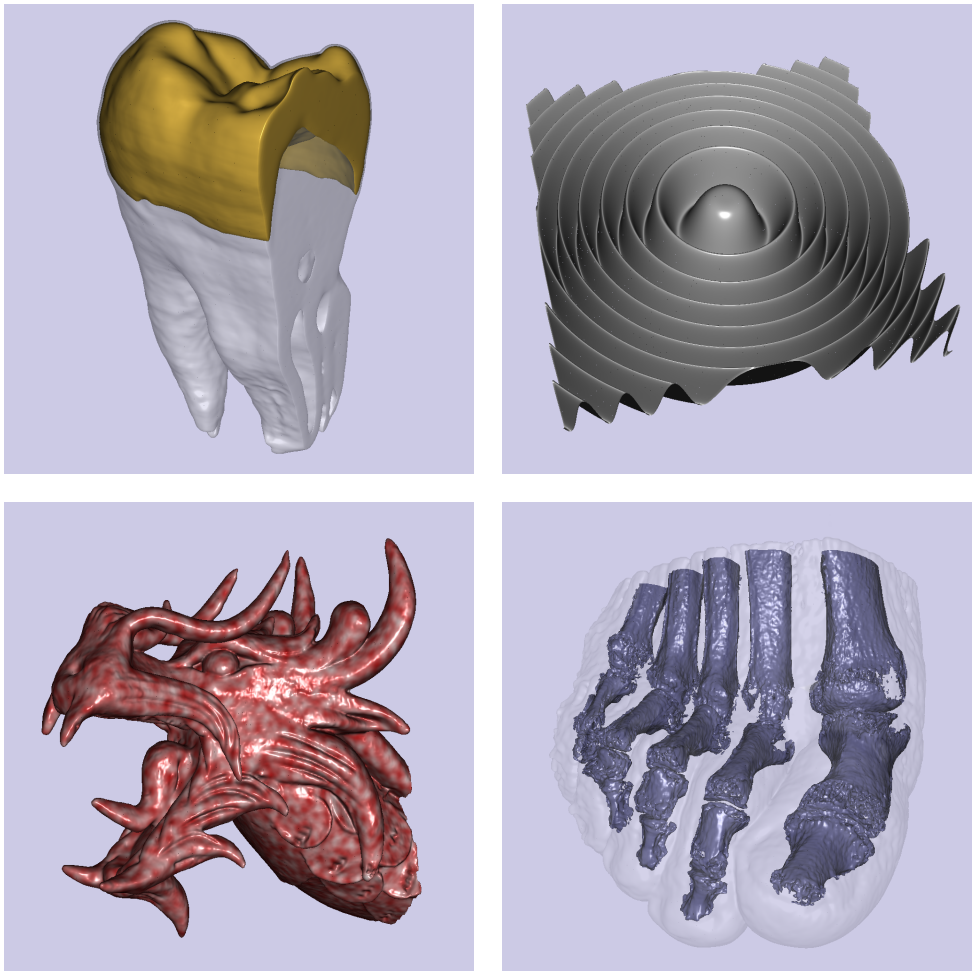
Interaktive Visualisierung variierender Isoflächen mit trivariaten Splines auf massiv parallelen Prozessoren

Diplomarbeit

Thomas Koch

TECHNISCHE
UNIVERSITÄT
DARMSTADT



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker (TU)

Interaktive Visualisierung variierender Isoflächen mit trivariaten Splines auf massiv parallelen Prozessoren

Thomas Koch
April 2009

betreut von
Herrn Dipl.-Inform. Thomas Kalbe
und
Herrn Prof. Dr.-Ing. Michael Goesele

Thomas Koch
Matrikelnummer: 965763
Studiengang: Diplom Informatik
e-mail: ThomasDidiKoch@gmx.net

Diplomarbeit
Thema: Interaktive Visualisierung variierender Isoflächen
mit trivariaten Splines auf massiv parallelen Prozessoren

Cover:
oben links: Tooth $256 \times 256 \times 161$ (Blending kubischer C^1-Splines)
oben rechts: Marschnerlobb-Testfunktion 256^3 (quadratische Super-Splines)
unten links: Asian Dragon (SDF) 256^3 (kubische C^1-Splines und prozedurale Textur)
unten rechts: Foot 256^3 (Blending quadratischer Super-Splines)
Tooth und Foot wurden vor der Rekonstruktion mit einem Gauß-Filter geglättet.

