenmodell ermöglicht, das in Kapitel 3.3 beschrieben wird.
- Compute limit points (L): Voraussetzung für die Limitpunktberechnung sind die in Kapitel 3.4 erläuterten Orientierungsanpassungen. Die Implementierungsdetails zur Limitpunktberechnung selbst werden dann in Kapitel 3.5 aufgezeigt.
- Approx. via CGLS with default param. (A): Kapitel 3.6
- Approx. via CGLS with input for param. (P): ebenfalls Kapitel 3.6
- Convert to GNAGG (G): Kapitel 3.7
- Difference limit pts <-> subdiv. pts (D): Kapitel 3.8.1
- Error computation: distance LPs <-> SPs (E): Kapitel 3.8.2
- Reset coordinates (R): Kapitel 3.8.3

3.3 Konvertierung bestehender Algorithmen auf ein neues Datenmodell

Mit dem in Kapitel 3.1 beschriebenen Datenmodell existiert immer nur ein Mesh. Will man nach Durchführung von Subdivision-Schritten aber z.B. für Demonstrationszwecke wieder eine oder mehrere Stufen auf ein gröberes Level zurück gehen, ist es sinnvoll, die Meshes in einem Array oder einer ähnlichen Datenstruktur aus der Standard Template Library von C++ zu speichern. Ein Problem hierbei ergibt sich daraus, dass man zum Programmstart nicht unbedingt weiß, wie viele Unterteilungen man durchführen wird. Man könnte dem Array eine Maximalgröße von z.B. sieben Meshes geben (d.h., es sind sechs Subdivision-Schritte möglich), doch mit dieser Methode belegt das Programm direkt nach seinem Start ca. 250 MB im Speicher. Zudem benötigt es einige Sekunden für den Start, wenn die Mesh-Objekte angelegt werden, und auch für die Beendigung, wenn sie zerstört werden. Dieser hohe Speicherbedarf folgt aus dem weiteren Nachteil, dass zum Programmstart nicht bekannt ist, wie viele Vertices, Edges und Faces jeweils auf einem Subdivision-Level vorhanden sind, für diese aber auch für alle sieben möglichen Meshes der Speicher, ebenfalls durch die Vorgabe von Maximalwerten, reserviert werden muss. Um diese Probleme zu umgehen, kann man die doppelt verkettete Liste aus der Standard Template Library verwenden. Wird durch eine Subdivision ein neues Mesh-Objekt benötigt, kann dieses neu erstellt und hinten an die Liste angehängt werden. Geht man auf ein gröberes Level zurück, wird das Objekt des feineren Levels sozusagen ausgehängt (d.h., die beiden Zeiger auf das Element davor

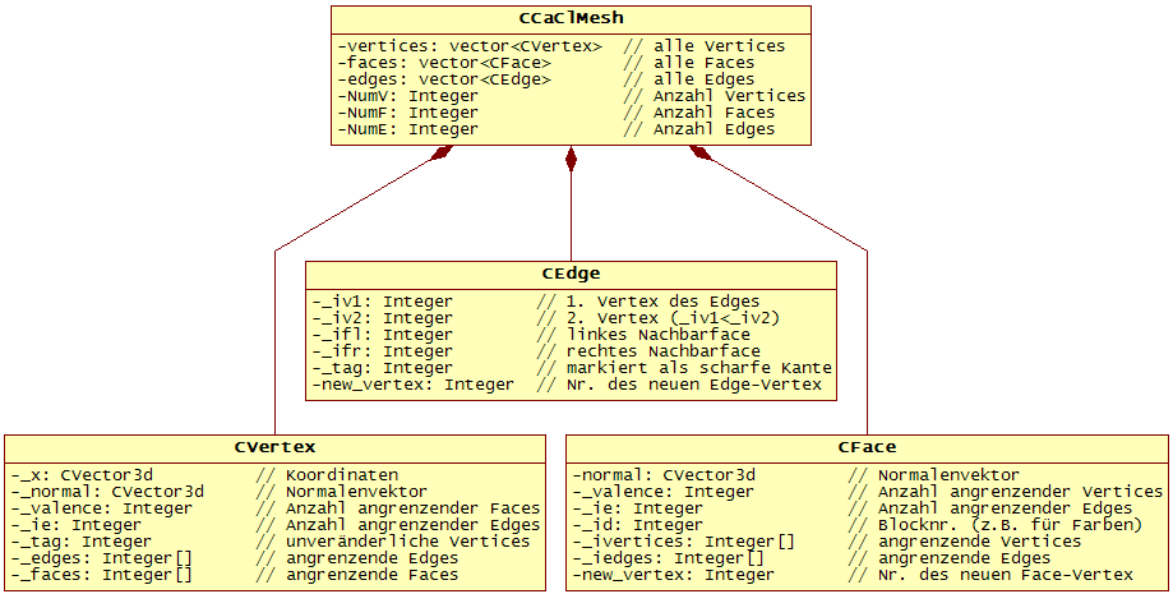

Abbildung 3.6: Klassendiagramm des neuen Datenmodells: Vertices, Edges und Faces werden jetzt als Vektoren aus der Standard Template Library angelegt, Mesh-Objekte vom Typ `CCaClMesh` in einer Liste gespeichert.

und das dahinter, in diesem Fall das Ende der Liste, werden entfernt). Da die Meshes auf diese Weise zur Laufzeit angelegt werden und man so die Anzahl der Vertices, Edges und Faces, die nach einer Subdivision für das dann neu entstehende Mesh benötigt werden, vor Ausführung der Subdivision kennt (z.B. für ein reines Viereckgitter nach Gl. (2.26)), kann man Vertices, Edges und Faces ebenfalls zur Laufzeit mit der jeweils benötigten Anzahl als Vektoren (wie die Liste aus der Standard Template Library) anlegen. Die Vektoren haben gegenüber einer Liste den Vorteil des direkten Zugriffs auf Elemente über den Indexoperator [ ] (wie beim Array), ihre Größe muss im Gegensatz zur Liste aber festgelegt werden.

Das neue Datenmodell, das sich aus den genannten Änderungen ergibt, ist im Klassendiagramm in Abbildung 3.6 zu sehen. Auffällig ist, dass die einzelnen Klassen weniger Variablen enthalten. Da jetzt mehrere Mesh-Objekte gleichzeitig zur Verfügung stehen, können bei Ausführung eines Subdivision-Schrittes neu berechnete Werte für Variablen, die auf dem gröberen Level aber noch mit altem Wert benötigt werden, direkt in einem neuen Mesh gespeichert werden. Vorher wurden hierfür die „new“-Variablen (z.B. `_new_edges` und `_new_faces` in der Klasse `CVertex`, s. Abb. 3.3) als Zwischenspeicherung vor dem Überschreiben des Meshes benötigt.

Die Anpassung der bestehenden Algorithmen an das neue Datenmodell erforderte vor allem die Implementierung von Kopierkonstruktoren und der Überladung der Zuweisungsoperation für die Klassen `CCaClMesh`, `CVertex`, `CEdge` und `CFace`. Die Funktion `push_back(·)` beispielsweise kopiert das in Klammern angegebene Objekt mit Hilfe des Kopierkonstruktors und fügt es am Ende der Liste bzw. des Vektors ein. Zusätzlich ergaben sich viele Detailänderungen am Code, die aber dadurch erleichtert wurden, dass auf die neu eingesetzten Vektoren über den Indexoperator

```
    1   -0.6 -0.6  0.0    0
    2   -0.6  0.6  0.0    0
    3    0.6  0.6  0.0    0
    4    0.6 -0.6  0.0    0
    5   -0.2 -0.2  0.0    0
    6   -0.2  0.2  0.0    0
    7    0.2  0.2  0.0    0
    8    0.2 -0.2  0.0    0
0
    1    4      1 5 6 2    1
    2    4      2 6 7 3    1
    3    4      3 7 8 4    1
    4    4      4 8 5 1    1
    5    4      5 8 7 6    1
0
    1    2 6    2
0
```

Abbildung 3.7: Vorgabe der Geometrie wie in Abbildung 3.2: Eine Zeile für die Angabe der Anzahl der einzulesenden Vertices, Faces und tagged Edges wird nicht mehr benötigt. Die Blöcke für die Vertexkoordinaten, den Aufbau der Faces und die Definition der tagged Edges werden durch die 0 voneinander getrennt.

genauso zugegriffen werden kann wie vorher auf die Arrays.

Eine größere Änderung betraf das Einlesen der Geometrie, das in den Konstruktor der Klasse `CCaClMesh` verlagert wurde. Ein Mesh-Objekt wird darin nach Aufruf von z.B. `CCaClMesh new_mesh(„Geometriedatei.txt“)` mit der richtigen Anzahl an Vertices, Edges und Faces angelegt und danach hinten an die Mesh-Liste angehängt. Wie weiter oben erwähnt, führt diese Vorgehensweise zu einer drastischen Reduzierung des Speicherbedarfs, so dass beim Programmstart jetzt nur noch ca. 8 MB belegt werden. Die der Geometriedatei aus Abbildung 3.2 entsprechende Datei sieht jetzt so aus wie in Abbildung 3.7. Wie viele Vertices, Faces und tagged Edges einzulesen sind, muss nicht mehr angegeben werden. Solange keine 0 als Trennzeichen auftaucht, werden die drei Variablen jeweils gefüllt. Hierfür werden im Mesh-Konstruktor Listen verwendet, da wie angesprochen für Listen die Größe nicht vorher festgelegt werden muss. Im Anschluss, wenn die jeweilige Listengröße bekannt ist, werden die Vektoren `vertices`, `faces` und `edges` mit dieser Größe angelegt und die Elemente aus den Listen können in die Vektoren kopiert werden.

Der Code, der bei Auswahl der Subdivision aus dem Kontextmenü des Programms ausgeführt wird, steht in der main-Funktion und sieht mit dem neuen Datenmodell folgendermaßen aus:

```
if (ccMeshes.size() == 0) ccMeshes.push_back(ccMesh0);
ccMeshes.push_back(ccMeshes.back());
ccMeshes.back().Subdivision();
ccMeshes.back().GenerateGLLists();
```

Wenn in der Mesh-Liste `ccMeshes` noch keine Objekte vorhanden sind, wird das beim Programmstart erzeugte Mesh `ccMesh0` eingefügt. Das bisher letzte Mesh in der Liste wird über `push_back(·)` kopiert und an das Ende der Liste angefügt. Die letzten beiden in der Liste gespeicherten Meshes sind nun also identisch. Für das gerade hinten angefügte Mesh kann jetzt ein Subdivision-Schritt durchgeführt werden. Die Funktion `GenerateGLLists()` ist schließ-

lich für die grafische Darstellung des neuen unterteilten Meshes über OpenGL zuständig. Als neue Funktion im Kontextmenü kann jetzt auch das Zurücknehmen eines Unterteilungsschrittes ausgewählt werden, man erhält wieder ein gröberes Gitter. Dabei wird das jeweils letzte Mesh in der Liste über `pop_back()` gelöscht:

```
if (ccMeshes.size() > 1) {
     ccMeshes.pop_back();
     ccMeshes.back().GenerateGLLists();
}
```

3.4 Anpassung der Orientierung von Vertices, Edges und Faces

Wie in den Klassendiagrammen (Abb. 3.3 und 3.6) zu erkennen ist, enthalten die Vertices die Nummern ihrer angrenzenden Edges und Faces in den Arrays `_edges` bzw. `_faces`. Ähnliches gilt für die Faces (`_ivertices` bzw. `_iedges`). Im regulären Fall (nur Vierecke und Valenz $N = 4$) weisen diese vier Arrays jeweils vier Einträge auf. Das Ziel sollte sein, diese Einträge in einer linksdrehenden Art und Weise, d.h. gegen den Uhrzeigersinn, zu ordnen, was in Abbildung 3.8 grafisch veranschaulicht ist. Der Vorteil dieser einheitlichen Sortierung wird verdeutlicht, wenn man sich z.B. Abbildung 2.17 ansieht. Die richtige Reihenfolge der Nummerierung von Nachbarvertices bei der Limitpunktberechnung kann nur gewährleistet sein, wenn alle Vertices und Faces für ihre Arrays die angesprochene einheitliche Orientierung erhalten, da die Limitpunktberechnung faceweise ausgeführt wird und dafür die benötigte Nachbarschaft immer auf die gleiche Weise zu durchsuchen sein muss.

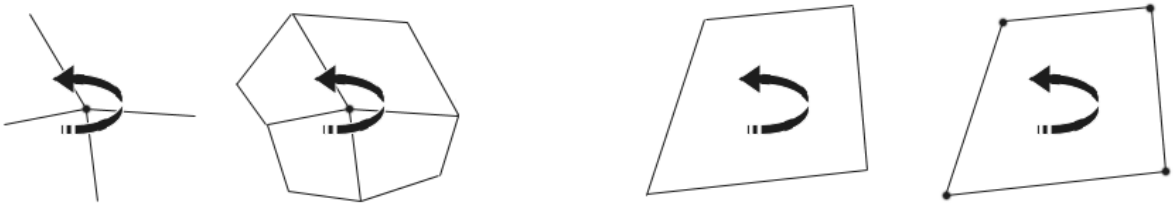

Abbildung 3.8: Links: gewünschte Orientierung der angrenzenden Edges und Faces bei den Vertices, rechts: entsprechende Orientierung der angrenzenden Edges und Vertices bei den Faces

Für die Faces (rechts in Abb. 3.8) wird die Orientierung der angrenzenden Edges und Vertices direkt durch das Einlesen der Geometrie vorgegeben. Die Geometrie aus Abbildung 3.7 beispielsweise ergibt eine linksdrehende Sortierung bei allen Faces. Würde man aber z.B. in der ersten Zeile des Faceaufbaus die Vorgabe 1 5 6 2 durch 1 2 6 5 ersetzen, wäre dieses Face rechtsdrehend, was in diesem Fall zu einem Normalenvektor führt, der im Vergleich zu den Normalenvektoren der anderen Faces in die entgegengesetzte Richtung zeigt. Dies würde zu Fehlern bei der Subdivision (und auch bei der später beschriebenen Limitpunktberechnung) führen. Um dem Nutzer die Vorgabe der Geometrie zu erleichtern, wurde daher ein Algorithmus

implementiert, der alle Faces in der Orientierung aufbaut, wie sie beim ersten Face vorgegeben wird. Dies wurde realisiert, indem zunächst die Nachbarfaces des ersten Faces gesucht werden und dann ihre Orientierung überprüft und gegebenenfalls angepasst wird. Danach werden wiederum die Nachbarfaces der gerade kontrollierten Faces untersucht usw. Die richtige Sortierung der jeweils an die Faces angrenzenden Edges folgt dann automatisch, da sich die Reihenfolge der Nummerierung beim Erstellen der Edges nach der Orientierung der Vertices richtet.

Für die Vertices (links in Abb. 3.8) werden zunächst die angrenzenden Faces sortiert. Jedes Edge besitzt zwei Nachbarfaces. Handelt es sich um ein Randedge, wird die außerhalb des Gitters liegende Nachbarschaft mit der Nummer -1 bezeichnet. Sämtliche Nachbarfacenummern der an einen Vertex angrenzenden Edges werden in einem Array gespeichert. Hierbei muss die „Richtung“ der Edges beachtet werden. Bezeichnet man `_iv1` als Anfangs- und `_iv2` als Endpunkt eines Edges (s. auch Klassendiagramme in den Abb. 3.3 und 3.6), wird zuerst das rechte und dann das linke Nachbarface zum Array hinzugefügt, wenn der untersuchte Vertex `_iv1` des jeweiligen Edges entspricht. Entspricht er dagegen `_iv2`, wird zuerst das linke und danach das rechte Nachbarface im Array gespeichert. So erhält man bei Valenz $N = 4$ beispielsweise ein Array mit acht Einträgen, wobei jeder Eintrag doppelt vorhanden ist. Das Array wird dann so sortiert, dass der zweite Eintrag dem dritten entspricht, der vierte dem fünften usw. Anschließend wird jeder zweite Eintrag gelöscht. Man erhält damit die gewünschte linksdrehende Sortierung für die angrenzenden Faces des gerade betrachteten Vertex. Die Sortierung der an den Vertex angrenzenden Edges fällt dann mit Hilfe des Arrays leicht. Das erste Edge muss als Nachbarfaces den ersten und zweiten Eintrag des Arrays haben, das zweite Edge den zweiten und dritten Eintrag usw. Sind diese Sortierungen für einen Vertex abgeschlossen, kann mit dem nächsten auf die gleiche Weise verfahren werden, bis schließlich alle Vertices ihre linksdrehende Nachbarschaftsorientierung erhalten haben.

3.5 Berechnung von Limitpunkten

Voraussetzung für die Limitpunktberechnung ist, wie in Kapitel 2.4.3 beschrieben, ein Gitter, das ausschließlich Vierecke enthält und dessen extraordinary Vertices isoliert sind, d.h., jedes Face darf maximal einen extraordinary Vertex enthalten. Um die Erfüllung dieser Bedingungen zu überprüfen, wurde der Klasse `CCaClMesh` eine private Integer-Variable `level` hinzugefügt mit folgender Bedeutung:

- `level = -2`: Das Gitter enthält nicht nur Vierecke *und* mindestens ein Face weist mehr als einen extraordinary Vertex auf.
- `level = -1`: Das Gitter enthält nicht nur Vierecke *oder* mindestens ein Face weist mehr als einen extraordinary Vertex auf.
- `level` $\geq$ `0`: Das Gitter enthält ausschließlich Vierecke und jedes Face weist höchstens einen extraordinary Vertex auf. Level 0 wird spätestens nach zwei Subdivision-Schritten

erreicht.

Solange noch nicht Level 0 erreicht ist, wird der Benutzer vom Programm aufgefordert, einen Subdivision-Schritt durchzuführen. Erst dann lässt sich die Funktion zur Berechnung der Limitpunkte aufrufen. Dies geschieht wie bei der Subdivision aus der main-Funktion heraus. Der Code dabei lautet:

```
if (ccMeshes.size() == 0) ccMeshes.push_back(ccMesh0);
if (ccMeshes.back().GetLevel() >= 0)
    ccMeshes.back().LimitPoints();
```

In der Funktion `LimitPoints()` wird die Limitpunktberechnung faceweise durchgeführt, d.h., für jedes Face werden bis zu neun Limitpunkte ermittelt. Gespeichert werden sämtliche $\#V + \#E + \#F$ Limitpunkte eines Gitters in einer weiteren privaten Variable der Klasse `CCaClMesh`. Sie heißt `limit_pts` und ist vom Typ `vector<CVertex>`. Die Nummerierung der Limitpunkte entspricht der Nummerierung der Vertices, die sich bei Durchführung eines Subdivision-Schrittes ergeben würde (s. Abb. 3.4). Damit Limitpunkte nicht mehrfach berechnet werden, wurde in der Klasse `CVertex` eine private `bool`-Variable `lp_set` hinzugefügt, die für jeden Limitpunkt mit `false` initialisiert wird. Nach der Berechnung eines Limitpunkts erhält die Variable für diesen Punkt den Wert `true`. Bevor mit der Berechnung begonnen werden kann, muss festgelegt sein, um was für ein Face es sich handelt. Die verschiedenen Typen werden in folgender Reihenfolge abgearbeitet (1. bis 4.), hinzu kommt die Berechnung von Limitpunkten auf festgelegten Kurven (5. bis 6.):

1. innere Faces ohne tagged Edges und ohne Randvertex (s. Kap. 3.5.1)
2. innere Faces ohne tagged Edges, aber mit einem Randvertex (s. Kap. 3.5.2)
3. Randfaces (s. Kap. 3.5.3 und 3.5.4)
4. innere Faces mit einem oder zwei tagged Edges (s. Kap. 3.5.5)
5. Randkurve (s. Kap. 3.5.6)
6. innere tagged Edges (s. Kap. 3.5.7)

An dieser Stelle kann man schon erkennen, dass ein Face nicht mehr als zwei tagged Edges aufweisen darf. Zudem dürfen sich diese beiden nicht gegenüber liegen. In beiden Fällen müsste zunächst ein Subdivision-Schritt durchgeführt werden, wozu der Benutzer vom Programm auch aufgefordert würde.

3.5.1 Berechnung für innere Faces ohne tagged Edges und ohne Randvertex

Wie die neun Limitpunkte eines inneren Faces ohne tagged Edges berechnet werden, wurde im Theorieteil in Kapitel 2.4.3 für ein Face mit einem extraordinary Vertex der Valenz $N = 3$ hergeleitet. Abbildung 3.9 zeigt die analoge Situation für ein Face mit einem extraordinary Vertex der

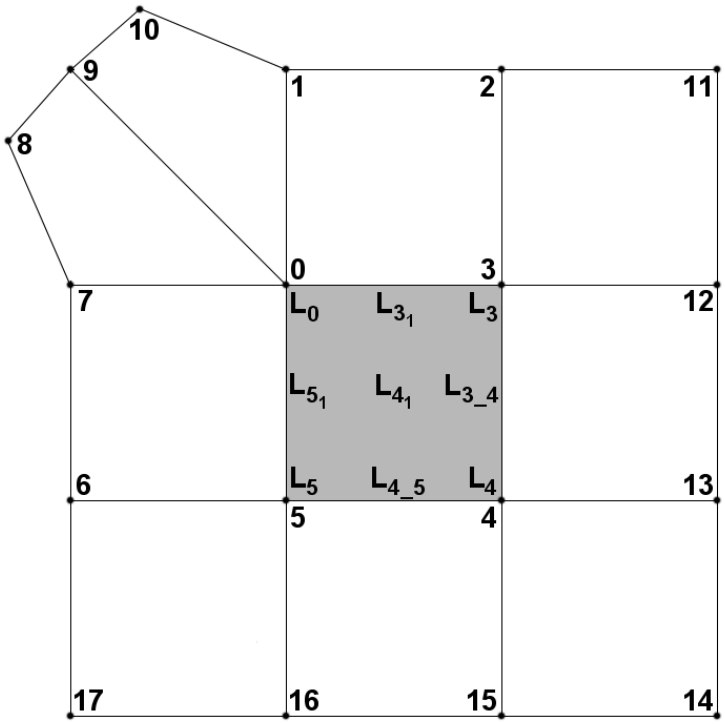

Abbildung 3.9: Limitpunktberechnung bei einem inneren Face ohne tagged Edges: Es sind $2N + 8$ Vertices beteiligt (in diesem Beispiel mit $N = 5$ sind es 18), deren Nummerierung mit dem extraordinary Vertex beginnt. Alle Koeffizienten für die neun Limitpunkte können über die Subdivision-Matrix $\hat{S}$ wie in Gleichung (2.69) ermittelt werden.

