HUMBOLDT–UNIVERSITÄT ZU BERLIN
INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSGEBIET
ALGORITHMEN UND KOMPLEXITÄT

Diplomarbeit

Effiziente Algorithmen zur Isoflächengenerierung aus Volumendaten

Thomas Anders
anders@informatik.hu-berlin.de

Betreuer: Dr. sc. Ernst–Günter Giessmann, Dipl.–Phys. Detlev Stalling

Berlin, den 29. März 1996

Eigenständigkeitserklärung

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit zum Thema „Effiziente Algorithmen zur Isoflächengenerierung aus Volumendaten“ selbständig und nur unter Zuhilfenahme der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den 29. März 1996 Thomas Anders

Vorwort

In dieser Diplomarbeit werden Verfahren zur Erzeugung von Isoflächen aus dreidimensionalen Skalarfeldern beschrieben und bewertet. Den Standardverfahren wie dem *Marching Cubes*–Algorithmus werden neuere Ansätze gegenübergestellt, die topologisch konsistente Isoflächen generieren, die Anzahl der erzeugten Polygone reduzieren oder hierarchische Datenstrukturen zur Vermeidung des Durchsuchens „leerer" Regionen benutzen. Darüberhinaus wird untersucht, inwieweit parallele Versionen der Algorithmen möglich sind und ein im Rahmen dieser Arbeit selbst entwickeltes paralleles Isoflächenprogramm für den Parallelrechner CRAY T3D SC 256 näher vorgestellt.

Die vorliegende Arbeit entstand am Fachbereich Informatik der Humboldt–Universität zu Berlin in Zusammenarbeit mit dem Konrad–Zuse–Zentrum für Informationstechnik Berlin (ZIB). Sie erhebt nicht den Anspruch, alles Wissen dieser Welt zum Thema Isoflächengenerierung in sich zu vereinen. Sie bemüht sich aber, so komplett wie möglich den momentanen Forschungsstand im Bereich sequentieller und paralleler Isoflächenalgorithmen zusammenzufassen. Sie möchte damit insbesondere dem Wissenschaftler bzw. Anwender, der ein Isoflächenmodul in ein Softwarepaket implementieren oder die Tauglichkeit eines vorhandenenen Isoflächenprogramms für seine Zwecke bewerten will, eine wertvolle Entscheidungshilfe an die Hand geben. Dem Autor ist keine wissenschaftliche Arbeit mit gleicher Zielstellung bekannt.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Visualisierung von Volumendaten 1

1.2 Zielstellung 5

1.3 Gliederung und Umfang der Arbeit 6

2 Isoflächen **7**

3 Sequentielle Ansätze **16**

3.1 Algorithmus für soft objects 17

3.2 Marching Cubes 23

3.3 Korrigierter Marching Cubes 27

3.3.1 Modifizierte Topologietabelle 27

3.3.2 Implementation in HyperPlan 30

3.3.3 Patches 30

3.4 Tetraederzerlegung 31

3.5 Trilineares Modell 34

3.6 Heuristiken 37

3.7 Effizienzverbesserungen 39

3.7.1 Skalarfelder als Octrees 41

3.8 Cuberille 43
3.8.1 Dividing Cubes 44

4 Verfahren mit interner Polygonreduktion 45

4.1 Adaptive Algorithmen 46
4.1.1 Splitting Box 47
4.1.2 Adaptive Marching Cubes 47
4.1.3 Isoflächen variabler Auflösung 49
4.2 Compact Cubes 49
4.3 Discretized Marching Cubes 52
4.4 Separate Polygonreduktion 53
4.4.1 Mesh Optimization 54
4.4.2 Triangle Decimation 55

5 Parallele Algorithmen 58

5.1 Bisherige parallele Implementierungen 59
5.2 Eigener Ansatz: PARADISO 61
5.2.1 CRAY T3D 61
5.2.2 Softwareentwurf 62
5.2.3 Implementation 64
5.2.4 Ergebnisse und Ausblick 66

6 Zusammenfassung 68

Farbige Abbildungen 71

Literaturverzeichnis 77

Kapitel 1

Einführung

1.1 Visualisierung von Volumendaten

Die wissenschaftliche Visualisierung (*Scientific Visualization*) ist ein verhältnismäßig junges, aber rasant an Bedeutung gewinnendes Fachgebiet mit stark interdisziplinärem Charakter, in dem Resultate aus einer Reihe anderer Teilgebiete der Informatik mit denen verschiedenster naturwissenschaftlich–technischer Disziplinen zusammenfließen. Ihr Hauptziel ist es, Anwendern die Interpretation visuell nicht zugänglicher Daten mittels computerberechneter Bilder so weit wie möglich zu erleichtern oder im Falle großer oder komplexer Datenmengen gar erst zu ermöglichen. Eine zweite wichtige Aufgabe der wissenschaftlichen Visualisierung besteht darin, dem Nutzer einen interaktiven Einfluß auf das Ergebnis der Darstellung zu gestatten.

Die Grundlage für den bestmöglichen Zugang zu den Daten bilden die hochentwickelten visuellen Wahrnehmungsfähigkeiten des Menschen, insbesondere hinsichtlich des Erkennens von Strukturen, Formen und Entwicklungen und des automatischen Herausfilterns unwesentlicher Informationen. Mittels des Sehapparates ist ein intuitives Verständnis von entsprechend aufbereiteten komplexen Sachverhalten möglich, die in alphanumerischer Form kaum erfaßbar wären. Die ausgeprägten Fähigkeiten des Menschen zum ganzheitlichen Erfassen bergen ein enormes Potential, das es bei der Visualisierung auszuschöpfen gilt.

Verstärkt durch die ebenso rasante Entwicklung des *Scientific Computing*, stellt dies hohe Anforderungen an die Entwicklung leistungsfähiger Visualisierungsmethoden und -algorithmen. Wichtige Kriterien für Visualisierungstechniken sind deren breite Anwendbarkeit, hohe Qualität und Genauigkeit, sowie die intuitive Interpretation der Ausgabe. An Visualisierungsprogramme stellen sich neben den Anforderungen an die implementierten Algorithmen auch softwareergonomische Ansprüche, insbesondere an die einfache Bedienbarkeit, die es Fachwissenschaftlern ermöglichen soll, komplexere Visualisierungsaufgaben hinsichtlich ihrer Daten selbst zu lösen.

Der Einsatz von im Bereich der wissenschaftlichen Visualisierung entwickelten Verfahren beschränkt sich natürlich nicht allein auf die wissenschaftliche Welt. Kommerzielle Anwendungen

von Computergrafik und verwandten Fachgebieten sind allgegenwärtig und verdrängen in atemberaubendem Tempo herkömmliche Herangehensweisen. Am anschaulichsten tritt dieser Prozeß in der Film– und Fernsehbranche zutage, wo bereits jetzt Personen durch animierte computergenerierte Modelle ersetzt werden können und sich Werbeagenturen in der Anwendung von Effekten und Tricktechniken gegenseitig überbieten. Auch die industrielle Produktion basiert zunehmend auf der Erstellung und Visualisierung von Modellen im Computer. Prototypen von Automobilen, Flugzeugen oder Maschinenteilen werden kostengünstig am Computer entworfen und getestet. Architekten konzipieren Gebäudekomplexe, die vor dem ersten Spatenstich nahezu komplett erfaßbar und „begehbar" sind. *Telemedizin* erlaubt die Erprobung, Planung und visuelle Kontrolle von Operationen oder gar die Steuerung von Eingriffen durch einen nur per Netzverbindung „anwesenden" Spezialisten. In den kommenden Jahren und Jahrzehnten wird der Trend in dieser Richtung weiter fortschreiten, wenn es gelingt, die Entwicklung leistungsfähiger Algorithmen in gleichem Maße voranzutreiben. Dies macht kommerzielle Anwendungen von Erfolgen in der wissenschaftlichen Forschung abhängig, die wiederum in zunehmendem Maße auf finanzielle Unterstützung durch die Industrie angewiesen ist.

Beiden Welten ist gemeinsam, daß in ihrem Umfeld *Volumendaten* zu den am häufigsten anfallenden Primärdaten gehören. Unter Volumendaten versteht man im allgemeinen höherdimensionale (Dimension $d \geq 3$) skalare, vektorielle oder tensorielle Felder. Während tensorielle Felder, z.B. aus der Differentialgeometrie oder Kontinuumsmechanik, zu jedem Punkt des d–dimensionalen Raumes ein komplexes mathematisches Objekt, den *Tensor*, enthalten, definieren Vektorfelder zu jedem Punkt einen Vektor. Dreidimensionale Vektorfelder können demnach als Funktionen aufgefaßt werden, die Punkte des Raumes nach $\mathbb{R}^3$ abbilden. Skalarfelder als einfachste Form von Volumendaten enthalten zu jedem Punkt im $\mathbb{R}^d$ genau einen skalaren Wert aus $\mathbb{R}$.

Die Felder können sowohl berechnet sein, d.h. ihnen liegen mathematische Funktionen oder numerische Berechnungen zugrunde, als auch aus Experimenten oder Messungen (z.B. von medizinischen oder industriellen Scannern) stammen. Kontinuierlich definierte Funktionen können beispielsweise ein elektrisches Potential oder implizite geometrische Modelle (z.B. algebraische Flächen) beschreiben. In kontinuierlichen Feldern kann für jede Position im Raum (innerhalb des Definitionsbereiches) der Feldwert exakt bestimmt werden. In den meisten Anwendungen, insbesondere bei Messungen, sind jedoch die Feldwerte nur als Wertepaare, d.h. an diskreten Punkten eines Gitters, gegeben. Das Gitter beschreibt die Lage und Topologie der Punkte im $\mathbb{R}^d$, für die Feldwerte definiert sind.

Volumendaten werden in nahezu allen wissenschaftlichen Disziplinen erzeugt. Die populärsten Anwendungsgebiete sind die Strömungsmechanik (*Computational Fluid Dynamics, CFD*), Materialforschung, Modellierung, Meteorologie (komplexe Klimasimulationen), Chemie (*Computational Chemistry*)[1] und die Medizin.

Die verschiedenen Möglichkeiten zur Visualisierung von Volumendaten lassen sich im weitesten Sinne unter dem Begriff *Volume–Rendering* zusammenfassen. Die große Gruppe der Verfahren zur Darstellung von Vektorfeldern ist Gegenstand aktueller Forschungen. Die Vektoren können dabei

[1] Die gewählten deutschen Bezeichnungen sind nicht gleichbedeutend mit den englischen Begriffen in Klammern, sondern schließen diese ein. In der deutschsprachigen wissenschaftlichen Welt werden die aufgeführten englischen Begriffe aus dem Bereich *Computational Sciences* in Ermangelung einer adäquaten Übersetzung meist als Eigennamen gebraucht.

z.B. Kräfte, die elektrische oder magnetische Feldstärke oder die Geschwindigkeit von Teilchen einer strömenden Flüssigkeit repräsentieren. Interessante Aspekte der Vektorfeldvisualisierung sind neben Feldstärke und Feldrichtung häufig auch topologische Gegebenheiten wie Unstetigkeitsstellen und Singularitäten. Symbolische Darstellungen gehen deshalb heute oft über die hinlänglich bekannte Repräsentation von Betrag und Richtung durch Pfeile hinaus. Zusammengesetzte Symbole erlauben die Darstellung einer Vielzahl von Visualisierungsaspekten, allerdings nur an ausgesuchten Positionen. Animierte Partikel, die der Wirkung des Vektorfeldes ausgesetzt werden, vermitteln einen globalen Eindruck, erfordern jedoch eine geeignete Wahl der Startpunkte. Für turbulente oder dichte Vektorfelder eignen sich die in jüngster Zeit entwickelten texturbasierten Verfahren (*spot noise* [vW91], *Linienintegralfaltung* [SH95b]) deutlich besser. Allen texturbasierten Vektorfeldvisualisierungsmethoden liegt die Idee zugrunde, entlang der Feldrichtung eine hohe und senkrecht zur Feldrichtung eine niedrige Pixelwertkorrelation zu erzeugen. Mit der Verfügbarkeit moderner Grafikhardware zur 3D–Texturgenerierung wird seit neuestem auch an dreidimensionalen Texturverfahren geforscht [Bat96]. Die Abbildungen 1.1 und 1.2 veranschaulichen die Wirkung solcher Methoden.

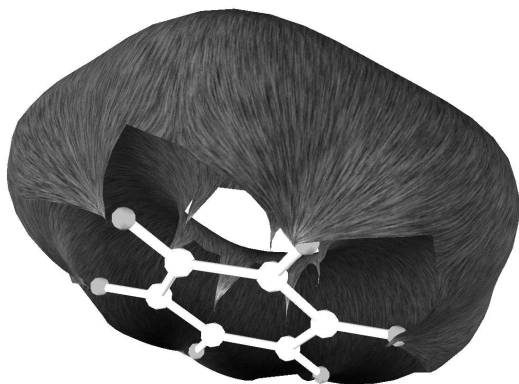


Abbildung 1.1: *Elektrisches Feld auf einer Strömungsfläche im Benzolmolekül, texturiert mit der Linienintegralfaltung*

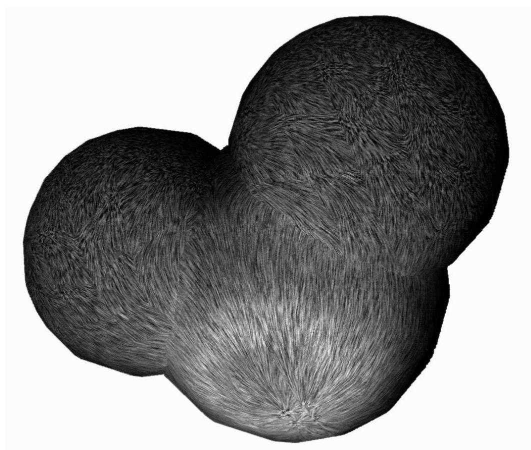


Abbildung 1.2: *Elektrisches Feld auf der* CONNOLLY*–Fläche in einem Wasser–Molekül, visualisiert mit der Linienintegralfaltung*

Im engeren Sinne versteht man unter Volume–Rendering oft auch Verfahren zur Darstellung des einfachsten Falles von Volumendaten — den dreidimensionalen Skalarfeldern. Skalarfelder können die verschiedensten Zustandsgrößen wie die Temperaturverteilung in einem Raum, den Druck in einer Strömung, die Dichte in einem inhomogenen Medium, elektrische Spannung oder das Potential eines Kraftfeldes repräsentieren. Medizinische und industrielle Meßmethoden wie die Computertomographie (CT), bei der das räumliche Verhalten des Dämpfungskoeffizienten der Röntgenstrahlung gemessen wird, und die Magnetfeld–Resonanz–Tomographie (MRT), die das Verhalten der Wasserstoff–Atomkerne erfaßt, generieren ebenfalls 3D–Skalarfelder [Mei90]. Als Feldwerte sind auch abgeleitete Größen, z.B. der Betrag eines Geschwindigkeitsvektors, denkbar.

In der Praxis treten in der Regel diskrete Skalarfelder auf. Die Gitter müssen nicht unbedingt uniform, d.h. rechtwinklig und äquidistant, sondern können auch gedehnt oder krummlinig sein und die verschiedensten topologischen Merkmale besitzen. In typischen Fällen weisen sie in in-

teressanteren Gebieten (*regions of interest*) kleinere Gitterabstände auf als in Gebieten, wo sich das Skalarfeld kaum verändert. Ungeachtet der verschiedenen möglichen Abbildungen zwischen dem logischen Gitterraum und dem physikalischen Raum beschränkt sich diese Arbeit auf solche Gitter, deren logische Struktur sich eineindeutig auf ein uniformes kubisches Gitter abbilden läßt (siehe Kapitel 2). Die kleinsten Gitterzellen (auch Gitterprimitive genannt) sind dann hexaedrische Volumenelemente, die als *Voxel*[2] oder *Kuben* (siehe Abschnitt 3.2) bezeichnet werden. Diese Einschränkung schließt viele im wissenschaftlichen Alltag oder von gängigen medizinischen und industriellen Scannern produzierten 3D–Skalardaten ein. Reguläre Gitter haben die angenehme Eigenschaft, daß sich die Voxel über einfache Integer–Operationen indizieren lassen. Dies vereinfacht die Vermeidung doppelter Untersuchungen desselben Voxels. Nicht näher eingegangen wird insbesondere auf unstrukturierte Gitter, wie sie beispielsweise in Simulationen auf Basis der *3D–Finite–Elemente–Methode (FEM)* auftreten können ([Ell95, Mei90]).

Skalarfelder aus dreidimensional verteilten Werten kann man nicht unmittelbar in grafischer Form darstellen, denn der Graph einer Funktion aus $\mathbb{R}^3$ in $\mathbb{R}$ ist eine Hyperfläche im vierdimensionalen Raum. Um solch ein Skalarfeld zu visualisieren, ist demzufolge eine Abbildung in den drei– oder zweidimensionalen Raum vonnöten. Dies kann z.B. über Schnittebenen, Flächen gleicher Skalarwerte oder die Abbildung der Funktionswerte auf Farb– und Transparenzwerte erfolgen.

Beim *direkten Volume–Rendering* werden den Funktionswerten an jedem Punkt des Gitters Materialeigenschaften wie Farbe und Opazität zugewiesen und auf dieser Grundlage das Skalarfeld direkt, d.h. ohne Umwandlung in andere geometrische Repräsentationen, als ein wolkenähnliches 3D–Objekt auf eine Projektionsfläche abgebildet. Das entstehende zweidimensionale Bild spiegelt trotzdem nur einen Teil der den Daten zugrundeliegenden Struktur wider, weshalb es oft notwendig ist, dem Wissenschaftler die Wahl von Blickwinkeln und die interaktive Anpassung von Parametern zu überlassen bzw. mit der Erzeugung von bewegten Bildern die menschliche Tiefenwahrnehmung zu unterstützen. Direktes Volume–Rendering eignet sich besonders zur visuellen Aufbereitung von amorphen Strukturen (Gase, Wolken etc.) bis hin zur Erzeugung fotorealistischer Bilder von Phänomenen dieser Art. Prinzipiell läßt sich zwischen *Raycasting*– und Projektionsmethoden unterscheiden. Ziel ist es jeweils, aus den emittierten, absorbierten und gestreuten[3] Intensitätsbeiträgen der einzelnen Volumenelemente Farbe und Transparenz der Bildpixel zu berechnen. Während man beim Raycasting für jedes Pixel Sichtstrahlen von einem Projektionszentrum (Auge oder Kamera) in das Volumen hineinsendet und Teilbeträge unmittelbar aufsummiert, werden bei Projektionsverfahren die einzelnen Volumenelemente in einer bestimmten Reihenfolge durchlaufen und in die Bildebene projiziert, um damit ihren Beitrag zum Gesamtbild zu berechnen. Projektionsverfahren wie *Zellprojektion* oder *Splatting* erlauben die Nutzung der Möglichkeiten moderner Grafikhardware, z.B. bei der Darstellung semitransparenter Polygone und sind meist schneller als vergleichbare Raycasting–Renderer. Direktes Volume–Rendering wird mittlerweile oft mit dem allgemeineren Begriff Volume–Rendering (siehe oben) gleichgesetzt. Dieser sprachlichen Vereinfachung schließt sich der Autor im folgenden an.

Mit Schnittebenen (*cutting planes*) reduziert man das Skalarfeld auf eine Schicht in der xy–, xz– oder yz–Ebene. Die Erzeugung eines zweidimensionalen Bildes erfolgt dann mittels Einfärben (*Pseudocoloring*), bei der ähnlichen skalaren Werten gleiche Farben zugeordnet werden, oder dem

[2]Im Unterschied zum Pixel (rechteckiges 2D–Element) ist ein Voxel nicht Element des Bildraumes, sondern des zugrundeliegenden Objektraumes

[3]Im allgemeinen werden Streueffekte höherer Ordnung aus Effizienzgründen vernachlässigt.

Berechnen von Isolinien (siehe auch Kapitel 2). 2D–Schnitte geben nur einen lokalen Einblick in eine kleine Untermenge der Gesamtdaten. Läßt man die Schnittebene jedoch entlang einer Koordinatenachse durch das Skalarfeld wandern (*sweeping plane*), werden nacheinander alle Teile des Skalarfelds durch Bildabfolgen visualisiert.

Isoflächen (*isosurfaces*), deren Generierung in dieser Arbeit thematisiert wird, geben einen globaleren Einblick in die Natur der Daten durch Berechnung einer kontinuierlichen Fläche, die alle Voxel schneidet, die einen gewählten konstanten Wert enthalten. Im Gegensatz zum direkten Volume–Rendering, wo alle Voxel im Objektraum einen Beitrag zum engültigen Bild leisten können, tragen hier nur diejenigen Daten zum Ergebnis bei, welche die Isofläche definieren. Eine animierte Sequenz von Isoflächen, bei der der Isowert mit einer kleinen Schrittweite vom minimalen zum maximalen Datenwert anschwillt, kann einen Eindruck von der Verteilung *aller* Werte im Skalarfeld vermitteln. Im Gegensatz zu Volume–Rendering– und Schnittbildverfahren bieten polygonal repräsentierte Isoflächen den Vorteil, daß mit den erzeugten 3D–Objekten interagiert werden kann, um z.B. wichtige Details aus einem speziellen Blickwinkel zu betrachten. Die Möglichkeit, unabhängig von der Größe der Daten und der gewählten Auflösung jedes Detail der Isofläche durch Ausschnittsvergrößerung sichtbar zu machen, erfordert eine topologisch exakte Berechnung, um falsche Interpretationen vermeiden zu helfen. Isoflächenkonstruktion ist im Vergleich zu den eingangs erwähnten Methoden allerdings anfälliger gegen starkes Rauschen in den Eingabedaten, da sie aus ihnen sowohl die genaue Lage als auch die topologischen Merkmale der Isofläche berechnet. Isoflächenberechnungen haben sich insbesondere im medizinischen Bereich als geeignet erwiesen, um beispielsweise genaue Umrisse von Organen auf der Grundlage von CT– oder MRT–Daten zu liefern.

Während Schnittbildverfahren mit einer Laufzeit von $O(N^2)$ für nahezu alle auftretenden Datensätze (der Größe N^3) Ergebnisse in kürzester Zeit versprechen, sind bei Echtzeitdarstellungen mittels Isoflächen– oder direkten Volume–Rendering–Verfahren schnell die Grenzen der Leistungsfähigkeit moderner Workstations erreicht. Bei dem rechenintensivsten der aufgeführten Verfahren, dem direkten Volume–Rendering, scheint derzeit bereits bei kleineren Datensätzen ein paralleler Ansatz die einzige Lösung dieses Problems zu sein. Die typische Herangehensweise über eine Zerlegung des Objektraums bzw. Bildraums wird bereits durch das prinzipielle Vorgehen beim Projektionsverfahren bzw. Raycasting (siehe oben) nahegelegt. Bei Isoflächenmethoden ist ein paralleler Ansatz (siehe Kapitel 5) erst für größere Datensätze[4] erforderlich.

1.2 Zielstellung

Diese Arbeit stellt sich das Ziel, einen nahezu kompletten Überblick über den aktuellen Stand der Forschung im Bereich sequentieller und paralleler Isoflächenalgorithmen zu geben. Sie möchte damit insbesondere den Wissenschaftler bzw. Anwender unterstützen, der vor der Aufgabe steht, wissenschaftliche Daten mittels Isoflächen zu visualisieren. Die kompakte Darlegung, die dem Leser eine mühevolle und zeitraubende Literaturrecherche erspart, enthält bewußt viele Verweise auf

[4] *Große* Daten beginnen nach heutigem Verständnis bei ca. 10^6 Gitterpunkten. Diese Grenze verschiebt sich jedoch, in Einklang mit der Entwicklung der Meß– und Rechentechnik, ständig nach oben.

elektronische Ressourcen im Internet[5], um einen schnellen Zugriff auf weiterführende Informationen, Literatur und Software zu ermöglichen. Durch eine wertende Darstellung der Verfahren soll der Leser in die Lage versetzt werden, die Tauglichkeit vorhandener Software für seine Zwecke zu beurteilen, diese anzupassen oder gar eigene „maßgeschneiderte“ Programme zu entwickeln. Da letzteres für die meisten an der Visualisierung ihrer Ergebnisse interessierten Wissenschaftler eine unzumutbare Hürde darstellt, sollen im Rahmen dieser Arbeit auch Werkzeuge entwickelt werden, die einer breiten Anwenderschicht eine schnelle und verläßliche Visualisierung selbst größter Datensätze mittels effizienter Isoflächengenerierung erlaubt. Ein Schwerpunkt liegt dabei auf der Unterstützung von Anwendungen auf Höchstleistungsrechnern wie dem CRAY T3D am ZIB, für den bislang keine adäquate Software zur Isoflächengenerierung verfügbar war.

1.3 Gliederung und Umfang der Arbeit

Die vorliegende Arbeit ist thematisch gegliedert. Kapitel 2 definiert die begrifflichen Grundlagen und zeigt die Problematik bei der Konstruktion von Isoflächen aus diskreten Skalarfeldern auf. In Kapitel 3 werden grundlegende sequentielle Ansätze vorgestellt und diskutiert und im besonderen auf den weit verbreiteten *Marching Cubes*–Algorithmus und Abwandlungen davon näher eingegangen. Betrachtungen zur Effizienzproblematik, insbesondere zur Verwendung von hierarchischen Datenstrukturen, und die Vorstellung alternativer Ansätze runden die Darstellung ab. Algorithmen, die die Größe der polygonalen Repräsentation gegenüber herkömmlichen Verfahren deutlich verringern können, sind Thema von Kapitel 4. Das vorletzte Kapitel widmet sich, wie bereits erwähnt, parallelen Ansätzen zur Isoflächengenerierung, und Kapitel 6 faßt die Ergebnisse der Arbeit zusammen.

Über die vorliegende schriftliche Arbeit hinaus wurden zwei umfangreiche Softwareentwicklungen vorgenommen. Auf die Implementation eines Isoflächenmoduls in das Therapie–Planungssystem *HyperPlan* wird in den Abschnitten 3.2 und 4.2 näher eingegangen. Die Ergebnisse dieses Softwareentwurfs fließen unmittelbar in das Projekt „Unterstützung der Hyperthermieplanung durch Verfahren der Bildverarbeitung und Computergraphik“[6] ein.

Die Entwicklung des parallelen Isoflächenprogramms PARADISO für den Parallelrechner CRAY T3D SC 256 wird in Kapitel 5 motiviert und genauer beschrieben. Das Programm soll zur allgemeinen Nutzung zur Verfügung stehen und unter anderem in Projekten aus den Bereichen Visualisierung, CFD und Chemie eingesetzt werden.

Eine Demonstration der Softwarekomponente dieser Arbeit ist vorzugsweise am ZIB auf einer SGI–Workstation mit schneller Netzanbindung[7] an den Parallelrechner CRAY T3D SC 256 (`cray.zib-berlin.de`) möglich.

[5] Die angegebenen Verweise sollten zum Zeitpunkt des Erscheinens dieser Arbeit korrekt sein. Aufgrund der Dynamik des Internet kann zwar keine Garantie übernommen werden, daß die entsprechenden Dokumente jederzeit unter der angegebenen Adresse verfügbar sind, doch der Versuch, schnellsten Informationszugriff zu gewährleisten, soll dennoch nicht unterbleiben.

[6] Projekt des ZIB in Zusammenarbeit mit dem Rudolf–Virchow–Klinkum Berlin, finanziell unterstützt durch die Deutsche Forschungsgemeinschaft (Sonderforschungsbereich 273 „Hyperthermie: Methodik und Klinik“)

[7] Am ZIB ist eine Verbindung via FDDI zwischen ausgewählten SGI–Workstations und dem Rechner `cray` mit einer theoretischen Bandbreite von 100 Mbit/s möglich, In der Praxis lassen sich bei Einzelverbindungen im Tagesbetrieb Übertragungsraten bis ca. 40 Mbit/s erreichen.

Kapitel 2

Isoflächen

In der einschlägigen Literatur wird auf eine formale Definition der Begrifflichkeiten zumeist verzichtet, obwohl die entsprechenden Termini durchaus unterschiedlich gebraucht werden. Diesem Beispiel möchte ich in der vorliegenden Arbeit nicht folgen, weshalb kurz die mathematischen und computergrafischen Grundlagen der Isoflächenberechnung dargelegt und einige Sprachregelungen getroffen werden sollen.

Definition 2.1 *Ein d–dimensionales **Skalarfeld** ist eine Funktion $f : X \to \mathbb{R}$, wobei $X \subseteq \mathbb{R}^d$, $d \in \mathbb{N}$. X heißt **Koordinatenraum** von f. Das Skalarfeld heißt **kontinuierlich**, falls X zusammenhängend ist.*

Da kontinuierliche Skalarfelder auf Funktionen zurückgeführt werden, sind insbesondere Begriffe wie Stetigkeit oder Differenzierbarkeit analog anzuwenden. In dieser Arbeit konzentrieren wir uns allerdings auf eine andere Klasse von Skalarfeldern (siehe auch Kapitel 1), die sich folgendermaßen definieren läßt:

Definition 2.2 *Seien N_x, N_y, N_z positive natürliche Zahlen, $X \subseteq \mathbb{R}^3$ und f eine Funktion von X in $\mathbb{R}$.*

1. *Ein **hexaedrisches Gitter** ist eine Menge $G = \{p \mid p = (x_i, y_j, z_k) \in \mathbb{R}^3,\ i = 0, \ldots, N_x - 1,\ j = 0, \ldots, N_y - 1,\ k = 0, \ldots, N_z - 1\}$ aufsteigender Koordinaten, d. h. die Folge der x_i, y_j und z_k soll jeweils monoton steigend sein.*

   Die Elemente von G heißen Gitterpunkte mit den drei Koordinaten x_i, y_j und z_k.

2. *Ist $G \subseteq X$, so heißt die Menge $S = \{(p, f(p)) \mid p \in G\}$ **dreidimensionales diskretes Skalarfeld** zu f auf dem Gitter G. Den Wert $f(p)$ nennt man auch **Skalarwert** des Gitterpunktes p.*

*N_x, N_y bzw. N_z nennt man **Auflösung** oder **Größe** des Skalarfeldes und des Gitters in x–, y– bzw. z–Richtung.*

Für Komplexitätsbetrachtungen sei im übrigen $N = max\{N_x, N_y, N_z\}$. Im Gegensatz zu den ihnen zugrundeliegenden Funktionen sind so definierte diskrete Skalarfelder prinzipiell unstetig. Die dem diskreten Skalarfeld zugrundeliegende Funktion läßt sich im allgemeinen nicht eindeutig bestimmen. Algorithmen auf solchen Feldern machen deshalb implizit Annahmen über diese Funktion, auf die wir in Kürze zu sprechen kommen. Die identische Abbildung aus dem Koordinatenraum von f (vgl. Definition 2.1) auf den diskreten Gitterraum wird auch als *Abtastung* oder *Diskretisierung* bezeichnet. Zur Vereinfachung der Notation und der sprachlichen Darstellung betrachten wir im folgenden nur noch eine spezielle Klasse von hexaedrischen Gittern, ohne dabei die Problemklasse prinzipiell einzuschränken:

Definition 2.3 *Es sei G ein hexaedrisches Gitter der Größe $N_x \times N_y \times N_z$. G heißt* ***uniformes kubisches Gitter****, wenn es ein $\delta \in \mathbb{R}$ gibt, so daß*

$$G = \{p \mid p = (x_o+i\delta, y_0+j\delta, z_0+k\delta) \in \mathbb{R}^3,\ i = 0,\ldots,N_x-1,\ j = 0,\ldots,N_y-1,\ k = 0,\ldots,N_z-1\}.$$

δ heißt ***Gitterabstand****, das Punkttupel $((x_0, y_0, z_0), (x_0 + (N_x - 1)\delta, y_0 + (N_y - 1)\delta, z_0 + (N_z - 1)\delta))$ nennt man* ***Bounding Box*** *von G. Zwei Gitterpunkte heißen* ***k–adjazent***[1]*, wenn sie sich in höchstens k Koordinaten unterscheiden und in jeder Koordinate um höchstens δ differieren. 1–adjazente Gitterpunkte heißen auch* ***benachbart****, 2–adjazente Punkte* ***diagonal gegenüberliegend****. Eine* ***Gitterzelle*** *wird aus acht zueinander 3–adjazenten Gitterpunkten gebildet. Eine* ***Fläche*** *der Gitterzelle besteht aus je vier paarweise 2–adjazenten Punkten der Gitterzelle. Zwei benachbarte Gitterpunkte bilden eine* ***Gitterkante****.*

Bemerkung 2.4 *Ein Gitter der Größe $N_x \times N_y \times N_z$ hat*

$N_x N_y N_z$	*Gitterpunkte*	(2.1)
$(N_x - 1)(N_y - 1)(N_z - 1)$	*Gitterzellen*	(2.2)
$3N_x N_y N_z - (N_x N_y + N_x N_z + N_y N_z)$	*Zellkanten*	(2.3)

Auf Grundlage der bildhaften Vorstellung eines kubischen Drahtgitters sind die Begriffe aus 2.3 intuitiv verständlich. Alle Erkenntnisse, die auf der Betrachtung dieser Gitterklasse beruhen, lassen sich durch Koordinatentransformation direkt auf allgemeine Gitter mit hexaedrischen Gitterzellen übertragen. Es erfolgt jedoch, anders als in einiger Literatur (z.B. [Nat94]), keine Beschränkung auf uniform dimensionierte Gitter mit $N_x = N_y = N_z$, denn diese lassen sich nicht direkt auf den allgemeineren Fall transformieren.