Eingereicht: 30.April 2009

Betreuer: Dipl.-Inform. Thomas Kalbe

Prof. Dr.-Ing. Michael Goesele
Graphisch-Interaktive Systeme (GRIS)
Technische Universität Darmstadt
Fraunhoferstr. 5
64283 Darmstadt

Kurzfassung

Die Visualisierung von skalaren Volumendaten, wie sie beispielsweise bei CT- oder MRT-Scans generiert werden, ist Thema aktueller Forschungen. Bei den Daten handelt es sich um Dichtewerte auf einem dreidimensionalen regulären Gitter. Unterschieden wird zwischen der direkten Volumenvisualisierung und der Darstellung sogenannter Isoflächen. Anwendung findet die Volumenvisualisierung im Bereich der modernen medizinischen Diagnostik und der industriellen Qualitätskontrolle.

In dieser Diplomarbeit wird untersucht, inwieweit moderne Grafikkarten mit ihren massiv parallelen Prozessoren in der Lage sind, interaktive Rekonstruktionen von qualitativ hochwertigen Isoflächen zu liefern. Dabei wird das gesamte Volumen mittels trivariater Splines in Bernstein-Bézier-Form beschrieben. Alle Isoflächen liegen somit implizit vor und werden mit einem präzisen Ray-Casting-Verfahren interaktiv visualisiert. Die verwendeten Splines verfügen über garantierte Glattheitsbedingungen bei niedrigem Grad, um eine hochqualitative sowie effiziente Visualisierung gegenüber Standardverfahren zu erreichen. Der vorgestellte parallele GPU-Algorithmus nutzt die Bernstein-Bézier-Form, Instancing-Techniken und neueste Shader-4.0-Funktionalitäten, um die Rekonstruktionszeiten gegenüber einer optimierten CPU-Version bis zu 68-fach zu beschleunigen. Zusätzlich wird der benötigte Speicher deutlich reduziert und die Bildrate erhöht. Die Zeitmessungen einer kombinierten CUDA-OpenGL-Implementierung belegen, dass ein interaktives Variieren der Isowerte auch bei großen Datensätzen durch Auslagerung aller Berechnungsschritte auf die Grafikkarte möglich ist. Folglich ist der gezeigte Algorithmus eine signifikante Verbesserung der praktischen Anwendung von trivariaten Splines für die Echtzeitvisualisierung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Standard-Grafikpipeline nutzt Visualisierungsprimitive niedriger Ordnung (Punkte, Linien, Dreiecke), um Geometrien darzustellen. Die flexible Programmierbarkeit aktueller Grafikprozessoren (GPUs) erlaubt die Verwendung von Primitiven höherer Ordnung. Die Visualisierung der Isoflächen in dieser Arbeit basiert auf volumetrischen Primitiven in Tetraederform, auf denen Polynomfunktionen (mit drei Veränderlichen) von totalem Grad 2 bzw. 3 definiert sind. Setzt man diese Tetraeder zu einem kontinuierlichen Volumen zusammen und wählt die Polynomkoeffizienten so, dass stetige Differenzierbarkeit zu einem bestimmten Grad eingehalten wird, spricht man von trivariaten Splines. Trivariate Splines sind ideal für die hochwertige Rekonstruktion von Dichtefunktionen, basierend auf diskreten Volumendaten. Solche Volumendaten werden in der modernen Medizin für die Diagnostik und in der Industrie für die Qualitätskontrolle mittels CT- oder MRT-Scans generiert.