Valenz $N = 5$. Eingetragen sind die neun zu berechnenden Limitpunkte $\mathbf{L}_0$, $\mathbf{L}_{3_1}$, $\mathbf{L}_3$, $\mathbf{L}_{5_1}$, $\mathbf{L}_{4_1}$, $\mathbf{L}_{3\_4}$, $\mathbf{L}_5$, $\mathbf{L}_{4\_5}$ und $\mathbf{L}_4$. Hierfür wurde die zweite Vorgehensweise nach den Gleichungen (2.68) und (2.69) bzw. Abbildung 2.19 implementiert. Zunächst mussten für die vier möglichen Valenzen $N \in \{3, 4, 5, 6\}$ jeweils die Subdivision-Matrizen $\hat{S}$ aufgestellt werden, was mit Hilfe von Maple erledigt wurde. Anschließend konnte aus den Matrizen über die Konvertierungsfunktion von Maple C-Code erstellt werden. Gespeichert wurden die Werte der Matrizen in einem dreidimensionalen Array der Größe [4][29][20] (vier mögliche Valenzen, maximal 29 Zeilen und 20 Spalten in $\hat{S}$). Des Weiteren wurde ein ebenfalls dreidimensionales Koeffizientenarray der Größe [9][4][20] angelegt (neun Limitpunkte pro Face, vier mögliche Valenzen, $2N + 8$ an der Berechnung eines Limitpunkts beteiligte Vertices, also mit $N = 6$ maximal 20). In diesem Koeffizientenarray wurden dann die Ergebnisse für c_k^m entsprechend Gleichung (2.69) gespeichert. Beispielhaft sei hier der Code für die Berechnung der Koeffizienten c_k^3 gezeigt (entspricht im Code `c[1][N-3][j]`), die mit Valenz $N = 3$ zur Berechnung des Limitpunkts $\mathbf{L}_3$ aus Gleichung (2.67) führen, wobei $\hat{S}$ im Code `A` heißt:

```
for (int j=0; j<2*N+8; j++) {
    c[1][N-3][j] = 1./36. * A[N-3][2][j]
                 + 1./9. * A[N-3][7+(N-3)*2][j]
```

```
                    + 1./36. * A[N-3][14+(N-3)*2][j]
                    + 1./9. * A[N-3][3][j]
                    + 4./9. * A[N-3][8+(N-3)*2][j]
                    + 1./9. * A[N-3][15+(N-3)*2][j]
                    + 1./36. * A[N-3][4][j]
                    + 1./9. * A[N-3][9+(N-3)*2][j]
                    + 1./36. * A[N-3][16+(N-3)*2][j];
}
```

Die Matrizen $\hat{S}$ und die Berechnungen der Koeffizienten c_k^m wurden in eine eigene Header-Datei ausgelagert. In der Funktion `LimitPoints()` wird dann zur Limitpunktberechnung auf die Koeffizienten zugegriffen. Folgender Code wurde implementiert:

```
for (int i=0; i<9; i++) {
    xl.SetVals(0., 0., 0.);
    if (limit_pts[position[i]].Get_lp_set() == false) {
        for (int j=0; j<2*extraord_val+8; j++)
            xl += c[i][extraord_val-3][j] * vertices[order[j]].Vec();
        limit_pts[position[i]].SetVals(xl);
        limit_pts[position[i]].Set_lp_set(true);
    }
}
```

Hierbei ist `xl` eine Variable vom Typ `CVector3d` für die Koordinaten des jeweils zu berechnenden Limitpunkts und `extraord_val` entspricht dem `N` von oben. Die Berechnung wird nur durchgeführt, wenn die oben erwähnte `bool`-Variable `lp_set` für den gerade betrachteten Limitpunkt noch nicht auf `true` gesetzt wurde. Das Array `order` enthält die Nummern der an der Berechnung beteiligten $2N + 8$ Vertices in der Reihenfolge, wie sie in Abbildung 3.9 dargestellt ist. Hierfür müssen die Nachbarschaftsverhältnisse des momentan bearbeiteten Faces abgesucht werden, wobei immer beim extraordinary Vertex begonnen wird, falls es einen gibt. Handelt es sich um ein reguläres Face (alle vier Vertices haben Valenz $N = 4$), ist es egal, von welchem Vertex aus die Nummerierung gestartet wird. In jedem Fall ist die linksdrehende Orientierung (s. Kap. 3.4) wichtig, um die richtigen Vertices zu finden und das Array `order` in der korrekten Reihenfolge zu füllen. Das Array `position` schließlich besteht für jedes Face aus den Nummern der zu berechnenden Limitpunkte, die zwischen 0 und $\#V + \#E + \#F - 1$ liegen. `order` und `position` werden auch für die Facetypen, deren Limitpunktberechnung in den folgenden Abschnitten erläutert wird, verwendet. Die Berechnung unterscheidet sich dann nur durch die Anzahl der beteiligten Vertices (d.h. `order` kann eine kleinere Größe aufweisen als hier) und die Koeffizienten (d.h. `c[i][extraord_val-3][j]` wird durch ein anderes Array ersetzt).

3.5.2 Berechnung für innere Faces ohne tagged Edges mit einem Randvertex

Betrachtet wird jetzt die Situation aus Abbildung 3.10. Es geht um die Limitpunktberechnung für das in der Abbildung grau hinterlegte Face, das zwar keine tagged Edges enthält, aber einen Vertex, der zur Randkurve gehört. Die Nummerierung wird immer so gewählt, dass der Randvertex an der Stelle 5 liegt. Statt der $2N + 8$ beim regulären Face an der Limitpunktberechnung beteiligten Vertices treten in diesem Fall nur noch $2N + 7$ Vertices auf. Um nicht zu viele Fallunterscheidungen implementieren zu müssen, dürfen die Vertices an den Stellen 0, 3 und 4 ausschließlich die Valenz $N = 4$ aufweisen (damit ergibt sich immer $2N + 7 = 15$). Ist dies nicht der Fall, muss vor der Berechnung der Limitpunkte ein Subdivision-Schritt durchgeführt werden, wozu der Benutzer des Programms aufgefordert wird. Der auf dem Rand liegende Limitpunkt $\mathbf{L}_5$ bekommt seine Koordinaten bei der Abarbeitung der Randkurve (s. Kap. 3.5.6). Im Gegensatz zu den Koeffizienten für die fünf Limitpunkte $\mathbf{L}_0$, $\mathbf{L}_{3_1}$, $\mathbf{L}_3$, $\mathbf{L}_{3\_4}$ und $\mathbf{L}_4$ können die Koeffizienten für die drei rot gekennzeichneten Limitpunkte $\mathbf{L}_{5_1}$, $\mathbf{L}_{4_1}$ und $\mathbf{L}_{4\_5}$ nicht mit Hilfe von $\hat{S}$ bestimmt werden, da sie von denen für ein reguläres Face verschieden sind. Stattdessen müssen sie explizit

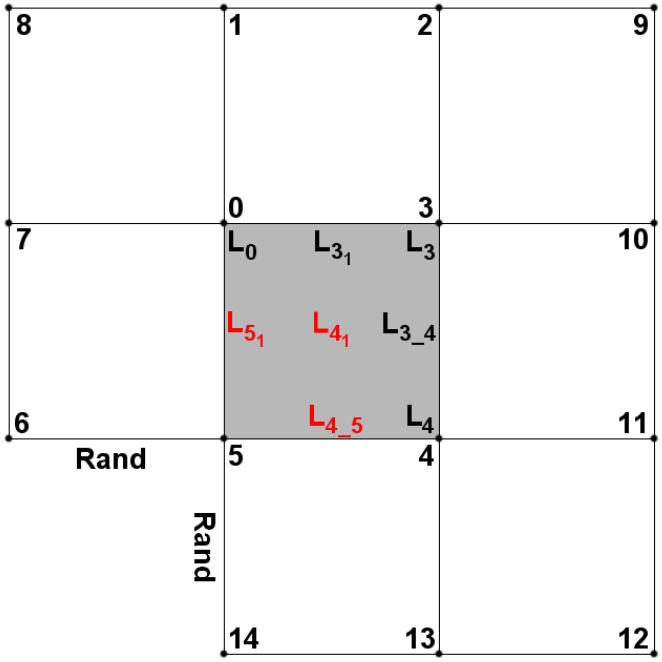

Abbildung 3.10: Limitpunktberechnung bei einem inneren Face ohne tagged Edges, aber mit einem Randvertex: Es sind $2N + 7$ ($= 15$, da nur $N = 4$ erlaubt ist) Vertices beteiligt. Die Berechnung der rot markierten Limitpunkte weicht von der Berechnung dieser Punkte in einem regulären Face ab. Der Limitpunkt zum Vertex an der Stelle 5 wird erst später bei der Bestimmung von Limitpunkten auf der Randkurve ermittelt.

aufgestellt werden, was für den Facepunkt $\mathbf{L}_{4_1}$ jetzt vorgeführt wird:

$$
\begin{aligned}
\mathbf{L}_{4_1} &= \tfrac{1}{36}\ (\tfrac{1}{64}p8 + \tfrac{3}{32}p1 + \tfrac{1}{64}p2 + \tfrac{3}{32}p7 + \tfrac{9}{16}p0 + \tfrac{3}{32}p3 + \tfrac{1}{64}p6 + \tfrac{3}{32}p5 + \tfrac{1}{64}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p1 + \tfrac{1}{16}p2 + \tfrac{3}{8}p0 + \tfrac{3}{8}p3 + \tfrac{1}{16}p5 + \tfrac{1}{16}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{64}p1 + \tfrac{3}{32}p2 + \tfrac{1}{64}p9 + \tfrac{3}{32}p0 + \tfrac{9}{16}p3 + \tfrac{3}{32}p10 + \tfrac{1}{64}p5 + \tfrac{3}{32}p4 + \tfrac{1}{64}p11) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p7 + \tfrac{1}{16}p6 + \tfrac{3}{8}p0 + \tfrac{3}{8}p5 + \tfrac{1}{16}p3 + \tfrac{1}{16}p4) \\
&+ \tfrac{4}{9}\ (\tfrac{1}{4}p0 + \tfrac{1}{4}p3 + \tfrac{1}{4}p5 + \tfrac{1}{4}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p0 + \tfrac{1}{16}p5 + \tfrac{3}{8}p3 + \tfrac{3}{8}p4 + \tfrac{1}{16}p10 + \tfrac{1}{16}p11) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p6 + \tfrac{3}{4}p5 + \tfrac{1}{8}p14) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p0 + \tfrac{1}{16}p3 + \tfrac{3}{8}p5 + \tfrac{3}{8}p4 + \tfrac{1}{16}p14 + \tfrac{1}{16}p13) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{64}p0 + \tfrac{3}{32}p3 + \tfrac{1}{64}p10 + \tfrac{3}{32}p5 + \tfrac{9}{16}p4 + \tfrac{3}{32}p11 + \tfrac{1}{64}p14 + \tfrac{3}{32}p13 + \tfrac{1}{64}p12) \\
&= \tfrac{1}{2304}p8 + \tfrac{23}{2304}p1 + \tfrac{23}{2304}p2 + \tfrac{1}{2304}p9 \\
&+ \tfrac{22}{2304}p7 + \tfrac{523}{2304}p0 + \tfrac{528}{2304}p3 + \tfrac{23}{2304}p10 \\
&+ \tfrac{25}{2304}p6 + \tfrac{541}{2304}p5 + \tfrac{523}{2304}p4 + \tfrac{23}{2304}p11 \\
&+ \tfrac{25}{2304}p14 + \tfrac{22}{2304}p13 + \tfrac{1}{2304}p12
\end{aligned}
$$

Diese Gleichung entspricht den Gleichungen (2.68) und (2.69) mit dem Unterschied, dass für die Koeffizientenberechnung hier wie erwähnt nicht $\hat{S}$ verwendet werden kann. Für $p0$ bis $p14$ müssen jeweils die Vertices eingesetzt werden, die sich an der betreffenden Stelle befinden. Das Array `order` wird dabei mit den Nummern dieser Vertices gefüllt. Rot markiert ist die Zeile, für die sich die Berechnung des Limitpunkts $\mathbf{L}_{4_1}$ in einem Face mit Randvertex von der Berechnung in einem regulären Face unterscheidet. Da der Vertex an der Stelle 5 auf der Randkurve liegt, müssen in dieser Zeile statt der Subdivision-Koeffizienten für Oberflächen die für Kurven eingesetzt werden. Die Folge ist eine im Vergleich zur Maske C_F aus Gleichung (2.44) abweichende Maske. Die anderen beiden rot markierten Limitpunkte auf den Edges ($\mathbf{L}_{5_1}$ und $\mathbf{L}_{4\_5}$) werden analog berechnet. Die speziellen Koeffizienten für diese drei besonderen Limitpunkte werden im Code im Array `c_tag_1` der Größe [3][15] (drei besondere Limitpunkte, $2N + 7 = 15$ beteiligte Vertices) gespeichert.

3.5.3 Berechnung für Faces mit einem Randedge

Das untere Edge der in Abbildung 3.11 grau hinterlegten Faces liegt auf der Randkurve. Hat ein solches Face einen extraordinary Vertex, was im Beispiel mit $N = 3$ der Fall ist, kann dieser links oben (im linken Teil der Abb. zu sehen) oder rechts oben (rechts in der Abb.) liegen. Damit die Berechnung der Limitpunkte des Faces in beiden Fällen auf die gleiche Weise erfolgen kann, muss die Nummerierung entsprechend angepasst werden. Bei der Durchsuchung der Nachbarschaft muss daher zwischen den beiden Möglichkeiten unterschieden werden, was über eine Variable geregelt wird, die den Wert 1 erhält, wenn der extraordinary Vertex links oben liegt oder 2, wenn er rechts oben liegt. Die Nummerierung wird dann wie bei den inneren Faces ohne tagged

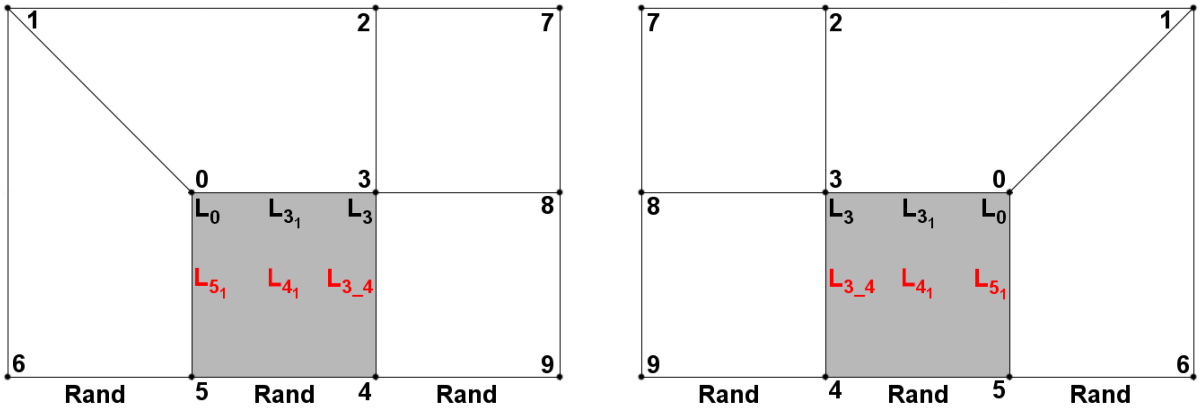

Abbildung 3.11: Limitpunktberechnung bei einem Face mit einem Randedge: Es sind $2N + 4$ Vertices beteiligt. Für dieses Beispiel mit $N = 3$ ergibt sich $2N + 4 = 10$. Der extraordinary Vertex kann links oben (im Bild links) oder rechts oben (im Bild rechts) liegen. Die Nummerierung der beteiligten Vertices muss entsprechend angepasst werden. Die Berechnung der rot markierten Limitpunkte weicht von der Berechnung dieser Punkte in einem regulären Face ab. Die drei unteren Limitpunkte werden erst später bei der Bestimmung von Limitpunkten auf der Randkurve ermittelt.

Edges jeweils so gewählt, dass der extraordinary Vertex an der Stelle 0 liegt. Haben die oberen Vertices beide Valenz $N = 4$, wird immer der linke der beiden als Startpunkt der Nummerierung gewählt. An der Limitpunktberechnung sind hier nur noch $2N + 4$ Vertices beteiligt. Die auf dem Rand liegenden Limitpunkte $\mathbf{L}_5$, $\mathbf{L}_{4\_5}$ und $\mathbf{L}_4$ werden später im Rahmen der Ermittlung der Limitpunkte auf der Randkurve bestimmt (s. Kap. 3.5.6). Für die drei Limitpunkte $\mathbf{L}_0$, $\mathbf{L}_{3_1}$ und $\mathbf{L}_3$ können die mit Hilfe von $\hat{S}$ bestimmten Koeffizienten verwendet werden, für die übrigen drei, in der Abbildung rot markierten Limitpunkte $\mathbf{L}_{5_1}$, $\mathbf{L}_{4_1}$ und $\mathbf{L}_{3\_4}$ müssen die Koeffizienten ähnlich wie schon bei den inneren Faces mit Randvertex explizit bestimmt werden. Erneut soll dies für den Facepunkt $\mathbf{L}_{4_1}$ gezeigt werden:

$$
\begin{aligned}
\mathbf{L}_{4_1} &= \tfrac{1}{36}\ (\tfrac{5}{12}p0 + \tfrac{1}{6}p1 + \tfrac{1}{36}p2 + \tfrac{1}{6}p3 + \tfrac{1}{36}p4 + \tfrac{1}{6}p5 + \tfrac{1}{36}p6) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p1 + \tfrac{1}{16}p2 + \tfrac{3}{8}p0 + \tfrac{3}{8}p3 + \tfrac{1}{16}p5 + \tfrac{1}{16}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{64}p1 + \tfrac{3}{32}p2 + \tfrac{1}{64}p7 + \tfrac{3}{32}p0 + \tfrac{9}{16}p3 + \tfrac{3}{32}p8 + \tfrac{1}{64}p5 + \tfrac{3}{32}p4 + \tfrac{1}{64}p9) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p1 + \tfrac{1}{16}p6 + \tfrac{3}{8}p0 + \tfrac{3}{8}p5 + \tfrac{1}{16}p3 + \tfrac{1}{16}p4) \\
&+ \tfrac{4}{9}\ (\tfrac{1}{4}p0 + \tfrac{1}{4}p3 + \tfrac{1}{4}p5 + \tfrac{1}{4}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p0 + \tfrac{1}{16}p5 + \tfrac{3}{8}p3 + \tfrac{3}{8}p4 + \tfrac{1}{16}p8 + \tfrac{1}{16}p9) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p6 + \tfrac{3}{4}p5 + \tfrac{1}{8}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{2}p5 + \tfrac{1}{2}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p5 + \tfrac{3}{4}p4 + \tfrac{1}{8}p9) \\
&= \tfrac{131}{6912}p1 + \tfrac{107}{10368}p2 + \tfrac{1}{2304}p7 \\
&+ \tfrac{745}{3456}p0 + \tfrac{383}{1728}p3 + \tfrac{22}{2304}p8 \\
&+ \tfrac{29}{2592}p6 + \tfrac{1739}{6912}p5 + \tfrac{2591}{10368}p4 + \tfrac{25}{2304}p9
\end{aligned}
$$

Für $p0$ bis $p9$ sind wiederum die Vertices einzusetzen, die sich an der betreffenden Stelle befinden und die Vertexnummern bilden den Inhalt des Arrays `order`. Da drei Limitpunkte auf der Randkurve liegen, treten hier auch drei Zeilen auf, in denen statt der Subdivision-Koeffizienten für Oberflächen die für Kurven verwendet werden, zu erkennen an der roten Markierung. Die beiden anderen rot gekennzeichneten Limitpunkte auf den Edges ($\mathbf{L}_{5_1}$ und $\mathbf{L}_{3\_4}$) werden analog zu $\mathbf{L}_{4_1}$ berechnet. Gespeichert werden die Koeffizienten für diese besonderen Limitpunkte im Array `c_tag_2` der Größe [3][4][16] (drei besondere Limitpunkte, vier mögliche Valenzen und $2N+4$ beteiligte Vertices, was bei $N=6$ maximal 16 ergibt).