In Hinblick auf die Isoflächengenerierung treffen wir noch einige spezielle Sprachregelungen für das Gitter und definieren im Anschluß den Kernpunkt dieser Arbeit, die Isofläche.

---

[1] In der Literatur wird leider eine sehr unterschiedliche Terminologie zur Notation von Nachbarschaftsbeziehungen verwendet. $O(6)$–, $O(18)$– bzw. $O(26)$–Adjazenz beispielsweise entspricht 1–, 2– bzw. 3–Adjazenz nach meiner an [VGW94] angelehnten Notation.

Definition 2.5 *Das* ***Vorzeichen*** *eines Gitterpunktes wird als Vorzeichen der Differenz von Skalar– und Isowert definiert. Ein Gitterpunkt heißt* ***negativ****, falls sein Skalarwert kleiner als der betrachtete Isowert ist, sonst heißt er* ***positiv***[2].

Eine Gitterkante heißt ***positiv*** *bzw.* ***negativ****, wenn beide Eckpunkte positiv bzw. negativ sind. Eine Gitterzellfläche heißt* ***mehrdeutig****, falls sie von je zwei positiven und negativen Gitterpunkten gebildet wird, die sich diagonal gegenüberliegen; ansonsten heißt sie* ***eindeutig****.*

Positive bzw. negative Gitterpunkte werden in bildlichen Darstellungen im folgenden als große schwarz (●) bzw. weiß (○) ausgefüllte Kreise repräsentiert. Im Gegensatz dazu erscheinen Eckpunkte von Polygonen als kleine weiß ausgefüllte Kreise ($\circ$).

Definition 2.6 *Sei* $d \in \{2, 3\}$ *und* $X \subseteq \mathbb{R}^d$ *und* $f : X \to \mathbb{R}$ *eine stetige Funktion. Dann heißt*

$$I_c = \{(x_1, \ldots, x_d) \in X \mid f(x_1, \ldots, x_d) = c\}$$

Isolinie *(falls* $d = 2$*) bzw.* ***Isofläche*** *(falls* $d = 3$*) von* f *mit dem* ***Isowert*** $c \in W(f)$.

Isoflächen sind demnach das dreidimensionale Pendant zu Isolinien, die hier nur für stetige Funktionen definiert sind. In der Mathematik sind solche Objekte allgemein als *Niveauflächen* bzw. *Niveaulinien* bekannt ([BS89]). Niveauflächen stetiger Funktionen haben die angenehme Eigenschaft, stetig zu sein, was in unserem Fall bedeutet:

Satz 2.7 *Isoflächen (stetiger Funktionen) sind* C^0*–stetig.*

Im Falle eines diskretes Skalarfeldes kann die Isofläche der zugrundeliegenden Funktion wegen der fehlenden Informationen zwischen den Gitterpunkten nur approximiert werden. Zur schnellen Darstellung unter Nutzung moderner Grafikhardware ist es günstig, eine *polygonale* Approximation vorzunehmen. Dreiecke sind dabei die bevorzugten, einfachsten und am schnellsten zu visualisierenden Grafikprimitive. Die Beschränkung der Betrachtung auf Dreiecke stellt keine Einschränkung dar, weil sich bekanntlich jedes andere einfache n-Eck mit $n > 3$ nach der Formel von EULER in $n - 2$ Dreiecke unterteilen lassen. Die Genauigkeit der *stückweisen Dreiecksapproximation* hängt natürlich von der Auflösung des Gitters ab.

Die Realitätsnähe der Darstellung der polygonalen Isofläche ist stark von der Schattierung (*Shading*), d.h. dem Erzeugen von Farbverläufen in der 3D–Szene zur Unterstützung des räumlichen Sichteindrucks, abhängig. Eine schnelle Schattierung verlangt das Vorhandensein berechneter Normalen, wobei zwischen Flächennormalen (Normalenvektor zu jeder Fläche) und Punktnormalen (Normalenvektor zu jedem Eckpunkt der Fläche) zu unterscheiden ist. Erstere erlauben nur ein individuelles konstantes Schattieren jedes einzelnen Flächenelements, was abrupte Farbunterschiede an Kanten bzw. Ecken zwischen verschieden orientierten benachbarten Polygonen hervorruft

[2] In der Literatur tauchen häufig auch die Bezeichnungen *hot* (für positiv) bzw. *cold* auf. Die deutsche Übersetzung dieser Begriffe trägt allerdings wenig zum intuitiven Verständnis des Sachverhaltes bei.

und dem gesamten 3D–Objekt ein kantiges Aussehen gibt. Punktnormalen erlauben GOURAUD–Shading, bei dem die Farben linear über die Fläche interpoliert werden, oder das komplexere PHONG–Shading, bei dem die Normalen über der Fläche interpoliert werden und daraus die Farben gemäß dem verwendeten Beleuchtungsmodell berechnet werden. Diese Schattierungsverfahren geben dem dargestellten 3D–Objekt ein weicheres Aussehen durch kontinuierlichere Farbübergänge an den Kanten bzw. Ecken und erzeugen damit einen besseren visuellen Eindruck von der Gestalt der zu approximierenden Isofläche. Für nähere Einzelheiten zu Schattierungsverfahren sei auf [FvDFH90] verwiesen. Für kontinuierliche Skalarfelder gilt, daß in jedem Punkt der Niveaufläche Gradient und Normalenvektor die gleiche Richtung (senkrecht zur Fläche) haben müssen ([BS89]). Diese Tatsache kann man sich für die Normalenberechnung zunutze machen. Für diskrete Skalarfelder muß der Gradient $n = (n_x, n_y, n_z)$ der zugrundeliegenden Funktion in einem Gitterpunkt $p = (x_i, y_j, z_k)$ allerdings näherungsweise bestimmt werden, z.B. über zentrale Differenzen entlang der Koordinatenachsen ([LC87]):

$$
\begin{aligned}
n_x &= \frac{G(x_i+\delta, y_j, z_k) - G(x_i-\delta, y_j, z_k)}{2\delta} \\
n_y &= \frac{G(x_i, y_j+\delta, z_k) - G(x_i, y_j-\delta, z_k)}{2\delta} \\
n_z &= \frac{G(x_i, y_j, z_k+\delta) - G(x_i, y_j, z_k-\delta)}{2\delta}
\end{aligned} \tag{2.4}
$$

Die Punktnormalen an den Eckpunkten der Polygone werden dann meist durch lineare Interpolation aus den so berechneten Gitterpunktgradienten bestimmt (siehe auch Abschnitt 3.2).

Bei der Bestimmung unbekannter Funktionswerte an räumlichen Koordinaten wird meist das folgende Interpolationsverfahren, hier speziell für kubische Intervalle, eingesetzt:

Definition 2.8 (Trilineare Interpolation in einem Kubus) *Gegeben seien eine Menge $X \subseteq \mathbb{R}^3$ und Zahlen $x_0, y_0, z_0, \delta \in \mathbb{R}$ mit*

$$\forall\, i, j, k \in \{0, 1\} : \quad (x_0 + i\delta, y_0 + j\delta, z_0 + k\delta) \in X$$

1. *Das **trilineare Interpolant** einer Funktion $f : X \to \mathbb{R}$ ($X \subseteq \mathbb{R}^3$) auf dem Bereich $X_{x_0,y_0,z_0,\delta} = ([x_0, x_0+\delta] \times [y_0, y_0+\delta] \times [z_0, z_0+\delta]) \subseteq \mathbb{R}^3$ ist die Funktion $T : X_{x_0,y_0,z_0,\delta} \to \mathbb{R}$*

mit

$$\begin{aligned}
T(x,y,z) &= A + Bx + Cy + Dz + Exy + Fyz + Gxz + Hxyz \\
A &= f(i,j,k) \\
B &= \frac{f(i+\delta,j,k) - f(i,j,k)}{\delta} \\
C &= \frac{f(i,j+\delta,k) - f(i,j,k)}{\delta} \\
D &= \frac{f(i,j,k+\delta) - f(i,j,k)}{\delta} \\
E &= \frac{f(i+\delta,j+\delta,k) - f(i,j+\delta,k) - f(i+\delta,j,k) + f(i,j,k)}{\delta^2} \\
F &= \frac{f(i,j+\delta,k+\delta) - f(i,j,k+\delta) - f(i,j+\delta,k) + f(i,j,k)}{\delta^2} \\
G &= \frac{f(i+\delta,j,k+\delta) - f(i,j,k+\delta) - f(i+\delta,j,k) + f(i,j,k)}{\delta^2} \\
H &= \frac{f(i+\delta,j+\delta,k+\delta) - f(i,j+\delta,k+\delta) - f(i+\delta,j,k+\delta)}{\delta^3} \\
&+ \frac{f(i,j,k+\delta) - f(i+\delta,j+\delta,k) + f(i,j+\delta,k) + f(i+\delta,j,k)}{\delta^3}
\end{aligned} \tag{2.5}$$

2. *Seien G ein uniformes kubisches Gitter der Größe $N_x \times N_y \times N_z$ mit Gitterabstand δ (gemäß Def. 2.3) und $S : G \to \mathbb{R}$ ein diskretes Skalarfeld darauf. Die Funktion $T : G \to \mathbb{R}$ heißt* ***stückweises trilineares Interpolant*** *von S auf G, wenn sie in jeder Gitterzelle von G das trilineare Interpolant von S ist.*

Mit dieser Definition läßt sich der Begriff sowohl auf kontinierliche Funktionen (wo die Berechnung der Funktionswerte im Intervall möglicherweise zu aufwendig sein kann) als auch auf diskrete Skalarfelder auf uniformen kubischen Gittern anwenden. Trilineare Interpolation ist ein gängiger Ansatz zur schnellen, näherungsweisen Bestimmung des Verlaufs einer Funktion dreier Veränderlicher zwischen diskreten Punkten. In Abschnitt 3.5 werden wir darauf noch einmal zu sprechen kommen. Da ein trilineares Interpolant stetig ist, ist es auch das stückweise trilineare Interpolant, da die einzelnen Stücke nach Definition 2.8 aneinander anschließen müssen. Prinzipiell sind selbstverständlich auch Interpolationen höherer Ordnung (z.B. trikubische Interpolation, siehe Abschnitt 3.6) möglich, verursachen aber einen deutlich höheren Rechenaufwand.

Eine hohe Auflösung der Datensätze ist unter dem Aspekt der topologischen Genauigkeit unabdingbar. Der Speicherplatzbedarf des Datensatzes hängt zusätzlich noch vom Datentyp an jedem Gitterpunkt ab. Die Skalarwerte können ganzzahlig oder reellwertig sein und sind von bestimmter Genauigkeit. Häufig ist hier ein 4–Byte–Gleitkommawert anzutreffen. Man verdeutliche sich, daß bereits ein z.B. von gängigen Computertomographen erzeugtes $256 \times 256 \times 192$–Skalarfeld mit 4–Byte–Werten einen 48 MB großen Datensatz darstellt. Mit der Auflösung und Größe der Datensätze steigt im allgemeinen die Laufzeit des Isoflächenalgorithmus.

Isoflächenalgorithmen, die auf diskreten Skalarfeldern arbeiten, müssen implizit Annahmen über die dem Skalarfeld zugrundeliegende kontinuierliche Funktion, deren Werte die Isofläche ja definieren, machen. Folgende implizite Annahmen sind nahezu allen Isoflächenalgorithmen (siehe auch [VGW94], [NB93]) gemein:

A1 Die Diskretisierung erfolgte in „genügend feiner Auflösung“, d.h. die Varianz der Funktionswerte ist klein in bezug auf die Zellgröße und es treten keine Aliasing–Effekte[3] auf. Es wird insbesondere davon ausgegangen, daß in einer Gitterzelle mit acht Zellecken gleichen Vorzeichens keine (zusammenhängende) Komponente der Isofläche enthalten ist, wie es bei grober Diskretisierung möglich wäre.

A2 Der Schnitt der Isofläche mit einer Gitterzellenfläche ist topologisch äquivalent zu einem (oder mehreren) Liniensegment(en).

A3 Die Gitterkante zwischen zwei benachbarten Gitterpunkten gleichen Vorzeichens hat keinen Schnittpunkt mit der Isofläche. Die Gitterkante zwischen zwei benachbarten Gitterpunkten mit unterschiedlichen Vorzeichen hat genau einen Schnittpunkt mit der Isofläche.

Punkt A2 ist die Grundlage für die Approximation der Isofläche mittels Polygonen, deren Genauigkeit sich an der Gitterauflösung orientiert. Annahme A3, basierend auf dem Zwischenwertsatz für stetige Funktionen, hängt eng mit A1 zusammen, denn bei einer zu groben Diskretisierung eines hochfrequenten Anteils der Funktion ist auch jede andere gerade bzw. ungerade Anzahl von Schnittpunkten möglich. Annahme A3 schlägt sich algorithmisch in der Anwendung von linearer Interpolation zur Bestimmung des Schnittpunktes der Isofläche mit einer Gitterzellenkante nieder. A2 und A3 sind insbesondere konsistent mit der Verwendung eines trilinearen Modells der Daten. In [VGW94] wird noch eine weitere Annahme zugrundegelegt:

> Die Isofläche ist eine stetige Funktion der Eingabedaten. Eine kleine Änderung des Isowertes oder einiger Skalarwerte impliziert nur eine geringe Änderung der Isofläche.

Diese Annahme ist jedoch nach meiner Auffassung nicht notwendig und beispielsweise bei Daten aus der Simulation von Schockwellen nicht erfüllt [Ell95].

Die aus einem diskreten Skalarfeld berechnete Isofläche kann trotz einer hohen Auflösung der Daten gemäß Annahme A1 nur eine (möglichst gute) Approximation der Isofläche der zugrundeliegenden Funktion sein. An einen auf eine breite Menge von solchen Daten anwendbaren polygonal approximierenden Isoflächenalgorithmus stellen sich einige wichtige konkrete Forderungen [VGW94, Nat94]:

F1 **Stetigkeit**: Die erzeugten Polygone sollen eine C^0–stetige Fläche bilden, d.h. an jeder Polygonkante sollten sich entweder genau zwei Polygonen treffen oder die Kante liegt in einem Polygon auf dem Rand des gesamten Skalarfeldes. Die Isofläche darf insbesondere keine „Löcher“ haben.

[3] *Aliasing* ist ein Begriff aus der Signalverarbeitung. Wird ein Signal zu grob abgetastet, so können hochfrequente Anteile des Originalsignals, die oberhalb der sogenannten *Nyquist–Frequenz* liegen, im resultierenden Signal fälschlicherweise als niedrigfrequente Komponenten erscheinen. In der Computergrafik werden alle Artefakte, die durch Digitalisierung analoger Signale oder durch Diskretisierung der Definitionsbereiche zu berechnender Funktionen entstehen, als Aliasing bezeichnet.

F2 **Konsistenz**: Die Isoflächenapproximation sollte *topologisch konsistent* mit der Isofläche des angenommenen Interpolants sein, d.h. beide Flächen sollten die gleichen diskreten Skalarwerte separieren und damit das Skalarfeld in die gleichen zusammenhängenden Komponenten teilen. Insbesondere sollten keine Artefakte wie „Beulen“ oder „Brücken“ entstehen, die nicht durch die Daten impliziert sind.

F3 **Genauigkeit**: Die stückweise Dreiecksapproximation und die Isofläche des angenommenen Interpolants sollten *geometrisch nah* sein.

F4 **Vorzeichen–Invarianz**: Die polygonal approximierte Isofläche sollte *neutral* bezüglich des Vorzeichens der Datenwerte sein, d.h. eine Multiplikation[4] der Datenwerte und des Isowertes mit -1 sollte zu der gleichen Approximation führen.

F5 **Polygonanzahl**: Die Anzahl der erzeugten Polygone (meist Dreiecke) sollte gering sein, um die Zeit für die Darstellung der Polygone zu minimieren.

F6 **Laufzeit**: Die Laufzeit des Algorithmus sollte möglichst gering sein, um ihn für interaktive Anwendungen einsetzen zu können.

Forderung F1 ergibt sich aus Satz 2.7, solange man im Einklang mit den oben gemachten Annahmen davon ausgeht, daß die dem Skalarfeld zugrundeliegende Funktion stetig ist. Treten Polygonkanten, die nicht auf dem Rand des Feldes liegen, einzeln auf und bilden eine geschlossene Kette, so sprechen wir im folgenden von einem *Loch*. Löcher in der generierten Fläche sind beim Erstellen von Bildern für Präsentationen schlichtweg lästig[5]. Für die Interpretation sensibler Daten (wie etwa in der medizinischen Diagnostik) oder die Anwendung analytischer Verfahren auf das Polygongitter sind sie hingegen in keinem Fall mehr hinnehmbar. In der Literatur wird häufig die Forderung nach *topologischer Korrektheit* der produzierten Approximation in bezug auf die Isofläche der dem Feld zugrundeliegenden Funktion gestellt ([NB93]). Man verdeutliche sich jedoch, daß sich auf der Grundlage der Diskretisierung einer kontinuierlichen Funktion keine eindeutigen Aussagen über die dazwischenliegenden Funktionswerte machen lassen. Selbst auf Grundlage der o.g. Annahmen ist die Topologie der produzierten Isofläche nicht eindeutig bestimmt, sondern von der Wahl des Interpolants abhängig, wie das einfache Beispiel in Abbildung 2 zeigt: Die beiden quadratischen Funktionen[6] haben an den Eckpunkten des dargestellten Bereiches, der bei einer Diskretisierung mit $\delta = 1$ einer Gitterzelle entsprechen könnte, exakt die gleichen Funktionswerte. Trotzdem sind die zugehörigen Isoflächen (Isowert 0) topologisch verschieden. Die Forderung F2 bezieht sich deshalb bewußt nur auf das angenommene Interpolant der Funktion, das den Verlauf der Funktion in einer Gitterzelle beschreiben soll.

Formal können wir die Begriffe aus Forderung F2 und F3 in Anlehnung an [Nat94] wie folgt definieren:

[4] Genau genommen sollte dabei eine Ersetzung der Werte $a_{i,j,k}$ durch $-a_{i,j,k} + \epsilon$ (ϵ sehr klein) erfolgen, damit wirklich auch jeder Gitterpunkt ein anderes Vorzeichen gemäß Def. 2.5 erhält.

[5] Sie können natürlich verdeckt, retuschiert oder über eine Darstellung als Drahtmodell „unsichtbar“ gemacht werden.

[6] $F_1(x, y, z) = 4y + 4(x - z)^2 - 5$, $F_2(x, y, z) = 4(y - 1)^2 + 2(x - z)^2 - 2(x + z - 3)^2 + 1$. Diese Funktionen werden in [VGW94] als Testfall für die verglichenen Verfahren verwendet.

Abbildung 2.1: *Isofläche zweier quadratischer Funktionen im Bereich (1,1,1) bis (2,2,2)*

Definition 2.9 *Es sei $\delta \in \mathbb{R}$ und P eine stückweise Dreiecksapproximation einer Fläche I im $\mathbb{R}^3$. P und I heißen*

1. ***topologisch konsistent**, falls für zwei beliebige Gitterpunkte in derselben Zelle gilt: Es existiert ein Pfad[7] zwischen beiden Punkten, der innerhalb der Zelle verläuft und nicht P schneidet, gdw. ein Pfad zwischen beiden Punkten existiert, der innerhalb der Zelle verläuft und nicht I schneidet.*
2. ***geometrisch nah** bzgl. δ, wenn gilt:*

$$\forall\, a \in P\ \exists\, b \in I: \quad b \text{ liegt in einem Würfel um } a \text{ mit der Seitenlänge } 2\delta$$
$$\forall\, b \in I\ \exists\, a \in P: \quad a \text{ liegt in einem Würfel um } b \text{ mit der Seitenlänge } 2\delta$$

Forderung F4 wird in der Literatur kontrovers bewertet. In einigen Arbeiten werden Werte oberhalb des Isowertes als Repräsentation des Objektes betrachtet und diese deshalb anders behandelt als Skalarwerte, die kleiner als der Isowert sind. In Abschnitt 3.3 wird diese Problematik nochmals aufgegriffen.

Das Problem der Generierung einer polygonal approximierten Isofläche läßt sich wie folgt zusammenfassen:

Definition 2.10 (polygonales Isoflächenproblem) *Gegeben seien ein beliebiges diskretes Skalarfeld auf einem kubischen Gitter der Größe $N_x \times N_y \times N_z$, ein Interpolant $T : X \to \mathbb{R}$ $(X \subseteq \mathbb{R}^3)$ und ein Isowert c. Die Fläche $T(x, y, z) = c$ schneide M Zellen des Gitters. Gesucht ist eine stückweise Dreiecksapproximation P der Fläche $T(x, y, z) = c$, die die Forderungen F1–F5 erfüllt.*

Satz 2.11 *Es gibt keinen Algorithmus, der das polygonale Isoflächenproblem in einer Laufzeit unterhalb von $O(M)$ löst.*

[7] Ein *Pfad* zwischen zwei Punkten ist deren Verbindung über beliebig viele Zwischenpunkte. Die Verbindung muß nicht unbedingt durch eine Gerade beschreibbar sein.

Beweis: Um eine topologisch konsistente Approximation der Isofläche zu generieren, muß der Algorithmus für jede Zelle, in denen Gitterpunkte auf verschiedenen Seiten bzgl. der Isofläche liegen, ein oder mehrere Polygone generieren, so daß die Gitterpunkte auch auf verschiedenen Seiten bzgl. dieser Polygone liegen. Nach Voraussetzung gibt es genau M solcher Zellen. Ist t die minimale Laufzeit, um Polygone auf diese Weise zu generieren, so ergibt sich für die Gesamtlaufzeit eines solchen Algorithmus eine untere Schranke von $t \cdot M$. □

Der Satz trifft keine Aussage darüber, ob diese untere Schranke auch die *größte* untere Schranke ist. Forderung F6 beinhaltet den Wunsch, einer solchen Schranke möglichst nahe zu kommen und die Laufzeit pro Zelle (trotzdem) minimal zu halten. Zur Effizienzproblematik gehört, wie bei jedem Algorithmus, auch die Frage nach dem Speicherplatzverbrauch.

Die Forderungen F1–F4 und F6 lassen sich mit einer konstanten Anzahl von Polygonen pro Zelle erfüllen, wie spätestens in den folgenden Kapiteln klar werden wird. Es ist demnach vernünftig, Forderung F5 dahingehend umzusetzen, daß die Anzahl der generierten Dreiecke ebenfalls $O(M)$ (mit einem möglichst kleinen Faktor) betragen soll. Es ist offensichtlich, daß die Forderungen eng miteinander verknüpft sind. Weiterhin bleibt anzumerken, daß M von den Daten, dem Gitter und dem Isowert abhängt und deshalb für gleichgroße Gitter sehr verschieden sein kann. Im durchschnittlichen Anwendungsfall wird M in der Größenordnung von N^2 liegen, weil der Wert durch eine Fläche im Raum bestimmt wird.

In den folgenden Kapiteln werden die prinzipiellen sequentiellen (Kapitel 3–4) und parallelen (Kapitel 5) Isoflächenalgorithmen ausführlich beschrieben und hinsichtlich der Erfüllung der genannten Forderungen, speziell ihrer topologischen Eigenschaften, der Komplexität der Implementation und der Anzahl der erzeugten Polygone verglichen. Während das erste Kriterium für die Verläßlichkeit der Darstellung von Bedeutung ist, sind die anderen vor allem von praktischer Bedeutung, letzteres insbesondere für die Tauglichkeit des Verfahrens für interaktive Anwendungen.

Kapitel 3

Sequentielle Ansätze

Dieses Kapitel konzentriert sich auf herkömmliche *sequentielle* Ansätze zur Isoflächengenerierung aus diskreten dreidimensionalen Skalardaten. Es existiert eine Fülle von Veröffentlichungen mit Vorschlägen für Algorithmen zur Isoflächenberechnung. Im folgenden sollen diese Vorgehensweisen klassifiziert und auf ihre Tauglichkeit zur Lösung des polygonalen Isoflächenproblems (siehe Def. 2.10) hin untersucht werden. Dabei wird insbesondere überprüft, inwieweit die in Kapitel 2 aufgestellten Forderungen erfüllt werden. Hinweise für eine effiziente Implementierung runden die Darstellung ab.

Methoden zur Repräsentation von Punkten konstanten Wertes in 3D–Skalardaten lassen sich in zwei prinzipielle Ansätze unterteilen.

Der erste Ansatz, mit dem sich diese Arbeit ausführlich auseinandersetzt, ist die Lösung des polygonalen Isoflächenproblems durch die Generierung einer zweidimensionalen Fläche im Raum. Methoden, die diesen Ansatz verfolgen, lassen sich unter dem Begriff *Beveled–Surface Methods* [VGW94] zusammenfassen. Frühere zweistufige Ansätze generierten zuerst Isolinien in benachbarten Schichten eines regulären Gitters und verbanden diese zweidimensionalen Konturumrisse in einem zweiten Schritt zu dreidimensionalen Gebilden, was mit erheblichen Schwierigkeiten und Laufzeitaufwand verbunden war. Der wesentliche Nachteil ist der Verlust der Konnektivitätsinformationen zwischen den Schichten nach dem ersten Schritt, was dazu führt, daß sich Mehrdeutigkeiten, z.B. wenn eine Schicht mehr als eine Kontur enthält, nicht mehr korrekt auflösen lassen. Heutige Verfahren betrachten ausnahmslos alle drei Dimensionen auf einmal. Bemerkenswerte Vertreter werden in den folgenden Unterkapiteln, thematisch und historisch geordnet, vorgestellt:

1. Erster *Beveled–Surface*–Algorithmus von WYVILL et al. 1986 (Kap. 3.1)
2. *Marching Cubes* von LORENSEN/CLINE 1987 (Kap. 3.2)
3. Modifizierter *Marching Cubes*, Vorschläge mehrerer Forscher (Kap. 3.3)
4. Gitterzerlegung in Tetraeder (Kap. 3.4)
5. Trilineares Modell von NATARAJAN 1991 (Kap. 3.5)

6. Heuristiken, speziell *Gradient Consistency Heuristics* von VAN GELDER/WILHELMS 1990 (Kap. 3.6)

Im Anschluß (Kap. 3.7) wird diskutiert, mit welchen Techniken sich diese Algorithmen noch weiter beschleunigen lassen.

Der zweite prinzipielle Ansatz, die sogenannte *Cuberille–Methode*, mit der sich Kapitel 3.8 beschäftigen wird, repräsentiert Regionen konstanten Wertes als Menge von Polyedern (meist Würfeln), die die Isofläche enthalten. Zu dieser Klasse von Algorithmen sei auch der *Dividing Cubes*–Algorithmus gezählt, der in Abschnitt 3.8.1 vorgestellt wird. Dieses Verfahren generiert zwar Punkte (und keine Würfel), aber auch diesen liegen nur binäre Entscheidungen darüber zugrunde, ob die Isofläche einen (sehr kleinen) kubischen Gitterausschnitt schneidet oder nicht. Die erzeugten Punkte, jeweils repräsentiert durch die Koordinaten und einen berechneten Normalenvektor, sind prinzipiell als direkte Entsprechung kleiner kubischer Voxel des Objektraums zu betrachten. Die Größe der betrachteten Voxel orientiert sich an der gewählten Auflösung und beeinflußt direkt auch die Anzahl der generierten Punkte. Mit Hilfe der Normalen läßt sich eine realistische Darstellung erreichen.

3.1 Algorithmus für soft objects

Im Jahre 1986 schlug, wenn man es so nennen möchte, die Geburtsstunde der *Beveled–Surface*–Algorithmen. GEOFF WYVILL[1], CRAIG MCPHEETERS, und BRIAN WYVILL[2] veröffentlichten in [WMW86] eine Arbeit über die Modellierung von Objekten, die sich unter der Einwirkung von Kräften verformen (Flüssigkeiten, Gewebe, „weiche“ Gegenstände) — sogenannten *soft objects*. Mengen solcher Objekte, die miteinander in Wechselwirkung stehen können (z.B. aufeinandertreffende Tropfen einer Flüssigkeit, siehe Abb. 3.1) lassen sich als Skalarfeld repräsentieren.

Das Gesamtobjekt ist dabei die Menge all derjenigen Punkte, für die der Skalarwert größer als eine bestimmte Zahl (*magic*) ist. Durch geeignete Wahl der dem Skalarfeld zugrundeliegenden Funktion kann so eine breite Formenvielfalt modelliert werden. Die Oberfläche der Gebilde ist demnach die Isofläche der Feldfunktion. Um ein handhabbares 3D–Objekt zu gewinnen, soll auf einem diskreten Gitter eine polygonale Approximation der Isofläche produziert werden. Obwohl der Isoflächenalgorithmus für ein spezielles Anwendungsgebiet konzipiert wurde, lassen sich die grundlegenden Ideen auch auf allgemeine Anwendungsfälle übertragen.

Die genaue Konstruktion der Feldfunktion, die über den Abstand sogenannter *key points* die Wechselwirkung der Objekte beschreibt, ist in unserem Zusammenhang nebensächlich. Wir konstatieren lediglich:

1. Die Feldfunktion ist kontinuierlich für alle Punkte im Raum definiert, aber die Berechnung ihres Wertes an einem gegebenen Punkt ist laufzeitaufwendig.

[1] Electronic Mail: `geoff@otago.ac.nz`

[2] Electronic Mail: `blob@cpsc.ucalgary.ca`, WWW: `http://www.cpsc.ucalgary.ca/-~blob/home.html`

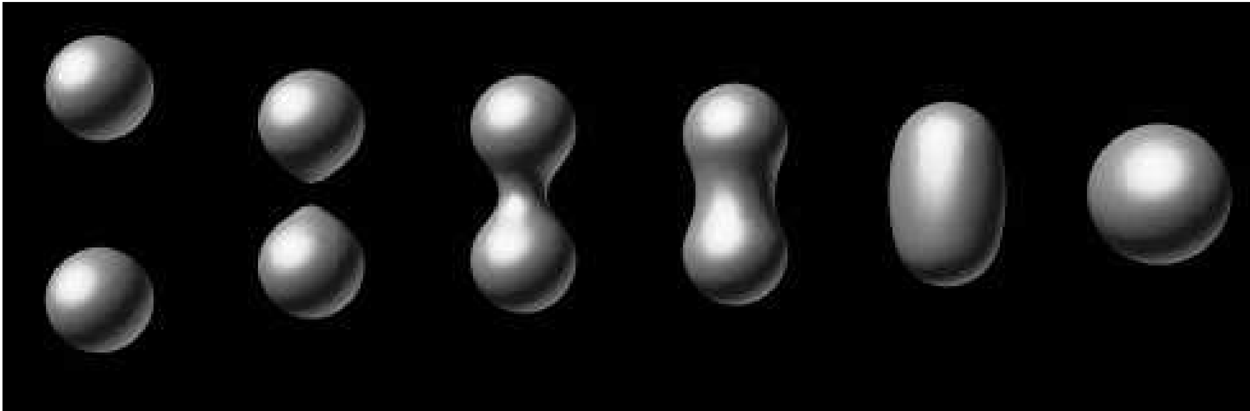


Abbildung 3.1: *„Soft objects“: Verschmelzung zweier Tropfen*

2. Jedes zusammenhängende Teilstück der Isofläche enthält einen *key point*, dessen Funktionswert über dem Isowert *magic* liegt.

Diese beiden Eigenschaften haben entscheidenden Einfluß auf das Design des Algorithmus, wie im folgenden klar werden wird.

Der Algorithmus läuft in zwei Stufen ab. Im ersten Schritt werden in dem gegebenen diskreten Gitter all diejenigen Gitterzellen gefunden, die von der Isofläche geschnitten werden, *ohne* an allen Punkten des Gitters eine teure Berechnung der Feldfunktion vorzunehmen. In einer zweiten Stufe werden nach einer bestimmten Logik für jede im ersten Schritt gefundene Gitterzelle Polygone berechnet, die in ihrer Gesamtheit letztendlich die gesuchte Isofläche bilden.