Mit der Darstellung von Isoflächen lassen sich wichtige Informationen von Volumendaten einsehen. Durch eine Spline-Rekonstruktion der skalaren Dichtefunktion liegen implizit sämtliche Isoflächen des Volumens vor. Die trivariaten Splines werden entlang von Strahlen auf univariate Splines des gleichen, niedrigen Grades reduziert. Dadurch kann der Schnitt eines Strahls mit der Isofläche exakt und effizient bestimmt werden. Die Vertex- und Fragmentshader-Programme visualisieren in dieser Arbeit die Isoflächen mit einem präzisen Ray-Casting-Verfahren. Anders als bei üblichen Standardverfahren garantiert die Spline-Rekonstruktion C^1-stetiges Verhalten im gesamten Volumen. So werden glatte Silhouetten und hochwertige Lichtberechnungen der Isoflächen erzielt. Abbildung 1.1 zeigt, dass höherwertige Elemente, im Gegensatz zu linearen Elementen (Dreiecke), eine höhere Approximationsordnung besitzen und damit die günstigere Wahl sind, um glatte Formen zu beschreiben.

1.1 Ziel dieser Arbeit

Der in dieser Arbeit vorgestellte Algorithmus verlagert alle Berechnungen zur Visualisierung von qualitativ hochwertigen Isoflächen auf die Grafikkarte. Aktuelle Grafikchips mit massiv parallelen Prozessoren verfügen über deutlich mehr Rechenleistung als derzeit verfügbare 4- oder 8-Kern CPUs. Folglich wird untersucht, ob hochwertige Rekonstruktionen und Visualisierungen durch Parallelisieren sämtlicher Berechnungsschritte in interaktiven Zeiten auch für große Datensätze möglich sind.

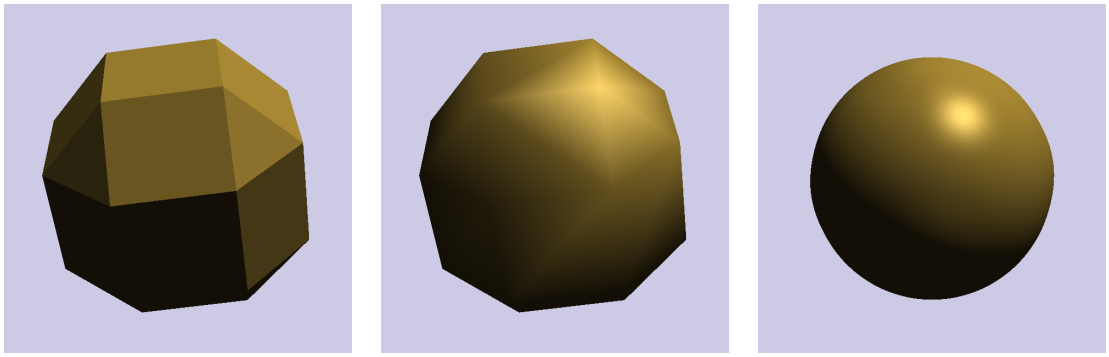


Abbildung 1.1: Bei diesem Extrembeispiel für sehr kleine Datensätze wird der Unterschied zwischen einem Standardverfahren (Marching-Cubes) und einer qualitativ hochwertigen Spline-Rekonstruktion (kubische C^1-Splines) deutlich. Alle drei Abbildungen wurden aus demselben 4^3 Volumendatensatz, der eine implizite Kugelfunktion abtastet, erstellt. *Links:* Marching Cubes Verfahren mit Flat-Shading. *Mitte:* Marching Cubes Verfahren mit Gouraud-Shading. Die Normalen wurden durch zentrale Differenzen approximiert. *Rechts:* Die in dieser Arbeit verwendeten kubischen C^1-Splines.

Interaktive Anwendungen benötigen meist zeitintensive Vorberechnungen, um Datenstrukturen zu erstellen, nicht relevante Elemente auszusortieren und *Level of Detail*-Entscheidungen zu treffen. Nur durch diesen Präprozess werden die benötigten hohen Bildraten erzielt. Verfahren zur interaktiven Visualisierung von Isoflächen, rekonstruiert aus großen 3D-Datensätzen, müssen Vorberechnungen dieser Art bei jedem Wechsel des eingestellten Isowertes durchführen. Auch die verwendeten Splines benötigen solche Vorberechnungen, um hohe Bildraten zu gewährleisten. Entscheidend ist hierbei das schnelle Bestimmen der Tetraederelemente, die Teile der Isofläche enthalten. Ein Ziel dieser Arbeit ist, herauszufinden, inwieweit diese Vorberechnungen parallelisiert werden können, um interaktive Rekonstruktionen für variierende Isowerte zu ermöglichen.