3.5.4 Berechnung für Faces mit zwei Randedges

Das untere und rechte Edge in Abbildung 3.12 liegen auf dem Rand. Daraus ergibt sich, dass sich nur der Vertex links oben im Inneren des Gitters befindet. Die Nummerierung wird so gewählt, dass dieser Vertex immer die Nummer 0 erhält. Für ihn muss bei der Berechnung wieder auf die Valenz geachtet werden, die im Beispiel $N=3$ beträgt. An der Limitpunktberechnung eines solchen Faces sind dann $2N+1$ Vertices beteiligt. Die fünf auf dem Rand liegenden Limitpunkte $\mathbf{L}_3$, $\mathbf{L}_{3\_4}$, $\mathbf{L}_4$, $\mathbf{L}_{4\_5}$ und $\mathbf{L}_5$ werden später berechnet, wenn die Limitpunkte auf der Randkurve bestimmt werden (s. Kap. 3.5.6). Für den Limitpunkt $\mathbf{L}_0$ können wie gehabt die Koeffizienten aus Gleichung (2.49) verwendet werden, für die übrigen drei, in der Abbildung rot markierten Limitpunkte $\mathbf{L}_{3_1}$, $\mathbf{L}_{4_1}$ und $\mathbf{L}_{5_1}$ müssen die Koeffizienten dagegen explizit bestimmt werden. Wie schon in den beiden vorangegangen Abschnitten soll dies für den Facepunkt $\mathbf{L}_{4_1}$ gezeigt werden:

$$
\begin{aligned}
\mathbf{L}_{4_1} &= \tfrac{1}{36}\ (\tfrac{5}{12}p0 + \tfrac{1}{6}p1 + \tfrac{1}{36}p2 + \tfrac{1}{6}p3 + \tfrac{1}{36}p4 + \tfrac{1}{6}p5 + \tfrac{1}{36}p6) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p1 + \tfrac{1}{16}p2 + \tfrac{3}{8}p0 + \tfrac{3}{8}p3 + \tfrac{1}{16}p5 + \tfrac{1}{16}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p2 + \tfrac{3}{4}p3 + \tfrac{1}{8}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{16}p1 + \tfrac{1}{16}p6 + \tfrac{3}{8}p0 + \tfrac{3}{8}p5 + \tfrac{1}{16}p3 + \tfrac{1}{16}p4) \\
&+ \tfrac{4}{9}\ (\tfrac{1}{4}p0 + \tfrac{1}{4}p3 + \tfrac{1}{4}p5 + \tfrac{1}{4}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{2}p3 + \tfrac{1}{2}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p6 + \tfrac{3}{4}p5 + \tfrac{1}{8}p4) \\
&+ \tfrac{1}{9}\ (\tfrac{1}{2}p5 + \tfrac{1}{2}p4) \\
&+ \tfrac{1}{36}\ (\tfrac{1}{8}p5 + \tfrac{3}{4}p4 + \tfrac{1}{8}p3) \\
&= \tfrac{1}{54}p1 + \tfrac{29}{2592}p2 \\
&+ \tfrac{89}{432}p0 + \tfrac{211}{864}p3 \\
&+ \tfrac{29}{2592}p6 + \tfrac{211}{864}p5 + \tfrac{343}{1296}p4
\end{aligned}
$$

Das Array `order` wird mit den Vertices gefüllt, die sich an den Stellen $p0$ bis $p6$ befinden. Die fünf rot markierten Zeilen weisen darauf hin, dass fünf der neun Limitpunkte auf dem Rand liegen. In diesen Zeilen finden die Subdivision-Koeffizienten für Kurven Anwendung. Die beiden anderen rot eingezeichneten Limitpunkte auf den Edges ($\mathbf{L}_{3_1}$ und $\mathbf{L}_{5_1}$) werden analog zu $\mathbf{L}_{4_1}$ berechnet. Die benötigten Koeffizienten zur Ermittlung dieser drei besonderen Limitpunkte werden im Array

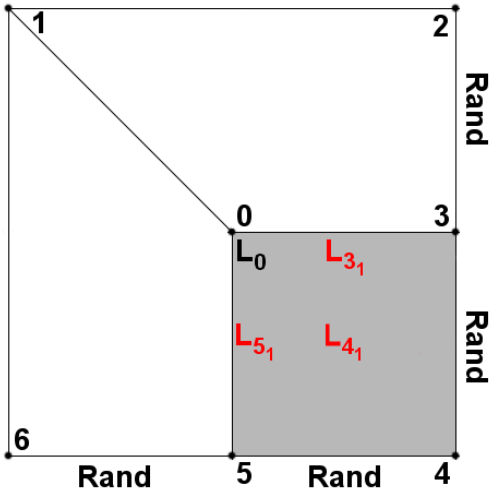

Abbildung 3.12: Limitpunktberechnung bei einem Face mit zwei Randedges: Es sind $2N + 1$ Vertices beteiligt. Für dieses Beispiel mit $N = 3$ ergibt sich $2N + 1 = 7$. Die Berechnung der rot markierten Limitpunkte weicht von der Berechnung dieser Punkte in einem regulären Face ab. Die fünf Limitpunkte unten und rechts werden erst später bei der Bestimmung von Limitpunkten auf der Randkurve ermittelt.

`c_tag_3` der Größe [3][4][13] gespeichert (drei besondere Limitpunkte, vier mögliche Valenzen und $2N+1$ beteiligte Vertices, was bei der maximal möglichen Valenz von $N = 6$ zu $2N+1 = 13$ führt).

3.5.5 Berechnung für innere Faces mit einem oder zwei tagged Edges

Die Limitpunkte eines inneren Faces mit einem bzw. zwei tagged Edges werden genauso berechnet wie die eines entsprechenden Randfaces. Anhand des Beispiels in Abbildung 3.13 soll gezeigt werden, dass diese Vereinfachung zu einigen geringfügigen Ungenauigkeiten führen kann. Die mit t gekennzeichneten Edges sind tagged. Dieses Beispiel soll als ein Ausschnitt aus einer Gittergeometrie verstanden werden, d.h., die acht Faces, die das mittlere Face umgeben, liegen nicht am Rand. Die rot gekennzeichneten Limitpunkte für Face F5 werden so berechnet wie die eines Faces mit zwei Randedges, also so, als ob die mit E1 und E4 bezeichneten Edges ebenfalls beide tagged wären. Zur Berechnung werden die Vertices aus den Faces F1, F2, F4 und F5 verwendet. Da die Edges E1 und E4 aber in Wahrheit nicht tagged sind, müssten zur vollständig korrekten Bestimmung der drei roten Limitpunkte auch die Faces F3, F6, F7 und F8 herangezogen werden, da Vertices, die nur ein angrenzendes tagged Edge aufweisen (in Face F5 links oben und rechts unten) wie Vertices ohne angrenzende tagged Edges behandelt werden. Die Vertices in den zusätzlichen Faces haben jedoch nur geringen Einfluss auf das Ergebnis, was die Vereinfachung der Implementierung rechtfertigt. Dies kann mit Hilfe der in Kapitel 3.8.1 beschriebenen Funktion zur Fehlerberechnung zwischen berechneten Limitpunkten und Subdivisionpunkten nach einem oder mehreren darauf folgenden Unterteilungsschritten überprüft werden. Ohne Vereinfachung müssten viele Fallunterscheidungen programmiert werden, z.B. könnte auch nur eines der beiden

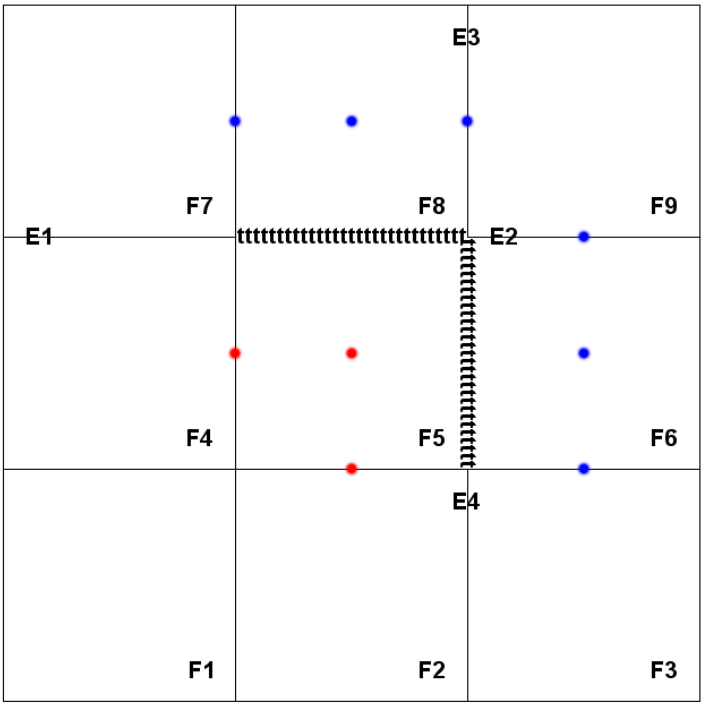

Abbildung 3.13: Vereinfachungen bei der Limitpunktberechnung: Die blau markierten Limitpunkte für innere Faces mit einem tagged Edge (F6 und F8) werden so berechnet wie die Limitpunkte in einem Face mit einem Randedge, d.h. für F6 so, als ob die Edges E3 und E4 auch tagged wären bzw. entsprechend für F8 so, als ob E1 und E2 auch tagged wären. Analog werden die rot markierten Limitpunkte für innere Faces mit zwei tagged Edges (F5) so berechnet wie die Limitpunkte in einem Face mit zwei Randedges, d.h. so, als ob die Edges E1 und E4 auch tagged wären.

Edges E1 oder E4 tagged sein, was die korrekte Berechnung erneut verändern würde.

Ähnliches gilt für die blau markierten Limitpunkte in den Faces F6 und F8, die so berechnet werden, als ob es sich um Faces mit einem Randedge handeln würde. In Face F6 werden sie also so bestimmt, als ob die Edges E3 und E4 ebenfalls beide tagged wären. Es werden die Faces F3, F6, F9 und die drei rechts davon liegenden, in der Abbildung nicht eingezeichneten Faces verwendet, statt korrekterweise zusätzlich F2 und F5 mit einzubeziehen. Da der links oben in Face F6 liegende Vertex auf einer festgelegten Kurve liegt (es laufen zwei tagged Edges in den Vertex, s. auch Kap. 3.5.7), spielt Face F8 bei der eigentlich richtigen Berechnung der drei Limitpunkte von Face F6 keine Rolle. Eine solche Situation verkompliziert die vollständig korrekte Implementierung der Limitpunktberechnung erneut und man kann sich vorstellen, dass viele weitere Fälle möglich sind.

Analog zu Face F6 werden die drei blauen Limitpunkte in Face F8 so berechnet, als ob die Edges E1 und E2 beide tagged wären.

3.5.6 Berechnung auf der Randkurve

Falls ein Gitter eine Randkurve aufweist, werden die Limitpunkte auf dieser Kurve separat ermittelt. Hierfür werden zunächst alle Nummern der Edges, die als linkes oder rechtes Nachbarface den Eintrag -1 enthalten, der „außerhalb des Gitters“ bedeutet, in einer Liste `bound_edges` gespeichert. Diese Liste wird daraufhin so in eine neue Liste `bound_edges_sorted` sortiert, dass der Endpunkt des einen Edges dem Anfangspunkt des nächsten Edges entspricht und sich somit die geschlossene Randkurve ergibt. Analog erhält das Array `bound_verts` die Nummern der sortierten Vertices der Randkurve. Für diese Kurve werden mit den Koeffizienten $\left(\frac{1}{6} \quad \frac{2}{3} \quad \frac{1}{6}\right)$ wie in Gleichung (2.43) zuerst die Limitpunkte zu den Vertices berechnet, solange die Tag-Summe eines Vertex gleich 2 ist. Die Tag-Summe ist die Anzahl der an einen Vertex angrenzenden tagged Edges, auf der Randkurve also für jeden Vertex größer oder gleich 2. Ist die Tag-Summe aber größer oder gleich 3, führt dies dazu, dass der entsprechende Vertex unveränderlich ist und seine Koordinaten bei einer Subdivision und dementsprechend auch bei der Limitpunktberechnung beibehält. Der Code für die Berechnung des Limitpunkts zum Vertex an der Stelle `bound_verts[i]` im Fall Tag-Summe = 2 lautet folgendermaßen (`bound_edges` wird abgekürzt geschrieben als `b_e`, `bound_verts` als `b_v`):

```
limit_pts[b_v[i]].SetVals(
     1./6. * vertices[b_v[(i+b_e.size()-1)%b_e.size()]].Vec()
   + 2./3. * vertices[b_v[i]].Vec()
   + 1./6. * vertices[b_v[(i+1)%b_e.size()]].Vec());
```

Das Prozentzeichen steht hierbei für die Modulo-Funktion. Der Limitpunkt erhält die gleiche Nummer wie sein Ausgangsvertex (also `bound_verts[i]`). Die beiden benachbarten Vertices befinden sich im Array an den Stellen `(i+b_e.size()-1)%b_e.size()` (entspricht normalerweise $i-1$; für $i = 0$ muss aber der letzte in `bound_verts` gespeicherte Vertex gewählt werden) und `(i+1)%b_e.size()`. Die Funktion `Vec()` liefert den Koordinatenvektor eines Vertex.

Bei der Bestimmung der Limitpunkte auf den Edges müssen vier Fallunterscheidungen getroffen werden:

1. Anfangs- und Endvertex des Edges haben beide eine Tag-Summe größer oder gleich 3:

   ```
   limit_pts[NumV+NumF+*iter].SetVals(
        1./2. * vertices[b_v[i]].Vec()
      + 1./2. * vertices[b_v[(i+1)%b_e.size()]].Vec());
   ```

   `iter` ist hierbei ein Listeniterator, dessen Dereferenzierung die Nummer des betrachteten Edges angibt. `b_v[i]` und `b_v[(i+1)%b_e.size()]` entsprechen den Nummern von Anfangs- und Endvertex des Edges.

2. Anfangs- und Endvertex des Edges haben beide eine Tag-Summe gleich 2. Es werden die Koeffizienten $\left(\frac{1}{48} \quad \frac{23}{48} \quad \frac{23}{48} \quad \frac{1}{48}\right)$ aus Gleichung (2.42) verwendet:

   ```
   limit_pts[NumV+NumF+*iter].SetVals(
        1./48. * vertices[b_v[(i+b_e.size()-1)%b_e.size()]].Vec()
   ```

```
    + 23./48. * vertices[b_v[i]].Vec()
    + 23./48. * vertices[b_v[(i+1)%b_e.size()]].Vec()
    + 1./48. * vertices[b_v[(i+2)%b_e.size()]].Vec());
```

3. Der Anfangsvertex des Edges hat eine Tag-Summe größer oder gleich 3, der Endvertex die Tag-Summe 2:

```
limit_pts[NumV+NumF+*iter].SetVals(
    + 25./48. * vertices[b_v[i]].Vec()
    + 22./48. * vertices[b_v[(i+1)%b_e.size()]].Vec()
    + 1./48. * vertices[b_v[(i+2)%b_e.size()]].Vec());
```

4. Der Anfangsvertex des Edges hat die Tag-Summe 2, der Endvertex eine Tag-Summe größer oder gleich 3:

```
limit_pts[NumV+NumF+*iter].SetVals(
      1./48. * vertices[b_v[(i+b_e.size()-1)%b_e.size()]].Vec()
    + 22./48. * vertices[b_v[i]].Vec()
    + 25./48. * vertices[b_v[(i+1)%b_e.size()]].Vec());
```

3.5.7 Berechnung auf inneren tagged Edges

Bei der Berechnung von Limitpunkten auf inneren tagged Edges wurden abgerundete Kanten nicht berücksichtigt, d.h., ein solcher Limitpunkt wird immer so berechnet, als ob es sich bei dem Edge um eine scharfe Kante handeln und das Edge somit für unendlich viele Unterteilungsschritte tagged bleiben würde.

Die Berechnung von Limitpunkten zu Vertices auf inneren tagged Edges läuft genauso ab wie bei Vertices auf der Randkurve. Es gibt zwar den zusätzlichen Fall der Tag-Summe gleich 1, aber die Vertices, für die dies gilt, werden wie Vertices mit einer Tag-Summe gleich 0 behandelt und ihre Limitpunkte schon vorher im Rahmen der Faceabarbeitung ermittelt.

Bei der Berechnung der Edge-Limitpunkte auf inneren tagged Edges erhält der Fall Tag-Summe gleich 1 jedoch eine eigene Behandlung. Zur besseren Übersicht werden die möglichen Fälle in Tupelschreibweise dargestellt: (Tag-Summe des Edge-Anfangsvertex, Tag-Summe des Edge-Endvertex). Dann ergeben sich zusätzlich zu den im vorigen Abschnitt erläuterten Fallunterscheidungen $(\geq 3, \geq 3)$, $(2, 2)$, $(\geq 3, 2)$ und $(2, \geq 3)$ die folgenden Fälle:

5. $(1, 1)$
6. $(1, 2)$
7. $(1, \geq 3)$
8. $(2, 1)$
9. $(\geq 3, 1)$

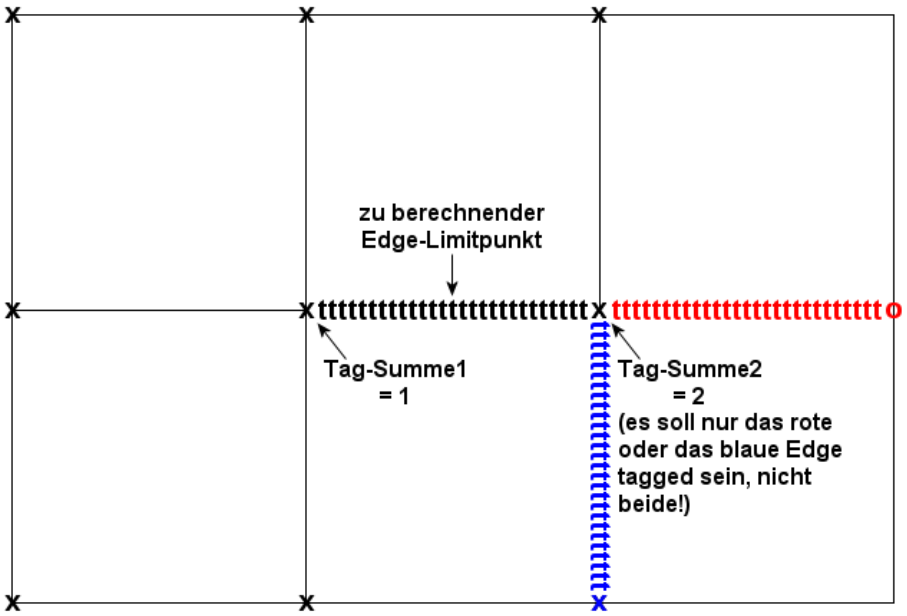

Abbildung 3.14: Unterscheidung bei der Berechnung eines Edge-Limitpunkts für den Fall „(Tag-Summe des Edge-Anfangsvertex = 1, Tag-Summe des Edge-Endvertex = 2)“: Ist das blaue Edge tagged, verstärkt sich der Einfluss des Vertex mit dem blau gefärbten x. Ist dagegen das rote Edge tagged, fließt stattdessen zusätzlich der mit dem roten o markierte Vertex in die Berechnung ein.

Da die Berechnung prinzipiell ähnlich verläuft wie die für die Edge-Limitpunkte auf der Randkurve, soll hier nur anhand des Falls $(1, 2)$ auf eine Besonderheit aufmerksam gemacht werden, die bei der Implementierung beachtet werden musste und die in Abbildung 3.14 veranschaulicht ist. Der Anfangspunkt des Edges, zu dem der Limitpunkt berechnet werden soll, hat die Tag-Summe 1, der Endpunkt die Tag-Summe 2 (es soll entweder das rote oder das blaue Edge tagged sein, aber nicht beide, da die Tag-Summe dann gleich 3 wäre). Da Vertices mit Tag-Summe gleich 1 wie Vertices mit Tag-Summe gleich 0 behandelt werden, fließen in die Berechnung des Limitpunkts auf dem Edge neun Vertices durch den Anfangsvertex ein, nämlich der Anfangsvertex selbst und die acht benachbarten Vertices, die alle mit x markiert sind. Ist nun das blaue das tagged Edge, wird der Einfluss des Vertex an der Stelle des blau gefärbten x verstärkt. Ist dagegen das rote das tagged Edge, fließt in die Berechnung des Edge-Limitpunkts ein weiterer Vertex ein (rotes o), der im anderen Fall keine Rolle spielt. Diese Besonderheit tritt nur bei den Fällen $(1, 2)$ und $(2, 1)$ auf. Im Beispiel könnte natürlich auch statt des blauen oder roten das oberhalb des blauen Edges liegende Edge tagged sein, was dann den Einfluss des mit x markierten Vertex rechts oben vergrößern würde.

3.5.8 Zusammenfassung der Bedingungen bei der Limitpunktberechnung

Zusammenfassend seien hier die Voraussetzungen genannt, die ein Gitter erfüllen muss, damit die Limitpunktberechnung möglich ist. Bei Verletzung einer der Bedingungen reicht es, zunächst einen oder mehrere Subdivision-Schritte durchzuführen, wozu der Benutzer in jedem der Fälle auch vom Programm aufgefordert wird. Danach ist die Limitpunktberechnung möglich.

- Das Gitter muss ausschließlich aus Vierecken bestehen.
- Alle extraordinary Vertices müssen isoliert sein, d.h., jedes Face darf maximal einen extraordinary Vertex aufweisen.
- Es sind maximal zwei tagged Edges pro Face erlaubt.
- Diese zwei tagged Edges dürfen sich nicht gegenüber liegen, d.h., sie müssen zusammenhängend sein.
- Ein Face, das genau einen Randvertex enthält, darf keine tagged Edges aufweisen.
- Bei einem Face, das genau einen Randvertex enthält, müssen die übrigen drei Vertices regulär sein, also Valenz $N = 4$ haben.
- Bei einem inneren Edge, das nicht tagged ist, darf nur einer der beiden zum Edge gehörenden Vertices eine Tag-Summe größer als 0 haben, nicht beide. Da diese Bedingung bisher noch nicht genannt wurde, ist sie in Abbildung 3.15 veranschaulicht.