Erste Stufe: Gitterzellenauslese

Das Vorgehen zu Beginn beruht auf Eigenschaft 2 der Feldfunktion. Für jeden *key point* wird, beginnend beim nächstliegenden Gitterpunkt, in einer fest vorgegebenen Richtung (z.B. entlang der Abszisse) solange der Feldwert des nächsten Gitterpunktes berechnet, bis wir einen Punkt gefunden haben, dessen Funktionswert kleiner als *magic* ist. Dieser Punkt und sein Vorgänger bilden die Kante einer Gitterzelle, die von der Isofläche geschnitten wird. Der gesamte Prozeß liefert uns eine Menge von initialen Gitterzellen (*seed cubes*) mit der Eigenschaft, daß jedes zusammenhängende Teilstück der Isofläche mindestens einen dieser Würfel schneiden muß. Aus dieser Menge werden dann mit einem *Greedy*–Algorithmus *alle* Gitterzellen gefunden, die die Isofläche enthalten. Dieses Vorgehen wirft zwei Teilprobleme auf, die jedes für sich durch die Verwendung geeigneter Datenstrukturen gelöst werden.

Das erste Problem ist die effiziente Berechnung des Wertes der Feldfunktion für einen gegebenen Punkt. Dazu wird der Raum in ein diskretes Gitter[3] aufgeteilt. Da es im Gitter viele Zellen geben

[3]Das in diesem Schritt verwendete grobe Gitter ist *nicht* identisch mit dem feinen Gitter, auf dem später die polygonale Approximation der Isofläche berechnet wird.

kann, die so weit von allen *key points* entfernt sind, daß die Funktionswerte ihrer Eckpunkte gleich Null sind, empfiehlt sich die Nutzung einer Hashtabelle, die nur „nichtleere“ Zellen enthält. Jede Zelle ist durch ihre logische Position im Gitter, also ein Tripel (l, m, n) natürlicher Zahlen, repräsentiert, aus dem auch die Hashadresse berechnet wird. Jede Zeile der Hashtabelle enthält eine Liste von Zellen mit Zeigern auf diejenigen *key points*, die „Einfluß“ auf die Gitterzelle haben, d.h. für die Berechnung mindestens eines Funktionswertes an den Eckpunkten von Bedeutung sind. Die Hashtabelle erlaubt also für eine gegebene Gitterzelle einen schnellen Zugriff auf die relevanten *key points*, aus denen sich der Funktionswert in vertretbarer Laufzeit berechnen läßt.

Trotz aller Bemühungen bleibt die Berechnung der Feldfunktion für einen gegebenen Punkt laufzeitaufwendig. Dies wirft das zweite Teilproblem auf — die Vermeidung doppelter Berechnungen von Feldwerten beim Finden aller von der Isofläche geschnittenen Zellen des fein aufgelösten Gitters, ausgehend von den *seed cubes*. Aus dem gleichen prinzipiellen Grund wie zuvor kommt auch hier eine Hashtabelle zum Einsatz, diesmal enthält sie allerdings Quintupel $(i, j, k, f, \text{done})$. Die Integer–Werte i, j und k, aus denen auch die Hashadresse berechnet wird, sind die Indizes eines Gitterpunktes und repräsentieren gleichzeitig diejenige Gitterzelle, deren „linke untere vordere“[4] Ecke dieser Punkt ist. Wenn auf einen Gitterpunkt das erste Mal zugegriffen wird, schlägt der Zugriff fehl. Daraufhin wird der Feldwert f an diesem Punkt berechnet und das Quintupel $(i, j, k, f, \text{done})$ (Defaultwert für done ist *false*) in die entsprechende Position in der Hashtabelle eingefügt. Das done–Flag dient der Überprüfung, ob die zugehörige Zelle bereits bearbeitet (konkret: done auf *true* gesetzt) wurde. Der Algorithmus zur Bestimmung der von der Isofläche geschnittenen Gitterzellen aus initialen *seed cubes* läßt sich wie folgt formalisieren:

procedure find_all_cubes
    Setze das *done*–Flag des *seed cube* c_s auf **true**
    Füge c_s in eine Queue Q ein
    while $Q \neq \emptyset$ **do**
        Nimm eine Zelle c_1 aus Q
        for each Fläche f_i von c_1 **do**
            if Isofläche schneidet f_i **then**
                finde die Nachbarzelle c_2, die an f_i angrenzt
                if not (done–Flag von c_2) **then**
                    Setze das done–Flag von c_2 auf **true**
                    Füge c_2 in Q ein
                endif
            endif
        enddo
        Gib Eckpunkte und Werte von c_1 an Stufe 2 des Algorithmus weiter
    enddo
endprocedure

[4]Formal ausgedrückt: derjenige Eckpunkt des Gitters mit dem kleinsten Wert $(i + j + k)$.

Zweite Stufe: Polygongenerierung

Als Ergebnis der ersten Stufe haben wir eine Menge von Gitterzellen (mit den Funktionswerten an ihren Eckpunkten), von denen wir wissen, daß die Isofläche sie schneidet. Wir finden nun die Schnittpunkte der Isofläche mit den Gitterkanten (im folgenden kurz Schnittpunkte genannt) und verbinden diese zu Polygonen, deren Gesamtheit die gesuchte Approximation der Isofläche darstellt. Gemäß Annahme A3 (siehe Seite 12) treten diese Schnittpunkte genau auf solchen Kanten auf, deren Endpunkte v_a und v_b ein verschiedenes Vorzeichen bzgl. *magic* haben. Die genaue Position des Schnittpunkts v_s wird näherungsweise (um wiederum teure Berechnungen der Feldfunktion zu vermeiden) über lineare Interpolation der Feldwerte auf der Gitterkante bestimmt:

$$v_s = v_a + \frac{magic - f(v_a)}{f(v_b) - f(v_a)}(v_b - v_a) \tag{3.1}$$

Diese Methode ergibt für alle Zellen, die diese Kante gemeinsam haben, den gleichen Schnittpunkt. Die Verbindung der Schnittpunkte erfolgt über Betrachtung der Gitterzellflächen. Die Flächen lassen sich vier prinzipiellen Konfigurationen[5] zuordnen, die in Abbildung 3.2 dargestellt[6] sind.

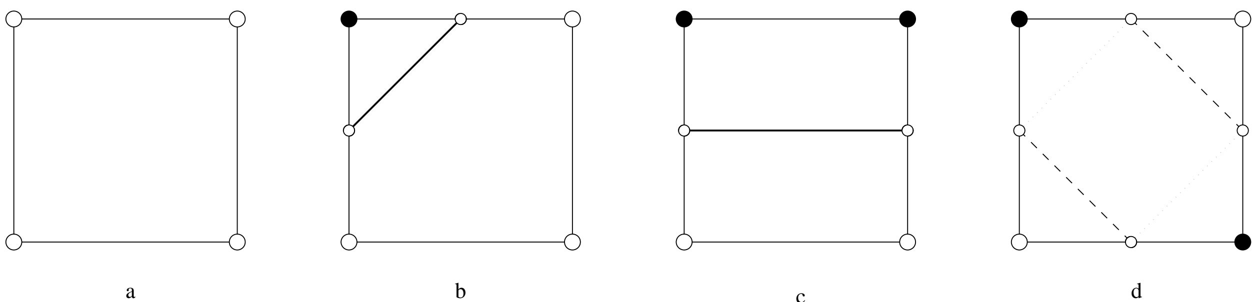


Abbildung 3.2: *Die 4 Fälle von Schnittpunkten auf einer Zellfläche*

Für die Fälle 3.2a–c lassen sich die Schnittpunkte und deren Verbindungen gemäß der Annahmen A2 und A3 (auf Seite 12) eindeutig auf der Grundlage des Feldwertes bzw. Vorzeichens der vier Eckpunkte bestimmen. Fall d stellt jedoch eine mehrdeutige Fläche[7] dar, denn hier ist unklar, ob (durch Teilung entlang der gepunkteten Linien) die negativen Eckpunkte verbunden (und die positiven separiert) oder (entlang der gestrichelten Linien) die positiven Eckpunkte verbunden (und die negativen getrennt) werden sollen. Die Entscheidung darüber trifft der Algorithmus auf der Grundlage des Flächendurchschnitts (*facial average*) f_m, der sich aus den Funktionswerten an den Eckpunkten v_1, v_2, v_3 und v_4 wie folgt berechnet:

$$f_m = \frac{f(v_1) + f(v_2) + f(v_3) + f(v_4)}{4} \tag{3.2}$$

Der Algorithmus folgt dem sogenannten **Flächendurchschnittsprinzip**, indem er die Schnittpunkte entlang der gestrichelten Linie (in Abb. 3.2) verbindet, falls $f_m < magic$ ist, und sonst entlang

[5] Alle $2^4 = 16$ Fälle von „Einfärbungen“ der vier Punkte lassen sich offensichtlich durch Rotationen und/oder Invertieren der Vorzeichen der Gitterpunkte in eine dieser Konfigurationen überführen.

[6] Zur Erinnerung: ● bzw. ○ repräsentieren positive bzw. negative Gitterpunkte.

[7] Spätestens hier wird klar, warum dieser Begriff in Def. 2.5 auf Seite 8 so und nicht anders definiert wurde.

der gepunkteten Linie. WYVILL et al. [WMW86] legen außerdem noch eine Richtung der Linien fest, und zwar immer, von den negativen Eckpunkten aus betrachtet, nach rechts. Die Polygonkanten können dann als geordnete Paare der Schnittpunkte repräsentiert und in einem Array (Index ist der erste Schnittpunkt) gespeichert werden. Der zweite Schnittpunkt ist immer auch erster Schnittpunkt einer anderen Polygonkante, so daß wir das gesamte Polygon erhalten, indem wir sukzessiv den „Nachfolger“ der Kante bestimmen, bis wir wieder bei der Anfangskante angekommen sind. Das Vorgehen zur Generierung der Polygone in einer Zelle c läßt sich wie folgt zusammenfassen:

```
procedure generate_polygons (c)
    for each Kante (v_a, v_b) von c do
        if v_a und v_b haben verschiedene Vorzeichen then
            berechne den Schnittpunkt v_m über lineare Interpolation
        endif
    enddo
    for each Fläche von c do
        generiere Polygonkanten gemäß Flächendurchschnittsprinzip
    enddo
    while (Polygon-)Kanten übrig do
        start ← irgendeine der Kanten
        polygon ← {start}
        entferne start aus dem Kanten-Array
        next ← Nachfolger von start
        while next ≠ start do
            polygon ← polygon + {next}
            entferne next aus dem Kanten-Array
        enddo
        gib polygon aus
    enddo
endprocedure
```

Die so erzeugten Polygone sind nicht notwendigerweise planar. Um das Ziel einer stückweisen Dreiecksapproximation zu erreichen, müssen die Polygone noch tesselliert[8] werden. Dabei wendet der Algorithmus ein Verfahren an, bei dem die Polygoneckpunkte mit einem Punkt im Innern der Zelle (*Centroid*) wie folgt verbunden werden:

Definition 3.1 (barizentrische Tessellation) *Gegeben sei ein n–Tupel $(p_0, \ldots, p_{n-1}), n \geq 4$, von Polygonpunkten. Dann sei*

$$C = (\frac{1}{n}\sum_{i=0}^{n-1} x_i, \frac{1}{n}\sum_{i=0}^{n-1} y_i, \frac{1}{n}\sum_{i=0}^{n-1} z_i) \tag{3.3}$$

und die gesuchten Dreiecke sind $\langle p_{i-1}, p_i, C\rangle$ $(i = 1, \ldots, n-1)$ und $\langle p_{n-1}, p_0, C\rangle$.

[8] Tessellation, also die Zerlegung nichtplanarer Polygone in planare, ist ein Forschungsthema für sich [VGW94]. Hier werden die damit zusammenhängenden Probleme nur ansatzweise gestreift.

Dieses Verfahren erzeugt nach Definition n Dreiecke und arbeitet korrekt für die durch den Algorithmus produzierten Polygone, jedoch nicht für beliebige nichtplanare Polygone.

Der vorgestellte Algorithmus von WYVILL et al. [WMW86] generiert aus einem diskreten Skalarfeld mit bekannter Feldfunktion eine C^0–stetige stückweise Dreiecksapproximation. Das Vorgehen muß sich an den eingangs aufgestellten Forderungen F1–F6 (siehe Seite 12) messen lassen. Zuerst wollen wir Forderung F1 (Stetigkeit) überprüfen und stellen dazu die folgende Behauptung (siehe auch [VGW94]) auf:

Satz 3.2 (Flächenebenenprinzip) *Benutzt ein Isoflächenalgorithmus zur Generierung von Polygonen für eine Gitterzellfläche stets nur Werte aus der Ebene, in der auch die Zellfläche liegt, und ist das Vorgehen invariant gegenüber Rotationen und Reflektionen, so ist die entstehende polygonale Approximation C^0–stetig.*

Beweis: An jeder Zellfläche, die nicht auf dem Rand des Gitters liegt, stoßen genau zwei Gitterzellen zusammen. Nach Voraussetzung werden für jede Zelle in dieser Fläche die gleichen Polygonkanten definiert, d.h. an jeder solchen Kante treffen sich genau zwei Polygone. Liegt die Zellfläche auf dem Rand des Gitters, so tritt die produzierte Kante nur einmal auf. □

Das Flächendurchschnittsprinzip genügt offensichtlich den Voraussetzungen von Satz 3.2 und damit Forderung F1. Das Vorgehen ist außerdem invariant gegenüber einer Invertierung der Datenwerte, d.h. auch Forderung F4 (Vorzeichen–Invarianz) wird erfüllt.

Der Algorithmus hat aber auch mehrere entscheidende Nachteile. Erstens produziert der Tessellationsalgorithmus sehr viele Dreiecke pro Zelle. Im maximalen Fall, der auftritt, wenn jede Zellfläche mehrdeutig ist und kein einzelnes Dreieck generiert wird, entstehen 12 Dreiecke. Zweitens verursacht das Vorgehen hohe Speicherplatzkosten durch die Verwendung der Hashtabellen. Drittens macht die hohe Laufzeit das Programm untauglich für interaktive Anwendungen (für die Generierung solcher Bilder wie in Abb. 3.1 wird eine Größenordnung von mehreren Minuten angegeben). Über die Komplexität der Implementation mag sich der Leser nach Studium dieses Abschnitts selbst ein Bild machen. Sie ist insofern nicht direkt mit der anderer in dieser Arbeit vorgestellter Isoflächenalgorithmen vergleichbar, als daß ein großer Teil des Aufwands mit der Vermeidung der Berechnung von Funktionswerten zu tun hat, die in den meisten anderen Fällen an jedem Gitterpunkt fest vorgegeben sind.

Die Erfüllung von Forderung F2 (Konsistenz) bedeutet hier, da die zugrundeliegende Funktion bekannt ist, daß die stückweise Dreiecksapproximation topologisch konsistent mit der Isofläche dieser Funktion ist. Für die in [WMW86] verwendete Funktion mit relativ einfachem Verlauf scheint dies zuzutreffen. Inwieweit dies auch für andere für den Anwendungsfall *soft objects* denkbare Funktionen zutrifft, kann nur vermutet werden.

3.2 Marching Cubes

Während der Algorithmus von WYVILL et al. [WMW86] als „Spezialanwendung“ kaum Einzug in gängige Softwareprodukte fand, entwickelte sich der 1987 von WILLIAM E. LORENSEN[9] und HARVEY E. CLINE vorgestellte *Marching Cubes*–Algorithmus ([LC87]) rasch zu *dem* Standardalgorithmus für die Isoflächenberechnung. Dies ist sowohl verständlich, als auch, was seinen Fortbestand bis in die heutige Zeit angeht, verwunderlich. Bevor ich aber seine Vor– und Nachteile diskutiere, möchte ich ihn erst einmal in der gebotenen Ausführlichkeit beschreiben.

Der Algorithmus wurde grundsätzlich für die Anwendung auf medizinische 3D–Daten (CT, MRT, SPECT) zur Untersuchung der Strukturen von Knochen, Blutgefäßen oder inneren Organen konzipiert, macht aber, abgesehen von der Regelmäßigkeit des Gitters, keine speziellen impliziten Annahmen bezüglich der Natur der Daten. Output des Algorithmus ist ein Dreiecksmodell der Isofläche. Dabei geht man wie folgt vor:

Wie der Name des Algorithmus, sinngemäß als „Methode des wandernden Kubus“ [Mei90] übersetzt, nahelegt, wird das Gitter nacheinander in einer festen Reihenfolge Zelle für Zelle durchlaufen und dabei zu einem Zeitpunkt stets nur die aktuelle (kubische) Zelle betrachtet. Mit der Maßgabe der Annahmen A2 und A3 (siehe Seite 12) wird auch hier, wie im Algorithmus von WYVILL et al., das Vorzeichen der Gitterpunkte bestimmt, um auf die Topologie der zu erzeugenden Polygone zu schließen. Der wesentliche Unterschied ist, daß der *Marching Cubes*–Algorithmus die Konfiguration der gesamten Gitterzelle (statt jeder Fläche einzeln) auf einmal untersucht und daraus direkt ohne weitere Berechnungen (wie die eines Flächendurchschnitts) oder Anwendung von Tessellationsverfahren die Topologie der Dreiecks–Repräsentation des entsprechenden Teilstücks der Isofläche generiert.

In einer Zelle kann jeder der acht Eckpunkte ein positives oder negatives Vorzeichen haben, womit sich $2^8 = 256$ zu betrachtende Fälle ergeben. Unter Berücksichtigung von Symmetrien (Rotationen, Reflexionen und Invertieren der Vorzeichen) reduziert sich diese Anzahl auf 14 Basisfälle (*major cases* [VGW94]). LORENSEN/CLINE nennen 15 Fälle, die in Abbildung 3.3 dargestellt sind. Hier irren die Autoren, denn bei genauer Betrachtung der 15 Fälle fällt auf, daß sich Fall 11 und Fall 14 ineinander überführen lassen. Zu jedem der 15 Fälle ist jeweils eine Verbindung der Schnittpunkte zu (maximal vier) Dreiecken dargestellt. Diese Informationen können statisch in einer Lookup–Tabelle[10] gespeichert werden, die für jeden Fall die Topologie der zu erzeugenden Dreiecke als geordnete Tripel von (Nummern von) Kanten enthält, auf denen die Schnittpunkte liegen. Die genaue Position der Schnittpunkte wird durch lineare Interpolation (gemäß Gleichung 3.1) bestimmt.

Konkret läuft der Algorithmus für jede Zelle folgendermaßen ab: Das Vorzeichen als Ergebnis des Vergleichs von Feldwert und Isowert wird an jeder der acht Zellecken mit einem Bit (z.B. 1

[9] Electronic Mail: `lorensen@graphics.stanford.edu`, WWW: `http://www-graphics.stanford.edu/~lorensen/`

[10] Die korrekte englische Bezeichnung lautet *look–up table*. Der Autor bittet um Nachsicht wegen der saloppen deutsch–englischen Mischform des gewählten Begriffes.

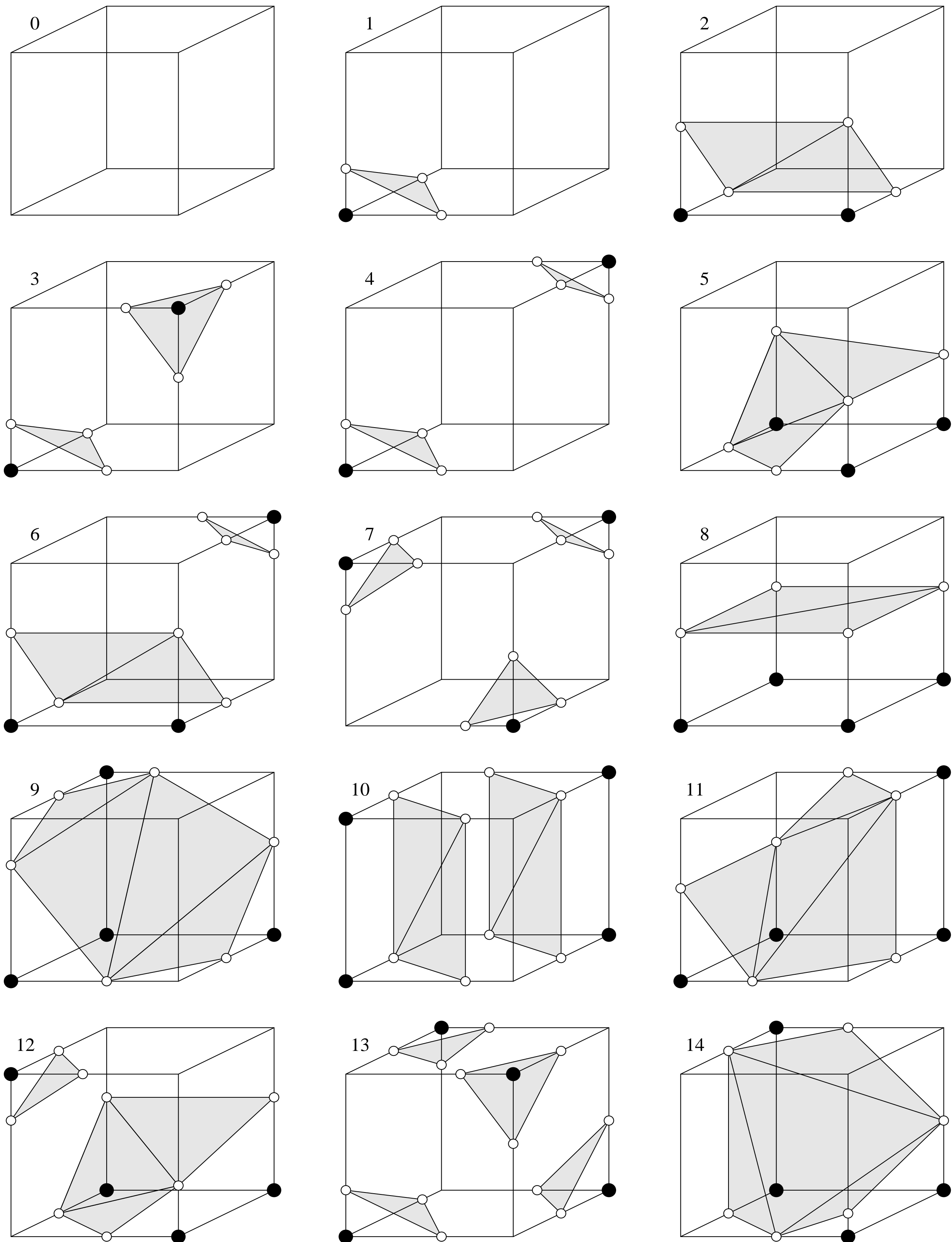


Abbildung 3.3: *Die 15 Basisfälle für die Zelltriangulierung beim Marching Cubes (negative Gitterpunkte sind aus Gründen der Übersichtlichkeit nicht gekennzeichnet).*

für positives und 0 für negatives Vorzeichen) kodiert. Daraus wird auf Grundlage einer festen Bezeichnungsweise ein 8bit–Index gebildet (siehe Abb. 3.4), der als Zeiger in die Lookup–Tabelle dient. Für jedes der aus der Tabelle gelesenen Kantentripel wird die genaue Position des gesuch-

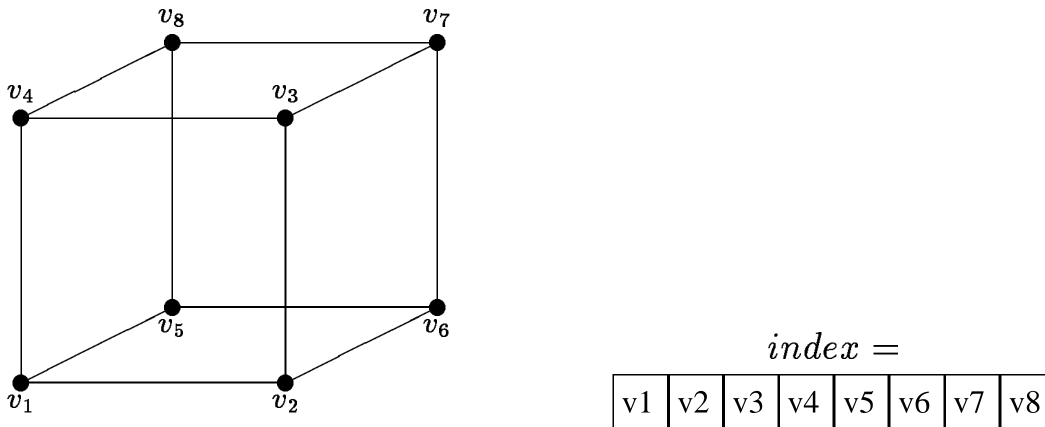


Abbildung 3.4: *Bezeichnung der Würfelecken und Berechnungsweise des 8bit–Indexes*

ten Schnittpunkts auf dieser Kante über lineare Interpolation bestimmt. Die Tripel dieser Schnittpunktpositionen bilden die letzendliche Dreiecksfläche. Zusätzlich können zu jedem Schnittpunkt Punktnormalen gemäß Gleichung 2.4 auf Seite 10 berechnet werden, die der Dreiecksfläche ein glattes Aussehen durch geeignetes Shading verschaffen.

Der *Marching Cubes*–Algorithmus liest sich in Pseudocode wie folgt:

algorithm marching_cubes
    for each Gitterzelle **do**
        berechne 8bit–Index
        finde über den Index eine Menge von Kantentripeln in der Lookup-Tabelle
        for each Kantentripel (e_1, e_2, e_3) **do**
            for $i \leftarrow 1$ **to** 3 **do**
                berechne Schnittpunkt s_i auf der Kante e_i
                berechne Einheitsnormale n_i im Punkt s_i
            enddo
        gib das Dreieck (s_1, s_2, s_3) mit den zugehörigen Normalen aus
        enddo
    enddo
endalgorithm

Wie man sieht, ist der *Marching Cubes*–Algorithmus simpel und problemlos zu implementieren. Er kommt ohne einen expliziten Tessellationsschritt aus, denn die Dreiecksaufteilung ist bereits durch die Topologietabelle vorgegeben. Dabei produziert er zwischen einem und vier Dreiecken

pro Zelle, also insgesamt zwischen M und $4M$. Eine geringere Dreiecksanzahl läßt sich nur mittels polygonreduzierender Verfahren (siehe Kap. 4) erreichen. Seine Laufzeitkomplexität beträgt allerdings $O(N^3)$ und ist damit weit von der konkretisierten Forderung F6 (Laufzeit) entfernt. Die Zeit für das Bearbeiten einer Gitterzelle ist wiederum dermaßen gering, daß die Laufzeit des Algorithmus für die meisten kleineren Anwendungen der wissenschaftlichen Visualisierung ausreicht. Die aufgezählten Vorteile dürften genügen, um die breite Verwendung von *Marching Cubes* in vielen kommerziellen und frei verfügbaren Grafik–und Visualisierungstools zu erklären. Für hochaufgelöste Datensätze, wie sie mittlerweile gerade in der Medizin, für die der Algorithmus ja eigentlich entwickelt wurde, gang und gäbe sind, ist die Laufzeit allerdings für eine interaktive Anwendung zu hoch. Hier bieten sich als Ausweg vor allem parallele Ansätze an. Das Prinzip der unabhängigen Bearbeitung der Gitterzellen birgt ein enormes Maß an Datenparallelität. Doch auch ein sequentieller *Marching Cubes* und ihm ähnliche Verfahren lassen sich durch einige Abwandlungen noch effizienter gestalten, worauf ich in Abschnitt 3.7 näher eingehen werde.

Den entscheidenden Nachteil des originalen *Marching Cubes* formulierte bereits 1988 DÜRST in [Dü88]: der Algorithmus produziert nicht notwendigerweise eine C^0–stetige Dreiecksfläche! Abbildung 3.5 zeigt einen solchen Fall: In dem an der gemeinsamen, mehrdeutigen Zellfläche generierten Teil der Isofläche tritt ein Loch auf. Daß dieses Loch ein topologischer Fehler sein muß,

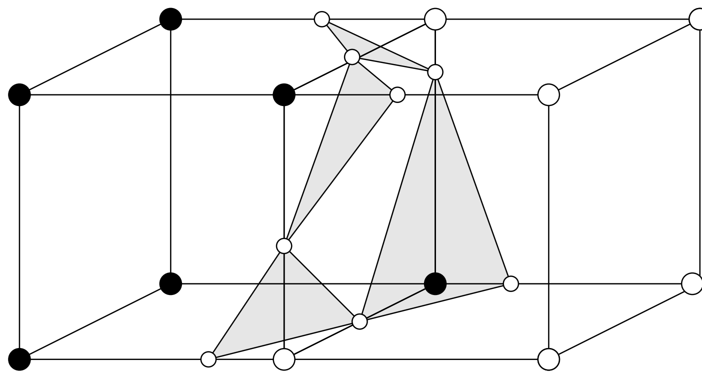


Abbildung 3.5: *Loch in einer durch den originalen Marching Cubes–Algorithmus generierten Isofläche*

wird spätestens dann deutlich, wenn man die Raumdiagonale quer durch beide Gitterzellen betrachtet, die die Dreiecksfläche nicht schneidet, obwohl beide Eckpunkte ein anderes Vorzeichen haben und deswegen auf verschiedenen Seiten der Isofläche liegen müßten. In der rechten Gitterzelle liegt Basisfall 3 vor und auch die linke wird durch Invertieren der Vorzeichen in diesen überführt. Die Verbindung der Schnittpunkte auf der gemeinsamen mehrdeutigen Zellfläche wird in den beiden Zellen dadurch unterschiedlich gehandhabt: links werden die negativen Gitterpunkte separiert, rechts die positiven. Die Behandlung der gemeinsamen Fläche ist hier abhängig vom Aussehen der gegenüberliegenden Fläche in der Zelle — diese Inkonsistenz muß zwangsläufig zu topologischen Fehlern führen. Die Anzahl solcher Löcher variiert je nach Daten und Isowert stark, und liegt meist im Bereich von 0–5% der Dreiecke [RHvK95]. Die Häufigkeit des Auftretens mehrdeutiger Fälle hängt insbesondere mit der „Glattheit“ der Isofläche zusammen.

Aufgrund dieses schwerwiegenden Fehlers im *Marching Cubes*–Algorithmus sollte er nie in der Originalform angewendet werden, sondern nur noch in geeigneter Abwandlung. Varianten dafür werden im folgenden Kapitel 3.3 vorgestellt.

Leider ist der originale *Marching Cubes* auch heute noch in den aktuellen Versionen einiger Visualisierungspakete, z.B. IRIS Explorer 3.0[11] implementiert. Allgemein ist zu bemängeln, daß in der meisten Software Isoflächenmodule als „Black Box“ ohne Angabe des zugrundeliegenden Algorithmus erscheinen. Aufgrund der feinen bis massiven Unterschiede zwischen den Algorithmen sollte dem kompetenten Anwender zumindest die Möglichkeit gegeben werden, die Eignung des Moduls für seine Zwecke im vorhinein einzuschätzen.

3.3 Korrigierter Marching Cubes

Für eine Abwandlung des *Marching Cubes*–Algorithmus zur Sicherstellung der Stetigkeit der generierten Fläche wurden seit 1988 viele Vorschläge verschiedener Forscher (siehe auch [VGW94], [NB93]) gemacht. Die Grenzen zwischen *Korrekturen* und *Alternativen* sind dabei fließend. Ich stelle mich auf den Standpunkt, daß eine Korrektur des *Marching Cubes*–Algorithmus lediglich Details betreffen darf und dabei die prinzipielle Vorgehensweise, dokumentiert im aufgeführten Pseudocode, beibehalten werden muß. Zu solchen Korrekturen zählen insbesondere Veränderungen der Topologietabelle.

Eine inhärente Eigenschaft des *Marching Cubes*–Algorithmus ist, die Topologie der polygonalen Approximation allein aus dem Vorzeichen der Gitterpunkte zu bestimmen. Eine andere, „neue“ Herangehensweise machen sich die Algorithmen, die in den nachfolgenden Abschnitten 3.4, 3.5 und 3.6 vorgestellt werden, zu eigen. Sie treffen zur Laufzeit Entscheidungen über die Topologie in Abhängigkeit von den genauen Skalarwerten an den Gitterpunkten. Diese Variante ist so neu wiederum nicht, denn sie wird ja bereits im Verfahren von WYVILL et al. (Kap. 3.1) in Form des einfachen Flächendurchschnittsprinzips verwendet; es kommt jedoch sehr auf ihre genaue Ausprägung an.

In manchen Grundzügen, z. B. dem Prinzip, einen „wandernden Kubus“ zu verwenden, sprich zu einem Zeitpunkt nur eine Gitterzelle[12] zu betrachten, stimmen einige der Algorithmen überein. Ich werde für jeden vorgestellten neuen Algorithmus die Unterschiede und Gemeinsamkeiten im jeweiligen Abschnitt deutlich herausarbeiten.

3.3.1 Modifizierte Topologietabelle

Der einfachste und naheliegendste Weg für eine direkte Korrektur des *Marching Cubes*–Algorithmus wird in [MSS94b] und [Lor94] vorgeschlagen und soll im folgenden eingehend erläutert werden.

[11]Der Name des Moduls ist *IsosurfaceLat*. In der Beschreibung wird lapidar vermerkt: „Known Problems: This module can produce surfaces with holes“.