Viele Standardverfahren zur Darstellung von Isoflächen nutzen polygonale Approximationen der tatsächlichen Fläche. Diese Geometrie kann eine große Menge an Speicherplatz benötigen, welcher auf der Grafikkarte nur begrenzt zur Verfügung steht. Die in dieser Arbeit verwendeten Splines werden allein durch die Polynomkoeffizienten festgelegt und basieren auf einer strukturierten Tetraeder-Partition. Somit wird theoretisch ein minimaler Speicheranteil für die Geometrie der Tetraeder benötigt. Neue hardwareunterstützte Grafik-API-Funktionen wie das *Geometry Instancing* bieten hierfür das ideale Werkzeug zur Visualisierung der auf Splines basierenden Isoflächen. Diese und andere neue Funktionalitäten, die seit dem Shader-Modell 4.0 zur Verfügung stehen, werden untersucht, um mögliche Speicherplatzeinsparung und maximale Bildraten zu realisieren.

Ein weiteres Ziel der Arbeit ist, das Zusammenspiel zwischen der verwendeten Grafik-API OpenGL und der NVidia CUDA-Technologie zu untersuchen. Darüber hinaus werden die beim Ausschöpfen der enormen parallelen Rechenleistung der zahlreichen GPU-Multiprozessoren gesammelten Erfahrungen weitergegeben.

1.2 Vorgehensweise

Es folgt eine kurze Übersicht der weiteren Kapitel.

Kapitel 2, Volumenvisualisierung. In diesem Kapitel werden Volumendaten vorgestellt und aufgezeigt, wo diese generiert werden. Weiterhin wird verdeutlicht, wie wichtig eine qualitativ hochwertige Rekonstruktion der skalaren Volumenfunktion aus den diskret abgetasteten Volumendaten ist. Die zwei grundlegenden Visualisierungsmethoden von Volumendaten, die direkte Volumenvisualisierung und das im Rahmen dieser Arbeit untersuchte Visualisieren von Isoflächen, werden erklärt.

Kapitel 3, Isoflächen - Techniken und Stand der Forschung. Hier werden die verschiedenen Techniken zur Darstellung von Isoflächen vorgestellt. Besonderes Augenmerk gilt der möglichen Parallelisierung und GPU-Unterstützung der Algorithmen, die Stand der Technik und Thema der Forschung sind. Weiterhin werden die bereits untersuchten Algorithmen zur Visualisierung mit trivariaten Splines vorgestellt. Die Defizite dieser Algorithmen und ihrer erzielten Performance, die diese Arbeit auszugleichen versucht, werden aufgezeigt.

Kapitel 4, Trivariate Splines in BB-Form. Für die Rekonstruktion des kontinuierlichen Volumens aus einem 3D-Datensatz werden Polynome mit drei Veränderlichen (trivariate Polynome) benötigt. Die Bernstein-Bézier-Form ist eine alternative Darstellung zu Polynomen in Monom-Basis. Zu Beginn des Kapitels werden die wesentlichen Aspekte der Bernstein-Bézier-Techniken anhand von Bézier-Kurven gezeigt. Die Bézier-Kurven beschreiben den Fall einer Veränderlichen (univariat). Mit diesen Grundlagen werden trivariate Tetraederelemente in BB-Form und den zugehörigen Techniken vorgestellt. Die zur Rekonstruktion verwendete Partition - das Zusammensetzen der Tetraederelemente zu einem kontinuierlichen Gebiet - wird dargestellt. Für diese Partition werden zwei Spline-Modelle vorgestellt, welche die diskreten Datenwerte aus dem Volumendatensatz approximieren und weiterhin die für die hochwertige Visualisierung wichtigen Glattheitsbedingungen über das gesamte Volumen erfüllen.