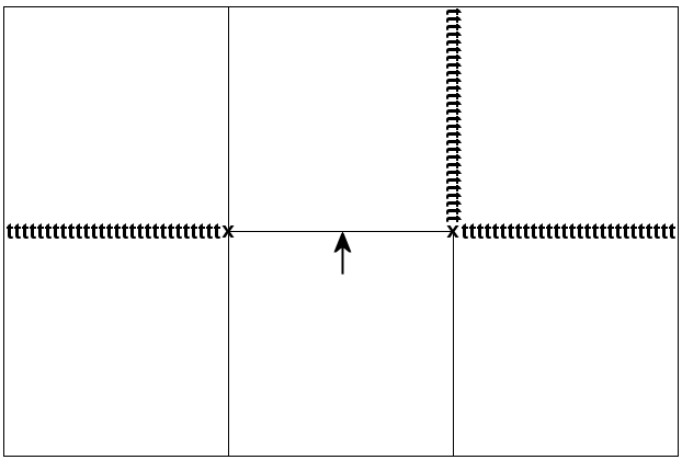

Abbildung 3.15: Nicht erlaubte Situation bei der Limitpunktberechnung: Das mit dem Pfeil gekennzeichnete Edge in diesem Ausschnitt aus einer Gittergeometrie ist nicht tagged, die beiden zu ihm gehörenden Vertices (markiert mit x) haben aber die Tag-Summen 1 (links) und 2 (rechts). Dies ist bei der Limitpunktberechnung nicht erlaubt und erfordert zunächst die Durchführung eines Subdivision-Schrittes.

3.6 Approximation gegebener Oberflächen mit Catmull-Clark-Gittern

3.6.1 Aufstellen der Koeffizientenmatrizen A und A^T

Für die Lösung des linearen Ausgleichsproblems $\|A\mathbf{V} - \mathbf{S}\|_2 \rightarrow min$, das in Kapitel 2.5 beschrieben wurde, müssen zuerst die Koeffizientenmatrix A und ihre Transponierte A^T aufgestellt werden. Um die dünnbesetzte Struktur beider Matrizen auszunutzen und dadurch Berechnungen wie z.B. Matrix-Vektor-Multiplikationen effizient zu gestalten, sollen nur Einträge ungleich 0 gespeichert werden. Da jede Matrixzeile unterschiedlich viele Elemente enthält, müssen die benötigten Zeilenlängen für beide Matrizen im Voraus ermittelt werden. Danach können die Einträge von A vollständig bestimmt und anschließend in A^T kopiert werden. Die Klasse `CCaClMesh` wird um zwei private Variablen `vector< vector<double> > matrix` und `vector< vector<double> > matrix_T` erweitert. Der innere Vektor vom Typ `double` enthält die Einträge einer Zeile der jeweiligen Matrix und im äußeren Vektor werden alle diese Zeilen gespeichert. Eine Matrixzeile besteht hierbei abwechselnd aus einer zu einem `double`-Wert konvertierten Vertexnummer und dem Koeffizienten, mit dem der Vertex mit dieser Nummer multipliziert werden muss. Der Vektor aller Limitpunkte $\mathbf{L} = A\mathbf{V}$ z.B. würde sich im Code dann folgendermaßen berechnen lassen:

```
for (int i=0; i<A.size(); i++) {
    AV[i] = 0.;
    for (int j=0; j<A[i].size(); j=j+2)
        AV[i] = AV[i] + A[i][j+1] * vertices[A[i][j]];
}
```

Die Matrix A besteht aus $\#V + \#E + \#F$ Zeilen, die wie erwähnt Vektoren unterschiedlicher Länge entsprechen. Jede Zeilenlänge von A ergibt sich aus der jeweiligen Anzahl der an der Berechnung des mit der Nummer der Zeile versehenen Limitpunkts beteiligten Vertices. Sie wird zu Beginn der Funktion `LimitPoints()` ermittelt und im, in der Funktion dynamisch angelegten, Array `rowsize` vom Typ `int` mit der Größe `NumV+NumE+NumF` gespeichert. Zuerst wird dies für die Limitpunkte zu den Vertices erledigt. Hierbei spielt die Tag-Summe wieder eine wichtige Rolle. Ist diese für einen Vertex z.B. größer oder gleich 3, gelten die Koordinaten dieses Vertex als unveränderlich und die entsprechende Matrixzeile erhält die Länge 2. Die beiden Einträge sind dann die Nummer dieses Vertex und der zugehörige Koeffizient 1. Bei einer Tag-Summe gleich 2 muss die Matrixzeile wegen der Beteiligung von drei Vertices an der Limitpunktberechnung sechs Elemente fassen können, während bei einer Tag-Summe kleiner oder gleich 1 die Zeilenlänge von der Valenz des Vertex abhängt und $2 \cdot (2N + 1)$ beträgt. Die Zeilenlängen zu den Vertexpunkten werden im Array `rowsize` an den Stellen `0` bis `NumV-1` abgelegt. Danach werden die benötigten Zeilenlängen der Facepunkte ermittelt. Dabei ist entscheidend, um was für einen Facetyp es sich handelt. Entsprechend der Abbildungen 3.9 bis 3.12 muss eine Zeile $2 \cdot (2N + 8)$, $2 \cdot (2N + 7)$,

$2\cdot(2N+4)$ oder $2\cdot(2N+1)$ Elemente aufnehmen können. Diese Zeilenlängen stehen in `rowsize` an den Stellen `NumV` bis `NumV+NumF-1`. Es folgen die notwendigen Zeilenlängen der Edgepunkte von `NumV+NumF` bis `NumV+NumF+NumE-1`, die über die Tag-Summen des Anfangs- und Endvertex des jeweiligen Edges bestimmt werden. Hierbei gibt es vor allem für tagged Edges, wie in den Kapiteln 3.5.6 und 3.5.7 gesehen, viele verschiedene Kombinationen, die beachtet werden müssen. Ist das Array `rowsize` schließlich komplett gefüllt, wird die Koeffizientenmatrix wie folgt initialisiert:

```
matrix.resize(NumV + NumF + NumE);
for (int i=0; i<matrix.size(); i++) {
     matrix[i].reserve(rowsize[i]);
     for (int j=0; j<rowsize[i]; j++)
          matrix[i].push_back(0.);
}
```

Zu beachten ist hier der Unterschied zwischen den Funktionen `resize(·)` und `reserve(·)`: Während `resize(·)` die über `size()` abzufragende Größe eines Vektors direkt festlegt, reserviert `reserve(·)` nur den hierfür nötigen Speicher, d.h., die Abfrage `size()` würde noch den Wert 0 ergeben. Erst mit jedem `push_back(·)` wird die Größe des Vektors um 1 erhöht. Die Vorbelegung sämtlicher Einträge mit 0 kann als Vorsichtsmaßnahme angesehen werden: Sollte es vorkommen, dass eine Zeilenlänge zu groß bestimmt wird und damit später Einträge am Ende der Zeile keinen Wert zugewiesen bekommen, ist dennoch ein richtiges Ergebnis gewährleistet. Ohne diese Vorbelegung könnte ein undefiniertes Verhalten auftreten, da nicht vorherzusehen wäre, welche Werte an den nicht zugewiesenen Stellen stehen würden.

Im Anschluss an die Berechnung der Zeilenlängen der Matrix A erfolgt wie gehabt die Limitpunktberechnung. Die hierbei verwendeten Koeffizienten und die Vertices, auf die diese angewendet werden, werden dabei in A (bzw. `matrix` im Code) gespeichert. Für ein Face ohne tagged Edges z.B. wurde der aus Kapitel 3.5.1 bekannte Code erweitert und sieht nun wie folgt aus:

```
for (int i=0; i<9; i++) {
     xl.SetVals(0., 0., 0.);
     k = 0;
     if (limit_pts[position[i]].Get_lp_set() == false) {
          for (int j=0; j<2*extraord_val+8; j++) {
               xl += c[i][extraord_val-3][j] * vertices[order[j]].Vec();
               if (c[i][extraord_val-3][j] != 0.
                         && k < matrix[position[i]].size()) {
                    matrix[position[i]][k] = order[j];
                    matrix[position[i]][k+1] = c[i][extraord_val-3][j];
                    k = k + 2;
               }
          }
```

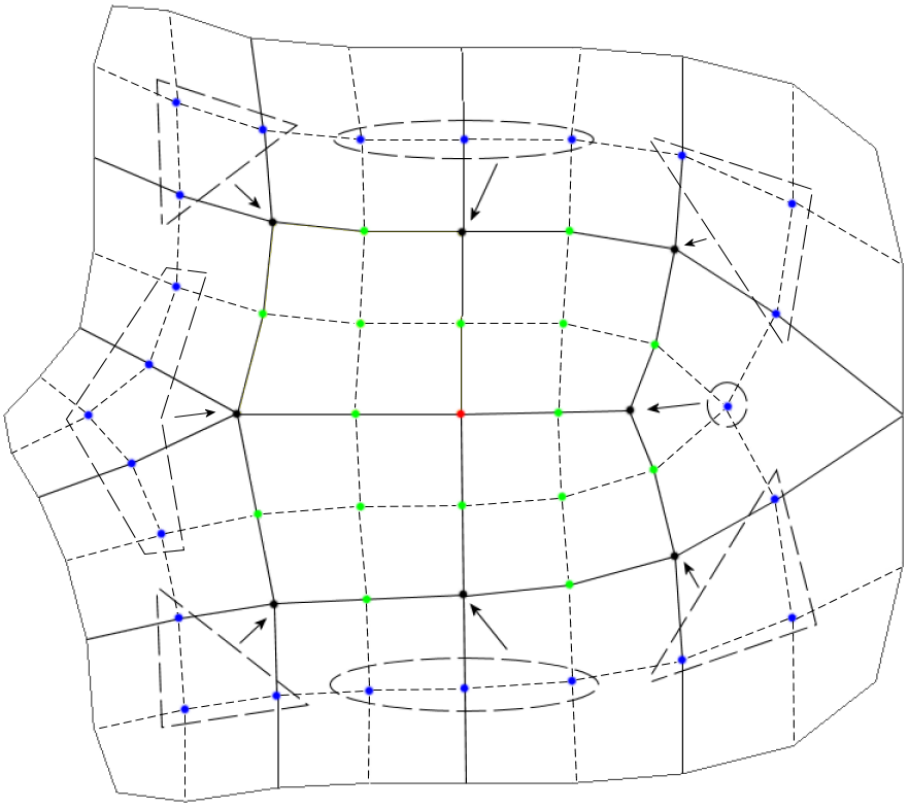

Abbildung 3.16: Ermittlung der Zeilenlängen von A^T: Der rot markierte Vertex beeinflusst die Berechnung der $6N+1$ Limitpunkte (rot, grün und schwarz) in den angrenzenden N Faces. Durch jeden schwarz gekennzeichneten Nachbarvertex kommen $2 \cdot (N-3)+1$ weitere Limitpunkte (blau) hinzu. Für jeden Vertex wird die Gesamtzahl der Limitpunkte, die er beeinflusst, im Code ermittelt und die Zeilenlängen der Matrix A^T können dementsprechend vorgegeben werden.

```
            limit_pts[position[i]].SetVals(xl);
            limit_pts[position[i]].Set_lp_set(true);
        }
    }
```

Bei den Berechnungen der Limitpunkte für andere Facetypen oder auf Rand- bzw. tagged Edges wird die Matrix A auf ähnliche Weise gefüllt.

Nach Abschluss der Limitpunktberechnung ist die Matrix A vollständig bestimmt und ihre Transponierte kann aufgestellt werden. A^T hat $\#V$ Zeilen, für die zunächst wieder über ein dynamisches `int`-Array `rowsize_T` der Größe `NumV` die benötigte Länge ermittelt werden muss. Die zu beantwortende Frage für jeden Vertex lautet dabei: An der Berechnung wie vieler Limitpunkte ist der Vertex beteiligt? Abbildung 3.16 veranschaulicht diese Situation für einen inneren, regulären Vertex (rot markiert). Die mit unterschiedlichen Farben gekennzeichneten Punkte stellen die Limitpunkte dar, auf die der rote Vertex einen Einfluss hat. Dies sind zunächst $6N+1$ in den N Faces, die an den Vertex angrenzen, hier mit $N=4$ also 25 (roter, grüne und schwarze

Punkte). Die $2N$ Nachbarvertices (schwarze Punkte) führen jeweils zu $2 \cdot (N-3)+1$ Limitpunkten, auf die der rote Vertex Einfluss hat. In diesem Beispiel haben sechs Nachbarvertices die Valenz $N = 4$ (die oberen und die unteren drei), es ergeben sich also zusätzlich zu den 25 weitere $6 \cdot (2 \cdot (N-3)+1) = 18$ vom roten Vertex beeinflusste Limitpunkte. Dazu kommen noch fünf Limitpunkte durch den linken Vertex der Valenz $N = 5$ und ein weiterer durch den rechten Vertex mit $N = 3$. Insgesamt ergeben sich damit $25+18+5+1 = 49$ vom roten Vertex beeinflusste Limitpunkte. `rowsize_T` an der Stelle der Nummer des roten Vertex würde also die vorgegebene Zeilenlänge 98 erhalten ($2 \cdot 49$, weil jeder Limitpunktnummer noch ein Koeffizient zugeordnet werden muss, s.o.). In diesem Beispiel liegen alle Vertices im Inneren des Gitters. Die Zeilenlänge verringert sich entsprechend, wenn einer oder mehrere Vertices am Rand liegen. Nicht beachtet wurden bei der Implementierung innere tagged Edges. Liegt ein Vertex auf einem solchen Edge, kann das dazu führen, dass für die entsprechende Matrixzeile von A^T eine zu große Länge angelegt wird. Die zusätzlichen, eigentlich nicht benötigten Einträge enthalten den Wert 0 und beeinflussen das Ergebnis damit nicht. Da in der Regel auch nur wenige Zeilen betroffen sind, führt diese Vereinfachung nicht zu einer merklichen Verlangsamung der Füllung der Matrix A^T bzw. bei Multiplikationen von A^T mit Vektoren. Die Matrix kann mit Werten gefüllt werden, sobald alle Vertices abgearbeitet sind und `rowsize_T` sämtliche benötigten Zeilenlängen von A^T enthält. Die Initialisierung ähnelt im Code der Initialisierung von A (s.o.):

```
matrix_T.resize(NumV);
for (int i=0; i<matrix_T.size(); i++)
    matrix_T[i].reserve(rowsize_T[i]);
```

Danach wird A^T durch Kopieren der Werte aus A gefüllt:

```
for (int i=0; i<matrix.size(); i++) {
    for (int j=0; j<matrix[i].size(); j=j+2) {
        if (!(matrix[i][j] == 0 && matrix[i][j+1] == 0)) {
            matrix_T[matrix[i][j]].push_back(i);
            matrix_T[matrix[i][j]].push_back(matrix[i][j+1]);
        }
    }
}
```

3.6.2 Surfacepunktberechnung

Ein weiterer Bestandteil auf dem Weg zur Lösung des linearen Ausgleichsproblems $\|A\mathbf{V}-\mathbf{S}\|_2 \rightarrow min$ ist der Vektor $\mathbf{S}$, der die Surfacepunkte enthält. Diese können durch die Projektion der Limitpunkte auf eine beliebige Fläche ermittelt werden. Im hier entwickelten Programm wurde jedoch ein anderer Weg gewählt: Um den Vektor $\mathbf{S}$ zu bestimmen, wurde in der Klasse `CCaClMesh` die Funktion `SurfacePoints()` implementiert. Als zu approximierende gegebene Oberfläche

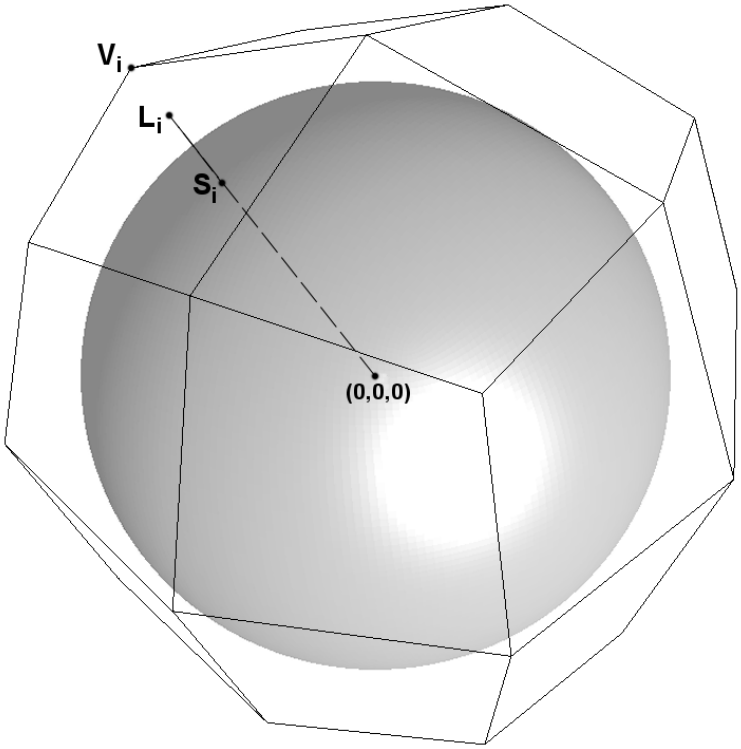

Abbildung 3.17: Surfacepunktberechnung: Die zu approximierende Kugeloberfläche liegt innerhalb des mindestens und hier genau ein Mal unterteilten Würfels. Jeder Surfacepunkt $\mathbf{S}_i$ wird berechnet, indem der entsprechende Limitpunkt $\mathbf{L}_i$ über eine Gerade mit dem Ursprung verbunden und diese Gerade mit der Kugeloberfläche geschnitten wird.

enthält diese ein Ellipsoid, dessen Gleichung

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{3.3}$$

lautet. Mit $a = b = c = 1$ erhält man die Einheitskugel (Radius $R = 1$). Wählt man als Startgitter einen Würfel, sind alle acht Vertices extraordinary, da sie die Valenz $N = 3$ haben. Um die Limitpunkte zu berechnen, müssen diese extraordinary Vertices bekanntlich isoliert werden, was nach einem Subdivision-Schritt der Fall ist. Jeder Limitpunkt wird dann über eine Gerade mit dem Ursprung verbunden. Die Schnittpunkte dieser Geraden mit der innerhalb des mindestens ein Mal unterteilten Würfels liegenden Kugel geben hier die Koordinaten der Surfacepunkte an. In Abbildung 3.17 ist dies grafisch dargestellt. Für die formale Herleitung benötigt man zusätzlich zur Ellipsoidgleichung (3.3) die Geradengleichung durch den Ursprung, die in Vektorschreibweise folgendermaßen formuliert werden kann:

$$\mathbf{x}_i = t\mathbf{L}_i \tag{3.4}$$

Diese Gleichung wird in die Ellipsoidgleichung (3.3) eingesetzt und man erhält

$$\frac{t^2 L_{i,x}^2}{a^2} + \frac{t^2 L_{i,y}^2}{b^2} + \frac{t^2 L_{i,z}^2}{c^2} - 1 = 0. \tag{3.5}$$

Nach t aufgelöst ergibt sich

$$t_{1,2} = \pm \frac{1}{\left(\frac{L_{i,x}^2}{a^2} + \frac{L_{i,y}^2}{b^2} + \frac{L_{i,z}^2}{c^2} \right)^{\frac{1}{2}}}. \tag{3.6}$$

Der Surfacepunkt $\mathbf{S}_i$ ist dann bestimmt über

$$\mathbf{S}_i = t_1 \mathbf{L}_i, \tag{3.7}$$

wobei t_1 die positive Lösung ist. Im Code werden die auf diese Weise berechneten Surfacepunkte für jedes Mesh in der privaten Variable `vector<CVertex> surface_pts` aus der Klasse `CCaClMesh` gespeichert.

3.6.3 CGLS-Methode

Die in Kapitel 2.5 vorgestellte CGLS-Methode wurde als Template-Funktion implementiert:

```
template<class T>
void CCaClMesh::CGLS(vector< vector<double> > &A,
        vector< vector<double> > &A_T, vector<T> &x,
        vector<T> &b, int use_defaults)
```

Der Typ `T` kann hierbei `double` oder `CVertex` sein. Damit sämtliche in der Methode benötigten Operationen auch mit Objekten vom Typ `CVertex` korrekt funktionieren, wurden in der Klasse `CVertex` die Operatoren =, +, −, ⋆ (in zwei Varianten: 1. Multiplikation `double`-Wert mit `CVertex`-Objekt, 2. Multiplikation zweier `CVertex`-Objekte), /, <= und « (für Ausgaben) sowie die mathematischen Funktionen `fabs` und `sqrt` überladen. Die Variablen `A`, `A_T`, `x` und `b` werden jeweils als Referenzen übergeben, so dass Änderungen an ihnen auch in der aufrufenden Funktion sichtbar werden. Beim Aufruf der Approximation aus dem Kontextmenü des Programms kann der Benutzer wählen, ob er für die maximale Iterationsanzahl K den Default-Wert 100 und für die Toleranz $\epsilon = 10^{-6}$ verwenden oder beide Werte selbst vorgeben möchte. Im zweiten Fall erhält die Variable `use_defaults` den Wert 0 und innerhalb der Funktion `CGLS` wird eine Eingabeaufforderung für die beiden Werte freigeschaltet. Anderenfalls wird `use_defaults` auf 1 gesetzt. Die in der Funktion `CGLS` verwendeten Variablen lauten wie folgt:

```
vector<T> r(A.size());
vector<T> s(A_T.size());
vector<T> d(A_T.size());
vector<T> Ax(A.size());
vector<T> A_Tr(A_T.size());
```

```
vector<T> Ad(A.size());
T alpha_k;
T beta_k;
T norm2_s_k;
T norm2_s_kplus1;
T norm2_Ad_k;
T residual;
```

Mit Hilfe dieser Variablen wurde der Algorithmus aus Kapitel 2.5 eins zu eins umgesetzt.