[12]Wir beziehen uns dabei nur auf die Erzeugung der Polygone an sich. Für die Berechnung der Normalen gemäß Gleichung 2.4 werden natürlich auch Skalarwerte außerhalb der aktuellen Zelle, und zwar von bis zu sechs Nachbarzellen, herangezogen.

Wir rufen uns ins Gedächtnis zurück, daß die Isoflächenapproximation genau dann C^0–stetig ist, wenn an jeder Dreieckskante genau zwei Dreiecke zusammentreffen oder die Kante auf dem Rand des Feldes liegt (siehe Seite 13). Da die Dreieckskanten innerhalb der Gitterzelle nach Definition der Topologietabelle offensichtlich verbunden sein müssen, können Löcher nur an den Zellflächen auftreten, an denen zwei Gitterzellen zusammenstoßen. Durch einfache vollständige Untersuchung[13] aller dabei auftretenden Fälle läßt sich folgende Aussage nachweisen:

Satz 3.3 *Ein Loch in der vom Marching Cubes–Algorithmus produzierten Isoflächenapproximation tritt genau an solchen mehrdeutigen Zellflächen auf, wo eine Zelle mit zwei, drei oder vier positiven Gitterpunkten (Basisfall) auf eine Zelle mit fünf oder sechs positiven Gitterpunkten (invertierter Basisfall) trifft.*

□

Gelingt es, für diejenigen inversen Basisfälle, in denen eine mehrdeutige Fläche vorliegt, eine neue konsistente Triangulierung zu finden, kann man dieses Dilemma beseitigen. Die betreffenden Basisfälle sind, wie sich aus Abbildung 3.3 leicht ersehen läßt, die Fälle 3, 6, 7, 10, 12 und 13. In jedem der Basisfälle werden auf mehrdeutigen Zellflächen positive Gitterpunkte separiert (vergleiche die gepunktete Linie in Abb. 3.2 auf Seite 20). Eine konsistente Dreiecksapproximation für die dazu inversen Fälle (im folgenden durch Anhängen des Buchstaben c gekennzeichnet) zu finden, bedeutet also, die gegebenen Schnittpunkte so zu verbinden, daß auf mehrdeutigen Zellflächen ebenfalls dieses Prinzip befolgt wird. Gemäß Satz 3.2 (Seite 22) führt dies zu einer C^0–stetigen Fläche. Für die Fälle 10c, 12c und 13c läßt sich die neue Zelltriangulierung durch Symmetriebetrachtungen direkt aus der im originalen *Marching Cubes*–Algorithmus zur Anwendung kommenden ableiten, d.h. es entsteht dabei auch die gleiche Anzahl von Dreiecken. Die Fälle 3c, 6c und 7c erfordern dagegen ein völlig neues Triangulationsschema. Dabei erhöht sich die Anzahl der generierten Dreiecke von zwei auf vier (Fall 3c) bzw. von drei auf fünf (Fälle 6c und 7c). Das genaue Schema [Lor94] ist in Abbildung 3.6 dargestellt. Es ergeben sich bei Anwendung dieser Methode also insgesamt 20 Basisfälle, wenn man wiederum davon ausgeht, daß sich die Fälle 11 und 14 (siehe Abb. 3.3) als ein Basisfall darstellen lassen. Der prinzipielle Ablauf bleibt jedoch identisch zum Vorgehen beim originalen *Marching Cubes*–Algorithmus (siehe Seite 25). Die Zeit für den Zugriff auf die Topologietabelle wird durch die neue Fallunterscheidung nicht erhöht, es vergrößert sich lediglich der Speicherplatzbedarf für eine minimale Lookup–Tabelle von 14 auf 20 Einträge. Oft wird jedoch aus Effizienzgründen (siehe Abschnitt 3.7) sowieso eine vollständige Tabelle mit 256 Einträgen verwendet, so daß sich auch hier kein Unterschied ergibt. Es werden auch nur unwesentlich mehr Dreiecke erzeugt, denn die Fälle 3c, 6c, 7c und 12c treten vergleichsweise selten[14] auf.

Der hier vorgestellte Ansatz wurde unabhängig voneinander[15] sowohl von LORENSEN selbst [Lor94] als auch von MONTANI et al. [MSS94b] entwickelt. Die Verfahren unterscheiden sich lediglich in der konkreten Unterteilung der Polygone in Dreiecke für die Fälle 3c, 6c, 7c und 12c.

---

[13] In [RHvK95] wird der Beweis mit Hilfe eines Computerprogramms geführt, das für alle $2^{12} = 4096$ möglichen Konfigurationen von Gitterpunkten benachbarter Gitterzellen testet, ob die an der gemeinsamen Zellfläche entstehenden Dreieckskanten paarweise auftreten.

[14] Eine kleine Statistik über die Häufigkeitsverteilung der Basisfälle in typischen wissenschaftlichen Daten findet sich z.B. in [VGW94], S. 345. Das Auftreten der Fälle 3, 6, 7 und 12 (inklusive der invertierten Fälle) bewegt sich dabei im Bereich von 0–2% von der Gesamtzahl M aller von der Isofläche geschnittenen Zellen.

[15] Diese Information entstammt einer privaten Korrespondenz mit W. E. LORENSEN.

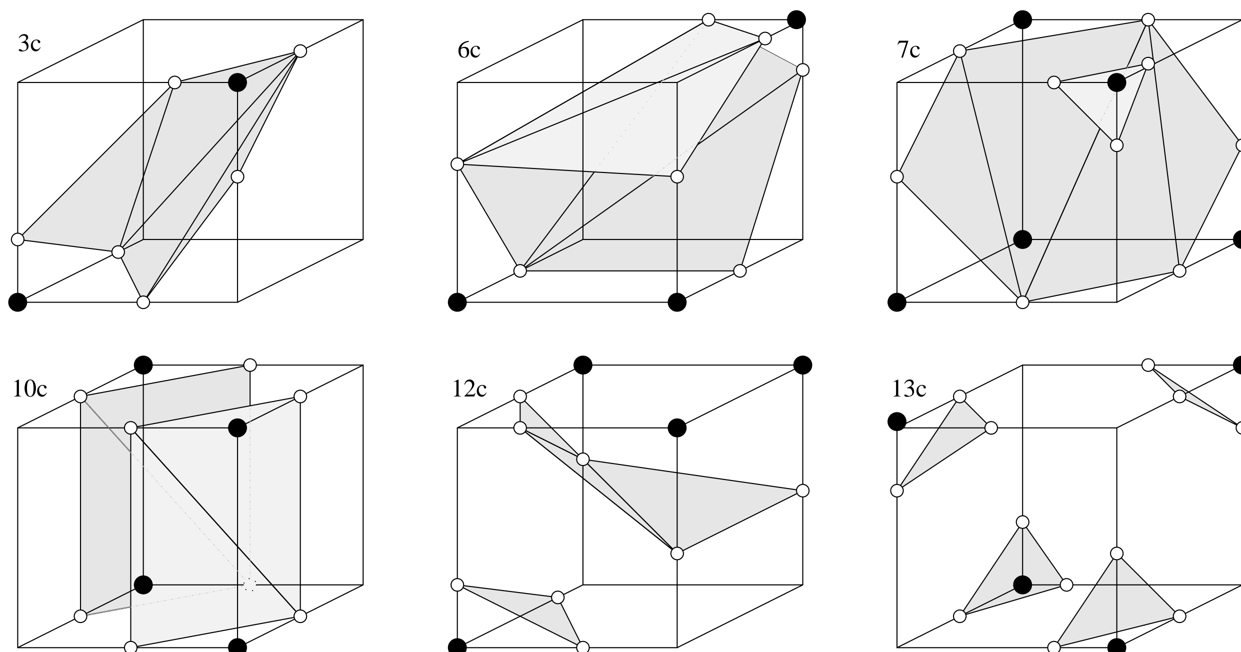


Abbildung 3.6: *Die sechs modifizierten komplementären Fälle*

Ein Nebeneffekt dieser Lösung ist, daß die (umstrittene) Forderung F4 (Vorzeichen–Invarianz) nicht mehr erfüllt wird. In [MSS94b] wird dies zur Kenntnis genommen und für vertretbar gehalten. Dieser Meinung können wir uns mit Einschränkungen anschließen. Es bleibt zu bemerken, daß das Verfahren nach unserer Terminologie keineswegs eine „korrekte“ Isofläche generiert, wie in [MSS94b] behauptet wird, sondern eine, die C^0–stetig, aber nicht topologisch konsistent gemäß Forderung F2 ist. Letzteres kann nicht erreicht werden, denn die Verbindung von Schnittpunkten auf einer mehrdeutigen Zellfläche wird lediglich vom Vorzeichen der Gitterpunkte und nicht vom genauen Verlauf eines numerischen Interpolants abhängig gemacht. In den meisten Anwendungsfällen wird eine solche Vereinfachung, zieht man die vielen Vorteile eines solchen *Marching Cubes*–Ansatzes in Betracht, genügen, insbesondere, wenn über die Natur der Daten keine genauen Aussagen getroffen werden können, also auch kein angenommenes Interpolant vorliegt. Die Topologie der erzeugten Approximation ist wohldefiniert und läßt sich in folgender Eigenschaft zusammenfassen:

Satz 3.4 *Die mit dem korrigierten Marching Cubes–Algorithmus generierte Fläche verbindet jedes Paar k–adjazenter positiver Gitterpunkte $(k \geq 1)$, das sich über positive Gitterkanten verbinden läßt. Jedes Paar k–adjazenter positiver Gitterpunkte, das sich nicht über positive Gitterkanten verbinden läßt, wird durch die Fläche getrennt.*

Insbesondere bedeutet dies, daß jedes Paar positiver benachbarter Gitterpunkte als zusammengehörig betrachtet und infolgedessen vom gleichen Teil der Fläche eingeschlossen wird.

3.3.2 Implementation in HyperPlan

Das im vorangegangenen Abschnitt ausführlich erläuterte Verfahren wurde von mir mit einigen Effizienzverbesserungen (siehe Kap. 3.7) als Referenzalgorithmus in das Hyperthermie–Planungssystem *HyperPlan* [SH95a] implementiert. HyperPlan stellt eine experimentelle Umgebung bereit, die höchste Ansprüche an Ergonomie, Erweiterbarkeit, Interaktion und Stabilität erfüllt. Die Software stellt modernste Visualisierungsmethoden wie Volume–Rendering mit 3D–Texturen, Textured Splats und Linienintegralfaltung [SH95b] über *Module* zur Verfügung. Mein Isoflächenmodul *HxIsosurface*, implementiert in C++, berechnet aus einem Skalarfeld auf einem hexaedrischen Gitter eine stückweise Dreiecksapproximation in Form eines dreidimensionalen *Open Inventor™*–Objekts, mit dem sich dann in einem Grafikfenster direkt interagieren läßt. Es bietet sich damit z.B. die Möglichkeit, das Objekt aus einem (oder mehreren) beliebigen Blickwinkel(n), mit einer wählbaren Vergrößerung, in der gewünschten Farbe und geeigneten Darstellungsformen (schattiert, Drahtmodell, Gitter mit verdeckten Kanten usw.) zu betrachten. Als Eingabegeräte dienen Tastatur und Tischmaus bzw. 3D–*Space Mouse*. Veränderbare Parameter für das Isoflächenmodul sind der Isowert und die Art der Berechnung. Neben einem korrigierten *Marching Cubes*–Algorithmus wurde ein Verfahren mit interner Polygonreduktion namens *Compact Cubes* implementiert, auf das in Abschnitt 4.2 näher eingegangen wird. Als Eingangsdaten kommen aufgrund des geplanten Einsatzes in der Krebstherapieplanung vor allem hochaufgelöste computertomographische Daten, häufig mit 150–200 Schichten und einer Auflösung von 256×256 pro Schicht, in Frage. Ein typischer Anwendungsfall für das Isoflächenmodul in HyperPlan ist auf Farbbild 3 (Seite 73) zu sehen.

In der Variante von [Lor94] steht der korrigierte *Marching Cubes*–Algorithmus seit neuestem auch als Klasse *VtkMarchingCubes* im *Visualization Toolkit*[16] zur Verfügung.

3.3.3 Patches

Ein weiterer Vorschlag zur Korrektur des *Marching Cubes*–Algorithmus stammt von RÖLL[17], HAASE und VON KIENLIN [RHvK95]. Eine genaue Untersuchung der durch Satz 3.3 beschriebenen mehrdeutigen Fälle fördert zutage, daß die vom *Marching Cubes*–Algorithmus produzierten „Löcher“ sämtlich aus vier einzeln auftretenden Kanten gebildet werden, die ein Viereck auf einer mehrdeutigen Zellfläche bilden. Dies führt direkt zu der Idee, die Löcher mit einem Viereck passender Form auszufüllen. Diese Idee taucht bereits in [NH91] auf, wird dort jedoch mit dem Hinweis verworfen, daß das Einführen eines zusätzlichen Vierecks für *jede* mehrdeutige Zellfläche dazu führen kann, daß eine Kante zu mehr als zwei Polygonen gehört. Diese Beobachtung ist natürlich korrekt, wie Bild 3.7b verdeutlicht. Solche „Flicken“ (*patches*) dürfen also nur dann

[16] VTK ist eine C++–Klassenbibliothek für Grafik und Visualisierung. Autoren sind W. SCHROEDER, K. MARTIN und W. E. LORENSEN. Die aktuelle Version (z. Z. 1.0 vom November 1995) ist im WWW unter `http://www.cs.rpi.edu/~martink/` mit Quellen, Beispielen und Dokumentation frei verfügbar. Sie läuft unter UNIX, Windows NT und Windows 95 und unterstützt OpenGL, Starbase, XGL und X.

[17] Electronic Mail: `roell@tomo.uni-bremen.de`. Die Arbeit stellte im Arbeitsgebiet der Gruppe am Physikalischen Institut der Uni Würzburg nur ein Randthema dar und wurde laut Information von S. RÖLL nicht weiter fortgeführt. Eine ANSI–C–Implementation des Algorithmus für wissenschaftliche Zwecke ist auf Nachfrage erhältlich.

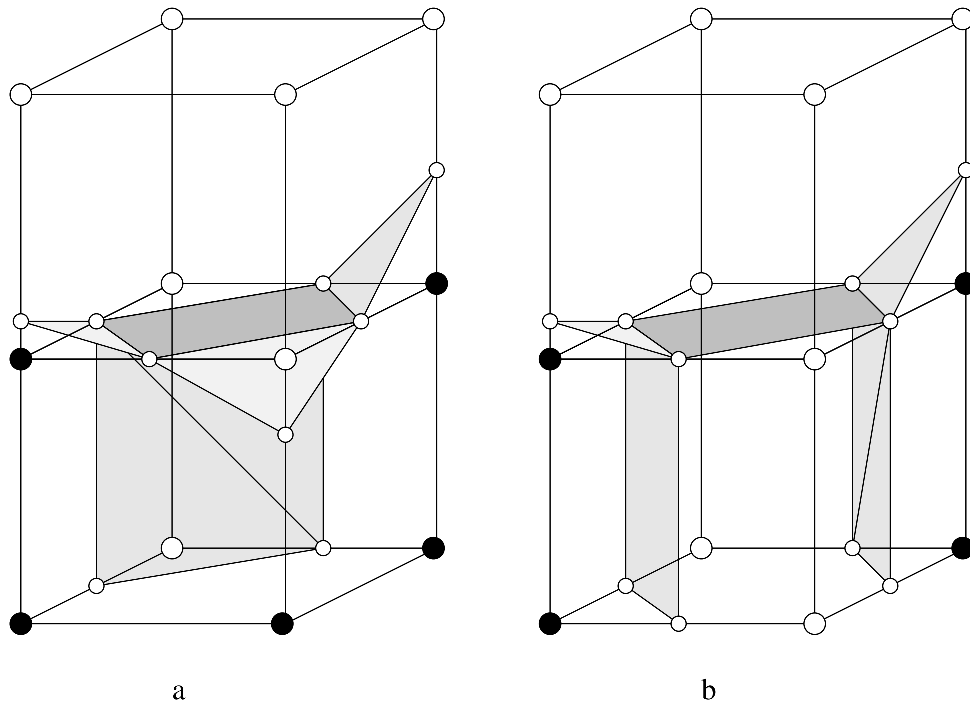


Abbildung 3.7: *Die Einführung zusätzlicher Vierecke (dunkel) auf mehrdeutigen Flächen darf nur erfolgen, wenn invertierte auf nicht invertierte Basisfälle treffen (links). Anderenfalls sind die entstehenden polygonalen Flächen nicht mehr einfach (rechts).*

eingesetzt werden, wenn die Bedingungen von Satz 3.3 erfüllt sind. Der Patch–Algorithmus teilt sich deshalb für jede elementare Gitterzelle in die Phasen Lochermittlung und Lochreparatur:

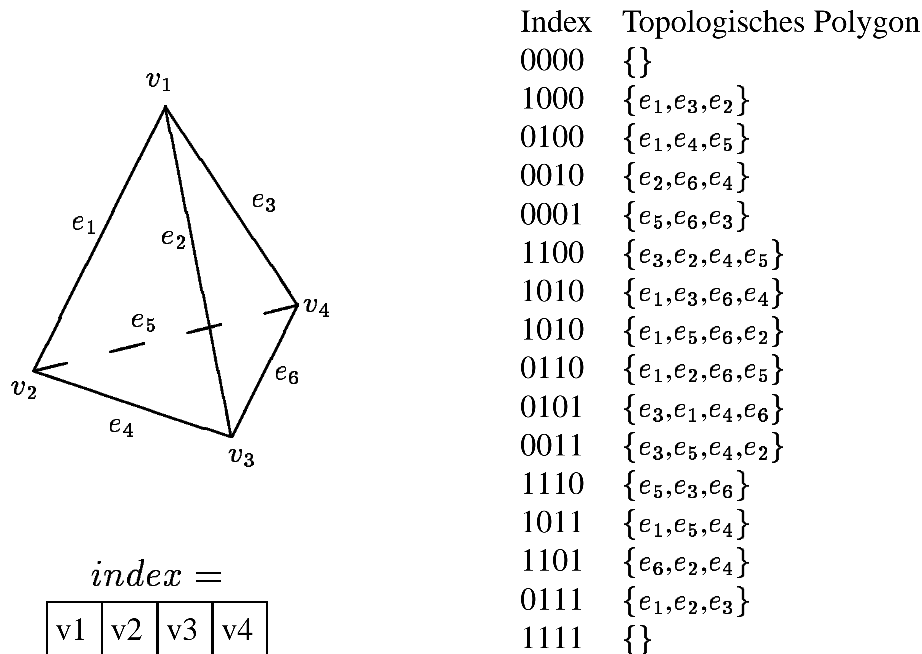
Die erste Phase basiert auf Satz 3.3. Das Auftreten von Löchern an einer Zellfläche kann direkt aus dem 8bit–Index der bezüglich dieser Fläche benachbarten Zellen abgeleitet werden. Tritt ein Loch auf, können im zweiten Schritt die Patches als Verbindung der Schnittpunkte zu einem planaren Viereck (evtl. unterteilt in zwei Dreiecke) generiert werden. Die topologische Definition der Patches läßt sich, analog zur Topologietabelle für die Basisfälle, in einer Lookup–Tabelle speichern. Selbstverständlich darf das Viereck in der an die mehrdeutige Fläche angrenzenden Zelle nicht noch einmal erzeugt werden. Mit diesem Vorgehen produziert der Algorithmus eine C^0–stetige und topologisch wohldefinierte Fläche, auf die ebenfalls Eigenschaft 3.4 zutrifft. Die Anzahl der generierten Dreiecke liegt wiederum nur knapp über der des *Marching Cubes*–Algorithmus.

RÖLL et al. [RHvK95] unternehmen daüberhinaus noch Versuche, die Laufzeit des Algorithmus weiter zu senken, indem sie ihn nahezu vollständig durch Lookup–Tabellen definieren. Diese und weitere Möglichkeiten zur Effizienzsteigerung werden in Abschnitt 3.7 näher erläutert.

3.4 Tetraederzerlegung

Das Prinzip des *Marching Cubes*–Algorithmus läßt sich nicht nur auf hexaedrische Gitter anwenden, sondern beispielsweise auch auf Tetraedergitter. Die möglichen Vorzeichenkonfigurationen

der Gitterpunkte reduzieren sich dabei auf $2^4 = 16$ Fälle, die sich in einem 4bit–Index kodieren lassen. Für jeden der 16 Fälle läßt sich eindeutig eine polygonale Approximation der Isofläche aus Drei– und Vierecken, wie in Abbildung 3.4 beschrieben, bestimmen: Die Fälle können in dieser



Index	Topologisches Polygon
0000	{}
1000	$\{e_1,e_3,e_2\}$
0100	$\{e_1,e_4,e_5\}$
0010	$\{e_2,e_6,e_4\}$
0001	$\{e_5,e_6,e_3\}$
1100	$\{e_3,e_2,e_4,e_5\}$
1010	$\{e_1,e_3,e_6,e_4\}$
1010	$\{e_1,e_5,e_6,e_2\}$
0110	$\{e_1,e_2,e_6,e_5\}$
0101	$\{e_3,e_1,e_4,e_6\}$
0011	$\{e_3,e_5,e_4,e_2\}$
1110	$\{e_5,e_3,e_6\}$
1011	$\{e_1,e_5,e_4\}$
1101	$\{e_6,e_2,e_4\}$
0111	$\{e_1,e_2,e_3\}$
1111	{}

Abbildung 3.8: *Tetraederpolygonalisierung über eine Topologietabelle mit 16 Einträgen*

Form in einer Lookup–Tabelle abgelegt werden. Da bei der Polygonalisierung von Tetraedern keine Mehrdeutigkeiten auftreten, ist die Überlegung naheliegend, im Falle von hexaedrischen Gittern jede Zelle in Tetraeder zu unterteilen und diese dann entsprechend Abb. 3.8 zu polygonalisieren, um so die Mehrdeutigkeiten zu beseitigen. Ansätze dazu stammen beispielsweise von BLOOMENTHAL[18] (1988) und KOIDE (1986, 1991) [NB93].

Da sich die Bestimmung der polygonalen Approximation für jedes Tetraeder problemlos und eindeutig gestaltet, reduziert sich die Aufgabe auf das Finden einer geeigneten Zerlegung. Eine hexaedrische Gitterzelle läßt sich, wie in Abbildung 3.9 zu sehen ist, allein unter Verwendung der acht Gitterpunkte als Eckpunkte in fünf (3.9a) oder sechs (3.9b) Tetraeder zerlegen. Die Tetraederkanten werden dabei von den Kanten der Zelle und Diagonalen auf der Zellfläche gebildet, bei der Zerlegung in sechs Teile zusätzlich noch von Raumdiagonalen im Innern der Zelle. Im Fall der Zerlegung eines Würfels in fünf Tetraeder sind die entstehenden Teile nicht alle gleich groß. Die Kanten des Tetraeders im Innern der Zelle sind alle Flächendiagonalen der Zelle, während je drei der vier Kanten der anderen Tetraeder Zellkanten sind. Auf jeder Zellfläche tritt genau eine Diagonale auf, deren Richtung man durch die Zerlegung vorgibt. Die gewählte Orientierung der Diagonalen bestimmt direkt die Verbindung der Schnittpunkte auf mehrdeutigen Flächen, wie Bild 3.10 zeigt. Verbindet die eingeführte Diagonale zwei positive (bzw. negative) Punkte, so separiert die nachfolgende Polygonalisierung (gemäß den Vorschriften in Abb. 3.8) die negativen (bzw. positiven) Gitterpunkte. Um Löcher zu vermeiden, müssen die Gitterzellen so zerlegt werden, daß auf gemeinsamen, mehrdeutigen Flächen gleiche Diagonalen entstehen. Dies gelingt dadurch, daß für benachbarte Gitterzellen jeweils alternierende Zerlegungen der gleichen Klasse (Fall 3.9a

[18]Electronic Mail: `bloomentr@parc.xerox.com`

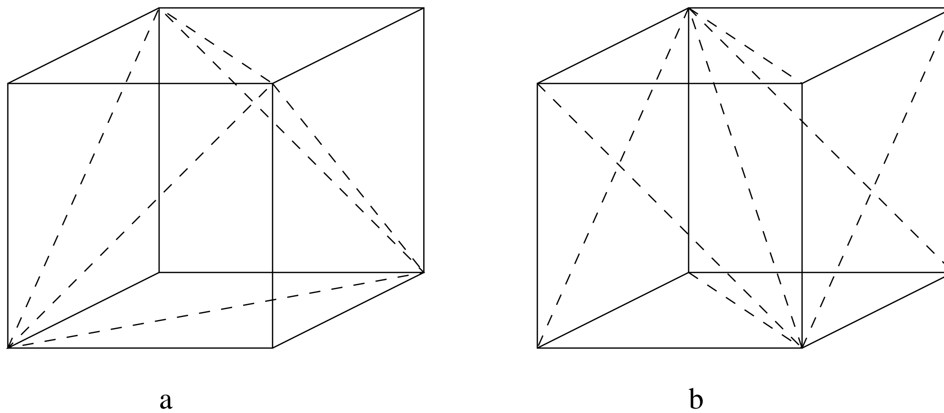


Abbildung 3.9: *Zerlegung eines hexaedrischen Volumenelements in fünf oder sechs Tetraeder ohne Einführung neuer Eckpunkte*

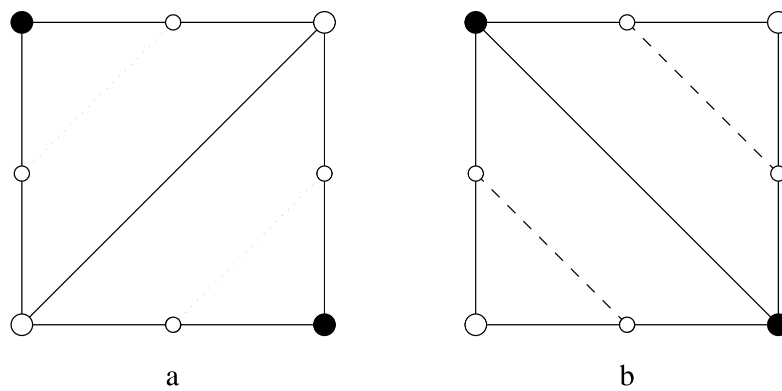


Abbildung 3.10: *Wirkung der Orientierung der bei der Gitterzerlegung eingeführten Diagonale auf die Verbindung der Schnittpunkte*

oder 3.9b) gewählt werden. Allein diese starre Festlegung impliziert die Topologie der generierten Fläche, die damit zwar Forderung F1 (Stetigkeit), aber nicht F2 (Konsistenz) erfüllt.

Die Einführung neuer Kanten durch die Zerlegung hat zur Folge, daß die berechneten Schnittpunkte nun auf jeder Tetraederkante, also nicht mehr nur auf den Zellkanten, sondern auch auf den Diagonalen der Zellflächen (3.9a) oder sogar Raumdiagonalen (3.9b) liegen können. Dies führt zu einer erheblich höheren Anzahl produzierter Dreiecke. NING und BLOOMENTHAL berichten in [NB93], daß die Anzahl der generierten Dreiecke häufig mehr als doppelt so hoch wie bei einem *Marching Cubes*–Ansatz ist.

Die Komplexität der Implementation ist ähnlich gering wie beim *Marching Cubes*–Algorithmus. Die Laufzeit ist etwas höher, liegt aber natürlich ebenso in der Größenordnung von $O(N^2)$. Die Berechnung der Schnittpunkte über lineare Interpolation auf den Tetraederkanten ist im übrigen nicht konsistent mit dem zumeist angenommenen trilinearen Interpolant innerhalb der Gitterzellen.

3.5 Trilineares Modell

Der Ansatz von NATARAJAN [Nat94] setzt sich von vorherein die topologische Konsistenz der erzeugten Fläche mit einem stückweisen trilinearen Interpolant (siehe Definition 2.8 auf Seite 10) zum Ziel. Er betrachtet auf Grundlage eines solchen Modells nicht nur die Mehrdeutigkeiten auf den Zellflächen, sondern auch im Innern der Zelle. Man führe sich vor Augen, daß im Basisfall 3 (siehe Seite 24) die Isofläche auch die Form einer „langen Röhre“ haben kann, die die zwei räumlich diagonal gegenüberliegenden Gitterpunkte verbindet. Die bisher vorgestellten Verfahren generieren in diesem Fall, unabhängig von den Skalarwerten, zwei Dreiecke, die diese Gitterpunkte separieren.

Wir betrachten im folgenden das stückweise trilineare Interpolant T eines Skalarfeldes S auf einem uniformen kubischen Gitter G. Die Isofläche von T zu einem Isowert t sei mit I bezeichnet. Ziel des Verfahrens ist es, eine stückweise Dreiecksapproximation P für I zu finden, die den bekannten Forderungen genügt. Zwei Gitterpunkte heißen *verbunden*, wenn es zwischen beiden Punkten einen beliebigen Pfad innerhalb der Zelle gibt, der I nicht schneidet. Sie heißen außerdem

1. **kantenverbunden**, wenn es einen solchen Pfad gibt, der nur aus Kanten der Zelle besteht
2. **flächenverbunden**, wenn es einen solchen Pfad gibt, der vollständig auf *einer* Zellfläche liegt
3. **außen verbunden**, wenn es einen solchen Pfad gibt, der vollständig auf Außenflächen der Zelle verläuft

Wenn wir wissen, welche Gitterpunkte verbunden sind, kennen wir die Topologie von I. Der Algorithmus aus [Nat94] beruht auf drei grundlegenden Aussagen darüber, wann solch eine Verbindung vorliegt. Der erste lautet:

Satz 3.5 *Zwei Gitterpunkte sind genau dann kantenverbunden, wenn sie durch einen Pfad verbunden sind, der nur aus Zellkanten besteht und nur Gitterpunkte gleichen Vorzeichens enthält.*

Wie eine genaue Untersuchung der Basisfälle auf Seite 24 zeigt, existiert in den Fällen 0, 1, 2, 5, 8, 9, 11 und 14 für jede Gitterzelle ein solcher Pfad. Mittels Satz 3.5 läßt sich die Topologie von P für diese Fälle eindeutig bestimmen. Die restlichen Fälle umfassen die Konfigurationen, in denen mindestens eine Zellfläche mehrdeutig ist oder Fall 4 vorliegt. Die Entscheidung, wie die Schnittpunkte in diesen Fällen zu verbinden sind, trifft der Algorithmus auf der Basis von Sattelpunkten:

Bereits NIELSON und HAMANN berichten in [NH91] vom Versuch, die topologische Entscheidung in mehrdeutigen Fällen auf der Basis eines Interpolants zu treffen. Sie betrachten dabei allerdings nur Mehrdeutigkeiten auf Zellflächen und lassen insbesondere Fall 4 außer Betracht. Da trilineare Interpolation auf einer Fläche der bilinearen Interpolation entspricht, ergibt sich die Gleichung aus Definition 2.8 auf Seite 10 mit $z = 0$:

$$\begin{aligned} T(x,y) = f(i,j) + (f(i+\delta,j) - f(i,j))x + (f(i,j+\delta) - f(i,j))y \\ +(f(i,j) + f(i+\delta,j+\delta) - f(i+\delta,j) - f(i,j+\delta))xy \end{aligned} \tag{3.4}$$

Legt man den Funktionswerten an den Eckpunkten einer Zellfläche ein bilineares Interpolant zugrunde, so werden die Schnittpunkte für ein festes t durch die Isolinien der bilinearen Funktion bestimmt, die die Form von Hyperbeln haben. Die Asymptoten der zwei Hyperbeln schneiden sich in einem Punkt innerhalb der Zellfläche. Bestimmt man den Wert der bilinearen Funktion an diesem Punkt, so läßt sich daraus auf die korrekte Verbindung der Schnittpunkte schließen. Ist der Wert kleiner als t, so separieren die Hyperbeln die positiven Gitterpunkte. Ist er größer als t, separieren sie die negativen Punkte. Abbildung 3.11 zeigt die beiden Fälle. Man kann sich dies

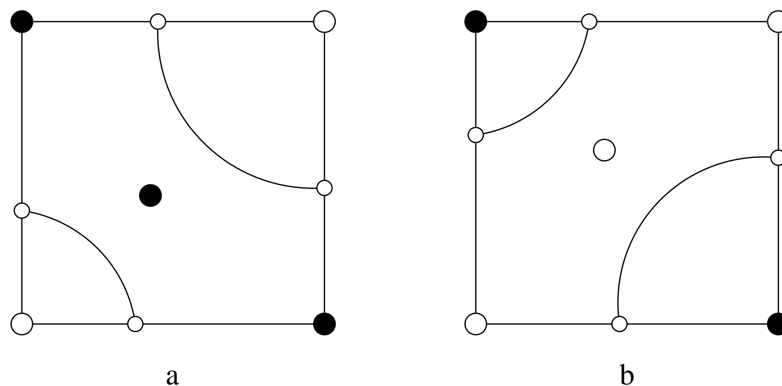


Abbildung 3.11: *Die Verbindung der Schnittpunkte einer mehrdeutigen Fläche auf Basis eines bilinearen Interpolants*

veranschaulichen, wenn man sich das bilineare Interpolant als dreidimensionales Gebilde über der Fläche vorstellt, das für mehrdeutige Flächen die Form eines „Sattels“ haben muß. Die Isokontur des Interpolants für einen Isowert t wird durch den Schnitt mit einer horizontalen Schnittebene der Höhe t bestimmt. Je nachdem, ob der Schnitt unter– oder oberhalb des Sattelpunktes (x_s, y_s) erfolgt, ergeben sich zwei „hufeisenförmige“ Schnittflächen, die von den in Abb. 3.11a bzw. 3.11b dargestellten Hyperbeln begrenzt werden.