Kapitel 5, Ein massiv paralleler Algorithmus. Beschrieben wird ein Algorithmus mit dem Ziel, die Isofläche zu einem gewählten Isowert schnellstmöglichst darzustellen. Dazu werden die Spline-Modelle aus Kapitel 4 genutzt. Diese beschreiben das gesamte Volumen und enthalten somit implizit die Informationen aller Isoflächen. Entscheidend ist die schnelle Bestimmung derjenigen Tetraederelemente, welche Teile der Fläche enthalten. Nur für diese Elemente wird mittels eines präzisen Ray-Casting-Verfahren die Grenzfläche visualisiert.

Der beschriebene Algorithmus verlagert sämtliche Berechnungen für das Darstellen variierender Isoflächen auf die Grafikkarte, da diese über eine große Anzahl parallel arbeitender Multiprozessoren verfügt. Somit wird ein Programmieransatz sowie Algorithmus benötigt, der nicht dem klassischen Single-Thread-Denken entspricht.

Kapitel 6, Arbeitstechniken & illustrative Methoden. Hier werden eine Reihe bekannter Techniken vorgestellt, die dabei helfen, die volumetrischen Daten zu analysieren. Dabei wird speziell auf die vorteilhaften Eigenschaften der genutzten trivariaten Splines eingegangen.

Kapitel 7, Numerische Ergebnisse. Die vorgestellten numerischen Ergebnisse sind Zeitmessungen, die mit dem implementierten Algorithmus aus Kapitel 5 erstellt wurden. Die entscheidende Messung ist die Latenzzeit, die vergeht, bis ein neu eingestellter Isowert dargestellt wird. Weiterhin wurden die Bildraten gemessen, mit denen die aktuell gewählte Isofläche dargestellt werden kann. Sämtliche Messungen wurden für beide Spline-Modelle aus Kapitel 4 durchgeführt: für die quadratischen Super-Splines und die kubischen C^1-Splines. Berücksichtigt wurden Volumendaten verschiedener Herkunft und unterschiedlicher Größe.

Kapitel 8, Schlussbetrachtung. Hier erfolgt eine Zusammenfassung der wichtigsten Ergebnisse, die in dieser Arbeit erzielt wurden: Das qualitativ hochwertige Visualisieren von Isoflächen mit der Möglichkeit den aktuell dargestellten Isowert interaktiv zu variieren. Mit den Ergebnissen werden mögliche Anwendungen vorgestellt und ein Ausblick auf weitere Arbeiten gegeben.

Anhänge. Anhang A enthält Bilder der Isoflächen, die bei den Zeitmessungen aus Kapitel 7 entstanden sind. Darüber hinaus sind weitere Beispiele von Abbildungen aus der Arbeit gegeben. In Anhang B findet sich eine bildliche Zusammenfassung des verwendeten quadratischen sowie kubischen Spline-Schemas. Weiterhin wurde eine Symboltabelle beigefügt (Anhang C).

An dieser Stelle möchte ich ganz besonders Herrn Thomas Kalbe für seine Arbeit in [KZ08] und Herrn Frank Zeilfelder für die Arbeiten an [RZNS03, SZ07, KZ08] danken.

Kapitel 2

Volumenvisualisierung

Volumenvisualisierung steht für das Darstellen eines skalaren dreidimensionalen Gebietes mit Hilfe eines zweidimensionalen Rasterbildschirmes. Dabei wird, entgegen dem *Volumenrendering*, kein Anspruch auf fotorealistische Darstellung erhoben. Das Volumen wird durch eine Funktion ϕ beschrieben, welche jedem Punkt im Raum einen skalaren Wert zuordnet:

$$\phi : \mathbb{R}^3 \rightarrow \mathbb{R}. \tag{2.1}$$

Hierbei beschreibt ϕ beispielsweise die Dichte, die Wassermenge im organischen Gewebe, die Temperatur oder eine beliebige andere skalare Größe. In der Regel liegt die Funktion ϕ nicht analytisch vor. Basierend auf einer Anzahl diskreter Datenwerte, welche zusammen einen Volumendatensatz bilden, muss ein mathematisches Modell zur Rekonstruktion von ϕ entwickelt werden. Mit Hilfe dieses Modells können verschiedene Darstellungstechniken genutzt werden, um die Informationen aus dem Volumen anzuzeigen. Dabei wird zwischen direkter Volumenvisualisierung und der Darstellung von Isoflächen unterschieden. Typische Volumendatensätze bilden sich aus einer Menge von 2D-Schichtbildern, die bei CT- oder MRT-Scans entstehen. Wie noch beschrieben wird, spielen Volumendaten auch in anderen Bereichen ein wichtige Rolle.