3.6.4 Ablauf eines Approximationsschrittes

Bei der Wahl der Approximation mit Default-Werten aus dem Kontextmenü ruft die main-Funktion über `ccMeshes.back().Approximation(1)` die Approximationsfunktion in der Klasse `CCaClMesh` auf. Wählt man dagegen die Approximation mit Eingabemöglichkeit für die maximale Iterationsanzahl und die Toleranz, wird statt der 1 als Parameter die 0 übergeben. Der daraufhin auszuführende Code in der Klasse `CCaClMesh` lautet:

```
void CCaClMesh::Approximation(int use_defaults)
{
    if (limit_pts.size() == 0) {
        cout « „Limit points have to be computed first!“ « endl;
        return;
    }
    SurfacePoints();
    vector<CVertex> x_0(vertices.size());
    for (int i=0; i<x_0.size(); i++)
        x_0[i].SetVals(vertices[i].Vec());
    CGLS(matrix, matrix_T, x_0, surface_pts, use_defaults);
    for (int i=0; i<vertices.size(); i++)
        vertices[i].SetVals(x_0[i].Vec());
    GenerateGLLists();
}
```

Zunächst wird geprüft, ob die Limitpunkte bereits berechnet wurden. Ist dies nicht der Fall, wird der Benutzer dazu aufgefordert und die Approximationsfunktion wird ergebnislos beendet. Sind die Limitpunkte bekannt, werden als erstes die Surfacepunkte ermittelt. Anschließend wird der Startvektor $\mathbf{x}^{(0)}$ für die CGLS-Methode angelegt und erhält als Werte die Vertices des aktuellen Meshes. Mit den beiden Koeffizientenmatrizen `matrix` und `matrix_T`, dem Startvektor `x_0`, den gerade berechneten Surfacepunkten `surface_pts` und dem „Schalter“ für die Default-Werte `use_defaults` wird die CGLS-Funktion aufgerufen. Diese berechnet den Vektor `x_0` iterativ neu. Nach Abschluss der CGLS-Methode werden die Vertices mit den Werten aus `x_0` überschrie-

ben und das dadurch veränderte Mesh über die Funktion `GenerateGLLists()` grafisch neu dargestellt.

3.7 Ausgabe der Gittergeometrie für den B-Spline-Gittergenerator GNAGG

Um ein mit dem hier entwickelten Programm generiertes Gitter für eine mit dem aus dem Sonderforschungsbereich 401 stammenden Strömungslöser QUADFLOW (für eine kurze Beschreibung s. Kap. 5.3) durchgeführte adaptive Rechnung verwenden zu können, muss die fertige Geometrie so ausgegeben werden, dass der institutseigene B-Spline-Gittergenerator GNAGG die Geometrie einlesen kann. GNAGG stellt einige elementare Funktionen zur Verfügung, um B-Spline-Kurven und -Flächen einzulesen bzw. sie interaktiv zu konstruieren und sie zu einem Multiblock-Gitter zu verbinden, das QUADFLOW verwenden kann. Ein Multiblock-Gitter ist eine Kombination von mehreren Blöcken, die in sich strukturiert sind. Jeder Block kann also als ein strukturiertes Gitter angesehen werden, wodurch der Übergang von einem Block zu einem anderen eine spezielle Behandlung erfordert, um die einzelnen Blöcke zu einem Gesamtgitter zusammenzusetzen. Ein strukturiertes Gitter zeichnet sich durch eine geordnete Nummerierung der Vertices aus. Für GNAGG ist eine Sortierung erforderlich, wie sie in Abbildung 3.18 beispielhaft für den Würfel veranschaulicht ist. Diese Sortierung muss erst herbeigeführt werden, da bei der Durchführung von Subdivision-Schritten oder bei der Limitpunktberechnung eine andere Sortierung entsteht

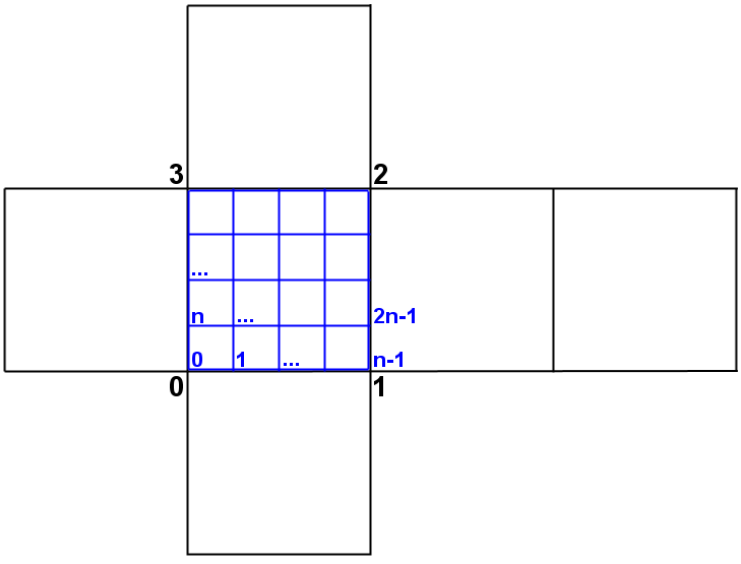

Abbildung 3.18: Aufteilung des Würfels und damit auch der später approximierten Kugel in sechs den Würfelfaces entsprechende Blöcke (schwarz): Demonstriert wird die Sortierung der Vertices bzw. Limitpunkte (blau) anhand eines der Blöcke, die beim ersten Vertex dieses Startfaces (schwarze 0) beginnt und beim Vertex mit der schwarzen 2 endet.

(s. Abb. 3.4). Der Würfel besteht aus sechs Faces, dementsprechend soll auch die durch mehrere Unterteilungen und Approximationsschritte entstehende Kugel sechs Blöcke enthalten. Jeder Block für sich erhält, wie in der Abbildung an einem der Blöcke demonstriert, die dargestellte Vertexsortierung.

Implementiert wurde allerdings nicht die blockweise, sortierte Ausgabe der aktuellen Vertices, sondern der von diesem Level aus berechneten Limitpunkte. Damit muss zunächst einmal klar sein, zu welchem der sechs Startfaces des Würfels jeder der Kugellimitpunkte gehört. Der Würfel besteht bereits zu Beginn nur aus Viereckfaces. Ist dies bei einer Gittergeometrie nicht der Fall, wird dieser Status nach einem Subdivision-Schritt erreicht. Als Ausgangspunkt für die Bildung der sortierten Blöcke für GNAGG wird immer das erste reine Viereckgitter gewählt. Hierfür erhält die Klasse `CFace` eine private Variable `_start_face`, damit jedes Face und damit auch jeder Vertex bzw. Limitpunkt auf jeder Unterteilungsstufe immer seinem Startface, aus dem es bzw. er hervorgegangen ist, zugeordnet werden kann. Die Variable `_start_face` wird aber erst dann gesetzt, wenn ein Gitter wie gerade beschrieben ausschließlich aus Vierecken besteht. Ist sie noch nicht gesetzt, enthält sie den Wert -1. Der Code zum Aufruf der Konvertierungsfunktion `ConvertToGNAGG(CCaClMesh& start_mesh)`, die in der Klasse `CCaClMesh` definiert wurde, aus der main-Funktion heraus lautet folgendermaßen:

```
list<CCaClMesh>::iterator iter;
for (iter=ccMeshes.begin(); iter!=ccMeshes.end(); ++iter)
    if ((*iter).GetFace(0).GetStartFace() != -1) {
        ccMeshes.back().ConvertToGNAGG(*iter);
        break;
    }
if (iter == ccMeshes.end())
    cout « „At least one subdivision has to be done first!“ « endl;
```

Die Mesh-Liste wird also nach dem ersten Gitter durchsucht, dessen Faces bereits ihre Startfaces zugewiesen bekommen haben. Dieses wird der Funktion `ConvertToGNAGG(CCaClMesh& start_mesh)` als Startmesh übergeben. Wird kein solches Gitter in der Mesh-Liste gefunden, ist noch kein reines Viereckgitter vorhanden und der Benutzer wird zur Durchführung eines Subdivision-Schrittes aufgefordert.

In der Funktion `ConvertToGNAGG(CCaClMesh& start_mesh)` wird zunächst überprüft, ob bereits Limitpunkte berechnet worden sind, da diese für GNAGG ausgegeben werden sollen. Ist dies nicht der Fall, wird der Benutzer dazu aufgefordert. Danach wird durch Absuchen der Nachbarschaften aller Faces, die sich in einem Startface befinden, die Sortierung ermittelt und die entsprechenden Limitpunkte in der dabei bestimmten Reihenfolge in eine Textdatei im Tecplot-Format geschrieben. Der Anwender kann dabei wählen, ob er alle oder nur einen der Blöcke ausgeben will, was in Abbildung 3.19 am Beispiel der Kugel zu sehen ist. Die blockweise Ausgabe wurde folgendermaßen realisiert:

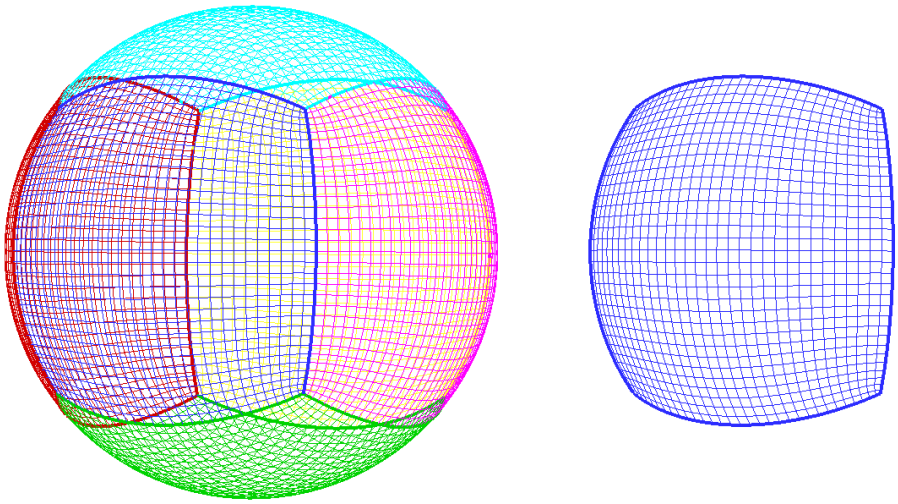

Abbildung 3.19: Darstellung einer Kugel (links) bzw. eines Kugelteils (rechts), entstanden über die Ausgabefunktion der Geometrie für GNAGG mit der Auswahl aller Blöcke (links) bzw. nur eines Blocks (rechts). Die Blöcke entsprechen jeweils einem der sechs Faces des Würfels, der für die Kugelapproximation als Ausgangspunkt gewählt wurde.

```
list<int> limit_pts_GNAGG;
cout « „Which block do you want to write out? (no. 1-‟ «
        start_mesh.GetNumF() « „ or 0 for all) ‟;
cin » block;
for (int nf=0; nf<start_mesh.GetNumF(); nf++) {
    if (block == 0 || nf == block-1) {
        limit_pts_GNAGG.clear();
        ⋮    // Absuchen der Nachbarschaften und
        ⋮    // Füllen von limit_pts_GNAGG
        for (iter=limit_pts_GNAGG.begin();
            iter!=limit_pts_GNAGG.end(); ++iter) {
                ⋮    // Ausgabe der Limitpunkte in Textdatei
        }
    }
}
```

Die Liste `limit_pts_GNAGG` dient für jeden Block zum zwischenzeitlichen Speichern der sortierten Limitpunkte, bevor diese am Ende eines `for`-Schleifendurchgangs in die Textdatei geschrieben werden und die Liste für den nächsten Durchlauf wieder freigegeben wird. Das Absuchen der Nachbarschaften beginnt für jedes Startface beim Limitpunkt mit der im Array `_ivertices` dieses Startfaces an erster Stelle gespeicherten Vertexnummer:

```
start_vert = start_mesh.faces[nf].GetVertex(0);
```

3.8 Weitere Funktionen

3.8.1 Fehlerberechnung zwischen Limitpunkten und den entsprechenden nach einem oder mehreren auf die Limitpunktberechnung folgenden Subdivision-Schritten neu berechneten Vertices

Um den Effekt der Vereinfachungen, die bei der Limitpunktberechnung im Bereich der inneren tagged Edges teilweise getroffen wurden (s. Kap. 3.5.5), überprüfen zu können, wurde eine Funktion zur Berechnung der prozentualen Abweichung der Limitpunkte von ihren entsprechenden Vertices implementiert. Die zu verwendenden Vertices sind Punkte, die nach mindestens einem oder aber mehreren auf die Limitpunktberechnung folgenden Subdivision-Schritten berechnet worden sind. Die prozentuale Abweichung sollte mit jeder Unterteilung sinken, da die Limitpunkte bekanntermaßen theoretisch den Subdivisionpunkten nach unendlich vielen Unterteilungen entsprechen. Um in dem ein- oder mehrfach unterteilten Mesh auf die auf dem gröberen Level ermittelten Limitpunkte zugreifen zu können, wurde in der Klasse `CCaClMesh` eine statische Klassenvariable `static list<CVertex> LP_total_0` angelegt, die immer die zuletzt berechneten Limitpunkte enthält. Die Formel für die prozentuale Abweichung des Limitpunkts $\mathbf{L}_i$ vom entsprechenden Vertex im weiter unterteilten Mesh $\mathbf{V}_i$ lautet dann

$$f_i = \frac{\|L_i - V_i\|_2}{\|V_i\|_2} \cdot 100. \tag{3.8}$$

Nach Aufruf der Fehlerberechnung aus dem Kontextmenü wird zunächst überprüft, ob nach der Limitpunktberechnung bereits mindestens ein weiterer Subdivision-Schritt durchgeführt wurde:

```
if (CCaClMesh::LP_total_0.size() == NumV+NumF+NumE) {
     cout « „At least one subdivision has to be done first!" « endl;
     return;
}
```

Ist mindestens eine Unterteilung vorgenommen worden, wird f_i für jeden in `LP_total_0` vorhandenen Limitpunkt berechnet und im Ausgabefenster aufgelistet. Um größere Abweichungen direkt erkennen zu können, werden bei solchen, die über 10 % betragen, zusätzlich zur Abweichung die jeweiligen Limitpunkt- und Vertex-Koordinaten zum direkten Vergleich ausgegeben.

3.8.2 Fehlerberechnung zwischen Surfacepunkten und den entsprechenden Limitpunkten

Diese Funktion wurde zur Untersuchung der Konvergenzordnung der Approximation implementiert. Die Approximationsaufgabe lautet bekanntlich $\|A\mathbf{V} - \mathbf{S}\|_2 = \|\mathbf{L} - \mathbf{S}\|_2 \rightarrow min$. Um die Güte dieser Minimierung zu überprüfen, wurden zwei verschiedene Fehler zwischen den Limit-

und den Surfacepunkten betrachtet, der maximale Fehler

$$max\left(\|\mathbf{L}_i - \mathbf{S}_i\|_2\right) \qquad \forall\ i = 0, \ldots, \#LP - 1 \tag{3.9}$$

und der gemittelte Fehler

$$\frac{1}{\#LP} \sum_{i=0}^{\#LP-1} \|\mathbf{L}_i - \mathbf{S}_i\|_2, \tag{3.10}$$

jeweils mit der Anzahl der Limitpunkte $\#LP = \#V + \#E + \#F$. Diese Berechnungen werden im Code in der main-Funktion gestartet. Hierbei wird zunächst überprüft, ob bereits eine Approximation erfolgt ist und Surfacepunkte vorliegen. Ist dies nicht der Fall, wird der Benutzer zur Durchführung eines Approximationsschrittes aufgefordert. Anschließend wird die Funktion `Error_LP_SP()` in der Klasse `CCaClMesh` aufgerufen, die folgendermaßen aussieht:

```
double error_max = 0.;
double error_sum = 0.;
int max_pos = 0;
for (int i=0; i<surface_pts.size(); i++) {
    CVector3d error(fabs(limit_pts[i].x() - surface_pts[i].x()),
                    fabs(limit_pts[i].y() - surface_pts[i].y()),
                    fabs(limit_pts[i].z() - surface_pts[i].z()));
    if (error.Norm2() > error_max) {
        error_max = error.Norm2();
        max_pos = i;
    }
    error_sum = error_sum + error.Norm2();
}
error_sum = error_sum / limit_pts.size();
```

`error_max` steht für den maximalen und `error_sum` für den gemittelten Fehler, `max_pos` gibt die Nummer des Limitpunkts mit dem maximalen Fehler an.

Für die Approximation der Kugel mit dem Radius $R = 1$ mit einem Würfel als Startgitter erhält man bei Verwendung der CGLS-Toleranz $\epsilon = 10^{-6}$ die in Tabelle 3.1 aufgelisteten Fehler. Wegen der Symmetrie des Gitters tritt der maximale Fehler jeweils an mehreren Stellen auf. Bei Level 0 und 1 liegt er jeweils in der Mitte der zwölf Verbindungslinien der extraordinary Vertices, ab Level 2 an der Stelle der Mittelpunkte der ursprünglichen sechs Würfelfaces. In der Tabelle ist auch der Faktor der Verringerung der Fehler von Level zu Level angegeben. Hier sieht man, dass die Reduzierung sowohl des maximalen als auch des gemittelten Fehlers gegen den Faktor 4 geht. Formal beschrieben ist h^p der bestimmende Wert für die Konvergenzordnung p (s. z.B. [13]). h ist die maximale Schrittweite der Kontrollpunkte, die sich in diesem äquidistanten Fall mit jeder Unterteilung halbiert. Mit der Viertelung des Fehlers gilt also: $\left(\frac{1}{2}\right)^p = \frac{1}{4} \Leftrightarrow p = 2$. Dieses Ergebnis lässt allgemein eine quadratische Konvergenzordnung der Approximation erwarten.

	maximaler Fehler nach Gl. (3.9)	Reduzierungsfaktor	gemittelter Fehler nach Gl. (3.10)	Reduzierungsfaktor
Level 0	$0,24$	-	$0,21$	-
Level 1	$1,56 \cdot 10^{-2}$	$15,33$	$5,71 \cdot 10^{-3}$	$36,89$
Level 2	$4,51 \cdot 10^{-3}$	$3,46$	$1,04 \cdot 10^{-3}$	$5,49$
Level 3	$6,41 \cdot 10^{-4}$	$7,03$	$2,58 \cdot 10^{-4}$	$4,04$
Level 4	$1,47 \cdot 10^{-4}$	$4,36$	$6,41 \cdot 10^{-5}$	$4,02$
Level 5	$3,55 \cdot 10^{-5}$	$4,15$	$1,60 \cdot 10^{-5}$	$4,01$
Level 6	$8,78 \cdot 10^{-6}$	$4,04$	$4,00 \cdot 10^{-6}$	$4,00$

Tabelle 3.1: Entwicklung des maximalen und des gemittelten Fehlers bei der Approximation der Kugel mit dem Radius $R = 1$ mit einem Würfel als Startgitter. Die CGLS-Toleranz beträgt $\epsilon = 10^{-6}$.

3.8.3 Manuelle Änderung von Vertexkoordinaten

Diese Funktion wurde implementiert, um während der Ausführung des Programms Vertexkoordinaten manuell anpassen zu können. Hierbei wird der Benutzer zunächst aufgefordert, die Anzahl der zu ändernden Vertices einzugeben. Danach werden so lange jeweils die Vertexnummer und die zugehörigen Koordinaten x, y und z abgefragt, bis diese Anzahl erreicht ist. Wofür diese Funktion nützlich sein kann, wird sich in Kapitel 5.2 zeigen.