Analog zum bilinearen Interpolant hat auch ein trilineares Interpolant Sattelpunkte. Die Topologie der Isofläche kann durch den Vergleich des Wertes des Interpolants in den Sattelpunkten mit dem Isowert bestimmt werden. Es müssen bis zu sieben Sattelpunkte untersucht werden, sechs auf den Gitterzellflächen (*Flächensattelpunkte*) und einer im Innern der Zelle (innerer Sattelpunkt). Flächensattelpunkte sind, genau genommen, keine wirklichen Sattelpunkte des trilinearen Interpolants T in einem Kubus, sondern die Sattelpunkte der Einschränkung von T auf eine seiner Flächen. Der innere Sattelpunkt ergibt sich aus der Lösung des Gleichungssystems:

$$\begin{aligned} \frac{\partial T}{\partial x} &= Hyz + Ey + Fz + B = 0 \\ \frac{\partial T}{\partial y} &= Hxz + Ex + Gz + C = 0 \\ \frac{\partial T}{\partial z} &= Hxy + Gy + Fx + D = 0 \end{aligned} \tag{3.5}$$

Für die Ermittlung der Topologie der zu generierenden Fläche sind dann die folgenden Aussagen [Nat94] von Bedeutung:

Satz 3.6 *Zwei Gitterpunkte sind genau dann auf einer Fläche verbunden, wenn mindestens eine der folgenden Bedingungen gilt:*

1. *Die zwei Punkte sind auf der Fläche kantenverbunden.*
2. *Die Funktionswerte an den zwei Punkten und der Flächensattelpunkt auf dieser Fläche sind alle kleiner als t.*
3. *Die Funktionswerte an den zwei Punkten und der Flächensattelpunkt auf dieser Fläche sind alle größer als oder gleich t.*

Die Sätze 3.5 und 3.6 sind hinreichend, um für zwei beliebige Gitterpunkte zu bestimmen, ob sie flächenverbunden sind. Es bleibt zu untersuchen, wie sich die Verbindungen im Innern der Zelle gestalten:

Satz 3.7 *Zwei beliebige Gitterpunkte v_a und v_b sind genau dann in einer Zelle verbunden, wenn mindestens eine der folgenden Bedingungen gilt:*

1. *Die zwei Punkte sind in der Zelle außen verbunden.*
2. *Die Funktionswerte in v_a und v_b und der innere Sattelpunkt der Zelle sind alle kleiner als t.*
3. *Die Funktionswerte in v_a und v_b und der innere Sattelpunkt der Zelle sind alle größer als oder gleich t.*

Betrachten wir noch einmal die Basisfälle aus Abbildung 3.3 auf Seite 24. Wie bereits erwähnt, ist Satz 3.5 ausreichend, um die Topologie für acht dieser Fälle zu bestimmen, denen keine Mehrdeutigkeiten innewohnen. In den Fällen 4, 6, 7, 10, 12 und 13 ist es möglich, daß zwei Gitterpunkte zwar verbunden, aber nicht flächenverbunden sind. Satz 3.7 ist also in sechs der sieben mehrdeutigen Fälle von Bedeutung. Dies zeigt, daß Ansätze, die nur Verbindungen auf den Zellflächen untersuchen [WMW86, LC87, NH91, MSS94b, Lor94] zu keiner topologisch konsistenten Fläche im Hinblick auf ein trilineares Interpolant führen können. Der Algorithmus von NATARAJAN läßt sich wie folgt zusammenfassen:

```
algorithm saddle
    for each Gitterzelle do
        bestimme das Vorzeichen der acht Eckpunkte
        wende die Sätze 3.5, 3.6 und 3.7 an, um die Topologie zu bestimmen
        berechne die Schnittpunkte
        verbinde die Schnittpunkte zu Dreiecken
        gib die Dreiecke aus
    enddo
endalgorithm
```

Das Vorgehen beim Verbinden von Schnittpunkte auf einer Fläche genügt dem Flächenebenenprinzip, so daß die erzeugte Approximation C^0–stetig sein muß. Pro Gitterzelle wird eine konstante Anzahl von Dreiecken erzeugt, deren Topologie sich nach den ermittelten Verbindungen der Gitterpunkte gemäß den Sätzen 3.5, 3.6 und 3.7 richtet. Durch die zusätzliche Betrachtung von Verbindungen innerhalb der Zelle wird die Anzahl der erzeugten Dreiecke, verglichen mit einem korrigierten *Marching Cubes*–Algorithmus, leicht höher liegen. Die Geschwindigkeit eines solchen Programms liegt in der gleichen Größenordnung und liegt bei Messungen nur um einige Prozent niedriger. Die Unterschiede sinken meist mit höherer Auflösung der Daten [Nat94].

In [Mat94] wird bewiesen, daß das Verbinden der Schnittpunkte auf einer Zellfläche auf Grundlage bilinearer Interpolation äquivalent zur Verbindung auf Grundlage der Sortierung der Punkte entlang einer der (auf einen Einheitswürfel transformierten) Koordinaten ist. MATVEYEV stellt auch einen alternativen Ansatz für eine topologisch konsistente Methode auf Grundlage eines trilinearen Modells vor. Er betrachtet die durch zwei diagonal gegenüberliegende parallele Gitterkanten definierten Schnittebenen, berechnet die Schnittpunkte der Isofläche mit den beiden Diagonalen der Schnittflächen und überführt alle diese Punkte in einen Graphen. Durch Anwendung von Methoden der Graphentheorie erhält er daraus die gesuchten Verbindungen. Für Einzelheiten sei auf [Mat94] verwiesen.

3.6 Heuristiken

Allen bisher vorgestellten Algorithmen ist gemeinsam, daß sie topologische Entscheidungen nur von den Skalarwerten der acht Eckpunkte der aktuellen Gitterzelle abhängig machen. Es sei an

Abbildung 2 auf Seite 14 erinnert, die zwei quadratische Funktionen zeigt, die in der Zelle trotz identischer Funktionswerte an den Eckpunkten einen völlig unterschiedlichen Verlauf aufweisen. Aufwendigere Methoden versuchen, mit Hilfe von Werten außerhalb der Zelle, auch für solche Funktionen die richtige Topologie zu ermitteln.

Ein möglicher Ansatz ist die Verwendung eines polynomialen Interpolants höherer Ordnung. In [VGW94] wird von der Verwendung eines trikubischen Interpolants berichtet, das eine $4 \times 4 \times 4$–Region um die aktuelle Zelle betrachtet, um die Funktion zu bestimmen. Konkret wird die dreidimensionale Version eines CATMUL–ROM–Splines [FvDFH90] verwendet, wodurch sich insgesamt eine C^1–stetige Funktion ergibt. Dieser Ansatz ist jedoch algorithmisch extrem teuer. Er erfordert für jede betrachtete Zelle über einhundert Berechnungen des Wertes der trikubischen Funktion. Für jede dieser Berechnungen muß jeweils eine vierstellige Anzahl arithmetischer Operationen ausgeführt werden [VGW94]. Im Unterschied zu den bisherigen Methoden muß man sie außerdem auf jede Zelle, nicht nur auf die mehrdeutigen, anwenden. Die Anwendung von trikubischer Interpolation in mehrdeutigen Zellen und linearer Interpolation bei der Berechnung von Schnittpunkten in eindeutigen Fällen führt zu Unstetigkeiten, weil das trikubische Interpolant insbesondere nicht konsistent mit den auf Seite 12 gemachten Annahmen A2 und A3 ist und so mehrere Schnittpunkte auf einer Zellkante entstehen können. Der Aufwand für die Anwendung trikubischer Interpolation auf alle Zellen ist so hoch, daß es den Ansatz für eine effiziente Isoflächenberechnung unbrauchbar macht.

WILHELMS[19] und VAN GELDER[20] [VGW94] beschreiten einen effizienteren Weg. Ihre zwei vorgestellten Gradientenheuristiken, *Center–Point Gradient Method* und *Quadratic Fit*, benutzen zusätzlich zu den Skalarwerten auch die Gradienten an den Eckpunkten, um auf diese Weise Informationen über das Verhalten der Funktion außerhalb der aktuellen Zelle mit einzubeziehen. Die Berechnung der Gradienten verursacht keinen zusätzlichen Aufwand, da diese gemäß Gleichung 2.4 auf Seite 10 auch zur Bestimmung der Normalen benötigt werden. Beide Verfahren verwenden eine bikubische Funktion, um die korrekte Topologie im Falle mehrdeutiger Zellflächen zu bestimmen. Diese Funktion wird so gewählt, daß sie an den Eckpunkten der Fläche mit den Skalarwerten übereinstimmt und den jeweiligen Gradienten so genau wie möglich, d.h. mit minimalem quadratischen Fehler, entspricht. Es werden nur die in einer Ebene mit der jeweiligen Fläche liegenden Komponenten des Gradienten benötigt. Die Gradientenheuristiken folgen damit dem Flächenebenenprinzip (Satz 3.2 auf Seite 22) und generieren C^0–stetige Flächen.

Die Entscheidung, welche der über lineare Interpolation bestimmten Schnittpunkte im Zweifelsfall zu verbinden ist, treffen die beiden Heuristiken auf verschiedene Weise. Während die einfachere *Center–Point Gradient Method* dazu in bekannter Weise den Wert des Interpolants im Mittelpunkt der Fläche benutzt, macht die *Quadratic Fit*–Methode die Entscheidung vom Vergleich des Wertes im Sattelpunkt mit dem Isowert abhängig. Letztere Berechnung ist deutlich aufwendiger, führt aber für die in [VGW94] betrachteten Testdaten zu besseren Ergebnissen.

Beide Gradientenheuristiken können, da sie nur auf mehrdeutige Flächen angewandt werden, nicht die Mehrdeutigkeiten im Innern der Zelle lösen, auf die in Abschnitt 3.5 hingewiesen wurde.

[19]Electronic Mail: `wilhelms@cse.ucsc.edu`
[20]Electronic Mail: `avg@cse.ucsc.edu`

3.7 Effizienzverbesserungen

In diesem Abschnitt sollen einige Prinzipien erläutert werden, wie sich die meisten der vorgestellten Verfahren effizienter gestalten lassen, ohne dabei ihre prinzipiellen Eigenschaften zu ändern. Einige dieser Ideen stammen aus der Literatur, andere aus den Erfahrungen des Autors beim Entwurf eigener Software zur Isoflächengenerierung (siehe Kapitel 3.3.2 und 5.2).

Die naheliegendsten Möglichkeiten zur Effizienzsteigerung liegen im Vermeiden doppelter Berechnungen durch Ausnutzen der Nachbarschaftsbeziehungen im Gitter. Solche Berechnungen betreffen die Funktionswerte an den Gitterpunkten und die Ermittlung von Koordinaten und Normalen des Schnittpunkts auf Gitterkanten. Aus Bemerkung 2.4 auf Seite 8 ergibt sich, daß das Verhältnis von Gitterkanten zu Gitterzellen für große Gitter gegen 3 geht, also im Idealfall in der aktuellen Zelle nur drei Kanten neu betrachtet werden müssen. Dieser Idealfall tritt genau dann ein, wenn man davon ausgeht, daß die Gitterzellen am Rand bereits durchlaufen wurden. Für innere Zellen können bei geeignetem Vorgehen dann die Koordinaten und Normalen von Schnittpunkten auf bis zu neun Kanten übernommen werden (falls diese Kanten überhaupt von der Isofläche geschnitten werden); nur auf den restlichen drei muß eventuell eine lineare Interpolation vorgenommen werden [RHvK95]. Das Merken der bereits berechneten Schnittpunkte zu den Kanten verursacht jedoch für große Gitter einen nicht zu vernachlässigenden Speicherplatzbedarf. In der Praxis wird man deshalb oft nur die Informationen zu den vier Kanten des aktuellen Würfels abspeichern, die er mit der nächsten Zelle gemeinsam hat [LC87]. Die Ausnutzung von Kohärenzen erschwert eine datenparallele Implementation. Parallele Ansätze, wie sie in Kapitel 5 vorgestellt werden, sollten nur lokal begrenzt Informationen mehrfach verwenden.

Die Berechnung des 8bit–Indexes setzt sich aus acht Vergleichen von den jeweiligen Funktionswerten und dem gegebenen Isowert zusammen. Obwohl ein solcher Vorgang für sich gesehen mit wenigen Operationen vonstatten gehen, schlägt die Indexberechnung für die Gesamtlaufzeit des Algorithmus mit $O(N^3)$ zu Buche, denn sie wird für jede der $(N_x - 1) \times (N_y - 1) \times (N_z - 1)$ Gitterzellen einmal ausgeführt. Infolgedessen ist das Optimierungspotential, bezogen auf die Gesamtlaufzeit des Algorithmus, hier besonders hoch. Der Zugriff auf die Funktionswerte an den Eckpunkten ist meist ein Zugriff auf ein Array und erfordert im Fall diskreter Skalarfelder u.U. Indexumrechnungen zur Abbildung der logischen Gitterposition auf den Feldindex. Muß der Funktionswert erst berechnet werden, ist das Einsparen solcher Zugriffe um so wichtiger. Der in Abschnitt 3.1 vorgestellte Algorithmus benutzt deshalb Hashtabellen. Der von mir in HyperPlan (siehe Kap. 3.3.2) und PARADISO (siehe Kap. 5.2) implementierte Algorithmus für diskrete Skalarfelder greift bei der Indexberechnung auf den Index der vorherigen Zelle zurück und ermittelt vier der Bits des neuen Indexes aus denen des alten durch einfache binäre Operationen. Mit der Bezeichnungsweise aus Bild 3.4 auf Seite 25 ergibt sich der 8bit–Index der rechten Nachbarzelle des dargestellten Würfels unter Verwendung C–ähnlicher Operatoren in Pseudocode wie folgt (t sei der Isowert):

procedure compute_index (c)
    index $= ((\text{index}\&34) \gg 1) \,|\, ((\text{index}\&68) \ll 1)$
    if $(f(v_2) \geq t)$ index $|= 2$
    if $(f(v_3) \geq t)$ index $|= 4$

if $(f(v_6) \geq t)$ index $| = 32$
if $(f(v_7) \geq t)$ index $| = 64$
gib *index* aus
endprocedure

Die erste Zeile berechnet die neuen Bits von v_1, v_4, v_5 und v_8 aus den Bits von v_2, v_3, v_6 bzw. v_7 im alten Index und setzt die restlichen Bits auf Null. Die anderen Bits werden ensprechend den jeweiligen Gitterpunktvorzeichen gesetzt. Die Gesamtkomplexität der Indexberechnung bleibt trotz aller Bemühungen weiterhin $O(N^3)$. Messungen mit meinem HyperPlan–Modul ergaben einen durchschnittlichen Anteil der Isoflächberechnung an der Gesamtlaufzeit von ca. 40%. Dies läßt sich nur dadurch umgehen, daß man die Indexberechnung lediglich auf Zellen anwendet, die wirklich von der Isofläche geschnitten werden. Das Unterkapitel 3.7.1 stellt eine solche Möglichkeit vor.

Eine der grundlegenden Ideen des *Marching Cubes*–Algorithmus ist die Verwendung einer vorher erzeugten Topologietabelle, um Untersuchungen zur Laufzeit einzusparen. Prinzipiell lassen sich bei jeder Methode diejenigen Schritte durch Verwendung von Lookup–Tabellen einsparen, die keine genauen numerischen Berechnungen auf Basis der Skalarwerte erfordern. Auch die Zuordnung der Fälle zu den Basisfällen zur Laufzeit kann eingespart werden, wenn eine vollständige Lookup–Tabelle mit 256 Einträgen verwendet wird. Die Erzeugung einer solchen Taballe von Hand ist mühselig und fehleranfällig und sollte deshalb einem Computerprogramm überlassen werden, das für jeden der 256 Fälle den Basisfall bestimmt und somit auf Grundlage einer minimalen Lookup–Tabelle für die Basisfälle die vollständige Tabelle erzeugt. Der höhere Speicherplatzaufwand für eine große Tabelle fällt in der Regel nicht ins Gewicht. Meine Programme verwenden eine solche computergenerierte Topologietabelle. Bei Verfahren, die die Verbindung der Schnittpunkte von den genauen Skalarwerten an den Gitterpunkten abhängig machen, können Berechnungen zur Laufzeit durch eine Erweiterung der Lookup–Tabelle eingespart werden. Für jeden mehrdeutigen Basisfall lassen sich Untertabellen [VGW94] generieren, die die verschiedenen Möglichkeiten für die Polygonalisierung der Zelle beschreiben. Zur Laufzeit wird dann nur noch zwischen diesen Untertabellen ausgewählt. Solch eine Untertabelle kann bis zu 64 Einträge (Basisfall 13) besitzen. Für die eindeutigen Fälle enthält die Tabelle die gleichen Informationen wie eine herkömmliche Topologietabelle.

Herkömmliche Lookup–Tabellen enthalten zu jedem Fall die Anzahl a der zu generierenden Dreiecke und eine Liste der Länge $3a$ der Kantennummern, auf denen die Dreieckspunkte liegen sollen. Treten Kantennummern doppelt auf, muß dies, will man eine doppelte Berechnung des Schnittpunktes vermeiden, zur Laufzeit erkannt werden. Wird eine Topologietabelle automatisch erzeugt, lassen sich die Informationen so aufbereiten bzw. erweitern, daß solche Untersuchungen zur Laufzeit nicht mehr notwendig sind. Die in meinen Implementationen zum Einsatz kommende Tabelle enthält für jeden Fall sowohl eine Liste der Kanten (ohne Doppelnennungen), die geschnitten werden, als auch die Dreiecke als Tripel dieser Kantennummern. Weiterhin ist die Zuordnung von Eckpunktbezeichnungen und Kantenummern zueinander und die Abbildung auf ihre logische Position im Raum darin untergebracht.

Der in Abschnitt 3.3.3 vorgestellte Algorithmus führt diese Idee soweit aus, daß er nahezu vollständig auf Lookup–Tabellen aufgebaut ist. Lineare Interpolation wird durch Bisektion (siehe

auch Kap. 4.3) ersetzt, so daß sich alle Schnittpunktkoordinaten als ganzzahlige diskrete logische Koordinaten ausdrücken lassen, deren auf die aktuelle Zelle bezogener Wert in Tabellen referenziert werden kann [RHvK95]. Bisektion führt natürlich, verglichen mit den Ergebnissen linearer Interpolation, zu stark abweichenden Schnittpunktkoordinaten. Die Dreiecksfläche liegt dann in ganzzahligen Koordinaten vor, was je nach Rechnerplattform gegenüber Gleitkommazahlen zu einer unterschiedlichen Speicherplatzersparnis führen kann.

Die erzeugte Approximation wird bei einer ungünstigen Vorgehensweise für jedes Dreieck die Koordinaten und Normalen seiner Eckpunkte enthalten. Werden doppelte Berechnungen von Schnittpunkten mit den oben genannten Methoden vermieden, kann die Dreiecksapproximation auch durch drei Listen von Indizes, Koordinaten und Normalen repräsentiert werden. Je drei aufeinanderfolgende Indizes in der Liste bilden ein Dreieck. Die Indizes dienen als Zeiger auf die jeweiligen Koordinaten und Normalen des Dreieckspunktes. Da die Dreiecke aneinander anschließen, treten alle Dreieckspunkte mehrfach auf. Auf ihre Koordinaten und Normalen, die nur einmal gespeichert sind, wird über den Index referenziert. Durch eine geschickte Vorgehensweise bei der Schnittpunktbestimmung läßt sich demnach die Größe, d.h. der Speicherplatzbedarf, der entstehenden Dreiecksfläche deutlich verringern.

Die Größe der Datensätze kann auf Workstations leicht den zur Verfügung stehenden Hauptspeicher übersteigen. Für Ansätze, die zur Ermittlung der polygonalen Approximation nur eine Gitterzelle pro Zeitschritt betrachten (Kap. 3.1–3.5), ist es aber gar nicht notwendig, den gesamten Datensatz im Speicher zu halten. Berücksichtigt man, daß für die Normalenberechnung Skalarwerte aus direkt benachbarten Zellen benötigt werden, so wird man bei beschränkten Speicherressourcen nur vier Schichten der Daten auf einmal im Hauptspeicher halten [LC87].

3.7.1 Skalarfelder als Octrees

Ein entscheidendes Manko aller aufgeführten Isoflächenalgorithmen bleibt, trotz aller bislang vorgestellten Möglichkeiten zur effizienten Implementation, die Diskrepanz zwischen der Anzahl M von der Isofläche geschnittener und der Anzahl $O(N^3)$ vom Algorithmus durchlaufener Gitterzellen. Die Datenstruktur des diskreten Skalarfeldes läßt keine Aussagen darüber zu, ob eine Zelle von der Isofläche geschnitten wird, ohne die Funktionswerte aller Gitterzelleneckpunkte zu bestimmen. Geeignete hierarchische Datenstrukturen können hier Abhilfe schaffen:

WILHELMS und VAN GELDER schlugen 1990 die Verwendung von Octrees zur Isoflächengenerierung vor [WVG90b]. Die Struktur von Octrees entspricht Bäumen, in denen jeder Knoten (außer der Blätter) im regelmäßigen Fall acht Kinder hat. Sie sind, analog zum Quadtree für zweidimensionale Vierecksgitter, speziell für die Repräsentation dreidimensionaler, hexaedrischer Gitter geeignet. Sie basieren auf einer rekursiven Unterteilung des Raumes in (normalerweise acht) Teilvolumina, wobei die Wurzel das Gesamtvolumen repräsentiert. In den Knoten des Octrees wird eine zusammenfassende Information über das darunterliegende Teilvolumen gespeichert. Dies macht Octrees für eine Vielzahl von Anwendungsgebieten tauglich, in denen dreidimensionale Volumen durchsucht werden müssen. Für die Isoflächengenerierung ist es wünschenswert, für ein Teilvolumen bestimmen zu können, ob es von der Isofläche geschnitten wird oder nicht. Dazu speichert man in jedem Knoten des Octrees den minimalen und maximalen Skalarwert innerhalb des durch

den Knoten repräsentierten Subvolumens. Der Octree zu einem diskreten Skalarfeld stellt lediglich eine geeignetere, komplexere Datenstruktur zur Repräsentation des Feldes dar und ist von keinem Parameter der Anwendung, z.B. dem Isowert, abhängig.

Das Vorgehen zur Isoflächengenerierung liegt auf der Hand. In der Setup–Phase ist der Octree aus einem gegebenen diskreten Skalarfeld zu konstruieren. Die untersten Knoten repräsentieren bis zu acht Gitterzellen und enthalten die entsprechenden Zeiger in das Datenarray. Im zweiten Schritt wird der Octree, beginnend beim Wurzelknoten, unter Betrachtung eines bestimmten Isowertes traversiert. Dabei wird nur an solche Knoten weiter im Baum hinabgestiegen, deren gespeichertes Minimum unter und deren Maximum über dem Isowert liegt. Regionen, für die dies nicht zutrifft, werden außer acht gelassen, weil es in ihnen gemäß Annahme A3 auf Seite 12 keine Gitterkante geben kann, die von der Isofläche geschnitten wird. Ist man bei einem untersten Knoten des Baumes angekommen, werden die bis zu acht von ihm repräsentierten Zellen mit einer der in den vorangegangenen Kapiteln vorgestellten Methoden polygonalisiert.

Die Konstruktion des Octrees ist unabhängig vom Isowert. Will man zu einem gegebenen Skalarfeld nacheinander Isoflächen mit verschiedenen Isowerten generieren, muß die Setup–Phase nur beim ersten Mal durchlaufen werden. Der Octree kann für jede weitere Isoflächenberechnung gespeichert und wiederverwendet werden. Für ein Skalarfeld der Größe $2^l \times 2^l \times 2^l$ ist die Konstruktion des Octrees einfach, indem ein eindeutig bestimmter vollständiger Octree angelegt wird, in dem jeder Knoten genau acht Kinder hat. Die Regularität eines solchen vollständigen Octrees erlaubt es, die gesamte Datenstruktur ohne explizite Adreßinformationen an den Knoten zu speichern. Im allgemeinen Fall, wo die Gitterdimensionen nicht gleiche Zweierpotenzen darstellen, stellen vollständige Octrees eine enorme Speicherplatzverschwendung dar, da viele der Knoten Teilvolumen außerhalb des Gitters repräsentieren. Pointer–Octrees [WVG92] legen Speicherplatz für Knoten nur dann an, wenn die entsprechende Region wirklich innerhalb der Feldgrenzen liegt. Die traditionelle und intuitive Strategie konstruiert einen Pointer–Octree von oben nach unten. Dabei wird das Gesamtvolumen, solange es geht, so gleichmäßig wie möglich zerteilt. Sie teilt demnach im oberen Teil des Octrees achtfach (solange sich alle Dimensionen noch teilen lassen), in der Mitte vierfach und ganz unten (wenn sich das Gitter nur noch in der größten Dimension unterteilen läßt) binär. Die in [WVG92] vorgestellte BONO(branch–on–need)–Strategie geht den umgekehrten Weg, indem die Unterteilung solange wie möglich herausgezögert wird. Die Methode versucht so, den Octree von unten in der effizientesten Weise aufzubauen. Die ideale Achtfach-Teilung findet so im unteren Bereich des Octrees statt, wo sich die meisten Knoten befinden. In [WVG92] wird bewiesen, daß für Gitter größer als $32 \times 32 \times 32$ das Verhältnis von Knoten zu Datenwerten zwischen 0.14286 und 0.162 liegen muß. Die Speicherung des Octrees führt also zu einem nicht unerheblichen zusätzlichen Speicherplatzaufwand, der für Gitter der Größe $2^l \times 2^l \times 2^l$ am geringsten ist.

Octrees ermöglichen einen enormen Laufzeitgewinn. Berücksichtigt man Setup– und Traversierungsphase, so ist die octreebasierte Methode in den allermeisten Fällen schneller als ein *Marching Cubes*–Ansatz. Für die nachfolgende Generierung einer Isofläche auf Grundlage der gleichen Daten, aber mit einem anderen Isowert, ist keine Setup–Phase mehr notwendig. Die Laufzeit des Algorithmus ist dann in jedem Falle erheblich geringer. Der Laufzeitunterschied zum entsprechenden Vorgehen ohne Octree hängt vor allem vom Verhältnis M/N_3, also dem Anteil von der Isofläche geschnittener Gitterzellen am Gesamtvolumen, ab. Konkrete Laufzeitstatistiken finden sich in [WVG92]. Während das Preprocessing, die Konstruktion des Octrees, eine Lauf-

zeitkomplexität von $O(N^3)$ aufweist, ist die eigentliche Isoflächengenerierung aus dem Octree in $O(MlogN)$ zu bewältigen. Dabei werden im Durchschnitt ca. doppelt so viele Gitterzellen untersucht, wie von der Isofläche geschnitten werden. In vielen Anwendungsfällen erfolgt die Untersuchung von Skalarfeldern über die Generierung von Isoflächen für mehr als einen Isowert. Octreebasierte Isoflächenalgorithmen stellen für solche Fälle die effizienteste bekannte Methode dar. Die Eigenschaften der generierten Isofläche werden von dem Vorgehen nicht beeinflußt, sondern hängen von dem verwendeten Polygonalisierungsschema ab. Dabei ist allerdings nur die Anwendung solcher Schemata möglich, die eng lokal auf den Daten arbeiten, z.B. zu einem Zeitpunkt nur auf einer Gitterzelle.

Ein weiteres Problem bei der octreebasierten Isoflächengenerierung ist die Vermeidung doppelter Berechnungen. Während bei einem schrittweisen Durchlaufen des Feldes in x–y–z–Richtung, wie am Anfang von Kapitel 3.7 beschrieben, auf bereits berechnete Informationen relativ leicht zurückgegriffen werden kann, wirft das unregelmäßigere Vorgehen beim Durchlaufen eines Octrees diesbezüglich technische Schwierigkeiten auf. WILHELMS und VAN GELDER schlagen die Verwendung eines speziellen, auf einer Hashtabelle basierenden Cache–Verfahrens vor, das entscheidet, wann eine Information wieder aus dem Cache entfernt werden kann. Für eine detaillierte Diskussion der zu beachtenden Aspekte bei einem Octree–Ansatz sei auf die exzellente Abhandlung in [WVG92] verwiesen.

Der Einsatz von Octrees birgt durch die gleichmäßige Aufteilung des Gitters implizit ein hohes Maß an Datenparallelität. Für Anwendungen, in denen Isoflächen schrittweise mit einem anschwellenden Isowert animiert werden sollen, eignet sich das vorgestellte Verfahren hervorragend. Für kontinuierliche Skalarfelder ist eine Vorverarbeitung zur Erzeugung solcher hierarchischen Datenstrukturen nicht praktikabel, da die dafür notwendigen Funktionswertberechnungen zu rechenaufwendig sind. Octrees bieten sich dagegen für die Generierung von Isoflächen variabler Auflösung (siehe Kapitel 4.1.3) an.

3.8 Cuberille

Eine Alternative zu den bisher diskutierten *Beveled–Surface*–Methoden stellt der *Cuberille*–Ansatz dar, der die durch die Gitterpunkte gegebene Unterteilung des Raums als Aufspaltung in atomare polyedrische Volumenelemente betrachtet. Konsequenterweise stellen sich solche Algorithmen zum Ziel, die kleinstmögliche Menge solcher Polyeder zu finden, die die Isofläche des Skalarfeldes vollständig enthält. Dies ergibt ein dreidimensionales Objekt, das einer Zusammensetzung aus Bauklötzen gleicht. Ein weicheres Aussehen kann durch zusätzliches Schattieren und Filtern erreicht werden [VGW94]. Der erste Cuberille–Algorithmus wurde bereits 1979 von HERMAN und LUI vorgestellt [Wal91].

Cuberille–Ansätze unternehmen im allgemeinen nicht den Versuch der Extraktion größerer Details, als die Daten unzweideutig vorgeben. Dies führt insbesondere zu einer sehr geringen Laufzeit, die sich durch eine naheliegende Verbindung mit der Verwendung von Octrees [WVG92] noch weiter senken läßt.

Durch den gewählten Ansatz führt eine Vorzeicheninvertierung zu unterschiedlichen Approximationen.

3.8.1 Dividing Cubes

Zu den Cuberille–Ansätzen sei der Einfachheit halber auch der 1988 erstmals vorgestellte *Dividing Cubes*–Algorithmus [Lor94] gezählt, obwohl er sich in einigen Punkten davon unterscheidet. Er betrachtet die gegebenen hexaedrischen Volumenelemente nicht als atomar, sondern unterteilt sie gegebenenfalls weiter in gleichförmige Teile:

Der Algorithmus durchläuft zunächst in bekannter Manier alle (zumeist kubischen) Gitterzellen und klassifizert sie. Sie heißen innere bzw. äußere Würfel, falls der Isowert an allen Ecken größer bzw. an allen Ecken kleiner als der Isowert ist. Ansonsten heißen sie *Surface Cubes* [Lor94]. Im zweiten Schritt werden nur die *Surface Cubes* entsprechend der gewählten Auflösung in kleinere kubische Voxel unterteilt. Für einen Datensatz der Größe $256 \times 256 \times 128$ und einer gewünschten Bildauflösung von 512×512 ist beispielsweise in x– und y–Richtung in zwei und in z-Richtung in vier Teile zu teilen. Die Daten an den Eckpunkten der kleinen Voxel werden über trilineare Interpolation bestimmt. Die Voxel werden auf dieser Grundlage analog in innere und äußere Voxel sowie *Surface Voxel* klassifizert. Für jedes *Surface Voxel* werden zum Schluß noch Normalen gemäß Gleichung 2.4 auf Seite 10 berechnet. Die gesuchte Approximation wird durch die Menge aller *Surface Voxel* mit den zugehörigen Normalen repräsentiert.

Die Darstellung eines solchen Modells ist nur mit spezieller Hard– oder Software möglich. Aus diesem Grund hat der Algorithmus bis jetzt keine weite Verbreitung gefunden, obwohl er schnell ist und eine topologisch konsistentes Repräsentation auf Basis eines trilinearen Interpolants generiert.