4 Anwendung des entwickelten Programms

4.1 „V“- und „W“-Anwendungsprofil

Über die von einem Würfel ausgehende Approximation der in der Funktion `SurfacePoints()` definierten Einheitskugel (s. Kap. 3.6.2) sollen zwei verschiedene Vorgehensweisen bei der Approximationsanwendung des Programms zur Gittererzeugung verglichen werden, das „V“-Profil oben und das „W“-Profil unten in Abbildung 4.1. Die Bezeichnungen sind an Begriffe aus dem Bereich der Mehrgitterverfahren angelehnt und folgen aus der in der Abbildung zu erkennenden Struktur, die sich durch die zyklische Vorgehensweise ergibt. Beim „V“-Profil führt man so lange Subdivision-Schritte durch, bis man die gewünschte Unterteilungsstufe erreicht hat. Anschließend kann man die Limitpunkte berechnen lassen und die gegebene Oberfläche approximieren. Beim „W“-Profil dagegen wird auf Level 0 mit der Limitpunktberechnung und anschließender Approximation begonnen, erst danach folgt der erste Subdivision-Schritt, um das nächste Level zu erreichen. Anschließend geht es im Wechsel weiter, d.h., es werden nie zwei oder mehr

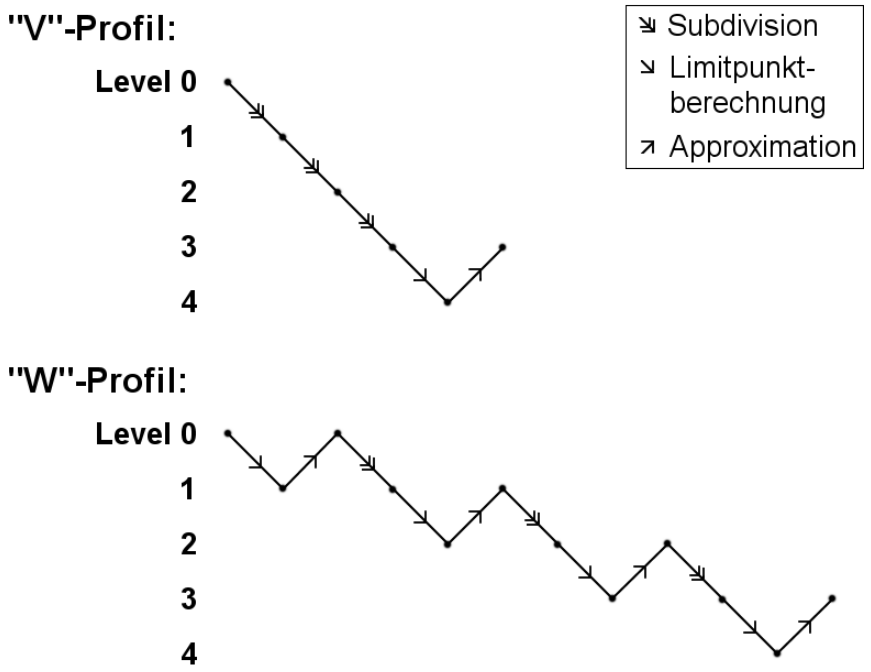

Abbildung 4.1: Mögliche Vorgehensweisen bei der Approximationsanwendung des Programms zur Gittererzeugung

Subdivision-Schritte direkt hintereinander ausgeführt. Dadurch müssen Limitpunktberechnung und Approximation im Gegensatz zum „V“-Profil mehrfach, nämlich auf unterschiedlichen Unterteilungsstufen, erfolgen. Auf den ersten Blick scheint die Anwendung des „W“-Profils komplexer und zeitaufwändiger zu sein, weil mehr Berechnungen notwendig sind. Dies ist auch in Tabelle 4.1 zu erkennen, die für das angesprochene Beispiel der von einem Würfel ausgehenden Kugelapproximation die Anzahl der benötigten Iterationen bei der CGLS-Methode mit einer Toleranz von $\epsilon = 10^{-6}$ angibt. Unterteilt man insgesamt bis Level 6, wird beim „V“-Profil erst nach diesen Unterteilungen der einzige Approximationsschritt ausgeführt. Dieser erreicht mit den aktuellen Vertices als Startvektor nach 80 CGLS-Iterationen die Toleranzgrenze, wobei auf dem eingesetzten Rechner (Einkern-CPU mit 2,2 GHz, 1 GB RAM) ungefähr 16 Sekunden für eine Iteration benötigt werden. Die Laufzeit beträgt also $80 \cdot 16s = 1280s \approx 21,3min$. Beim „W“-Profil dagegen wird auf jedem Level eine Approximation vorgenommen (ebenfalls jeweils mit den aktuellen Vertices als Startvektor), was dazu führt, dass insgesamt mit 168 mehr als doppelt so viele Iterationen nötig sind wie beim „V“-Profil. Die Laufzeit berechnet sich bei Vernachlässigung des sehr geringen Zeitaufwands von Level 0 bis 2 zu $48 \cdot 0,25s + 37 \cdot 1s + 20 \cdot 4s + 15 \cdot 16s = 369s \approx 6,2min$. Die Anwendung dieses Profils erfordert also weniger als ein Drittel der Gesamtzeit des „V“-Profils, da die Anzahl der Iterationen beim „W“-Profil auf den hohen rechen- und damit zeitintensiven Level sinkt. Das liegt daran, dass die Kugel durch vorhergehende Approximationen auf gröberen Level bereits gut angenähert ist.

	# Vertices	Dauer/Iter.	# Iter. „V“	# Iter. „W“	Ges.zeit „V“	Ges.zeit „W“
Level 0	26	$\ll 0,1$ s	2	2	$\ll 1$ s	$\ll 1$ s
Level 1	98	$< 0,1$ s	8	9, insg. 11	< 1 s	< 1 s
Level 2	386	$< 0,1$ s	38	37, insg. 48	< 2 s	< 2 s
Level 3	1538	ca. 0,25 s	70	48, insg. 96	ca. 18 s	ca. 12 s
Level 4	6146	ca. 1 s	77	37, insg. 133	ca. 77 s	ca. 49 s
Level 5	24578	ca. 4 s	80	20, insg. 153	ca. 320 s	ca. 129 s
Level 6	98306	ca. 16 s	80	15, insg. 168	ca. 1280 s	ca. 369 s

Tabelle 4.1: Vergleich der benötigten CGLS-Iterationen (jeweils aktuelle Vertices als Startvektor, Toleranz $\epsilon = 10^{-6}$) bei Verwendung des „V“- bzw. des „W“-Profils zur Approximation der Einheitskugel, ausgehend von einem Würfel. Der Zeitaufwand pro Iteration wächst mit jedem Level aufgrund der steigenden Vertex-Anzahl. Die Sekundenangaben beziehen sich auf den eingesetzten Rechner.

4.2 Beispielablauf bei der Anwendung

Analog zu der bereits mehrfach erwähnten Kugelapproximation mit einem Würfel als Startgitter soll hier aufgrund der besseren Darstellbarkeit im Zweidimensionalen ein Beispielablauf für die Anwendung des Programms zur Approximation eines Kreises mit dem Radius $R = 0,8$, ausgehend von einem Quadrat, gezeigt werden. Die Surfacepunktberechnung aus Kapitel 3.6.2 kann mit z-Koordinate gleich 0 übernommen werden, es muss allerdings darauf geachtet werden, dass

die Surfacepunkte nur für die auf der Berandung des Gitters liegenden Limitpunkte berechnet werden, da der Rand analog zur Kugeloberfläche die Kreis"oberfläche" darstellt. Anderenfalls würden sämtliche Vertices im Inneren des Gitters durch die Surfacepunktberechnung auf den Rand verschoben. Ein möglicher Ablauf bei der Verwendung des Programms gestaltet sich dann folgendermaßen:

1. Vorgabe der Geometrie: Die in Abbildung 4.2 gezeigte Textdatei ähnelt der aus Abbildung 3.7. Allerdings ist hier keines der Edges tagged. Die Geometrie, die sich daraus ergibt, ist in Abbildung 4.4 mit Angabe der Vertex-, Edge- und Face-Nummerierungen dargestellt. Zu beachten ist hierbei, dass die Nummerierungen in der Textdatei jeweils mit 1 beginnen und im Programm mit 0. Die linksdrehenden Nachbarschaftsverhältnisse lassen sich aus Abbildung 4.3 ablesen, die einen Ausschnitt des Ausgabefensters direkt nach dem Programmstart zeigt. So sind beispielsweise für die acht Vertices bei „ENr" und „FNr" die jeweils angrenzenden Edges bzw. Faces in einer linksdrehenden Weise angegeben. Gleiches gilt bei den fünf Faces für die unter „VNr" und „ENr" angegebenen angrenzenden Vertices bzw. Edges.

2. Durchführung eines Subdivision-Schrittes, um Level 0 zu erreichen: Das Startgitter befindet sich auf Level -1, d.h., es sind noch nicht alle extraordinary Vertices isoliert. Damit die Limitpunktberechnung möglich ist, muss dies aber der Fall sein. Deshalb ist ein Subdivision-Schritt nötig, nach dem Level 0 erreicht wird. Die daraus folgende Gittergeometrie ist in Abbildung 4.5 zu sehen. Wiederum sind die Vertex-, Edge- und Face-Nummerierungen angegeben, um die durch eine Unterteilung erfolgte Neunummerierung (s. Abb. 3.4) nachvollziehen zu können.

3. Anwendung des „W"-Profils, um den Kreis zu approximieren:

   a) Limitpunktberechnung

   b) Approximation

   c) Subdivision

   In diesem Beispiel soll als Verfeinerungsstufe Level 3 erreicht werden. Dafür müssen die Schritte a), b) und c) insgesamt drei Mal jeweils hintereinander ausgeführt werden. Anschließend folgen noch eine Limitpunktberechnung und ein Approximationsschritt. Es ergibt sich das Gitter aus Abbildung 4.6. Im Gegensatz zum Gitter aus Abbildung 4.7, das durch reine Subdivision bis Level 3 entsteht, ist es tatsächlich kreisrund, zu erkennen an den schwarzen Kreisen, die grafisch über die beiden Gitter gelegt wurden.

4. Ausgabe der Geometrie für GNAGG: Die durch die Ausgabe aller Blöcke generierte Tecplot-Geometrie ist in Abbildung 4.8 zu sehen. Mit GNAGG kann das so erstellte Gitter dann zu einem B-Spline-Gitter weiterverarbeitet werden.

```
    1   -0.9 -0.9  0.0    0
    2   -0.9  0.9  0.0    0
    3    0.9  0.9  0.0    0
    4    0.9 -0.9  0.0    0
    5   -0.3 -0.3  0.0    0
    6   -0.3  0.3  0.0    0
    7    0.3  0.3  0.0    0
    8    0.3 -0.3  0.0    0
0
    1    4     1 5 6 2    1
    2    4     2 6 7 3    1
    3    4     3 7 8 4    1
    4    4     4 8 5 1    1
    5    4     5 8 7 6    1
0
0
```

Abbildung 4.2: Beispielablauf, zu 1.: Vorgabe der Quadrat-Geometrie

```
                                                    8 vertices
V  0: (-0.9000,-0.9000, 0.0000)  3 E,  2 F : ENr:  0, 3,11,    FNr:  3, 0,
V  1: (-0.9000, 0.9000, 0.0000)  3 E,  2 F : ENr:  2, 6, 3,    FNr:  0, 1,
V  2: ( 0.9000, 0.9000, 0.0000)  3 E,  2 F : ENr:  5, 9, 6,    FNr:  1, 2,
V  3: ( 0.9000,-0.9000, 0.0000)  3 E,  2 F : ENr:  8,11, 9,    FNr:  2, 3,
V  4: (-0.3000,-0.3000, 0.0000)  3 E,  3 F : ENr:  0,10, 1,    FNr:  0, 3, 4,
V  5: (-0.3000, 0.3000, 0.0000)  3 E,  3 F : ENr:  1, 4, 2,    FNr:  0, 4, 1,
V  6: ( 0.3000, 0.3000, 0.0000)  3 E,  3 F : ENr:  4, 7, 5,    FNr:  1, 4, 2,
V  7: ( 0.3000,-0.3000, 0.0000)  3 E,  3 F : ENr:  7,10, 8,    FNr:  2, 4, 3,

                                                    5 faces
F  0:  4 V,  4 E : VNr:  0, 4, 5, 1, ENr:  0, 1, 2, 3,
F  1:  4 V,  4 E : VNr:  1, 5, 6, 2, ENr:  2, 4, 5, 6,
F  2:  4 V,  4 E : VNr:  2, 6, 7, 3, ENr:  5, 7, 8, 9,
F  3:  4 V,  4 E : VNr:  3, 7, 4, 0, ENr:  8,10, 0,11,
F  4:  4 V,  4 E : VNr:  4, 7, 6, 5, ENr: 10, 7, 4, 1,

                                                  12 edges:
edge  0  :  v1= 0, v2= 4, left face= 0, right face= 3, tag= 0
edge  1  :  v1= 4, v2= 5, left face= 0, right face= 4, tag= 0
edge  2  :  v1= 1, v2= 5, left face= 1, right face= 0, tag= 0
edge  3  :  v1= 0, v2= 1, left face=-1, right face= 0, tag= 1
edge  4  :  v1= 5, v2= 6, left face= 1, right face= 4, tag= 0
edge  5  :  v1= 2, v2= 6, left face= 2, right face= 1, tag= 0
edge  6  :  v1= 1, v2= 2, left face=-1, right face= 1, tag= 1
edge  7  :  v1= 6, v2= 7, left face= 2, right face= 4, tag= 0
edge  8  :  v1= 3, v2= 7, left face= 3, right face= 2, tag= 0
edge  9  :  v1= 2, v2= 3, left face=-1, right face= 2, tag= 1
edge 10  :  v1= 4, v2= 7, left face= 4, right face= 3, tag= 0
edge 11  :  v1= 0, v2= 3, left face= 3, right face=-1, tag= 1

overall 8 vertices, 5 faces, 12 edges
subdivision steps = 0, level = -1
```

Abbildung 4.3: Beispielablauf, zu 1.: Ausschnitt aus dem Ausgabefenster direkt nach dem Programmstart. Sämtliche Vertices, Edges und Faces werden mit Angabe ihrer Nachbarschaften aufgelistet.

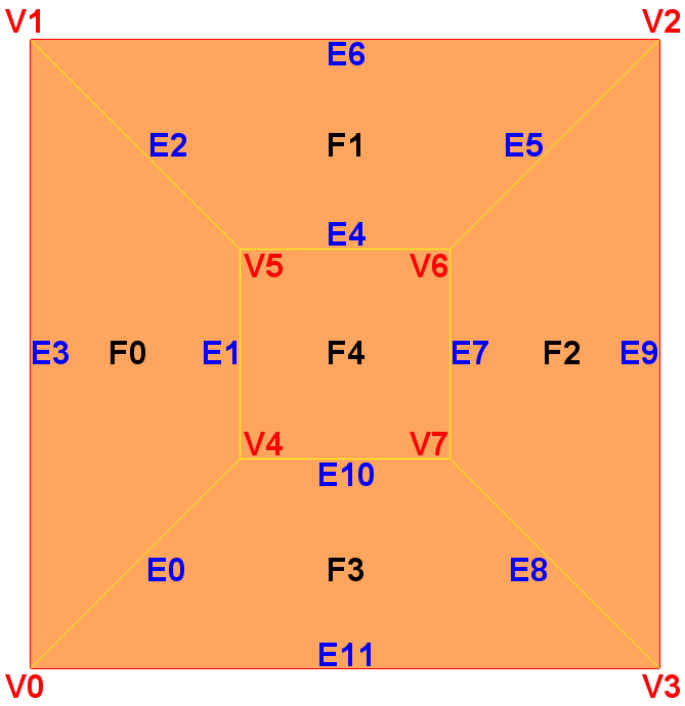

Abbildung 4.4: Beispielablauf, zu 1.: Darstellung der Geometrie auf Level -1 (direkt nach dem Programmstart) mit eingetragener Vertex-, Edge- und Face-Nummerierung

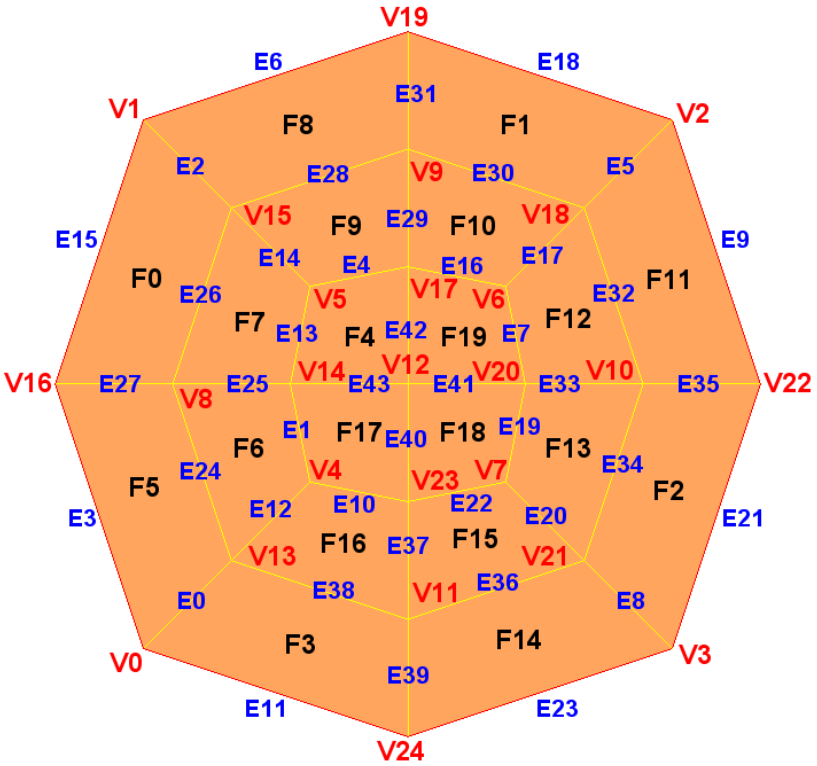

Abbildung 4.5: Beispielablauf, zu 2.: Darstellung der Geometrie auf Level 0 (nach einem Subdivision-Schritt) mit eingetragener Vertex-, Edge- und Face-Nummerierung. Die Neunummerierung ergibt sich durch die Anwendung der Regeln aus Abb. 3.4 auf die Geometrie des Levels -1 in Abb. 4.4.

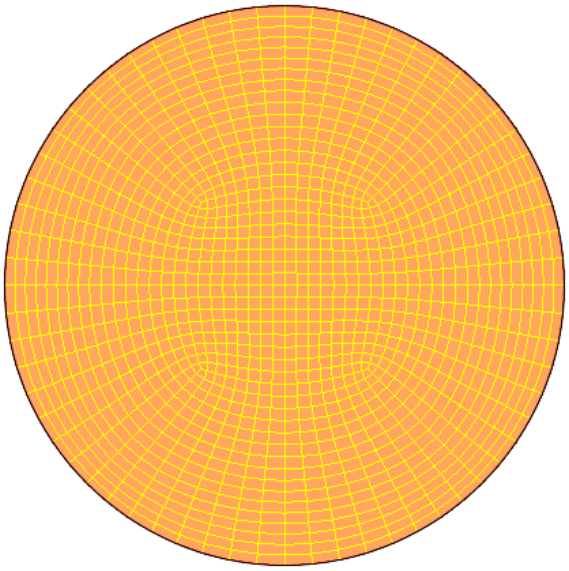

Abbildung 4.6: Beispielablauf, zu 3.: Nach dreimaliger Anwendung des „W“-Profils approximierter Kreis mit dem Radius $R = 0,8$

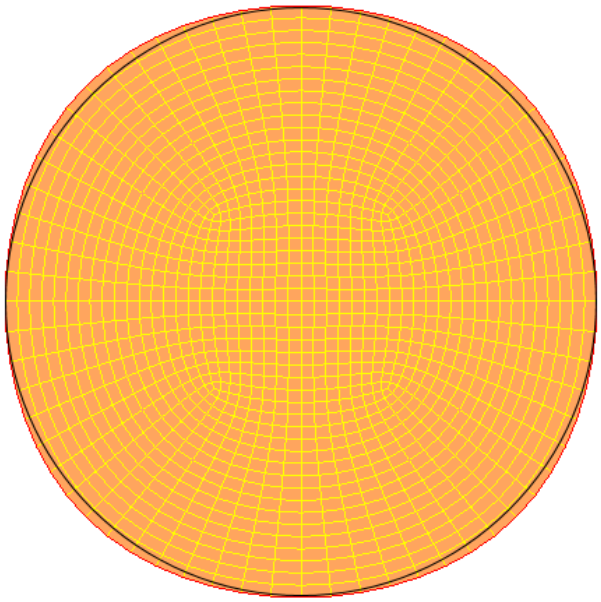

Abbildung 4.7: Ohne Approximation bis Level 3 unterteiltes Gitter, nicht exakt kreisrund

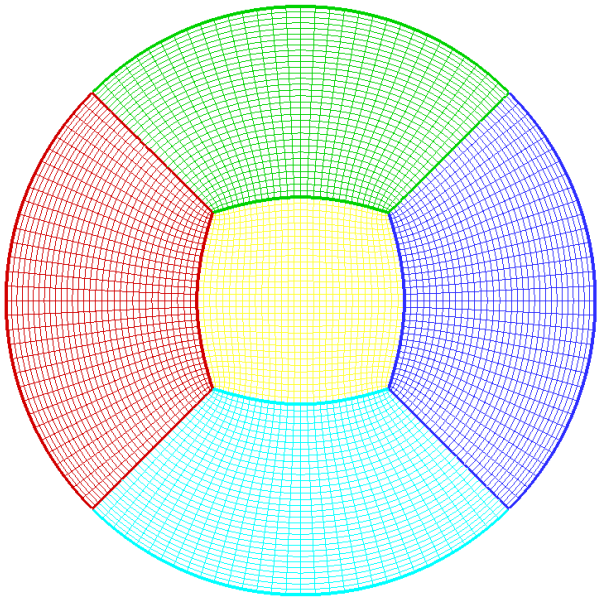

Abbildung 4.8: Beispielablauf, zu 4.: Approximierter Kreis in Tecplot-Darstellung; die fünf farblich unterschiedlich gekennzeichneten Blöcke sind aus den fünf Viereckfaces des Anfangsgitters entstanden.

5 Anwendung zur Strömungssimulation

Anhand der Simulation eines Strömungsbeispiels soll in diesem Kapitel gezeigt werden, wie ein mit dem entwickelten Programm generiertes Gitter praktisch eingesetzt werden kann. Hierzu muss das über mehrere Subdivision- und Approximationsschritte entstandene Oberflächengitter zunächst zu einem Volumengitter erweitert werden (s. Kap. 5.2).