Der Algorithmus unterscheidet sich in zwei wesentlichen Punkten deutlich von allen anderen Verfahren: Erstens hängt die Größe der berechneten Repräsentation prinzipiell nicht von der Auflösung der Daten, sondern nur von der Bildauflösung ab. Kurioserweise kann er für größere Datensätze schneller sein als für kleinere, bei denen erst trilinear interpoliert werden muß. Überschreitet die Gittergröße die Bildauflösung, so bleibt die Laufzeit nahezu konstant. Durch die lokalen Operationen muß auch nicht der gesamte Datensatz im Speicher gehalten werden, sondern kann schichtweise eingelesen werden. Dies macht den Algorithmus für beliebig große Datensätze tauglich. Die Herangehensweise läßt sich problemlos implementieren und auch ohne Schwierigkeiten datenparallel umsetzen. Die Eignung für einen spezifischen Einsatzfall hängt entscheidend davon ab, ob die Darstellung der Repräsentation (Punkte und Normalen) mit Hilfe der zur Verfügung stehenden Hard– und Software effektiv gestaltet werden kann.

Eine Implementation des Verfahrens steht seit neuestem als Klasse *VtkDividingCubes* im *Visualization Toolkit* (siehe Fußnote (16) auf Seite 30) jedermann[21] zur Verfügung.

[21] Die geneigte Leserin bzw. der geneigte Leser möge hier, wie auch anderswo, den „männerdominierten“ Begriff verzeihen. Ich gebrauche die gewählte Form aus Gründen der Einfachheit und Verständlichkeit, möchte sie aber nicht als Ausdruck von Diskriminierung, sondern in der Bedeutung von jedermann/frau verstanden wissen. Worte wie „der Leser“ richten sich natürlich auch und zuallererst an Leser*innen*.

Kapitel 4

Verfahren mit interner Polygonreduktion

Polygone sind in computergrafischen Anwendungen populäre Grafikprimitive, die von gängiger kommerzieller Grafikhardware und in vielen Softwareprodukten unterstützt werden. Im allgemeinen sind Polygone nicht planar und müssen erst tesselliert werden, um die schnellen Darstellungsmöglichkeiten für einfache planare Polygone ausnutzen zu können. Jedes einfache planare n–Eck läßt sich wiederum in $(n - 2)$ Dreiecke zerlegen. In diesem Kapitel betrachte ich deshalb der Einfachheit halber nur solche Flächen, die aus Dreiecken zusammengesetzt sind, ohne damit die Anwendungsbreite einzuschränken. Diese Vereinfachung ist insbesondere im Kontext der Isoflächengenerierung naheliegend, denn die im vorangegangenen Kapitel 3 vorgestellten *Beveled–Surface*–Methoden konstruieren ausnahmslos stückweise Dreiecksapproximationen.

Um komplexe Geometrien mit planaren Polygonen detailliert zu beschreiben, sind häufig Tausende oder Millionen von Dreiecken notwendig. Modelle solcher Größe sind allerdings kaum handhabbar, denn mit der Anzahl der Polygone nimmt die Darstellungsgeschwindigkeit ab und der Speicherbedarf zu. Ab einer gewissen Grenze, die heute bereits in Dutzenden von Applikationen überschritten werden kann, können die Modelle auf herkömmlichen Workstations nicht mehr dargestellt werden. Einer dieser Anwendungsfälle ist die Isoflächenberechnung auf Grundlage hochaufgelöster Daten. Gängige industrielle und medizinische Computertomographen können Datensätze erzeugen, die aus hunderten Schichten mit einer Auflösung von 512×512 oder gar 1024×1024 bestehen. Herkömmliche Isoflächenverfahren, wie die bisher vorgestellten, generieren aus solchen Daten häufig über eine Million Dreiecke. Eine hochaufgelöste Version $(512 \times 512 \times 178)$ des CT–Datensatzes, der den farbigen Abbildungen auf der Seite 73 zugrundeliegt, führt beispielsweise bei einem Isowert von -200 und Anwendung eines korrigierten *Marching Cubes*–Algorithmus zu einer Isoflächenapproximation mit 1.38 Millionen Dreiecken.

Eine Möglichkeit, Modelle solcher Größe zu vermeiden, ist das Reduzieren der Auflösung der gegebenen Daten. Eine naive Vorgehensweise zur Halbierung der Datengröße würde dabei in einer gewählten Richtung jede zweite Schicht entfernen, was den völligen Verlust dieser Informationen bedeutet. Üblicherweise wird man stattdessen mehrere Datenwerte mittels Interpolation zu einem neuen zusammenfassen. Nichtsdestotrotz ist eine solche Reduktion der Eingangsdaten bei Betrachtung der Annahme A1 auf Seite 12 höchst problematisch, denn durch das *Subsampling* können wichtige Details verlorengehen. Ein Halbieren der Auflösung in allen drei Richtungen verringert

die Anzahl der Polygone erfahrungsgemäß auf etwa ein Viertel [NH92], da M in der Größenordnung von N^2 liegt. Alternativen dazu sind Methoden zur Verringerung der Größe der polygonalen Approximation, mit denen sich dieses Kapitel beschäftigt, und parallele Darstellungsverfahren, die in Kapitel 5 Erwähnung finden.

Polygonreduzierende Verfahren lassen sich in zwei große Gruppen von Methoden unterteilen:

Die erste Gruppe umfaßt diejenigen Algorithmen, die nur für spezielle Anwendungen tauglich sind und dabei häufig direkt in das Verfahren eingreifen, das die polygonalen Modelle produziert. In dieser Arbeit sind speziell Isoflächenalgorithmen mit interner Polygonreduktion von Interesse. Ich werde in den folgenden Unterkapiteln den *Compact Cubes*– (Kap. 4.2) und den *Discretized Marching Cubes*–Algorithmus (Kap. 4.3) vorstellen. Zu den Vertretern der ersten Gruppe zähle ich auch adaptive Methoden (Kap. 4.1), die die Genauigkeit der Approximation lokal vom Verhalten der Feldfunktion anhängig machen. Drei solcher Arbeiten werden im Anschluß an diese Einleitung diskutiert.

Der zweite prinzipielle Ansatz ist, die Polygonreduktion separat als Postprocessing–Schritt auszuführen. Solche anwendungsunabhängigen Verfahren arbeiten nur mit der gegebenen polygonalen Fläche und beziehen keine Informationen über die Daten ein, auf deren Grundlage die Polygone erzeugt wurden.

Es ist offensichtlich, daß die erste Gruppe von Verfahren ein größeres Potential besitzt, eine möglichst adäquate Approximation der Isofläche mit geringerer Polygonanzahl zu erreichen, da Informationen über die Gittertopologie und die genauen skalaren Werte einbezogen werden können. Eine wesentliche Reduzierung der Polygonanzahl bringt natürlich im allgemeinen einen gewissen Genauigkeitsverlust in der Approximation der Isofläche mit sich. Die strengen Forderungen F2 (Konsistenz) und F3 (Genauigkeit) können hier deshalb nicht angelegt werden. Von den restlichen Forderungen, insbesondere Forderung F1 (Stetigkeit), soll dagegen nicht abgerückt werden. Forderung F5 (Polygonanzahl) ist natürlich gemeinsames Ziel aller Ansätze dieses Kapitels. Die meisten der Verfahren greifen für die Untersuchung der Gitterzellen auf einen bestimmten Algorithmus zurück, lassen sich aber prinzipiell auch mit einer anderen Zellpolygonalisierungsmethode implementieren. Die Erfüllung einiger topologischen Forderungen ist davon, wie im vorigen Kapitel verdeutlicht, stark beeinflußt. Durch die Polygonreduzierung entstehen oft zusätzliche, spezifische Probleme, auf die in den jeweiligen Abschnitten näher eingegangen wird.

4.1 Adaptive Algorithmen

Die Anzahl und Größe von mit herkömmlichen Isoflächenalgorithmen produzierten Dreiecken hängt direkt mit dem Gitterabstand zusammen, da die Dreieckspunkte als Schnittpunkte auf den Gitterkanten definiert sind. Die Verbindung der Dreiecke erfolgt nur innerhalb einer Zelle, so daß eine Dreieckskante maximal so lang wie die längste Raumdiagonale in der Gitterzelle sein kann. Man geht gemäß Annahme A1 auf Seite 12 also implizit davon aus, daß sich die Aufteilung des Gitters am Verlauf der Feldfunktion orientiert und in Gebieten mit hoher Varianz der Feldwerte kleinere Gitterabstände aufweist als in Regionen, wo sich die Skalarwerte weniger stark ändern.

In vielen Simulationen und Meßverfahren sind die Gitterabstände jedoch fest (zumeist uniform) vorgegeben. Adaptiven Algorithmen liegt die Idee zugrunde, die Größe der erzeugten Polygone der Gestalt der Funktion anzupassen. Während die in Kapitel 3 vorgestellten *Beveled–Surface–* Methoden für einen schwach gekrümmten Teil der Isofläche Unmengen in der gleichen Ebene liegender Dreiecke generieren, stellen es sich adaptive Verfahren zum Ziel, die Isofläche in diesem Fall mit einer deutlich geringeren Anzahl größerer Dreiecke zu approximieren.

4.1.1 Splitting Box

MÜLLER[1] und STARK[2] stellten 1993 erstmals einen solchen Algorithmus vor, den sie *Splitting Box* [MS93] tauften. *Boxen* sind dabei hexaedrische Volumina, die aus mehreren Gitterzellen gebildet werden. Die Anzahl der Gitterpunkte auf der Boxkante wird als ihre Länge bezeichnet. Der Algorithmus startet mit einer Box, die das gesamte Gitter umfaßt. Schritt für Schritt wird die Box dann jeweils entlang der längsten Kante der Box bisektiert, so daß man immer wieder zwei neue Unterboxen erhält. Der Unterteilungsprozeß endet, wenn eine $2 \times 2 \times 2$–Box erreicht ist. Die Zellen werden dann in herkömmlicher Weise polygonalisiert. Stößt man auf eine Box, deren zwölf Kanten allesamt mindestens eine Zellkante enthalten, deren Gitterpunkte ein unterschiedliches Vorzeichen aufweisen, wird die Unterteilung vorerst gestoppt. Über lineare Interpolation auf den Zellkanten (zwischen den benachbarten Gitterpunkten mit unterschiedlichem Vorzeichen) werden die Schnittpunkte bestimmt und entsprechend einem der Prinzipien aus Kapitel 3 zu Konturumrissen auf den Außenflächen der Box verbunden. Das so entstandene, im allgemeinen nichtplanare Polygon für die Box wird zumeist nur eine grobe Approximation des wirklichen Verlaufs der Isofläche in der Box sein. Deshalb wird im Anschluß die Unterteilung weiter fortgesetzt, nun allerdings mit der Maßgabe, die Qualität der Approximation zu überprüfen und gegebenenfalls Korrekturen vorzunehmen. Dafür wird ein Bisektionsbaum verwendet [MS93]. Das größte Problem bei einem solchen Vorgehen ist das Vermeiden von Löchern, die entstehen können, wenn zwei benachbarte Boxen eine unterschiedliche Unterteilungstiefe aufweisen. MÜLLER/STARK lösen dieses Problem, indem das Verfahren gemeinsame Boxflächen zwar unabhängig voneinander bearbeitet, aber auf ihnen erwiesenermaßen gleiche Konturlinien generiert. Die erreichte Polygonreduktion gegenüber einer Variante des Verfahrens von WYVILL et al. (Kap. 3.1) liegt im Bereich von 30–85%, stark abhängig vom Datensatz. Leider können jedoch beim *Splitting–Box–*Algorithmus selbst größere Teilstücke der Isofläche unberücksichtigt bleiben, denn er überprüft nicht alle Werte im Innern einer Box. Die Laufzeit des Verfahrens beträgt prinzipiell $O(N^3)$. Verglichen mit dem Algorithmus von RÖLL et al. (Kap. 3.3.3) liegt die Laufzeit allerdings um ca. den Faktor 5–15 höher [RHvK95].

4.1.2 Adaptive Marching Cubes

Ein neuerer adaptiver Ansatz stammt von SHU et al. [SZK95]. Ihr Algorithmus *Adaptive Marching Cubes (AMC)* zerteilt das Gesamtvolumen sukzessiv in kleinere Teile, geht aber bereits von einer initialen Unterteilung in Würfel gleicher Größe aus. Für jeden Würfel wird auf Basis der Eckpunkte eine Polygonalisierung in herkömmlicher Weise vorgenommen. Die Entscheidung, ob jeweils

[1] Electronic Mail: `mueller@informatik.uni-freiburg.de`
[2] Electronic Mail: `stark@informatik.uni-freiburg.de`

weiter unterteilt wird, trifft der Algorithmus auf Basis berechneter Normalen. Dahinter steckt die Idee, daß sich über den Vergleich von Normalenvektoren auf die lokale Glattheit der Funktion schließen läßt. Unterscheidet sich die Richtung der Normalenvektoren kaum, wird der Schnitt der Isofläche mit der Box durch eine Linie approximiert. Ansonsten wird weiter unterteilt, bis das Kriterium einmal erfüllt ist oder die Box die Größe einer Gitterzelle hat. Das Kriterium lautet wie folgt: Überprüfe, ob für irgendeines der gerade generierten Dreiecke (mit den Normalen n_0, n_1, und n_2 gilt:

$$\exists\, i \in \{0,1,2\} : \qquad \arccos\left(n_i n_{((i+1) \mod 3)}\right) < \Delta \tag{4.1}$$

Die Konstante Δ beschreibt den Winkel, den zwei Normalen maximal haben dürfen, damit das Kriterium erfüllt werden kann.

Der AMC–Ansatz löst das Problem des eventuellen Auftretens von Löchern an gemeinsamen Flächen von Boxen unterschiedlicher Größe über *Patches*, ähnlich dem Vorgehen zur Korrektur des *Marching Cubes*–Algorithmus in Kapitel 3.3.3. Hier sind die Bedingungen, unter denen Löcher auftreten, allerdings völlig andere. SHU et al. [SZK95] unterscheiden 22 Basiskonfigurationen für die Form der Löcher, aus denen sich die entsprechenden Patches ergeben. Die für diesen Schritt zu speichernden Informationen verbrauchen einen Speicherplatz von $O(N^2)$.

Der AMC–Algorithmus läßt sich wie folgt in Pseudocode zusammenfassen:

```
algorithm adaptive_marching_cubes
    Teile das Gesamtvolumen in initial cubes
    for each initial cube c do
        call process_cube(c)
        if Loch tritt auf then
            füge Patches von der Form und Größe der Löcher ein
        endif
    endfor
    procedure process_cube(c)
        if (c wird von der Isofläche geschnitten) then
            bestimme die Schnittpunkte auf den Kanten von c
            berechne die Normalen im Schnittpunkt
            if c ist Gitterzelle or Kriterium 4.1 erfüllt then
            gib die Dreiecke aus
            speichere Informationen für die Patches
            else
                unterteile c in acht Teilwürfel
                for each Teilwürfel c_i do
                    call process_cube(c)
                endfor
            endif
        else
            speichere Informationen für die Patches
        endif
    endprocedure
```

endalgorithm

Die Laufzeit des Algorithmus (prinzipiell $O(N^3)$), die Anzahl der generierten Polygone und die Genauigkeit der Approximation hängen entscheidend von der Wahl der Größe der *initial cubes* ab. Bei einer Größe von 2 und einem $256 \times 256 \times 113$–CT–Datensatz hat AMC gegenüber dem *Marching Cubes*–Algorithmus beispielsweise eine um etwa ein Viertel geringere Laufzeit und generiert nur halb so viele Polygone [SZK95]. In geringer Auflösung sind die Darstellungen dabei kaum zu unterscheiden. Die Genauigkeit der Approximation wird für glattere Isoflächen besser sein, da auch beim AMC Details des Verlaufs der Isofläche „verwischt" werden können. Die Komplexität der Implementation ist sowohl für den *Splitting Box*– als auch für den AMC–Algorithmus sehr hoch. Das bei hexaedrischen Gittern auftretenden Löcherproblem (*hängende Knoten*) läßt sich bei Tetraedergittern vermeiden.

4.1.3 Isoflächen variabler Auflösung

Eine andere Vorgehensweise, die man ebenfalls in dieses Gebiet einordnen kann, ist die von NING und HESSELINK. Sie beschreiben in [NH92] ein aufwendiges Verfahren, das ein gegebenes Skalarfeld adaptiv mit verschiedenen Unterteilungstiefen polygonalisiert. Das Skalarfeld wird dabei als Octree (siehe Kap. 3.7.1) gespeichert. Das Vorgehen wird im Kern von einem volumetrischen Fehlerkriterium gesteuert, das von einem trilinearen Modell ausgeht. Der Algorithmus berechnet zuerst ein polygonales Modell für den gesamten Octree. Auf Grundlage des Fehlerkriteriums werden dann solche Regionen (d.h. Teilbäume im Octree) zusammengefaßt, deren Anteil am Gesamtfehler am geringsten ist. Für eine gewählte „Auflösung" (Polygonanzahl), kann so die (auf der Basis des Fehlerkriteriums) ideale Isoflächenapproximation bestimmt werden. Um das Fehlerkriterium anwenden zu können, muß für jeden Teilbaum des Octree die zugehörige Isoflächenapproximation (auf der Basis des Algorithmus von WYVILL et al., siehe Kap. 3.1) einzeln bestimmt und abgespeichert werden. Der Laufzeitaufwand ist damit selbst bei kleinen Datensätzen extrem hoch. In [NH92] wird bereits für einen sehr kleinen Datensatz (31^3) über Laufzeiten von ca. 30 Stunden (alter Ansatz *stepwise optimal*) bzw. 8 Minuten (neuer Ansatz *slope–constrained*) berichtet. NING et al. ordnen ihren Ansatz deshalb eher in den Bereich der Datenkompression bzw. Optimierung ein. Die Generierung einer nahezu idealen Approximation einer Fläche mit einer vorgegebenen Anzahl von Polygonen ist für einige Anwendungsbereiche durchaus von Interesse. Die Implementation [NH92] des Verfahrens ist jedoch als höchst komplex einzuschätzen.

4.2 Compact Cubes

Der 1991 von D. MOORE[3] und J. WARREN[4] erstmals vorgestellte *Compact Cubes*–Algorithmus [MW91, MW92] stellt sich mehrere wichtige Ziele. Neben der Reduzierung der Polygonanzahl

[3]Electronic Mail: `dougm@cs.rice.edu`, WWW: `http://www.cs.rice.edu/~dougm/`
[4]Electronic Mail: `jwarren@cs.rice.edu`, WWW: `http://www.cs.rice.edu/~jwarren/`

wird eine Verbesserung der Qualität der Oberfläche bei kaum höherer Laufzeit angestrebt — drei Forderungen, die im allgemeinen kaum simultan zu erreichen sind. Unter Oberflächenqualität wird hier insbesondere verstanden, daß die Dreiecksrepräsentation möglichst viele wohlgeformte und wenig degenerierte Dreiecke enthält. In die letzte Gruppe fallen sowohl sehr langgetreckte als auch sehr kleine Dreiecke, die Linien bzw. Punkten ähneln. Solche Dreiecke tragen naturgemäß sehr wenig zur Gestalt der Fläche bei und können sowohl bei der Darstellung auf der Grundlage eines Beleuchtungsmodells als auch bei der Anwendung analytischer Verfahren auf das polygonale Gitter zu Problemen führen.

Der Algorithmus läßt sich auf jeder der in Kapitel 3 vorgestellten *Beveled–Surface*–Methoden aufbauen, die lineare Interpolation auf den Gitterkanten verwenden. Einen dieser Algorithmen, z.B. einen korrigierten *Marching Cubes*–Algorithmus, wenden wir wie üblich auf die gesamten Eingangsdaten an, merken uns dabei aber zusätzlich für jeden berechneten Schnittpunkt der Isofläche den nächstgelegenen Gitterpunkt. Dies kann ohne zusätzlichen Berechnungsaufwand bei der Interpolation erfolgen. Der Schnittpunkt heiße dann *Satellit* des Gitterpunkts, der wiederum als *Planet* des späteren Dreieckseckpunktes bezeichnet wird. Die vom korrigierten *Marching Cubes*–Algorithmus generierte Dreiecksfläche sei mit P_1 bezeichnet. Aus dieser Repräsentation sollen nun die degenerierten Dreiecke beseitigt werden, ohne die Genauigkeit der Approximation wesentlich zu verschlechtern. In einem ersten Schritt werden sehr kleine Dreiecke in Punkte und langgestreckte in Linien überführt:

procedure collapse
    for each Dreieck T in P_1 **do**
        if Planeten aller Eckpunkte von T paarweise verschieden **then**
            produziere ein Dreieck, das die Planeten (Gitterpunkte) verbindet
        else ignoriere T (T kollabiert zu Kante oder Punkt)
        endif endfor
endprocedure

Die nach diesem Schritt entstehende Approximation P_2 enthält deutlich weniger Dreiecke als P_1. Die Eckpunkte der Dreiecke sind allesamt Gitterpunkte der Eingangsdaten und weichen damit um bis zu $\delta/2$ von den eingangs berechneten Schnittpunkten aus P_1 ab. Im letzten Schritt wird dieser Fehler wieder korrigiert, indem die Punkte von P_2 wieder zurechtgerückt werden

procedure displace
    for each Punkt g in P_2 **do**
        verschiebe g auf die durchschnittliche Position seiner Satelliten
    endfor
endprocedure

Da gemäß der bekannten Annahmen aus Kapitel 2 davon ausgegangen wird, daß die Satelliten direkt auf der Isofläche der Feldfunktion liegen, können die so berechneten Punkte nur geringfügig von der Isofläche entfernt liegen. Bei Daten aus fehlerbehafteten Meßverfahren, wie beispielsweise

medizinischen CT–Daten, kann diese Abweichung vernachlässigt werden, da sie deutlich unterhalb von $\delta/2$ liegt.

Aus diesem Grund habe ich den Algorithmus in das Isoflächenmodul von HyperPlan (siehe Kap. 3.3.2) implementiert, wo solche Daten den Hauptanteil der auftretenden Problemstellungen ausmachen. Durch eine geschickte Zusammenfassung der Schritte *collapse* und *displace* konnte erreicht werden, daß die Laufzeit des Programms nur ca. 10% über der Laufzeit des korrigierten *Marching Cubes*–Algorithmus liegt, auf dem auch die implementierte Version des *Compact Cubes*–Algorithmus basiert. Diese Geschwindigkeitseinbuße wird allein schon durch die deutlich verminderte Zeit zum Darstellen der Isofläche aufgewogen. Die Anzahl der erzeugten Polygone liegt in den meisten Anwendungsfällen bei 50–60%, verglichen mit dem korrigierten *Marching Cubes*–Algorithmus. In Abbildung 4.2 lassen sich die mit meinem Programm auf Grundlage der beiden Verfahren erzeugten Isoflächen eines kleinen Datensatzes vergleichen. Die bessere Gestalt

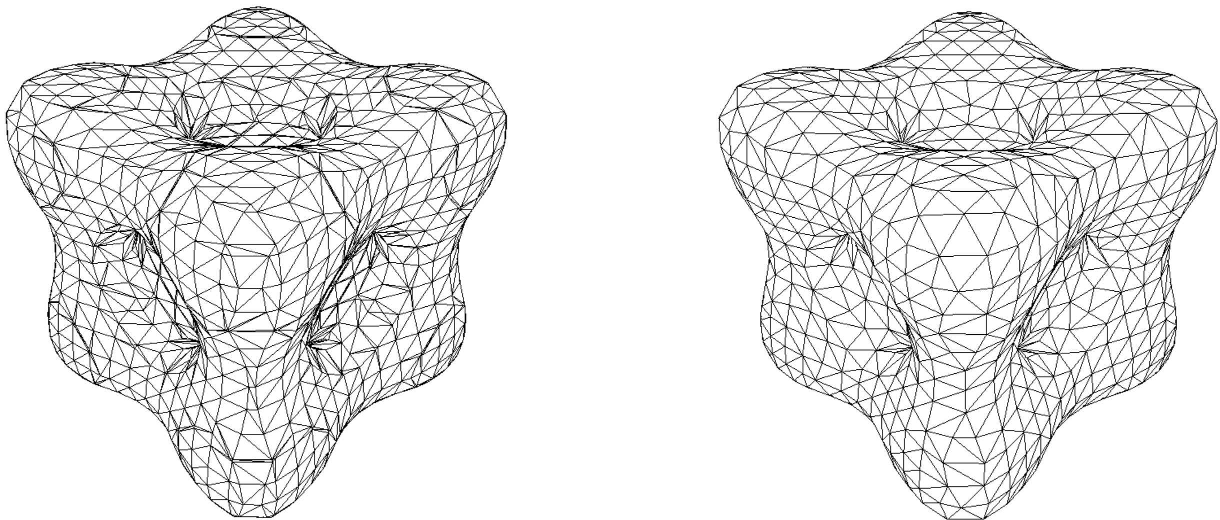


Abbildung 4.1: *Isofläche eines Beispieldatensatzes (*$20 \times 20 \times 20$*) aus IRIS Explorer mit Isowert 1.5, berechnet mit dem Compact Cubes–Algorithmus (rechts, 2200 Dreiecke) im Vergleich mit dem Ergebnis des (korrigierten) Marching Cubes–Algorithmus (links, 3808 Dreiecke).*

der Dreiecke ist deutlich sichtbar. Die Form der Isofläche unterscheidet sich kaum, obwohl die vom *Compact Cubes*–Algorithmus produzierte Fläche 42% weniger Polygone enthält. Die Verbesserung der Gestalt der Dreiecke ist nicht nur visuell auszumachen, sondern auch meßbar. Ein Standardverfahren ist die Berechnung des *aspect ratio*, dem Verhältnis von innerem zu äußerem Radius. In [MW91] wird bewiesen, daß dieser Wert im schlimmsten Fall 0.0503 beträgt. In der Praxis tritt sehr selten ein kleinerer *aspect ratio* als 0.25 auf. Bei herkömmlichen Ansätzen kann der Wert beliebig gegen Null gegen. Der *Compact Cubes*–Algorithmus stellt nach meiner Meinung einen geeigneten Kompromiß zwischen möglichst niedriger Polygonanzahl, genauer Approximation und geringer Laufzeit dar. In meinem HyperPlan-Modul wird das Verfahren deshalb als Standard verwendet und nur selten, vor allem zu Vergleichszwecken, auf den korrigierten *Marching Cubes*–Algorithmus zurückgegriffen. Meine Implementation des *Compact Cubes*–Algorithmus profitiert zusätzlich von dem Fakt, daß die Identifikation von gemeinsamen Eckpunkten der Dreiecke keinen

zusätzlichen Aufwand erfordert. Die Dreiecksfläche kann somit effizient über Listen[5] von Indizes, Koordinaten und Normalen (siehe Kap. 3.3.2) repräsentiert werden, ohne daß eine Koordinate mehrmals auftritt. Mein Isoflächenmodul wird z. B. in [SH95a] beschrieben.

Ein topologisches Problem des *Compact Cubes*–Algorithmus sei nicht verschwiegen. Wenn zwei Punkte von verschiedenen Teile der Isofläche nahe an einem gemeinsamen Gitterpunkt gelegen sind, verschmelzen sie im Schritt *collapse* zu einem neuen Punkt, d. h. die zwei Teile sind daraufhin über diesen Punkt verbunden. Durch eine spezielle Behandlung dieses Falles, die dafür sorgt, daß dann die zwei Satellitenpunkte erhalten bleiben, läßt sich dieses Problem unkompliziert lösen, ohne sich entscheidend auf Polygonanzahl oder Laufzeit auszuwirken. Die so generierte Fläche muß C^0–stetig sein, wenn P_1 C^0–stetig ist. Da in meiner Implementation des *Compact Cubes*–Algorithmus im Modul *HxIsosurface* ein korrigierter *Marching Cubes*–Algorithmus verwendet wird, der diese Eigenschaft hat, wird Forderung F1 (Stetigkeit) erfüllt. Die farbige Abbildung auf Seite 6 oben (Bild 5) zeigt eine mit dem Modul auf Grundlage des *Compact Cubes*–Algorithmus generierte Isofläche eines Benzol–Moleküls mit eingezeichnetem Molekülmodell und verdeutlicht damit gleichzeitig die Eignung von HyperPlan für Visualisierungsaufgaben auch jenseits medizinischer Anwendungen.

4.3 Discretized Marching Cubes

Während beim *Compact Cubes*–Algorithmus die Größe der Dreiecke direkt vom Gitterabstand abhängt und alle Dreiecke ein ähnliche Größe haben, versucht der 1994 von MONTANI[6], SCATENI[7] und SCOPIGNO[8] vorgestellte *Discretized Marching Cubes*–Algorithmus [MSS94a], diese Einschränkung zu überwinden und an schwach gekrümmten Flächen entsprechend größere Dreiecke zu produzieren. Das Verfahren basiert grundsätzlich auf einem *Marching Cubes*–Ansatz, mit der Besonderheit, daß statt linearer Interpolation eine Teilung in der Mitte der Kante (siehe auch Kap. 3.3.3) angewendet wird. Der damit implizierte Fehler in der Position der Polygoneckpunkte liegt unterhalb von $\delta/2$. Für binäre Daten führt das Vorgehen zu identischen Resultaten.

Die Wahl der Schnittpunkte als Mittelpunkte der Gitterkanten birgt einige Vorteile in sich. Erstens können die Dreieckseckpunkte nur an dreizehn unterschiedlichen Positionen in der Zelle liegen, nämlich auf der Mitte einer der zwölf Kanten oder auf dem Mittelpunkt der Zelle. Der Mittelpunkt einer quaderförmigen Zelle ist der Schnittpunkt der Raumdiagonalen. Zweitens ist die Ebene, in der ein generiertes Dreieck liegen muß, unabhängig von den Datenwerten an den Gitterpunkten. Es ergeben sich nur 13 mögliche Ebenen:

$$\begin{aligned} x = c,\ y = c,\ z = c, \\ x \pm y = c,\ x \pm z = c,\ y \pm z = c, \\ x \pm y \pm z = c \end{aligned} \tag{4.2}$$

[5] In *Open Inventor*™ konnte ich für solch eine Repräsentation die Klasse *SoIndexedFaceSet* benutzen.
[6] Electronic Mail: `montani@iei.pi.cnr.it`
[7] Electronic Mail: `riccardo@chianti.csr4.it`
[8] Electronic Mail: `R.Scopigno@cnuce.cnr.it`

Die Einführung des Zellmittelpunktes ermöglicht eine barizentrischen Tessellation (siehe Definition 3.1 auf Seite 21) aller auftretenden planaren und nichtplanaren Polygone in gleichseitige oder rechtwinklige Dreiecke und Rechtecke. Durch genaue Untersuchung der durch die barizentrische Tessellation leicht abgewandelten Basisfälle (vgl. Abb. 3.3 auf S. 24) läßt sich eine Numerierung der möglichen Formen der Drei– und Vierecke und ihrer Positionen innerhalb der topologischen Polygone vornehmen. Diese Informationen werden in eine neue Lookup–Tabelle aufgenommen. Die Topologietabelle basiert grundsätzlich auf der eines korrigierten *Marching Cubes*–Algorithmus(siehe Kap. 3.3.1), wodurch die C^0–Stetigkeit der berechneten Isoflächenapproximation sichergestellt wird. Der Isoflächengenerierungsschritt des *Discretized Marching Cubes*–Algorithmus läuft bis auf die erwähnten Unterschiede analog zum *Marching Cubes*–Ansatz ab. Die erzeugten planaren Polygone werden dabei als Tripel von Zellindex, einem *form code* und einem *incidence code* repräsentiert und in 26 Hashtabellen, einer für jede mögliche Richtung der Ebene (siehe Gleichung 4.2) abgespeichert. Während der *form code* die Form der Teilpolygone (z. B. Rechteck der Seitenlänge δ) repräsentiert, beschreibt der *incidence code* die Lage der Ebene, in der das planare Polygon liegt. Das Vorzeichen des *incidence code* entspricht der prinzipiellen Orientierung der Fläche, gegeben durch die Normalen. Die Isoflächengenerierung läuft sehr schnell ab. Die hohe Geschwindigkeit ergibt sich daraus, daß das Vorgehen nahezu vollständig auf Lookup–Tabellen basiert und, abgesehen von der Normalenberechnung, mit ganzzahliger Arithmetik auskommt.

In der *merging*–Phase können dann auf Grundlage der 26 Hashtabellen koplanare Polygone zusammengefaßt werden. Es werden grundsätzlich diejenigen Polygone vereint, die jeweils in der gleichen Hashtabelle liegen und eine gemeinsame Kante haben. Der *Discretized Marching Cubes*–Algorithmus nutzt dabei eine spezielle Datenstruktur, sog. FREEMAN *chains* [MSS94a]. Der Schritt *merging* nimmt typischerweise 85% der Laufzeit in Anspruch. Die restlichen 15% teilen sich auf die Zellpolygonalisierung (10%) und die Normalenberechnung (5%) auf.

Für nähere Einzelheiten sei auf die detaillierte Beschreibung in [MSS94a] verwiesen. Eine Implementation[9] des Algorithmus ist frei erhältlich.

4.4 Separate Polygonreduktion

Polygonreduktion auf beliebigen Polygongittern ist ein Forschungsthema für sich. Wir wollen hier nur diejenigen Verfahren kurz vorstellen, die häufig im Kontext der Isoflächenkonstruktion genannt und genutzt werden. Diese Algorithmen beschränken sich auf Dreiecksgitter, wie sie durch die in Kapitel 3 diskutierten *Beveled–Surface*–Methoden üblicherweise generiert werden.

[9] Entstanden in Zusammenarbeit der Visual Computing Group des I.E.I.–C.N.R. und CNUCE–C.N.R. (Pisa, Italien) und der Scientific Visualization Group des CRS4 (Cagliari, Italien). Programmautoren sind P. CRISCIONE, C. MONTANI, R. SCATENI und R. SCOPIGNO. Derzeit wird Release 1.3 (Februar 1996) angeboten. Die aktuelle Version ist im WWW unter `http://miles.cnuce.cnr.it/cg/swOnTheWeb.html` frei verfügbar.

4.4.1 Mesh Optimization

HOPPE[10] et al. [HDD+93] stellten 1993 ein Verfahren namens *Mesh Optimization* vor, das aus einem gegebenen Dreiecksgitter ein neues generiert, das die gleichen topologischen Eigenschaften hat, weniger Dreiecke enthält und eine gute Approximation der Punkte des gegebenen Gitters ist. Die Abbildungen 4.2 und 4.3 zeigen die Ergebnisse dieses Vorgehens. Die Optimierung basiert

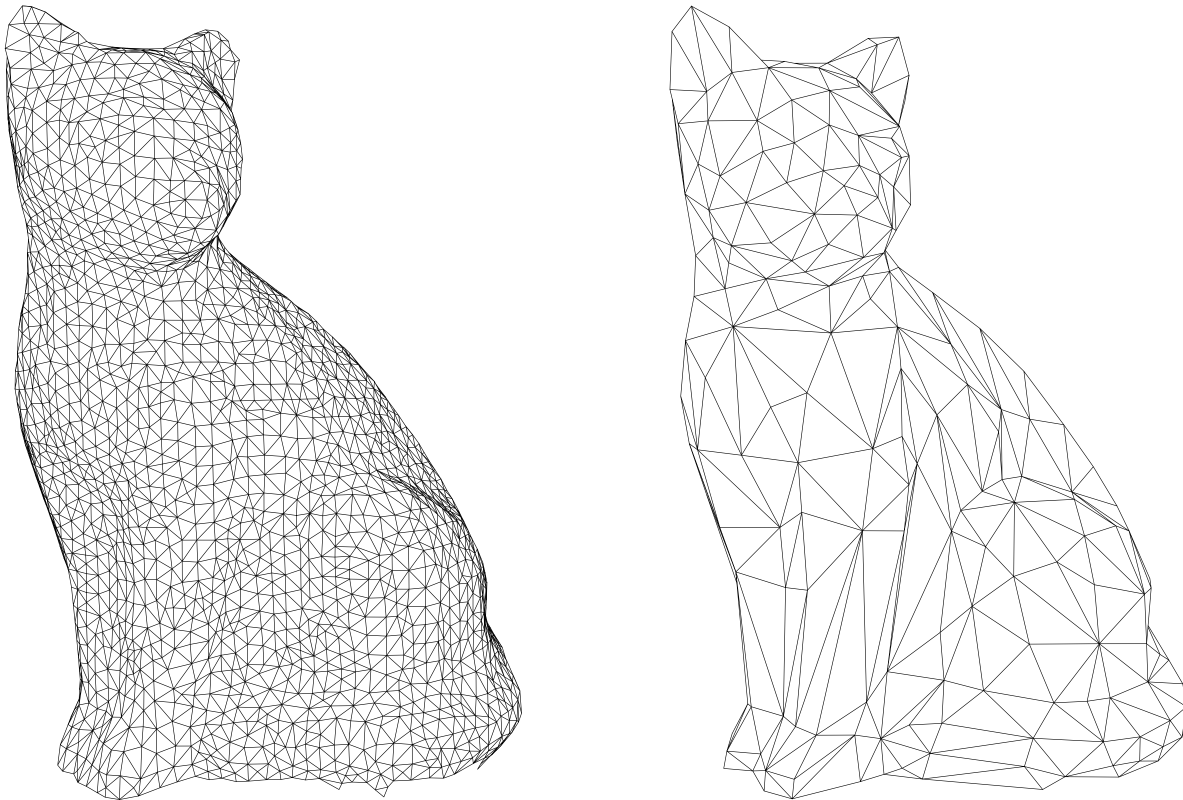


Abbildung 4.2: *Rekonstruktion mittels Polygonreduzierung nach* HOPPE *et al. Das optimierte Gitter (rechts) unterscheidet sich deutlich vom zugehörigen initialen Gitter (links).*

auf der Bewertung einer Energiefunktion über dem Gitter. Durch geschickte lokale Operationen, wie dem Entfernen oder Verschieben von Punkten bzw. dem Verschmelzen oder Vertauschen von Kanten soll diese Funktion minimiert werden. Das Vorgehen wird u. U. mehrfach wiederholt, bis keine weiteren Verbesserungen möglich sind. Der Algorithmus kann sowohl zur Vereinfachung gegebener Gitter als auch zur Rekonstruktion von Flächen (Abb. 4.2) dienen.

Die Komplexität der Implementation ist extrem hoch, weshalb bei Bedarf auf die Distribution[11] der Autoren zurückgegriffen werden sollte. Die größte Schwierigkeit stellt selbst dann noch die geeignete Parametrisierung dar, ohne die der Algorithmus keine zufriedenstellenden Ergebnisse liefert.

[10] Electronic Mail: `hhoppe@microsoft.com`

[11] Die Software ist per Anonymous FTP unter `/pub/graphics/Recon.distrib.940504.tar.Z` auf `ftp.cs.washington.edu` frei erhältlich. Die Distribution vom 04.05.1994 enthält den Programmcode und einige Beispiele zu den Veröffentlichungen „Surface reconstruction from unorganized points“ (SIGGRAPH ’92), „Mesh optimization“ (SIGGRAPH ’93) (siehe [HDD+93]) und „Piecewise smooth surface reconstruction“ (SIGGRAPH ’94). Alle Programme sind in C++ geschrieben und benutzen Templates. Die Distribution ist auch auf der zu den SIGGRAPH ’94 Proceedings gehörenden CD–ROM enthalten.

Bekommt man dieses Problem in den Griff, sind die Ergebnisse beim Optimieren von schwach gekrümmten polygonalen Flächen hervorragend, wie Abbildung 4.3 eindrucksvoll unter Beweis stellt.

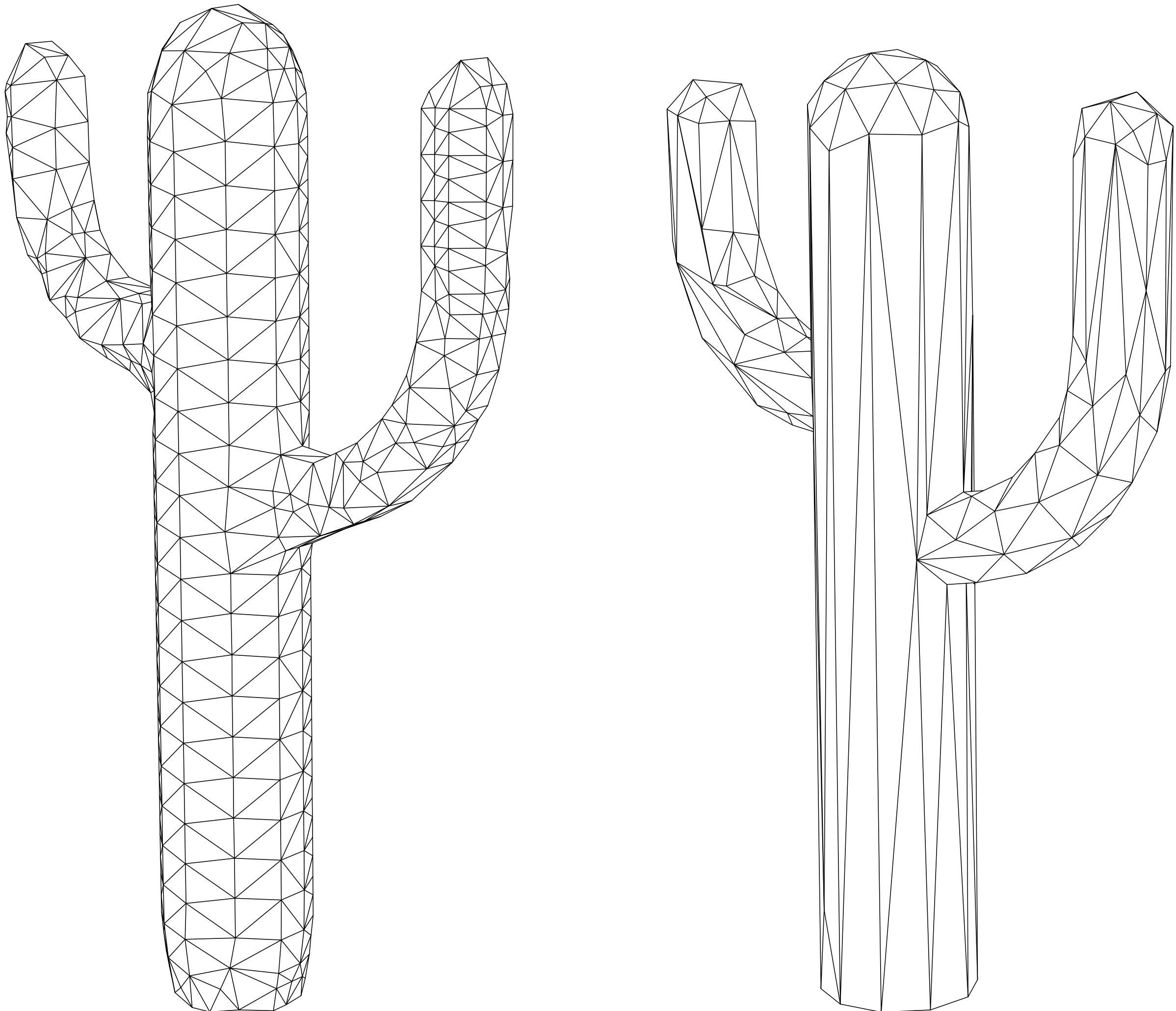


Abbildung 4.3: *Dieses Beispiel verdeutlicht die exzellente Triangulation schwach gekrümmter Flächen mit dem* HOPPE*–Algorithmus.*

Laufzeiten für die Optimierung der initialen Gitter, wie sie den Abbildungen 4.2 und 4.3 zugrundeliegen, liegen im Bereich von zehn Minuten.

4.4.2 Triangle Decimation

Das 1992 von WILLIAM J. SCHROEDER, JONATHAN A. ZARGE und WILLIAM E. LORENSEN vorgestellte Verfahren [SZL92] wurde ursprünglich direkt für die Anwendung auf die Ausgabedaten des *Marching Cubes*–Algorithmus entworfen, kann aber auch für Dreiecksgitter aus anderen Bereichen eingesetzt werden, insbesondere auch solche, die nicht einfach sind, d. h. wo nicht an jeder (inneren) Dreieckskante zwei Dreiecke zusammentreffen müssen. Das Verfahren generiert

daraus ein Drahtgitter, das zwar weniger Dreiecke enthält, dessen Punkte aber eine Untermenge der Originalpunkte sind. Für einige Anwendungen ist solch eine Eigenschaft von Wichtigkeit.

Das Prinzip des Algorithmus ist einfach. Das gegebene Dreiecksgitter P wird mehrfach durchlaufen. In jedem Schritt werden, unter Beachtung lokaler Topologie und Geometrie, diejenigen Knoten entfernt, die ein bestimmtes Abstands– oder Winkelkriterium erfüllen und die entstandenen Löcher jeweils retrianguliert:

algorithm decimate
    while not Endkriterium erfüllt **do**
        for each Punkt v von P **do**
            bestimme die lokale Topologie und Geometrie
            if not lokale Topologie ist *komplex* **then**
                if Dezimierungskriterium erfüllt **then**
                    entferne v aus P
                    trianguliere das Loch, ohne neue Eckpunkte einzuführen
                endif
            endif
        enddo
    enddo
endalgorithm

Das Endkriterium ist zumeist ein Zielkriterium, z. B. die Anzahl oder Prozentzahl zu reduzierender Polygone. Das Dezimierungskriterium richtet sich nach der lokalen Topologie, die in drei Klassen (siehe Abbildung 4.4) eingeteilt wird.

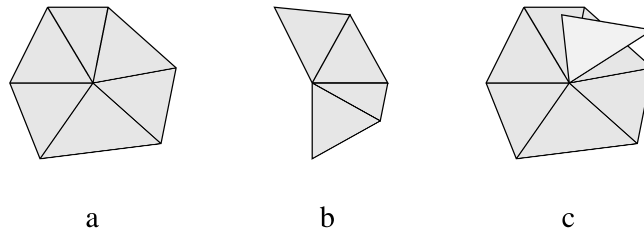


Abbildung 4.4: *Klassifikation der lokalen Topologie eines Punktes als simpel (a), außen (b) oder komplex (c)*

Im Falle einer *komplexen* Topologie (Abb. 4.4c), d. h. wenn die den Punkt lokal umgebende Dreiecksfläche nicht einfach ist, wird der Punkt auf gar keinen Fall entfernt. Ansonsten ist wird das vom topologischen Typ abhängende Kriterium überprüft. Die Entscheidung für den *simplen* Fall (Abb. 4.4a) wird vom Abstand des Punktes von einer von den umgebenden Punkten aufgespannten „durchschnittlichen Ebene“ [SZL92] abhängig gemacht. Weiterhin wird der Winkel der Kanten zueinander verglichen, um zu bestimmen, ob eine der Kanten *wichtig* ist und deshalb erhalten werden muß. Ist der Punkt vom Typ *außen* (Abb. 4.4b), so wird der Abstand des Punktes zu der

fehlenden, d. h. den Halbkreis schließenden Kante bestimmt. Ist der Abstand klein, wird die Kante eingezeichnet und der Punkt entfernt.

Das Verfahren ist wegen der eng lokal begrenzten Operationen für eine datenparallele Implementation geeignet. Der Algorithmus ist in sequentieller Form als Klasse *VtkDecimate* im *Visualization Toolkit* (siehe Fußnote (16) auf Seite 30) implementiert.

Kapitel 5

Parallele Algorithmen

Viele der Ausgangsdaten für die Isoflächengenerierung stammen aus numerischen Simulationen, die wegen ihrer Komplexität nicht auf normalen Workstations ausgeführt werden können, sondern auf Höchstleistungsrechner, z. B. massiv–parallele Systeme, verlagert werden. Die dabei anfallenden Daten sind, ebenso wie die Daten aus vielen modernen Meßmethoden, oft immens und übersteigen die Hauptspeicherkapazitäten herkömmlicher Einzelplatzrechner, an denen der Wissenschaftler die Visualisierung seiner Daten vornimmt. Es ist zwar, wie wir in Abschnitt 3.7 gezeigt haben, prinzipiell möglich, die Isoflächenberechnung auch dann durch schrittweises Einlesen von Teilen der Daten zu bewerkstelligen, dies ist aber mit erheblichen Geschwindigkeitseinbußen verbunden, denn die Daten müssen bei jeder Veränderung des Isowerts erneut eingelesen werden. Ein weiteres Manko ist, daß die gesamte Datenmenge erst zur Workstation, auf der die Isoflächenberechnung üblicherweise stattfindet, übertragen werden muß. Eine Visualisierung von Simulationsdaten während einer laufenden Berechnung, die dem Wissenschaftler sofortigen Aufschluß über die Ergebnisse geben kann und z. B. einen geeigneten Eingriff zur Variation von Parametern erlaubt, ist unter diesen Bedingungen nicht möglich. Es drängt sich der Wunsch auf, die Isoflächenberechnung ebenfalls auf dem Höchstleistungsrechner durchzuführen. Statt der Rohdaten müssen dann nur noch die Daten für die Repräsentation der Isofläche zur Grafikworkstation des Anwenders übertragen werden.

Wie wir in Kapitel 3 feststellen mußten, beträgt die Laufzeitkomplexität der Isoflächenalgorithmen prinzipiell $O(N^3)$. Mit der Verwendung von Octrees (Kap. 3.7.1) kann man die Laufzeit zwar auf $O(M \log N)$ senken, muß dafür aber einen Vorverarbeitungsschritt vornehmen, der nicht schneller als in $O(N^3)$ zu bewerkstelligen ist. Die Laufzeit bleibt bei großen Daten ein entscheidendes Problem, wie auch die Erfahrungen mit meinem Isoflächenmodul in HyperPlan (Kap. 3.3.2, 4.2) zeigen. Parallele Ansätze können die Geschwindigkeit von Algorithmen deutlich erhöhen, wenn es gelingt, die anfallenden Aufgaben effektiv auf mehrere Prozessoren aufzuteilen.

Die Sequentialität der in Kapitel 3 und 4 beschriebenen Algorithmen ist nicht in jedem Fall ein inhärentes Merkmal der prinzipiellen Vorgehensweise. Einigen der Verfahren wohnt eine implizite Parallelität inne, die von den jeweiligen Autoren zwar selten herausgestellt wird, aber häufig eine Parallelisierung mit vertretbarem Aufwand ermöglicht. Ich habe in den jeweiligen Abschnitten darauf aufmerksam gemacht. Es läßt sich zusammenfassen, daß für die Isoflächengenerierung

prinzipiell nur *lokale* Betrachtungen der Daten notwendig sind. Diese implizite Datenparallelität erleichtert und motiviert eine parallele Lösung des Isoflächenproblems. Welchen konkreten Algorithmus man verwendet, wird sehr von den Randbedingungen der Implementation abhängen. Ausgehend von der zur Verfügung stehenden Hardware und dem geplanten Einsatzszenario müssen dabei konkrete Entwurfsentscheidungen getroffen werden.

5.1 Bisherige parallele Implementierungen

Die Vorteile einer parallelen Isoflächenberechnung liegen, wie erläutert, auf der Hand. Es ist deshalb eher verwunderlich, daß es recht wenige Implementierungen für Parallelrechner gibt. Diese Tatsache läßt es zu, in diesem Kapitel über *alle* mir bekannten bisherigen parallelen Isoflächenprogramme zu berichten. Die zum Einsatz kommenden Hardwarearchitekturen sind, ebenso wie die ins Auge gefaßte Anwendung, sehr verschieden. Im folgenden sei eine kurze Übersicht über die jeweiligen Autoren und die Zielarchitektur, sowie Ort und Zeitpunkt der Entwicklung gegeben:

1. C. D. HANSEN[1], P. HINKER[2]; Thinking Machines Connection Machine CM–2; Los Alamos National Laboratory, New Mexico, USA; 1992 [HH92]
2. T. S. YOO, D. T. CHEN; Pixel–Plane 5; University of North Carolina, USA; 1994 [YC94]
3. P. MACKERRAS; Fujitsu AP1000; Australian National University, Australien; 1992 [Mac92]
4. P. J. CROSSNO[3]; Intel iPSC/860; Sandia National Laboratories, Albuquerque, New Mexico, USA; 1993 [CCJ93]

Der erste Ansatz [HH92] behandelt die Implementation auf einem SIMD–Rechner, einer CM–2 (Connection Machine) mit 2^{16} Prozessoren. SIMD–Rechner (single instrucion, multiple data) führen auf jedem Datenelement identische Operationen aus und bestehen aus vielen, meist nicht sehr leistungsfähigen Prozessoren. Bei der CM–2 kann zusätzlich noch jeder PE (*processing element*) in virtuelle PEs (*vp*) aufgeteilt werden. Für den Programmierer bietet sich der Vorteil, daß er die gesamte implizite Datenparallelität direkt nutzen kann, indem er jeden zu betrachtenden Datenblock auf einem solchen *vp* bearbeiten läßt. Wie in Kapitel 3 ausführlich diskutiert, kann die Isoflächengenerierung prinzipiell für jede Gitterzelle separat erfolgen. HANSEN et al. [HH92] verteilen in ihrem Ansatz alle Gitterzellen einzeln auf virtuelle PEs und wenden auf jede Zelle einen originalen *Marching Cubes*–Algorithmus an. Sie sind sich des Problems von Unstetigkeiten bewußt und konstatieren, daß sich das Programm leicht um Methoden zur Korrektur des Algorithmus, die höchstens *Nearest Neighbor*[4]–Kommunikation erfordern, erweitern lassen. Die Lookup–Tabelle enthält 256 Einträge, um Berechnungen zur Laufzeit zu sparen und wird auf jedem *vp* vollständig gespeichert. Das gleiche trifft für die Daten und Koordinaten der acht Gitterpunkte

[1]Electronic Mail: `hansen@acl.lanl.gov`
[2]Electronic Mail: `phinker@scisoft.com`
[3]Electronic Mail: `pjcross@cs.sandia.gov`
[4]Kommunikation zwischen direkt benachbarten Elementen

zu. Der Rechenschritt erfolgt auf jedem *vp* in bekannter Weise, nutzt aber *Nearest Neighbor*–Kommunikation, um doppelte Schnittpunktberechnungen zu vermeiden und die Normalen zu berechnen. Mit diesem Ansatz lassen sich beachtliche Resultate erzielen, sofern die Daten schon auf dem Rechner vorliegen. Die Laufzeit des Algorithmus ist bei gleicher Größe der Eingangsdaten nahezu unabhängig von deren Art, weil stets einer der virtuellen PEs die maximale Anzahl von Dreiecken pro Zelle erzeugt und damit die Gesamtlaufzeit bestimmt. Bei genügend großen Datensätzen, d. h. solchen mit mindestens so vielen Gitterzellen wie PEs, skaliert der Algorithmus nahezu linear mit der Anzahl dieser PEs. In [HH92] wird berichtet, daß sich mit dem Algorithmus unter den gemachten Voraussetzungen bis zu 170000 Dreiecke pro Sekunde generieren lassen, einer Zahl, die sehr nah an der maximalen Darstellungsleistung von Grafikworkstations liegt. In einer späteren Version[5] des Algorithmus wurden noch deutlich bessere Ergebnisse im Bereich von 1 Million Dreiecke pro Sekunde erreicht. Der Flaschenhals liegt hier also in der Übertragung und Darstellung der Polygone. Der Algorithmus ist mittlerweile auch auf andere Architekturen wie CM–5, SGI–SMP, IBM SP–2 und den CRAY T3D portiert, dürfte dort jedoch höchst unterschiedliche Ergebnisse liefern. Weiterhin wurde mit paralleler Polygonreduzierung experimentiert. Die Programme sind aufgrund von amerikanischen Exportbestimmungen nicht frei erhältlich.

Der zweite Ansatz stammt von YOO und CHEN [YC94]. Plattform ist eine *Pixel–Plane 5*, eine heterogene Grafik–Architektur, die sowohl MIMD[6]–, als auch SIMD–Parallelität bietet. Die Maschine enthält sowohl Intel–i860–basierte Grafikprozessoren als auch SIMD–Pixel–Prozessoren (Renderer), die von den Grafikprozessoren direkt angesprochen werden können. Der Isoflächenalgorithmus basiert grundsätzlich auf einer gleichmäßigen Datenverteilung und der Anwendung eines *Marching Cubes*–Algorithmus auf die Teildaten auf jedem PE. Die berichteten Ergebnisse sind dank der Fähigkeit der Hardware zu parallelem Rendering sehr gut und erlauben interaktive Anwendungen [YC94].

Der dritte Ansatz von MACKERRAS wurde für einen experimentellen MIMD–Rechner vom Typ Fujitsu AP1000 in einer Konfiguration mit 128 PEs entwickelt. Auf den auf die PEs schichtweise aufgeteilten Daten kommt jeweils ein identisches Programm zum Ablauf. Der Isoflächenalgorithmus trifft die Entscheidung über die Verbindung der Schnittpunkte auf Basis eines bilinearen Modells über eine Sortierung der Schnittpunkte (siehe Kap. 3.5). Die gewählte zweidimensionale Gebietszerlegung (*domain decomposition*) bringt Nachteile bezüglich des Lastausgleichs mit, da die Isofläche viele der Schichtausschnitte gar nicht schneidet. Es wird versucht, dieses Problem mittels *Interleaving* zu bekämpfen, so daß die jedem PE übergebenen Daten gleichmäßiger über die Schicht verteilt sind. Der Flaschenhals ist aber in jedem Fall das I/O über den Hostrechner, das dutzendfach mehr Zeit in Anspruch nimmt als der eigentliche Rechenvorgang. Es wurde deshalb ein einfacher Z–Buffer–Renderer [FvDFH90] getestet, der die Probleme deutlich entschärft.

Der vierte Ansatz bezieht sich auf einen MIMD–Rechner vom Typ Intel iPSC/860 und stellt eine Verbindung von Simulation und Isoflächenberechnung vor. Die Darstellung der Daten erfolgt über eine Netzwerkverbindung auf einer SGI–Workstation. Im Anwendungsfall, einer Strömungssimulation, dauern die numerischen Berechnungen allerdings deutlich länger als der Isoflächengenerierungsschritt, so daß mittlerweile[7] die Isoflächenkonstruktion wieder nach der Simulation stattfin-

[5] Diese Informationen entstammen einer privaten Korrespondenz mit HANSEN und HINKER.

[6] MIMD (multiple instruction, multiple data) bedeutet im Unterschied zu SIMD, daß auf jedem Prozessor unterschiedlicher Programmcode laufen kann.

[7] Diese neuen Informationen stammen direkt von der Programmautorin PATRICIA CROSSNO.

det. Das Verfahren ist nun mit einer parallelen Version der Polygonreduzierung nach SCHROEDER et al. [SZL92] (siehe Kapitel 4.4.2) kombiniert. Dabei muß beachtet werden, daß Punkte auf dem Rand des Teilvolumens im Dezimierungsschritt nicht entfernt werden dürfen.

Nachdem bisher nur Implementationen jenseits dieses Kontinents zur Sprache kamen, soll ein Ansatz aus Deutschland nicht vergessen werden, auch wenn er keine Parallelrechnerimplementierung darstellt. P. ELLSIEPEN[8] stellt in [Ell95] einen allgemeinen MIMD–Ansatz vor, der auf einem heterogenen Workstation–Cluster im einem Netzwerk (LAN oder WAN) arbeitet. Die Vorgehensweise konzentriert sich vor allem auf den Lastausgleichsmechanismus, den der Einsatz von Prozessoren und Peripherie unterschiedlicher Geschwindigkeit und Last erfordert. Der Isoflächenalgorithmus ist auf die speziellen Bedürfnisse des Anwendungsfalls (unstrukturierte Gitter aus Finite–Elemente–Methoden) abgestimmt.

5.2 Eigener Ansatz: PARADISO

Dem Wissenschaftler, der seine Berechnungen auf anderen als den im vorigen Kapitel genannten Parallelrechnern durchführt, nützen die Erkenntnisse über die Tauglichkeit und Erfolge der aufgeführten Ansätze wenig. Selbst wenn eine Implementation für die entsprechende Architektur vorliegt, verhindern, wie im Falle des T3D–Programms aus Los Alamos, häufig rechtliche Probleme deren Nutzung. Für Deutschland bedeutete dies bis jetzt, daß für die leistungsstärksten hierzulande verfügbaren Parallelrechner keine Software zur Isoflächengenerierung frei verfügbar ist. Mit der Entwicklung eines parallelen Isoflächenprogramms im Rahmen dieser Arbeit sollte diesem Zustand endlich Abhilfe geschaffen werden. Das Programm PARADISO (*PARAllel and Distributed ISOsurface generation*) wurde speziell für den bis dato schnellsten Rechner Deutschlands, den CRAY T3D SC 256 des Konrad–Zuse–Zentrums für Informationstechnik Berlins (ZIB), entworfen.

5.2.1 CRAY T3D

Bevor ich auf das von mir entwickelte parallele Isoflächenprogramm zu sprechen komme, möchte ich kurz wesentliche Eigenschaften des CRAY T3D–Systems vorstellen, die für den Entwurf des Programms entscheidend waren.

Der CRAY T3D ist ein skalierbarer, massiv–paralleler Rechner, der am ZIB in einer Konfiguration mit 256 Prozessoren vom Typ DEC Alpha 21064 betrieben wird. Die einzelnen Prozessoren mit je 64 MB Speicher sind in Knoten gruppiert und durch ein sehr schnelles Netzwerk mit einer nominellen Datentransferrate von 300MB/s in Form eines dreidimensionalen Torus (siehe Abbildung 5.1) miteinander verbunden. Auf jedem Knoten befinden sich zwei Prozessoren und zusätzliche Hardware, die unabhängig vom Prozessor Kommunikation und Datentransfer organisiert. Der Speicher ist physikalisch verteilt (*distributed memory*), kann aber global adressiert und damit als logisch gemeinsam betrachtet werden. Zugriffe auf Speicheradressen anderer Prozessoren bringen

[8] Electronic Mail: `Peter.Ellsiepen@mechbau.uni-stuttgart.de`

zwar eine Verzögerung um etwa den Faktor 10 (bei unbelastetem Netz) mit sich, die Zugriffszeit ist jedoch nahezu unbeeinflußt von der Lage der Knoten im Torus. Der CRAY T3D SC 256 ist physikalisch im gleichen Gehäuse wie seine Hostrechners untergebracht (siehe Abb. 5.1), welcher das I/O abwickelt und Teile des Betriebssystems sowie die Entwicklungsumgebung bereitstellt. Hostrechner ist am ZIB der Vektorrechner CRAY Y–MP4E/464. Eine *Barrier Synchronisation Hardware* ermöglicht die globale Synchronisation der PEs durch Semaphoren. Das T3D–System ist skalierbar, Programme müssen allerdings immer auf Zweierpotenzen von Prozessoren arbeiten.

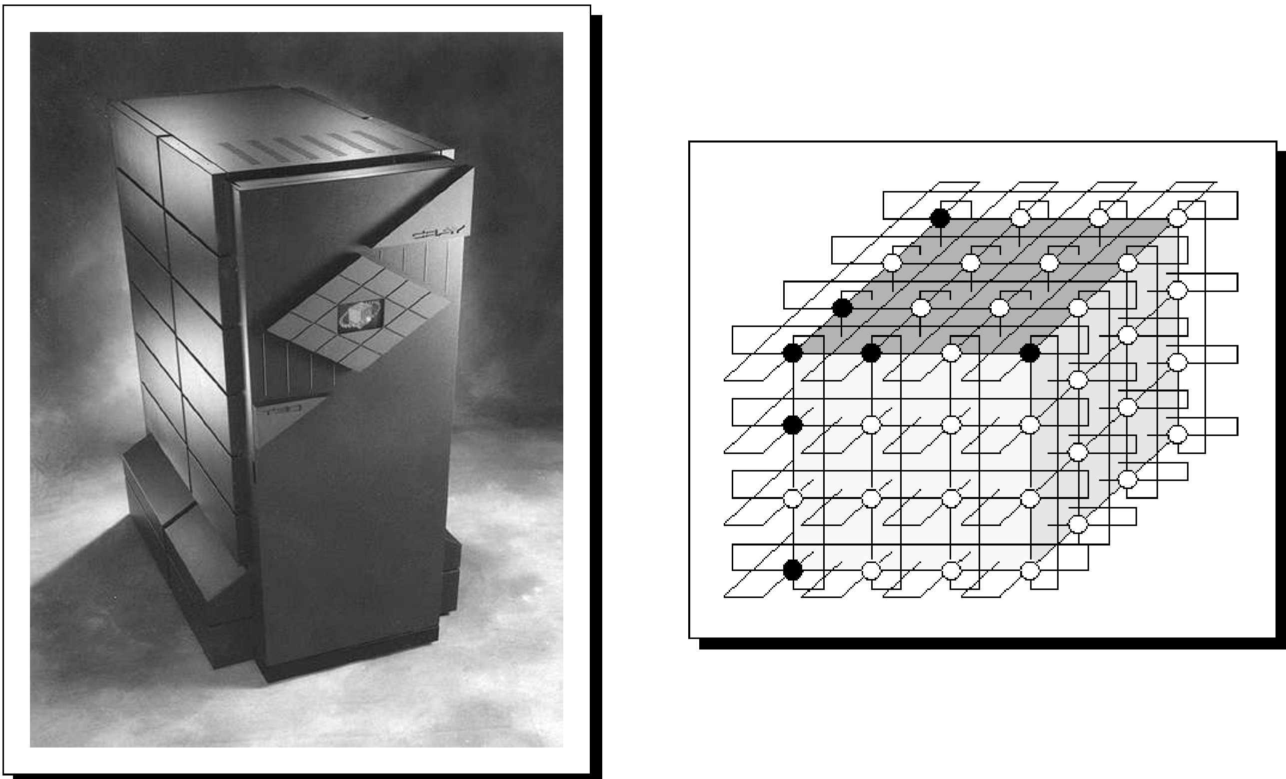


Abbildung 5.1: *Der Parallelrechner CRAY T3D SC 256 in Außenansicht (links) und aus topologischer Sicht (rechts)*

Als parallele Programmiermodelle sind PVM (*Parallel Virtual Machine*), CRAFT (*Cray Research Adaptive ForTran*) und SHMEM (*SHared MEMory functions*) nutzbar. Die SHMEM–Funktionsbibliothek ist speziell für den T3D entwickelt worden. Mit ihnen läßt sich der Speicher als global gemeinsam betrachten und direkt lesend oder schreibend auf den Speicher anderer Prozessoren zugreifen, ohne diese damit zu blockieren oder vorher eine Nachricht auszutauschen. Die SHMEM–Routinen werden als die effizienteste Methode zum Datenaustausch angesehen, erfordern allerdings größere Sorgfalt und sind programmtechnisch schwieriger zu handhaben.