5.1 Beschreibung des Testfalls

Als Testfall soll die Überschallanströmung einer Kugel betrachtet werden. Es bildet sich ein Verdichtungsstoß, der, wie immer bei stumpfen Körpern, von der Kugeloberfläche abgelöst ist. Der Abstand zwischen Stoßlage und Staupunkt hängt hierbei von der Anström-Mach-Zahl $M_\infty = \frac{u_\infty}{c_\infty}$ mit der Strömungsgeschwindigkeit u_∞ und der Schallgeschwindigkeit c_∞ ab. Je größer die Mach-Zahl M_∞ vor dem Stoß ist, desto kleiner wird der Stoßabstand.

Eine Analyse des abgelösten Verdichtungsstoßes kann anhand von Abbildung 5.1, die in ähnlicher Form in [14] zu finden ist, vorgenommen werden: Im Punkt 'a' steht die Stoßwelle senk-

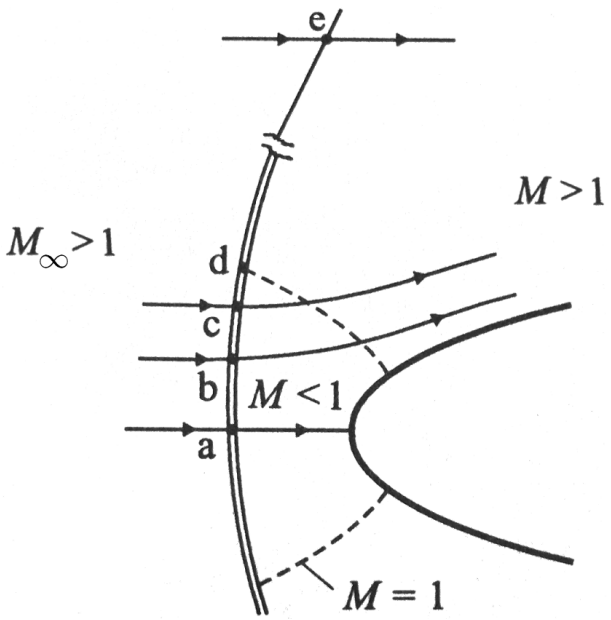

Abbildung 5.1: Abgelöster Verdichtungsstoß bei der Überschallanströmung eines stumpfen Körpers (vgl. [14])

recht zum Staupunkt, so dass die Strömung keine Umlenkung erfährt. In großer Entfernung vom stumpfen Körper wird der Verdichtungsstoß immer schwächer, weshalb die Strömung im Punkt 'e' ebenfalls nicht umgelenkt wird. Zwischen diesen beiden Punkten 'a' und 'e' ergeben sich sämtliche mögliche Lösungen für schräge Verdichtungsstöße. Im Punkt 'b' wird die Strömung leicht nach oben abgelenkt. Geht man von Punkt 'b' weiter in Richtung 'c', so erhöht sich die Umlenkung, bis sie im Punkt 'c' ihren maximalen Wert erreicht hat. Von Punkt 'd' aus liegt stromab des Stoßes der Schallzustand vor, der ein subsonisches Gebiet einschließt ($0.3 < M < 1$), während zwischen 'd' und 'e' die Strömung hinter dem Stoß supersonisch ist ($M > 1$).

5.2 Erstellung des Volumengitters für die Simulation

Für die Durchführung der Simulation wird ein Volumengitter benötigt, das die Anströmung der Kugel, den vor der Kugel entstehenden Stoß und das zwischen Stoß und Kugeloberfläche liegende Unterschallgebiet auflösen kann. Zunächst kann dafür mit dem hier entwickelten Programm die Kugeloberfläche erstellt werden. Den Ausgangspunkt hierfür bildet eine Würfelgeometrie, aus der durch die Anwendung des „W"-Profils eine Kugel mit dem Radius $R = 1$ approximiert wird. Für die Simulation wird nur der angeströmte Teil der Kugel benötigt. Also kann mit Hilfe der in Kapitel 3.7 beschriebenen GNAGG-Ausgabe einer der sechs Blöcke in eine Datei geschrieben werden. Wählt man diesen allerdings wie rechts in Abbildung 3.19, ist der modellierte Teil der Kugeloberfläche zu klein, da das zwischen Stoß und Kugel liegende Unterschallgebiet durch das später auf der Grundlage der Kugeloberfläche erstellte Volumengitter nicht vollständig erfasst werden kann. Aus diesem Grund wird die Funktion der manuellen Änderung von Vertexkoordinaten aus Kapitel 3.8.3 eingesetzt. Beim Würfel sind alle sechs Flächen gleich groß, weshalb dies auch für die sechs Blöcke der approximierten Kugel gilt (s. Abb. 3.19). Um einen der Blöcke zu vergrößern, werden nach einer Subdivision des Würfels einige Vertices versetzt, veranschaulicht

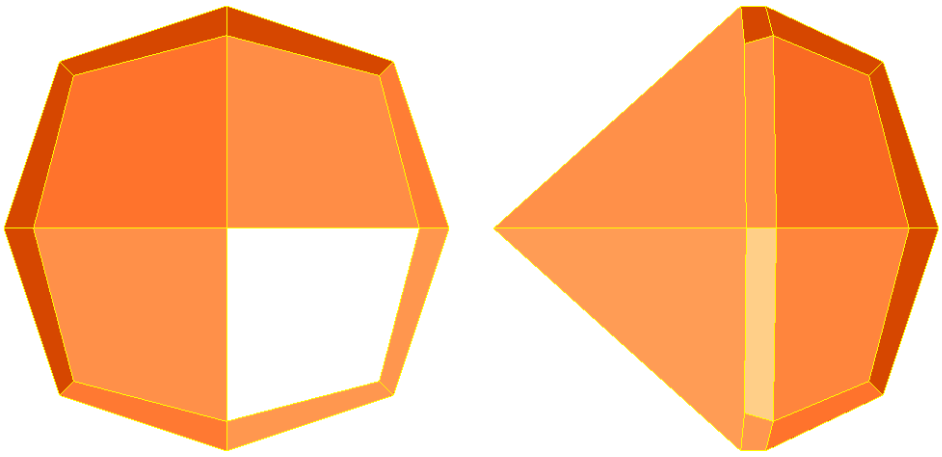

Abbildung 5.2: Ein Mal unterteilter Würfel vor (links) und nach manueller Vertexanpassung zur Vergrößerung eines der sechs Blöcke (rechts)

in Abbildung 5.2. Links in der Abbildung ist der ein Mal unterteilte Würfel zu sehen, rechts die neue Geometrie nach manueller Vertexanpassung. Hier erkennt man, dass die Vertices auf der mittleren vertikalen Linie und auch einige weiter links liegende Vertices nach rechts verschoben worden sind. Anschließend wird das „W“-Profil bis Level 5 angewendet, es werden also vier Subdivision- und fünf Approximationsschritte durchgeführt. Die auf diese Weise generierte Kugel ist in blockweiser Darstellung links in Abbildung 5.3 zu sehen, rechts ist der für die Erstellung des Volumengitters schließlich verwendete Block abgebildet, der $65 \cdot 65 = 4225$ Gitterpunkte enthält.

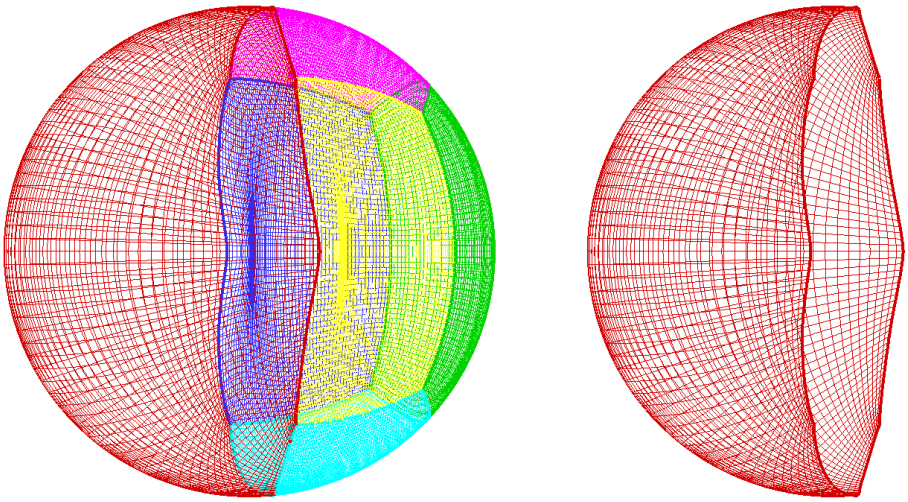

Abbildung 5.3: Approximierte Kugeloberfläche nach Anwendung des „W“-Profils bis Level 5: Links die gesamte, aus sechs Blöcken bestehende Oberfläche, rechts nur der für die Erstellung des Volumengitters verwendete, vorher nach Abb. 5.2 vergrößerte Block mit 4225 Gitterpunkten.

Aus dem Oberflächengitter des Kugelblocks kann mit Hilfe eines kleinen C++-Programms auf einfache Weise ein Volumengitter generiert werden. Hierbei sind vier wesentliche Schritte erforderlich:

1. Einlesen des Oberflächengitters: Die 65·65 Gitterpunkte werden eingelesen und in der dreidimensionalen Variable `grid` vom selbst definierten Typ `vektor3d` (ähnlich zu `CVector3d`, s. Kap. 3.1) an der Stelle `grid[i][j][0]` mit `i` und `j` jeweils von 0 bis 64 gespeichert.

2. Generierung einer Hyperboloidoberfläche: Die Gleichung für ein zweischaliges Hyperboloid lautet

$$\frac{y^2}{b_{Hyp}^2} + \frac{z^2}{c_{Hyp}^2} - \frac{(x - x_{Hyp})^2}{a_{Hyp}^2} = -1 \qquad \Leftrightarrow \qquad \frac{y^2}{b_{Hyp}^2} + \frac{z^2}{c_{Hyp}^2} - \frac{(x - x_{Hyp})^2}{a_{Hyp}^2} + 1 = 0, \quad (5.1)$$

   wobei der Parameter x_{Hyp} die Ausdehnung der Hyperboloidoberfläche in x-Richtung vorgibt. Die x-Achse ist in Abbildung 5.2, 5.3 und 5.4 die horizontale Achse. Über x_Hyp wird also auch der Abstand zwischen Kugel- und Hyperboloidoberfläche angepasst. Die

auf dem Hyperboloid liegenden Gitterpunkte werden jeweils über den Schnitt der Gerade durch den Ursprung und den entsprechenden Gitterpunkt auf der Kugeloberfläche mit dem Hyperboloid ermittelt, so dass die quadratische Gleichung

$$at^2 + bt + c = 0 \tag{5.2}$$

mit

$$\begin{aligned} a &= \frac{y^2}{b_{Hyp}^2} + \frac{z^2}{c_{Hyp}^2} - \frac{x^2}{a_{Hyp}^2}, \\ b &= \frac{2xx_{Hyp}}{a_{Hyp}^2}, \\ c &= 1 - \frac{x_{Hyp}^2}{a_{Hyp}^2} \end{aligned} \tag{5.3}$$

zu lösen ist. t ergibt sich aus den beiden Lösungen t_1 und t_2 zu $t = t_1$, falls $t_1 > 0$ gilt, sonst zu $t = t_2$. Die auf diese Weise berechneten Gitterpunkte (`t*grid[i][j][0].x`, `t*grid[i][j][0].y`, `t*grid[i][j][0].z`) auf der Hyperboloidoberfläche werden in

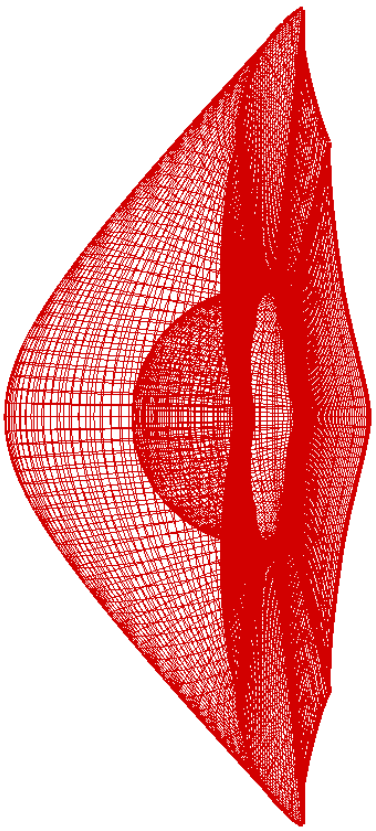

Abbildung 5.4: Für die Simulation erstelltes Volumengitter mit 278850 Gitterpunkten: In der Mitte ist die Kugeloberfläche aus Abb. 5.3 zu erkennen.

`grid[i][j][65]` gespeichert. `i` und `j` liegen hierbei wiederum zwischen 0 und 64.

3. Ausfüllen des Volumens zwischen Kugel- und Hyperboloidoberfläche mit Gitterpunkten: Das Volumen wird hier mit Hilfe einer Stretching-Funktion ausgefüllt. Diese sorgt dafür, dass das Gitter zur Kugeloberfläche hin immer feiner wird, um den Verdichtungsstoß und das dahinter liegende Unterschallgebiet gut auflösen zu können. Das Stretching ist nur für eine nicht-adaptiv ausgeführte Simulation notwendig. `grid[i][j][k]` wird mit `i` und `j` wie gehabt zwischen 0 und 64 und `k` zwischen 1 und 64 gesetzt zu `(1-s)*x0 + s*x1`, wobei `x0` dem jeweiligen Gitterpunkt `grid[i][j][0]` auf der Kugeloberfläche und `x1` dem jeweiligen Gitterpunkt `grid[i][j][65]` auf der Hyperboloidoberfläche entspricht. `s` wird berechnet über `k/65 * (m_stretch + k/65*(1-m_stretch))` mit dem auf 0.02 gesetzten Parameter `m_stretch`.

4. Ausgabe der Volumengittergeometrie in eine Datei im Tecplot-Format: Das ausgegebene Volumengitter mit $65 \cdot 65 \cdot 66 = 278850$ Gitterpunkten ist in Abbildung 5.4 dargestellt.

5.3 Durchführung der Simulation mit QUADFLOW

Die Simulation wurde mit dem Strömungslöser QUADFLOW (s. [15]) durchgeführt, der an dieser Stelle kurz beschrieben werden soll. QUADFLOW ist ein Löser für groß angelegte Simulationen kompressibler Strömungen und Strömungs-Struktur-Wechselwirkungen. Die zugrundeliegenden Gleichungen sind die instationären Navier-Stokes-Gleichungen. Unter Vernachlässigung von Volumenkräften können damit die Erhaltungsgleichungen für jedes Kontrollvolumen Ω mit der Berandung $\partial\Omega$ und dem nach außen zeigenden, auf dem Oberflächenelement $\partial S \subset \partial\Omega$ liegenden Normalenvektor $\mathbf{n}$ in Integralform formuliert werden als

$$\begin{aligned}
\int_\Omega \frac{\partial \rho}{\partial t} dV + \oint_{\partial\Omega} \rho \mathbf{v}\mathbf{n} dS &= 0 && \text{(Masse)} \\
\int_\Omega \frac{\partial (\rho \mathbf{v})}{\partial t} dV + \oint_{\partial\Omega} (\rho \mathbf{v} \circ \mathbf{v} - \tau)\mathbf{n} dS &= \mathbf{0} && \text{(Impuls)} \\
\int_\Omega \frac{\partial (\rho e_{tot})}{\partial t} dV + \oint_{\partial\Omega} (\rho e_{tot}\mathbf{v} - \mathbf{v}\tau + \mathbf{q})\mathbf{n} dS &= 0. && \text{(Energie)}
\end{aligned} \tag{5.4}$$

Hierbei bezeichnet $\tau = -pI + \tau^\nu$ den Spannungstensor, der den viskosen Spannungstensor τ^ν enthält. $\mathbf{q}$ ist der Wärmestromvektor, ρ die Dichte, $\mathbf{v}$ der Geschwindigkeitsvektor des Fluids und $e_{tot} = e + \frac{1}{2}|\mathbf{v}|^2$ die spezifische Gesamtenergie. Das Symbol $\circ$ steht für das dyadische Produkt. Die unbekannten Größen in den drei Gleichungen sind ρ, $\mathbf{v}$, der statische Druck p, e und die Temperatur T, die im Fourier'schen Gesetz für den Wärmestrom $\mathbf{q}$ auftritt. Zur Schließung des Systems sind zusätzlich zwei Zustandsgleichungen nötig, z.B. für den Fall eines idealen Gases die Gleichungen $p = \rho RT$ mit der spezifischen Gaskonstante R und $e = c_v T$ mit der spezifischen isochoren Wärmekapazität $c_v = \frac{\partial e}{\partial T}\big|_{1/\rho}$ bei konstantem spezifischem Volumen $\frac{1}{\rho}$.

Da singuläre Phänomene wie z.B. Diskontinuitäten zumindest lokal hoch aufgelöst werden müssen, würden Diskretisierungen auf uniformen Gittern oft eine viel zu große Anzahl von Gitterpunkten erfordern, die besonders bei dreidimensionalen, instationären Problemen die Ressourcen heutiger Computersysteme in Bezug auf Speicherplatz und Rechenzeit bei weitem übersteigen würden. Aus diesem Grund verwendet QUADFLOW das Konzept der Gitteradaption, um die jeweilige Problemgröße der diskreten Formulierungen in jedem Zustand so klein wie möglich zu halten. Das zentrale Ziel bei der Gitteradaption ist die Realisierung adaptiv generierter Diskretisierungen, die in der Lage sind, die physikalisch relevanten Phänomene bei größtmöglicher Reduzierung der Kosten (Rechenzeit, Speicherplatz) aufzulösen. Hierfür verwendet QUADFLOW den Ansatz der h-Adaption, bei der lokal Gitterpunkte hinzugefügt werden, ohne dass sie an anderer Stelle entfernt werden müssten. Diese Technik funktioniert in QUADFLOW nur bei Verwendung eines B-Spline-Gitters, wie es mit GNAGG erstellt werden kann.

Die Diskretisierung der zugrundeliegenden Navier-Stokes-Gleichungen (5.4) erfolgt, nachdem diese dimensionslos gemacht worden sind, durch eine zellzentrierte Finite-Volumen-Methode von zweiter Ordnung in Raum und Zeit.

Der übliche Ablauf bei der Durchführung einer Simulation mit QUADFLOW kann dem Flussdiagramm in Abbildung 5.5 entnommen werden. Dieses Diagramm ist in ausführlicherer Form in [16] zu finden. Dort und in [15] können weitergehende Details zu QUADFLOW nachgelesen werden.

Die Simulation wurde schließlich von Christian Windisch durchgeführt, in dessen Arbeit (s. [17]) weitere Beispiele zu finden sind. Da kein B-Spline-Gitter zur Verfügung stand, wurde die Simulation nicht-adaptiv mit dem Gitter aus Abbildung 5.4 durchgeführt. Die Parameter der

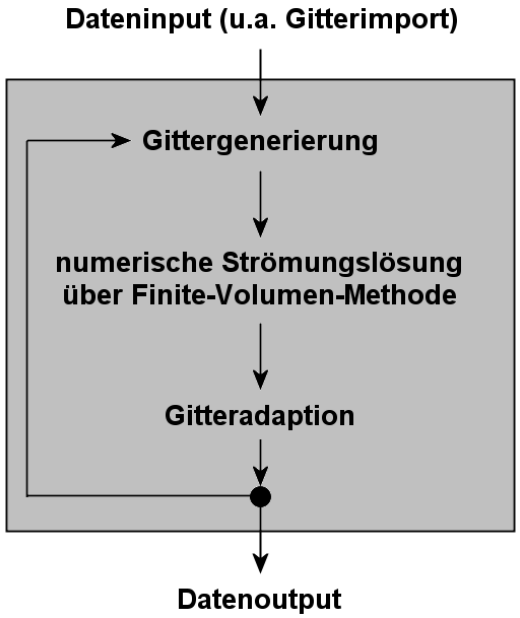

Abbildung 5.5: Flussdiagramm für den Ablauf einer Simulation mit QUADFLOW

Anströmung lauteten

$$M_\infty = 11.0 \tag{5.5}$$
$$\rho_\infty = 0.0016 \frac{kg}{m^3} \tag{5.6}$$
$$T_\infty = 196K. \tag{5.7}$$

Da es sich um eine reibungsfreie Idealgasströmung handelte, reduzierten sich die Navier-Stokes-Gleichungen (5.4) zu den Eulergleichungen, in denen die Temperatur T nicht auftritt und für die daher zur Schließung nur eine Zustandsgleichung benötigt wird. Als Randbedingungen wurden die Kugel mit dem Radius $R = 1$ als feste Wand, die Einströmseite als supersonischer Einströmrand und die Ausströmseite als supersonischer Ausströmrand festgelegt. Die Simulation wurde solange durchgeführt, bis sich ein Dichteresiduum kleiner als 10^{-3} ergab, was nach 2850 Iterationen der Fall war.

5.4 Auswertung der Simulation

Abbildung 5.6 zeigt einen Schnitt durch das Gitter bei $z = 0$. Man erkennt den vor der Kugeloberfläche ausgebildeten Verdichtungsstoß, hinter dem die Dichte von $\rho_\infty = 0.0016 \frac{kg}{m^3}$ nahezu sprunghaft auf einen höheren Wert ansteigt. An der Dichteverteilung erkennt man auch die Symmetrie der Lösung, die bei einer Kugel bei horizontaler Anströmung gegeben sein muss.