5.2.2 Softwareentwurf

Bei der Entwicklung meines parallelen Isoflächenprogramms PARADISO mußten im vorhinein einige wichtige Entscheidungen bezüglich des Softwaredesigns getroffen werden, die ich im folgenden erläutern möchte. Die Entwurfsentscheidungen ergeben sich, ausgehend vom aufzustellenden

Entwurfsziel, sowohl aus den hard– und softwareseitigen Gegebenheiten, als auch aus Spezifika der Isoflächengenerierung, wie sie in den vorangegangenen Kapiteln ausführlich diskutiert wurden.

Als Zielstellung ist zu formulieren, daß ein Programm für den Parallelrechner CRAY T3D SC 256 entwickelt werden soll, welches das polygonale Isoflächenproblem für große Datensätze in geringer Laufzeit löst und dabei unter Ausnutzung der sich durch die Hardware bietenden Möglichkeiten gut skaliert. Auf weitere Details komme ich noch zu sprechen.

Auf dem CRAY T3D stehen drei Compiler zur Verfügung, nämlich für Fortran 77, Standard C und C++. C++ bietet als objektorientierte Programmiersprache natürlich die besten Möglichkeiten für eine Softwareentwicklung, ist auf dem T3D allerdings nur über einen C–Front–Compiler verfügbar, der C++–Quelltext in C–Code übersetzt und diesen dann dem Standard–C–Compiler übergibt. Nach den Erfahrungen anderer Projekte am ZIB generiert der C++–Compiler dabei leider keinen sehr effizienten C–Code. Ich entschied mich deshalb, prinzipiell in C++ zu programmieren, aber laufzeitaufwendige Routinen und I/O–Operationen als reine C–Funktionen zu implementieren.

Bei der Wahl des Programmiermodells für Datenaustausch und Kommunikation fiel meine Wahl auf die SHMEM–Bibliothek, weil diese Funktionen die beste Performance erreichen. Im Hinblick auf einen möglichst schnellen Datendurchsatz sollte, wo irgend möglich, beim Datenaustausch zwischen PEs die `shmem_put()`-Funktion statt `shmem_get()` benutzt werden. Hintergrund dafür ist, daß es nach den Erfahrungen aus anderen Anwendungen etwa zweimal schneller möglich ist, Daten vom Sender an den Empfänger mittels `shmem_put()` zu schicken, als sie vom Empfänger via `shmem_get()` lesen zu lassen. Während letzterer Funktionsaufruf erst dann zurückkehrt, wenn die Daten im lokalen Speicher des empfangenden PEs stehen, schickt `shmem_put()` die Datenpakete lediglich ab, ohne ihre erfolgte Zustellung abzuwarten. Diese Variante ist zwar deutlich effizienter, hat aber den Nachteil, daß man im Programm zum entsprechenden Zeitpunkt selbst explizit dafür Sorge tragen muß, daß die Daten wirklich zur Verfügung stehen. In der aktuellen Version[9] der SHMEM–Bibliothek reicht dies sogar bis zur manuellen Sicherstellung der Cache–Kohärenz. Es führt zu kaum nachvollziehbaren Fehlern, wenn man mittels `shmem_put()` direkt eine Speichervariable verändert und nicht dafür sorgt, daß diese Variable auch im Cache des PEs aktualisiert bzw. ungültig gemacht wird. Alles in allem verursacht die Festlegung auf die `shmem_put()`–Funktion einen hohen programmtechnischen Aufwand, der nur mit der Zielsetzung maximaler Effizienz und Ausschöpfung der hard– und softwareseitigen Möglichkeiten zu rechtfertigen ist. Die Einfachheit der Implementation ist nur ein untergeordnetes Kriterium bei diesem Entwurf.

Ein– und Ausgabeoperationen erfolgen über den Hostrechner. Der Zugriff ist bei gleicher Datenmenge deutlich schneller, je höher die Blockgröße der geschriebenen bzw. gelesenen Daten ist. Es ist unbedingt zu beachten, daß die Reihenfolge der Datenpakete bei Ein– und Ausgaben nicht notwendigerweise der Reihenfolge ihres Absendens entsprechen muß. Dies hat zur Folge, daß das (synchronisierte) Schreiben mehrerer PEs auf die gleiche Datei eine nichtdeterministische Ausgabe erzeugt. Für die Programmentwicklung hat bedeutet das, daß bei Ein– und Ausgabeoperationen auf Dateien jeder PE auf einer anderen Datei arbeiten muß.

[9] In einer zukünftigen Version soll diese unerwünschte Eigenschaft nach Aussagen der Firma CRAY nicht mehr enthalten sein.

Da die Isoflächengenerierung zu einem Zeitpunkt mit eng lokal begrenzten Informationen auskommt, ergibt sich das Grundprinzip einer datenparallelen Vorgehensweise durch Verteilung der Daten auf die PEs nahezu von selbst. Die grundsätzliche Schwierigkeit, die mit jedem parallelen Ansatz einhergeht, ist das Erreichen einer gleichmäßigen Lastverteilung (*load balance*). Im Kontext der Isoflächengenerierung ergibt sich das Problem, daß nicht vorhergesagt werden kann, in welchem Bereich des Gesamtvolumens Teile der Isofläche auftauchen und in welchem nicht. Versuche, bei ungleicher Lastverteilung die Aufgaben umzuverteilen, werden unter dem Begriff *dynamic load balancing* zusammengefaßt. Eine dynamische Lastverteilung setzt aber voraus, daß aus der aktuellen Lastverteilung die zukünftige abgeschätzt werden kann. Bei der Isoflächengenerierung lassen sich aber aus der Betrachtung der aktuellen Zelle nur Aussagen über diejenigen direkt benachbarten Zellen machen, die mit der aktuellen Zelle eine solche Kante gemeinsam haben, auf der ein Schnittpunkt liegt. Berücksichtigt man, daß die Zeit zur Bearbeitung einer Zelle sehr gering ist, kann man schlußfolgern, daß die Lastverteilung *statisch* erreicht werden muß. Die geeignete Verteilung der Daten auf die Prozessoren ergibt sich direkt aus dem Vorgehen bei der Zellpolygonalisierung und der Latenz beim Zugriff auf entfernten Speicher: Es sollten möglichst viele Gitterzellen benachbart sein, damit doppelte Berechnungen auf gleichen Kanten vermieden werden und die für die Normalenberechnung relevanten Punkte im lokalen Speicher stehen. Anschaulich bedeutet dies, daß das Subvolumen bei fester Gitterzellenanzahl eine möglichst geringe Außenfläche haben sollte. Die ideale Zerlegung ist also eine dreidimensionale Gebietszerlegung in Würfel bzw. in zusammenhängenden Teilvolumina mit möglichst gleicher Ausdehnung in jeder der drei Richtungen. Unter Berücksichtigung der Normalenberechnung müssen sich zwei benachbarte Teilvolumina in drei Gitterpunktschichten überlappen (siehe Abb. 5.2).

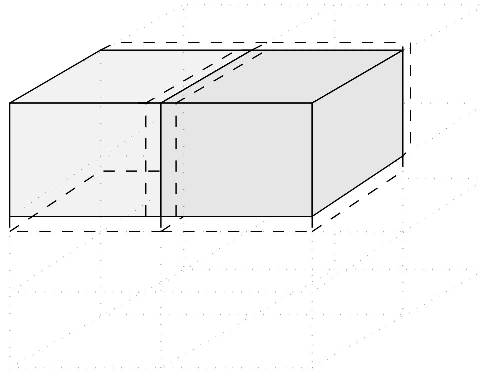


Abbildung 5.2: *Dreidimensionale Gebietszerlegung in möglichst gleichseitige Quader mit Überlappung*

Das Zellpolygonalisierungsverfahren für die Untervolumina sollte möglichst schnell und lokal arbeiten und eine C^0–stetige Fläche aus wenigen Dreiecken erzeugen. Dies leistet der *Marching Cubes*–Algorithmus mit modifizierter Lookup–Tabelle (siehe Kap. 3.3.1), dessen Laufzeit sich durch Anwendung geeigneter Feinheiten aus Abschnitt 3.7 noch senken läßt.

5.2.3 Implementation

Die Entwurfsentscheidungen aus dem vorangegangenen Abschnitt bestimmen das prinzipielle Vorgehen des Programms:

1. 3D–Zerlegung des Gitters
2. Einlesen und Verteilen der Felddaten
3. Randaustausch der Teilvolumina
4. Isoflächenberechnung für die Teilvolumina
5. Ausgabe der Teilflächen
6. Zusammenfügen zur Gesamtfläche
7. Darstellung der Gesamtfläche

Die dreidimensionale Gebietszerlegung erfolgt auf Grundlage der Gittergröße und der Anzahl der PEs. Die Gitterdimensionen werden aus einer separaten ASCII–Datei oder einem Vorspann der Gesamtrohdaten gelesen. Auf dieser Grundlage wird das Gitter in n möglichst würfelähnliche Teilvolumina zerteilt (siehe Abb. 5.2), wenn n die Anzahl der Prozessoren ist.

Die Felddaten liegen als Binärfile vor, in dem die Daten mit aufsteigenden Koordinaten (x wächst am schnellsten) gespeichert sind. Das Einlesen der Daten erfolgt durch den PE mit der Nummer $n-1$. Es soll die maximale Blocklänge gelesen werden, die im Ganzen an einen PE weitergereicht werden können. Die Ordnung der Rohdaten impliziert, daß dieser Block die Größe eines Teilvolumens in x–Richtung haben muß. Der Block wird an eines der Teilvolumina (d. h. den PE, der dieses speichert) gesendet, das diese *Zeile* gemäß der Zerlegung enthalten muß. Eine Zeile, die zu zwei (oder vier) Teilvolumina gehört, wird dem Teilvolumen zugeordnet, das am nächsten zum Gitterursprung liegt.

Im dritten Schritt, dem Randaustausch sendet jeder PE mittels `shmem_put()` bestimmte Teile seiner Ränder an andere PEs, was für die notwendigen Überlappungen sorgt. Sind Ränder der Teilvolumina auch Ränder des Gesamtvolumens, so wird eine Überlappung künstlich so berechnet, daß die Berechnung der Normalen auf Grundlage dieser Überlappung das gleiche Ergebnis bringt, als wenn man den Randfall bei der Normalenberechnung separat behandelt und in Gleichung 2.4 auf Seite 10 jeweils der Term „$+\delta$“ streicht [Mac92]. Diesen gesamten Schritt nur mit `shmem_put()` zu implementieren, erforderte eine sehr genaue manuelle Untersuchung der auftretenden Fälle und erscheint im Quellcode recht unübersichtlich. Ich bin davon überzeugt, daß dies trotz fehlender optischer Eleganz die schnellste Methode für einen Randaustausch darstellt.

Nach einer Synchronisation, die dafür sorgt, daß die Daten auf jedem PE wirklich im Speicher stehen, führt jeder PE die Isoflächenberechnung (*Marching Cubes*–Algorithmus mit modifizierter Lookup–Tabelle) auf seinem Teilvolumen durch und profitiert davon, daß Randfälle nicht extra betrachtet werden müssen. Doppelte Berechnungen werden weitestgehend durch Merken der letzten Schnittpunkte und 8bit–Indizes (siehe Kap. 3.7) vermieden.

Jeder PE schreibt seine Repräsentation der Teilfläche (Listen von Indizes, Koordinaten bzw. Normalen) in separate Dateien. Vorher tauschen sie die Information aus, wieviele Dreiecke sie erzeugt haben, so daß die Indizes, beginnend bei PE 0, insgesamt fortlaufend numeriert werden können.

Das Zusammenfügen zu einer Gesamtdatei kann dann einfach durch Aneinanderhängen der binären Dateien in einer festen Reihenfolge erfolgen. Die Darstellung der Polygone erfolgt derzeit auf einer Grafikwirkstation. Dazu habe ich einen Programm auf Basis der *Open Inventor*™–Bibliothek geschrieben, das die Binärdatei liest, interpretiert und die entsprechenden Polygone als 3D–Objekte darstellt, die interaktiv bewegt werden können.

5.2.4 Ergebnisse und Ausblick

Das Programm wurde mit Daten aus den verschiedensten Anwendungsbereichen getestet. Ein wichtiger Anwendungsfall betrifft Skalarfelder aus der *Computational Chemistry*. Skalarwerte sind hier zumeist elektrostatische Potentiale. Die farbigen Abbildungen auf Seite 71 und 75 zeigen die mit PARADISO berechneten Isoflächen für verschiedene Moleküle, jeweils mit Gittergröße $101 \times 101 \times 101$. Bild 1 zeigt die Isofläche für ein Benzen–Molekül mit Isowert -0.1, wobei hier die Dreieckskanten eingezeichnet wurden, um die mittels GORAUD–Shading dargestellte Fläche und das zugehörige Drahtmodell einmal direkt vergleichen zu können. Bild 2 liegen die gleichen Daten zugrunde, allerdings wurden hier andere Isowerte gewählt. Die Anzahl der Dreiecke liegt zwischen 2400 (gelbe Fläche) und 8400 (blaue und rötliche Fläche). Bild 6 zeigt die Isoflächen für ein C_{60}–Molekül (*buckyball*) mit entgegengesetzten Isowerten (0.01 und -0.01).

Ich führte mehrere Laufzeitvergleiche für kleine, mittlere und große Datensätze bei einer Mindestanzahl von 8 Prozessoren durch. Dabei konnte ich feststellen, daß das Programm erwartungsgemäß für den kleinen Datensatz ($20 \times 20 \times 20$, siehe Abb. 4.2 auf Seite 51) bei höherer Prozessoranzahl schlechtere Ergebnisse liefert, weil der Datensatz dann nicht mehr sinnvoll aufgeteilt werden kann. Die Laufzeit lag im Bereich von 0.3 Sekunden (8 PEs), wobei die Berechnung mit maximal 0.01 Sekunden zu Buche schlägt. Beim mittleren Datensatz (Benzen–Molekül, $101 \times 101 \times 101$, siehe Farbbilder 1 und 2) lag die Gesamtlaufzeit beim Isowert 0.2 im Bereich von 1.6 bis 1.8 Sekunden, nahezu unabhängig davon, ob 8, 16 oder 32 Prozessoren verwendet werden. Die Dauer des Berechnungsschrittes sinkt dabei etwa linear mit der Prozessoranzahl, die Zeit für die Ausgabe verhält sich genau umgekehrt und die Zeit für das Einlesen und Verteilen des Datensatzes ist etwa konstant. Bei größeren Datensätzen (z. B. CT–Daten $512 \times 512 \times 178$) ließ sich mit steigender Prozessoranzahl auch eine höhere Gesamtgeschwindigkeit erreichen. Bereits bei 16 PEs konnte eine Gesamtlaufzeit von knapp über einer Minute (Isowert -200, siehe auch Farbbild 3) erreicht werden. Der Berechnungsschritt nahm dabei nur einen Anteil von 7–11 Sekunden ein. Gemessen an der Größe des Datensatzes sind dies hervorragende Werte, wenn man sich auch noch vor Augen hält, daß über 1.3 Millionen Dreiecke erzeugt werden. Den Löwenanteil an der Laufzeit nimmt das Einlesen des Datensatzes (182 MB) in Anspruch.

Die Ergebnisse zeigen, daß PARADISO ab einer gewissen Problemgröße eine deutlich schnellere Isoflächenberechnung ermöglicht, als sich mit sequentiellen Verfahren ereichen läßt. Der größte Flaschenhals bei der Isoflächengenerierung ist dabei das Einlesen der teilweise riesigen Datenmengen von der Platte. Dies läßt erwarten, daß bei Kopplung des Programms an eine numerische Simulation auf dem CRAY T3D noch bessere Ergebnisse als die hier vorgestellten erreichbar sind.

Die schnelle Generierung polygonaler Isoflächenapproximationen aus großen Datenmengen verlangt natürlich auch nach adäquaten Darstellungsmethoden. Der versuchsweise praktizierte An-

satz, die Polygone zu einer Grafikworkstation zu übertragen und dort darzustellen, ist nur für mittlere Problemstellungen tauglich, verbietet sich jedoch ab einer gewissen Anzahl von Dreiecken. Hier drängt es sich auf, auch die Darstellung durch den Parallelrechner vornehmen zu lassen. Ich hätte gern einen parallelen Polygonrenderer verwendet, nur leider ist solch ein Programm für den T3D bisher nicht verfügbar. CRAY Research hatte eine parallele Version des *zorba*–Renderers mit einer neuen Version der Visualisierungssoftware *Cray Animation Theater (CAT)*[10] für März 1996 angekündigt, aber die Entwicklung dann leider kurzfristig eingestellt. Ein in den USA für den T3D entwickelter paralleler Polygonrenderer darf leider wegen strenger Exportbestimmungen nicht frei zur Verfügung gestellt werden.

Ein paralleler Dreiecksreduktionsalgorithmus wäre wünschenswert, um die polygonalen Flächen bei Bedarf gleich „ vor Ort" zu vereinfachen. Für sehr große Datensätze sollte die parallele Implementation eines *Dividing Cubes*–Ansatzes erwogen werden.

Die Entwicklung paralleler Verfahren, die Isoflächen erzeugen, sie verarbeiten oder darstellen, scheint erst am Anfang zu stehen. Hier ergibt sich ein Ansatzpunkt für weitere Arbeiten und eine spannende Aufgabe für die Zukunft.

[10]Eine von mir erstellte deutschsprachige Beschreibung zu CAT 1.0 findet sich im WWW unter `http://www.ZIB-Berlin.DE/German/Vektor/ZIBdoc/Anwendungssoftware/Grafik_und_Visualisierung/cat.html`

Kapitel 6

Zusammenfassung

In dieser Diplomarbeit wurden Verfahren zur Erzeugung von Isoflächen aus dreidimensionalen Skalarfeldern beschrieben.

Wegen der Unvollständigkeit von diskreten 3D–Skalarfeldern kann kein Isoflächenalgorithmus garantieren, daß die generierte polygonale Approximation topologisch exakt der Isofläche der zugrundeliegenden Funktion entspricht. Es wurde jedoch gezeigt, daß es möglich ist, *topologisch konsistente* Isoflächen, basierend auf einem trilinearen Modell, zu erzeugen. Bei effizienteren Algorithmen, insbesondere solche mit integrierter Polygonreduktion, nimmt man immer in Kauf, daß die erzeugte Approximation, abhängig von den Daten, zwar C^0-stetig ist, aber trotzdem Ungenauigkeiten oder gar topologische Fehler enthält. Will man bei gleichen Daten Isoflächen mit verschiedenen Isowerten berechnen, versprechen hierarchische Datenstrukturen (Octrees) bei vertretbarem Initialisierungsaufwand einen deutlichen Zeitgewinn. Sie lassen sich für nahezu jeden polygonalen Isoflächenalgorithmus verwenden.

Eine Real–Time–Darstellung fein aufgelöster, zeitlich veränderlicher Skalarfelder, etwa um in eine laufende Simulation interaktiv eingreifen zu können, ist mit den beschriebenen Verfahren auf Workstations nicht zu bewältigen. Es wurde ein im Rahmen dieser Arbeit entwickelter paralleler Isoflächengenerator (PARADISO) für den Parallelrechner CRAY T3D SC 256 vorgestellt, der die Volumendaten zerlegt, auf die einzelnen Prozessoren verteilt und für den jeweiligen Ausschnitt des Skalarfeldes unter Berücksichtigung der Randbedingungen den entsprechenden Teil der Isofläche berechnet. Zur grafischen Darstellung sind dann nur die Isoflächendaten, ein Bruchteil der umfangreichen Originaldaten, zur Grafikworkstation zu übertragen und dort zu rendern.

Danksagung

An dieser Stelle möchte ich mich bei *allen* bedanken, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben. Zuallererst geht mein Dank an Dipl.–Phys. Detlev Stalling und Dr. sc. Ernst–Günter Giessmann für die Betreuung und ihre Anregungen. Besonderer Dank gilt den Kollegen in den Abteilungen HLR[1] und VisPar[2] des ZIB für die hervorragende Arbeitsatmosphäre und viele kleine und große Ratschläge, insbesondere bei der Erstellung der Software. Für die den Bildern zugrundeliegenden Datensätze bedanke ich mich bei Hugues Hoppe (Microsoft Corporation) [Abb. 4.2, 4.3], Dr. Thomas Steinke, Detlev Stalling (ZIB) und dem Rudolf–Virchow–Klinikum Berlin. Ein großes Dankeschön geht an Henrik Battke, der mir bei der Erstellung des Einbands und der farbigen Abbildungen selbstlos zur Seite stand.

Persönlich möchte ich meiner Lebensgefährtin Katrin für ihre Liebe und ihr Verständnis und meinen Eltern für die bereitwillige Unterstützung meines Studiums danken.

[1] Bereich Rechenzentrum, Abt. Höchstleistungsrechner, Abteilungsleiter Hubert Busch
[2] Visualisierung und Paralleles Rechnen, Abteilungsleiter Hans–Christian Hege

Literaturverzeichnis

[Bat96] Henrik Battke. Entwicklung texturbasierter Verfahren zur Vektorfeldvisualisierung. Diplomarbeit, Humboldt-Universität zu Berlin, Fachbereich Informatik, März 1996.

[Bre] Thomas Brenner. Parallel Volume Rendering — Ein Projektansatz für den CRAY T3D. Diplomarbeit, Humboldt–Universität zu Berlin, Fachbereich Informatik. Erscheint demnächst.

[BS89] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Thun, Frankfurt/Main, 24. Auflage, 1989.

[CCJ93] Patricia J. Crossno, Douglas D. Cline, and Jeffrey N. Jortney. A Heterogeneous Graphics Procedure for Visualization of Massively Parallel Solutions. In *CFD Algoritms and Applications for Parallel Processors (Proceedings of ASME Fluids Engineering Conference)*, pages 65–70, 1993.

[Dü88] M. J. Dürst. Letters: Additional Reference to “Marching Cubes”. *ACM Computer Graphics*, 22(4):72–73, 1988.

[Ell95] Peter Ellsiepen. Parallel Isosurfacing in Large Unstrucured Datasets. In M. Göbel and B. Urban, editors, *Visualization in Scientific Computing (Proceedings of 5th EUROGRAPHICS Workshop on Visualization in Scientific Computing)*, pages 9–23, Wien, 1995. Springer Eurographics.

[FG91] M. Frühauf und M. Göbel (Hrsg.). *Visualisierung von Volumendaten*. Springer–Verlag Berlin, 1991.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1990.

[HB94] C. T. Howie and E. H. Blake. The Mesh Propagation Algorithm for Isosurface Construction. *Computer Graphics Forum*, 13(3):C–65–C–74, 1994. Eurographics ’94 Conference issue.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH ’93 Proceedings)*, volume 27, pages 19–26, August 1993. Also as Technical Report TR 93–01–01, University of Washington, Seattle, USA. Anonymous FTP: /tr/1993/01/UW-CSE-93-01-01.PS.Z on cs.washington.edu.

[HH92] C.D. Hansen and P. Hinker. Massively Parallel Isosurface Extraction. In A.E. Kaufman and G.M. Nielson, editors, *Visualization '92 Proceedings*, pages 77–83. IEEE Computer Society Press, 1992. WWW: http://www.acl.lanl.gov/Viz/vis92_paper.ps.

[(Hr93] Hans-Christian Hege (Hrsg.). Handbuch zur Visualisierung am ZIB. Technical Report TR–92–7, Konrad–Zuse–Zentrum für Informationstechnik Berlin, Dezember 1993.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[Lor94] Bill Lorensen. Extracting Surfaces from Medical Volumes. In *Visualizaion '94 Course Notes: Volume Visualization Algorithms and Applications*, pages 26–45, 1994.

[Mac92] Paul Mackerras. A Fast Parallel Marching–Cubes Implementation on the Fujitsu AP1000. Technical Report TR-CS-92-10, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, August 1992. WWW: http://cs.anu.edu.au/techreports/TR-CS-92-10.ps.gz.

[Mat94] Sergey V. Matveyev. Approximation of Isosurface in the Marching Cube: Ambiguity problem. In R.D. Bergeron and A.E. Kaufman, editors, *Visualization '94 Proceedings*, pages 288–292. IEEE Computer Society Press, 1994.

[Mei90] Gert-Andreas Meißner. *Der Einsatz schneller Visualisierungsmethoden für den grafisch–interaktiven Entwurfsprozeß*. Doktorarbeit, Technische Universität Berlin, Fachbereich für Verkehrswesen, 1990.

[MG93] Tom Meyer and Al Globus. Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments. Technical Report CS–93–54, Brown University, Department of Computer Science, P.O. Box 1910, Providence RI 02912, USA, December 1993. Anonymous FTP: /pub/techreports/93/cs93-54.ps.Z on ftp.cs.brown.edu.

[MS93] H. Muller and M. Stark. Adaptive Generation of Surfaces in Volume Data. *The Visual Computer*, 9(4):182–199, 1993.

[MS94] C. Montani and R. Scopigno. Using Marching Cubes on small machines. *CVIGP: Graphical Models and Image Processing*, 56(2):182–183, March 1994.

[MSS94a] C. Montani, R. Scateni, and R. Scopigno. Discretized Marching Cubes. In R.D. Bergeron and A.E. Kaufman, editors, *Visualization '94 Proceedings*, pages 281–287. IEEE Computer Society Press, 1994. WWW: http://miles.cnuce.cnr.it/cg/papers/DiscMCACM.ps.Z.

[MSS94b] C. Montani, R. Scateni, and R. Scopigno. A modified look-up table for implicit disambiguation of Marching Cubes. *The Visual Computer*, 10(6):353–355, 1994.

[MW91] D. Moore and J. Warren. Mesh Displacement: An Improved Contouring Method for Trivariate Data. Technical Report TR91–166, Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, USA, September 1991.

[MW92] D. Moore and J. Warren. Compact Isocontours from Sampled Data. In D. Kirk, editor, *Graphics Gems III*, pages 23–28. Academic Press, 1992. Short version of [MW91].

[Nat94] B. K. Natarajan. On generating topologically consistent isosurfaces from uniform samples. *Visual Computer*, 11(1):52–62, 1994.

[NB93] Paul Ning and Jules Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, November 1993.

[NH91] Gregory M. Nielson and Bernd Hamann. The Asymptotic Decider: Removing the Ambiguity in Marching Cubes. In *Visualization '91*, pages 83–91, 1991.

[NH92] P. Ning and L. Hesselink. Octree Pruning for Variable–Resolution Isosurfaces. In *Visual Computing (Proceedings of CG International '92)*, Tokyo, 1992. Springer–Verlag.

[RHvK95] Stefan Röll, Axel Haase, and Markus von Kienlin. Fast Generation of Leakproof Surfaces from Well-Defined Objects by a Modified Marching Cubes Algorithm. *Computer Graphics forum*, 14(2):127–138, 1995.

[SH95a] D. Stalling and H.C. Hege. Design and Implementation of a Hyperthermia Planning System. In B. Arnolds, H. Müller, T. Tolxdorff, and D. Saupe, editors, *Proceedings zum Workshop „Digitale Bildverarbeitung in der Medizin"*, März 1995.

[SH95b] Detlev Stalling and Hans-Christian Hege. Fast and Resolution Independent Line Integral Convolution. Preprint SC–94–37, Konrad–Zuse–Zentrum für Informationstechnik Berlin, June 1995.

[SZK95] Renben Shu, Chen Zhou, and Mohan S. Kankanhalli. Adaptive marching cubes. *The Visual Computer*, 11(4):202–217, 1995.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[Tha90] Daniel Thalmann, editor. *Scientific Visualization and Graphics Simulation*. John Wiley & Sons Ltd., Chichester, West Sussex, England, 1990.

[VGW94] Allen Van Gelder and Jane Wilhelms. Topological Considerations in Isosurface Generation. *ACM Transactions on Graphics*, 13(4):337–375, October 1994. Also as Technical Report UCSC–CRL–94–31, University of California, Santa Cruz, USA. Anonymous FTP: /pub/tr/ucsc-crl-94-31.ps.Z on ftp.cse.ucsc.edu. See corrigendum: [VGW95].

[VGW95] Allen Van Gelder and Jane Wilhelms. Corrigendum: "Topological Considerations in Isosurface Generation". *ACM Transactions on Graphics*, 14(3):307–308, July 1995. See [VGW94].

[vW91] Jarke J. van Wijk. Spot noise, texture synthesis for data visualization. *Computer Graphics*, 25(4):309–318, July 1991.

[Wal91] Ake Wallin. Constructing Isosurfaces from CT Data. *IEEE Computer Graphics and Applications*, 11(6):28–33, November 1991.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, August 1986.

[WVG90a] Jane Wilhelms and Allen Van Gelder. Octrees for Faster Isosurface Generation Extended Abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 57–62, November 1990. Also as Technical Report UCSC–CRL–90–28, University of California, Santa Cruz, USA. Preliminary version of [WVG92].

[WVG90b] Jane Wilhelms and Allen Van Gelder. Topological Considerations in Isosurface Generation Extended Abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 79–86, November 1990. Also as Technical Report UCSC–CRL–90–14, University of California, Santa Cruz, USA. Preliminary version of [VGW94].

[WVG92] Jane Wilhelms and Allen Van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[YC94] Terry S. Yoo and David T. Chen. Interactive 3D Medical Visualization: A Parallel Approach to Surface Rendering 3D Medical Data. In Johannes M. Boehme, Alan H. Rowberg, and Neil T. Wolfman, editors, *Computer Applications to Assist Radiology (Proceedings of S/CAR 94, Symposium for Computer Assisted Radiology)*, pages 100–105. Symposia Foundation, 1994. Anonymous FTP: /pub/technical-reports/94-059.ps.Z on ftp.cs.unc.edu.

Thesen zur Diplomarbeit

1. Die Darstellung von Isoflächen trägt wesentlich zum Verständnis der in den zugrundeliegenden 3D–Daten enthaltenen Informationen bei.

2. Die Beschäftigung mit anderen Herangehensweisen an ein Problem macht die Resultate eigener Arbeit vergleichbar und erleichtert das Finden bemerkenswerter Lösungen.

3. Veröffentlichungen, die es sich zum Ziel setzen, alle wichtigen Ansätze zur Generierung von Isoflächen aus diskreten Skalarfeldern zu diskutieren, sind in der Fachwelt bisher nicht bekannt.

4. Der originale *Marching Cubes*–Algorithmus ist, obwohl noch weit verbreitet, ein zwar schneller, aber nicht empfehlenswerter Isoflächenalgorithmus.

5. In Kapitel 3.3 wird gezeigt, wie sich der *Marching Cubes*–Algorithmus dahingehend korrigieren, daß er ohne deutlichen Laufzeitzuwachs C^0–stetige Isoflächen erzeugt. Die Erfahrungen auf Grundlage der vorgestellten eigenen Implementation verdeutlichen, daß ein solches Vorgehen für kleine und mittlere Datensätze gute Ergebnisse verspricht.

6. Es gibt keinen „idealen" Isoflächenalgorithmus, der aus einem beliebigen diskreten 3D–Skalarfeld eine topologisch exakte polygonale Approximation der Isofläche der zugrundeliegenden Funktion erzeugt.

7. Es existieren jedoch *topologisch konsistente* Isoflächenalgorithmen, die eine topologisch exakte polygonale Approximation der Isofläche *eines Modells* der dem Skalarfeld zugrundeliegenden Funktion berechnen.

8. Die Speicherung eines Skalarfeldes als *Octree* erhöht die Effizienz der Isoflächengenerierung bei mehrfacher Benutzung derselben Daten deutlich.

9. Polygonreduzierung läßt sich im allgemeinen nur auf Kosten von Laufzeitzuwachs und größeren Ungenauigkeiten erreichen, ist für einige Anwendungen aber trotzdem essentiell. Der *Compact Cubes*–Algorithmus ist ein geeigneter Kompromiß zwischen möglichst niedriger Polygonanzahl, genauer Approximation und geringer Laufzeit.

10. Das Problem der Isoflächenberechnung birgt ein enormes Parallelisierungspotential. Parallele Versionen spezieller Algorithmen sind sinnvoll und erschließen bei Nutzung adäquater Hardware neue Anwendungsfelder. Das in Kapitel 5.2 vorgestellte Programm PARADISO stellt die Tauglichkeit eines parallelen Ansatzes unter Beweis.

11. Die Leistungsfähigkeit moderner Isoflächenalgorithmen bleibt ein Forschungsthema und sollte weiter in einem größeren Rahmen untersucht werden.