Untersucht werden soll zunächst der Stoßabstand nach Billig (s. [18]), gegeben über

$$\frac{\Delta}{R} = 0.143 \; exp\left(\frac{3.24}{M_\infty^2}\right). \tag{5.8}$$

Diese Gleichung stellt eine empirische Formel dar, die aus Experimenten mit niedriger Temperatur und für Idealgas mit dem Isentropenexponenten $\gamma = 1.4$ hergeleitet wurde. Für $\frac{\Delta}{R}$ erhält man mit $M_\infty = 11.0$ einen Wert von ungefähr 0.147. Dieser Stoßabstand ist in Abbildung 5.7, die den Dichteverlauf auf der Symmetrielinie $y = 0$ darstellt, eingetragen. Es ergibt sich eine gute Übereinstimmung mit dem Stoßabstand der Simulation, der anhand des steilen Dichteanstiegs zu erkennen ist. In dieser Abbildung ist zusätzlich die Stoßintensität (s. z.B. [14])

$$\frac{\rho_2}{\rho_\infty} = \frac{(\gamma+1)M_\infty^2}{(\gamma-1)M_\infty^2 + 2} \tag{5.9}$$

eingezeichnet, die nur für den senkrechten Verdichtungsstoß gilt und sich mit $M_\infty = 11.0$ und $\gamma = 1.4$ zu ca. 5.762 berechnet. Sie liefert ebenfalls eine gute Übereinstimmung mit den Simulationsergebnissen. Wählt man für $\frac{\rho_2}{\rho_\infty}$ den ersten Wert des Plateaus nach dem starken Dichteanstieg (in der Abbildung mit x markiert), der bei 5.973 liegt, so ergibt sich zur vorberechneten Stoßintensität eine Abweichung von ungefähr 3.7%.

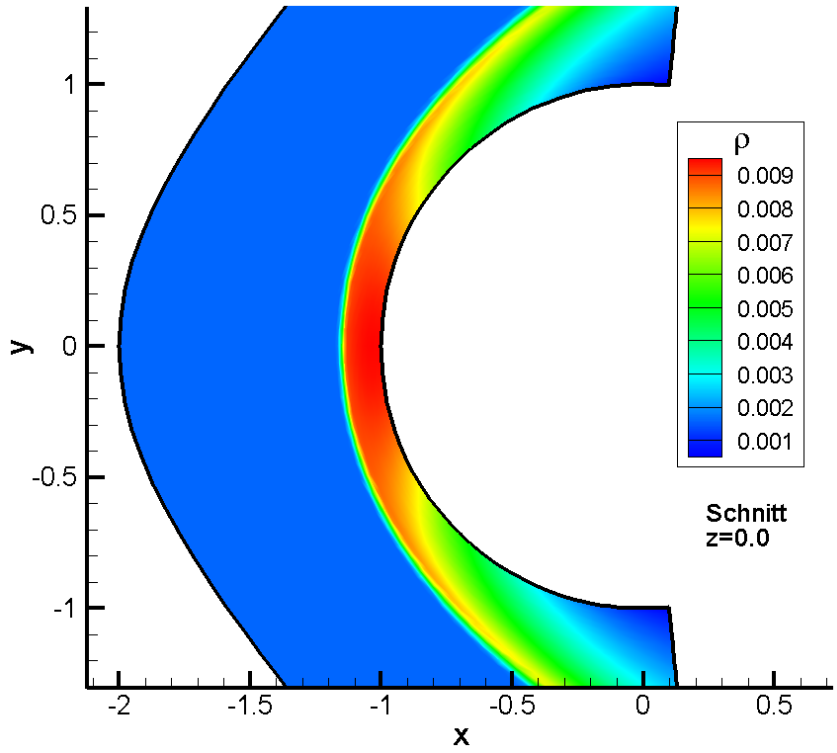

Abbildung 5.6: Konturplot für die Dichte ρ, Schnitt bei $z = 0$

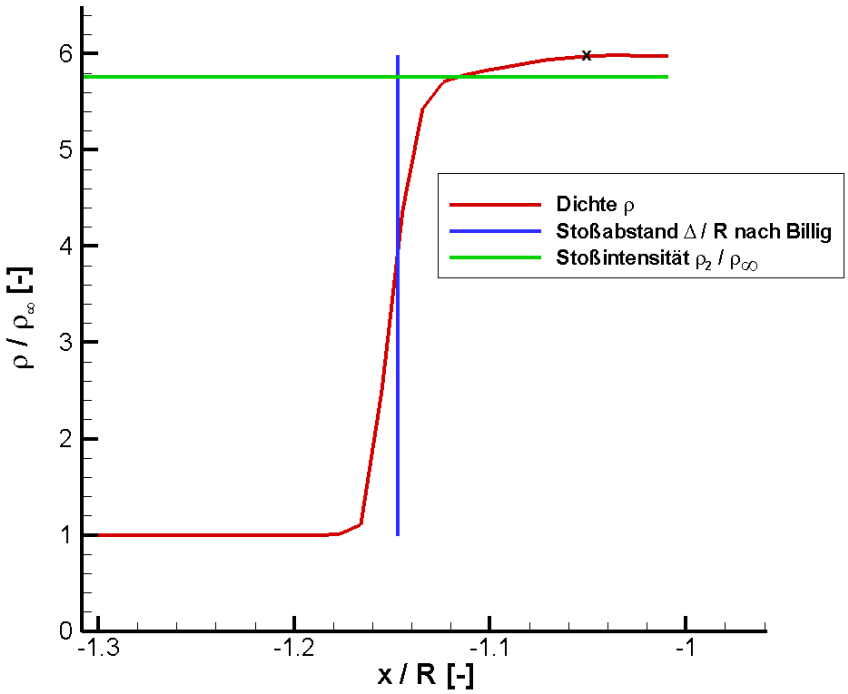

Abbildung 5.7: Verlauf der auf die Anströmdichte ρ_∞ bezogenen Dichte ρ auf der Symmetrielinie $y = 0$: Es ergibt sich eine gute Entsprechung der vorberechneten Werte für die Stoßintensität und den Stoßabstand.

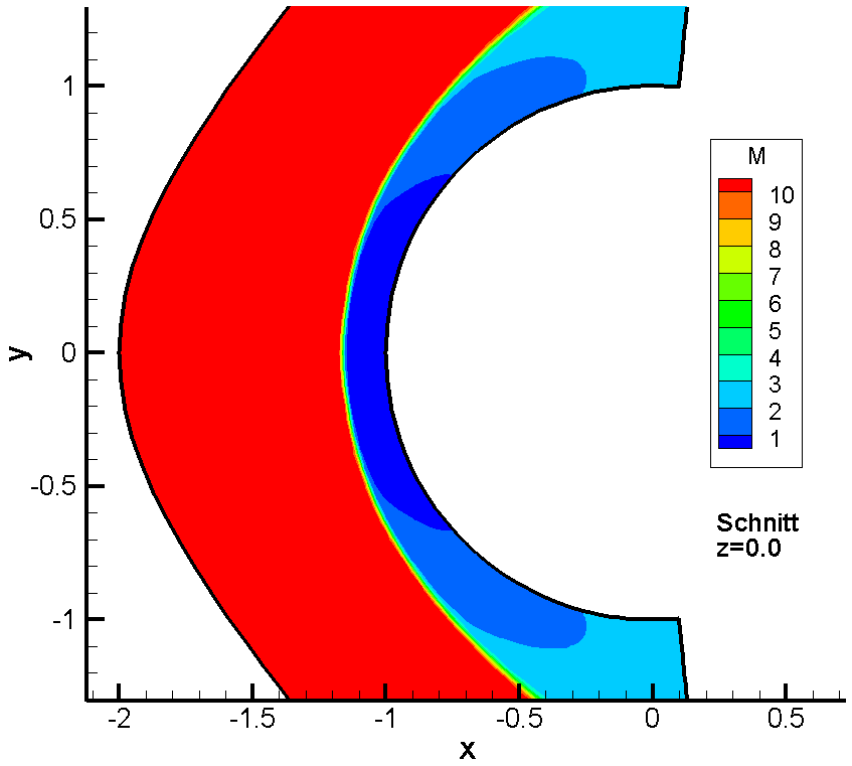

Abbildung 5.8: Konturplot für die Mach-Zahl M, Schnitt bei $z = 0$

Abbildung 5.8 zeigt abschließend einen Schnitt bei $z = 0$ zur Darstellung der Mach-Zahl. Erneut stellt sich eine symmetrische Lösung ein. Gut sichtbar ist zudem das hinter dem Verdichtungsstoß liegende Unterschallgebiet (dunkelblau).

Insgesamt lässt sich feststellen, dass trotz der nicht-adaptiv durchgeführten Rechnung zufriedenstellende Ergebnisse resultieren. Für weitere, detailliertere Ergebnisse sei an dieser Stelle erneut auf [17] verwiesen.

6 Zusammenfassung

Das Ziel der vorliegenden Arbeit war die Weiterentwicklung eines C++-Programms von Dr. K.-H. Brakhage (s. [4]), so dass dieses eingelesene Oberflächengittergeometrien nicht nur nach Catmull-Clark-Regeln unterteilen, sondern auch vorgegebene Oberflächen approximieren kann. Hierfür wurde zunächst die notwendige Theorie aufgezeigt (s. Kap. 2), angefangen beim Stetigkeitsbegriff, der entscheidend für die Glattheit der angenäherten Oberflächen ist. Im darauf folgenden Abschnitt wurden wichtige Begriffe aus dem Bereich der Subdivison geklärt. Da das Catmull-Clark-Schema wie viele andere Subdivisiontechniken auch auf B-Splines basiert, wurden danach grundlegende Details wie die Konstruktion und die Verfeinerbarkeit von B-Splines und die Subdivision von B-Spline-Kurven aufgezeigt. Der Übergang zu B-Spline-Flächen über einen Tensorproduktansatz wurde kurz skizziert. Es folgte die Beschreibung der Subdivision-Regeln und der Eigenschaften des Catmull-Clark-Schemas sowie die Darstellung von drei Beispielen für die Subdivision von Objekten aus dem Gebiet der Aeronautik. Zum Abschluss des Theorieteils wurden die für die Approximation gegebener Oberflächen benötigte Limitpunktberechnung und die Lösung des linearen Ausgleichsproblems, das sich aus der Approximationsaufgabe ergibt, mit Hilfe der CGLS-Methode beschrieben.

Zu Beginn der Implementierung (s. Kap. 3) wurde das vorhandene Datenmodell verändert: Vorher wurden Gitter bei der Durchführung eines Subdivision-Schrittes überschrieben, d.h., es konnte immer nur ein Gitter zur gleichen Zeit existieren. Nach der Implementierung einer Gitter-Liste mit Hilfe der Standard Template Library von C++ jedoch waren es beliebig viele Gitter. Hierdurch ergab sich die Möglichkeit, Subdivision-Schritte rückgängig zu machen und die Gitter auf gröberen Unterteilungsstufen wieder darstellen und nachträgliche Modifikationen vornehmen zu können. Außerdem konnte der Code vereinfacht werden, da Variablen, die zuvor vor dem Überschreiben eines Gitters zur Zwischenspeicherung von Werten benötigt wurden, jetzt wegfallen konnten. Die vorher für die Vertices, Edges und Faces eines Gitters verwendeten Arrays wurden durch Vektoren ersetzt, die ebenfalls aus der Standard Template Library stammen und wie bei der Liste Zugriffe durch einfache Funktionen wie `push_back(·)` oder `pop_back()` ermöglichen.

Durch die in der Folge implementierte linksdrehende Orientierung der Nachbarschaftsinformationen bei Vertices und Faces konnte gewährleistet werden, dass bei jeder Berechnung eines Limitpunkts die beteiligten Vertices in der korrekten Reihenfolge ausgewählt werden können. Die Limitpunktberechnung selbst machte dann den Großteil der Implementierungsarbeit aus. Neben dem in der Theorie vorgestellten Fall eines im Inneren des Gitters liegenden Faces ohne tagged

Edges mussten weitere Möglichkeiten berücksichtigt werden, z.B. Limitpunkte in Randfaces oder in inneren Faces mit einem oder zwei tagged Edges. Vor allem durch innere tagged Edges ergaben sich viele mögliche Fallunterscheidungen, die auch zur Notwendigkeit einer Vereinfachung bei der Implementierung führten: Limitpunkte in Faces mit einem oder zwei inneren tagged Edges werden so berechnet wie Limitpunkte in den entsprechenden Randfaces, auch wenn die inneren tagged Edges nicht zwingend eine geschlossene Kurve bilden. Der hierbei entstehende Fehler ist in der Regel klein und kann über eine Funktion innerhalb des Programms auch überprüft werden. Zur Eindämmung der vielen Fallunterscheidungen wurden des Weiteren einige wenige Einschränkungen für die Limitpunktberechung vorgeschrieben, die zunächst die Durchführung eines Subdivision-Schrittes erfordern, wozu der Benutzer des Programms in jedem der Fälle auch aufgefordert wird. So ist z.B. in einem Face, das genau einen Randvertex enthält, für die anderen drei Vertices jeweils nur die Valenz $N = 4$ erlaubt.

Um gegebene Oberflächen über die Lösung des linearen Ausgleichsproblems $\|A\mathbf{V} - \mathbf{S}\|_2 \rightarrow min$ approximieren zu können, wurde das Aufstellen der Matrizen A und ihrer Transponierten A^T, die die Koeffizienten der Limitpunktberechnung enthalten, innerhalb der Funktion für die Limitpunktberechnung realisiert. Da es sich um dünnbesetzte Matrizen handelt, wurde jeweils im Vorhinein die benötigte Anzahl an Einträgen in jeder Zeile ermittelt, um für die Zeilen, die als Vektoren angelegt wurden, nur den tatsächlich benötigten Speicherplatz reservieren zu müssen. Außerdem kann auf diese Weise die Dauer einer Berechnung wie der Multiplikation einer der Matrizen mit einem Vektor deutlich verkürzt werden. Die Limitpunkte sind schließlich für die Berechnung der Surfacepunkte im Vektor $\mathbf{S}$ erforderlich: Die Surfacepunkte werden hier berechnet, indem die Limitpunkte jeweils über eine Gerade mit dem Ursprung verbunden und diese Geraden mit der gegebenen Oberfläche geschnitten werden. Implementiert wurde dies für ein Ellipsoid. Mit A, A^T, den Vertices $\mathbf{V}$ und den Surfacepunkten $\mathbf{S}$ kann die Approximationsaufgabe über die CGLS-Methode gelöst werden. Diese wurde als Template-Funktion implementiert und kann sowohl mit Vektoren, deren Elemente `double`-Variablen entsprechen, als auch mit Vektoren, deren Elemente die dreidimensionalen Vertices vom Typ `CVertex` sind, umgehen.

Um ein mit Hilfe des in dieser Arbeit entwickelten Programms erstelltes Gitter mit dem B-Spline-Gittergenera-tor GNAGG einlesen zu können, wurde eine Funktion zur sortierten Ausgabe der Gittergeometrie implementiert. Hierbei werden die Koordinaten der aktuellen Limitpunkte blockweise in eine Datei geschrieben. Zu welchem Block ein Limitpunkt gehört, richtet sich nach dem Face im ersten reinen Viereckgitter, aus dem er hervorgegangen ist.

In Kapitel 4 wurden verschiedene Vorgehensweisen bei der Anwendung des entwickelten Programms demonstriert. Es ergab sich, dass das „W“-Anwendungsprofil, bei dem Limitpunktberechnung und Approximation nach jeder Subdivision durchgeführt werden, dem „V“-Profil, bei dem diese beiden Schritte nur am Ende nach sämtlichen Unterteilungen Anwendung finden, in der Regel aufgrund der signifikant verringerten Anzahl an CGLS-Iterationen auf den feinen, zeitintensiven Unterteilungsstufen vorzuziehen ist, auch wenn sich eine deutlich höhere Gesamtzahl an Iterationen ergibt.

Mit Hilfe der abschließenden Strömungssimulation (s. Kap. 5) sollte die praktische Anwendbarkeit eines mit dem in dieser Arbeit entwickelten Programm erstellten Oberflächengitters anhand einer Strömungssimulation gezeigt werden. Wichtig war hierbei vor allem der Schritt der Erweiterung des Oberflächengitters zu einem Volumengitter. Bei der Simulation der Überschallanströmung einer Kugel konnte schließlich gezeigt werden, dass trotz der nicht-adaptiv durchgeführten Berechnung zufriedenstellende Ergebnisse erreicht werden konnten: Der Verdichtungsstoß stellte sich wie erwartet ein, die Lösung war symmetrisch und die Stoßintensität $\frac{\rho_2}{\rho_\infty}$ oder der Stoßabstand $\frac{\Delta}{R}$ zeigten im Vergleich zu den vorberechneten Werten akzeptable Abweichungen.

Ausblick

Untersucht wurde bisher die generelle Anwendbarkeit von Flächenapproximationen und der Gittergenerierung mittels Catmull-Clark-Methoden unter Einsatz globaler Iterationsverfahren (hier CGLS). Als non-rectangular Beispiel wurde, wie bei der Strömungssimulation zu erkennen erfolgreich, eine Kugel gewählt.

Bei Anwendungen sind die Flächen meist als getrimmte Spline-Flächen gegeben, d.h. als Flächen, die über Trimmkurven zurechtgeschnitten sind. Ein Ziel für die Weiterentwicklung der implementierten Methoden ist die Projektion auf solche Flächen, statt ausschließlich eine Kugel bzw. ein Ellipsoid approximieren zu können.

Es kann auch vorkommen, dass Dreiecksnetze gegeben sind. Für diesen Fall sind zwei verschiedene Vorgehensweisen für eine Implementierung interessant:

- Projektion auf Dreiecke bzw. auf durch Dreiecke lokal gegebene Flächen
- Projektion der Dreiecksvertices auf eine Catmull-Clark-Fläche: Die Einträge der hierfür benötigten Ausgleichsmatrix könnten in diesem Fall wie in [11] beschrieben berechnet werden.

Für die beschriebenen Flächen sollte es außerdem ermöglicht werden, wie z.B. bei der Flügel-Rumpf-Konfiguration aus Abbildung 2.12 durch die Modifikation der Regeln scharfe oder abgerundete Kanten zu erzwingen. Bei Dreiecksnetzen wäre es wünschenswert, wenn solche Kanten automatisch erkannt werden könnten (sogenannte „Feature detection“).

Literaturverzeichnis

[1] W. Schröder. Der Tragflügel der nächsten Generation. *Highlights der Aachener Sonderforschungsbereiche*, S. 34–35, 2007.

[2] K.-H. Brakhage und Ph. Lamby. Application of B-Spline Techniques to the Modeling of Airplane Wings and Numerical Grid Generation. *CAGD*, 25(9):738–750, 2007.

[3] E. Catmull und J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *CAD*, 10(6):350–355, 1978.

[4] K.-H. Brakhage. Modified Catmull-Clark Methods for Modelling, Reparameterization and Grid Generation. *2. Internationales Symposium Geometrisches Modellieren, Visualisieren und Bildverarbeitung*, HfT Stuttgart, 2007.

[5] J. Hoschek und D. Lasser. *Grundlagen der geometrischen Datenverarbeitung*. B. G. Teubner Stuttgart, 1989.

[6] P. Schröder. Foundations I: Basic Ideas. In *Subdivision for Modeling and Animation*, S. 15–44. ACM SIGGRAPH Course Notes, 1999.

[7] D. Zorin. Subdivision Surfaces. In *Subdivision for Modeling and Animation*, S. 45–63. ACM SIGGRAPH Course Notes, 1999.

[8] G. de Rham. Un peu de mathématiques á propos d'une courbe plane. *Elemente der Mathematik*, 2(4):89–104, 1947.

[9] G. Chaikin. An algorithm for high speed curve generation. *Computer Graphics and Image Processing*, 3:346–349, 1974.

[10] T. DeRose, M. Kass und T. Truong. Subdivision Surfaces in Character Animation. In M. Cohen, Hrsg., *Proceedings of the 25th annual conference on Computer graphics and interactive techniques, ACM SIGGRAPH*, S. 85–94. Addison Wesley, 1998.

[11] J. Stam. Exact Evaluation Of Catmull-Clark Subdivision Surfaces At Arbitrary Parameter Values. In *Proceedings of SIGGRAPH*, S. 395–404, 1998. Überarbeitete Version über http://www.dgp.toronto.edu/people/stam.

[12] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, second edition, 2003.

[13] U. Reif. Grundzüge der Splinetheorie, 2004. Vorlesungsskript der Abteilung für Angewandte Mathematik, Universität Freiburg.

[14] W. Schröder. *Fluidmechanik*. Aachener Beiträge zur Strömungsmechanik, Band 7. Wissenschaftsverlag Mainz in Aachen, 2004.

[15] F. D. Bramkamp, B. Gottschlich-Müller, M. Hesse, Ph. Lamby, S. Müller, J. Ballmann, K.-H. Brakhage und W. Dahmen. H-Adaptive Multiscale Schemes for the Compressible Navier-Stokes Equations - Polyhedral Discretization, Data Compression and Mesh Generation. *Notes on Numerical Fluid Mechanics*, 84:125–204, 2003.

[16] F. D. Bramkamp. *Unstructured h-Adaptive Finite-Volume Schemes for Compressible Viscous Fluid Flow*. Dissertation, RWTH Aachen, 2003.

[17] C. Windisch. Numerische Simulation von Hochgeschwindigkeitsströmungen im Nichtgleichgewicht. Diplomarbeit, Lehrstuhl für Computergestützte Analyse Technischer Systeme, RWTH Aachen, 2009.

[18] F. S. Billig. Shock-Wave Shapes around Spherical- and Cylindrical-Nosed Bodies. *Journal of Spacecraft and Rockets*, 4(6):822–823, 1967.