2.1 Volumendaten

Volumendaten kann man sich als natürliche Erweiterung von zweidimensionalen Daten, wie bei einem digitalen Foto, vorstellen. Bei einem Foto liegen die Daten regulär angeordnet in einem Gitter. Jedem Gitterpunkt ist genau ein Pixel zugeordnet. Dabei verfügt jedes Pixel über Datenwerte, welche die einzelnen Farbzusammensetzungen, Helligkeiten und gegebenenfalls die Transparenz des Bildes beschreiben. Volumendaten entstehen durch Erweitern des Bildes von zwei auf drei Dimensionen, somit liegen die Daten auf einem 3D-Gitter. Aus dem Begriff Pixel (engl. *pix* „picture“ und *el* „element“) wird dann Voxel, was für „volumetric pixel“ oder „volume element“ steht. Ein Voxel verfügt ebenfalls über Datenwerte, die beispielsweise skalare Größen wie Temperatur, Dichte oder Farbwerte beschreiben. Weiterhin können die Daten aber auch vektorielle Größen wie Richtungen, Spannungen und andere höherdimensionale Werte repräsentieren.

Es gibt zwei unterschiedliche Betrachtungsweisen eines skalaren Voxelgitters. Zum einen kann ein Voxel als kleines quaderförmiges Volumen, in welchem der Datenwert konstant ist, interpretiert werden (Abbildung 2.1 *links* und in der *Mitte*), zum anderen

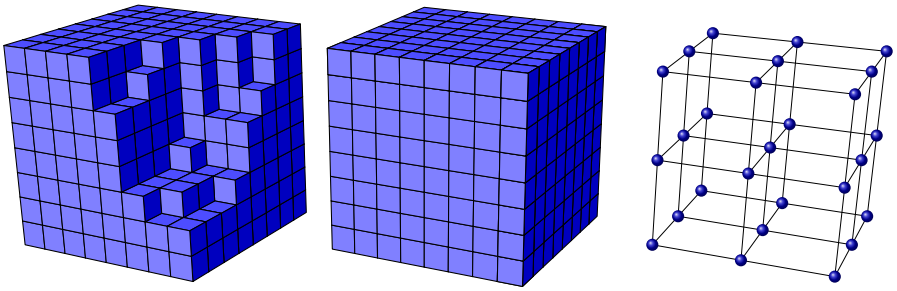


Abbildung 2.1: Die unterschiedliche Betrachtungsweise eines regulären Voxelgitters: *Links, Mitte:* Ein 8^3-Datensatz, bei dem ein Voxel den Datenwert in einem kleinen Volumen beschreibt. *Rechts:* Ein 3^3-Datensatz, bei welchem jedes Voxel einen diskret gemessenen Wert am jeweiligen Gitterpunkt repräsentiert. In dieser Arbeit wird die Sichtweise der Abbildung *rechts* verwendet, weil sich hierbei flexiblere mathematische Modelle definieren lassen.

kann jeder Voxel als diskret gemessener Wert am jeweiligen Gitterpunkt betrachtet werden (Abbildung 2.1 *rechts*). Skalare Werte auf quaderförmigen Gittern mit beliebiger Auflösung in jede Dimension bilden einen Standard für Volumendatensätze.

Im weiteren Verlauf dieser Arbeit werden die skalaren Daten als Dichtewerte interpretiert. Den Daten können aber auch andere physikalische Eigenschaften zu Grunde liegen. In der medizinischen Diagnostik könnten diese zum Beispiel die Wassermenge in einem Gewebe oder das Vorkommen eines Kontrastmittels im Körper eines Patienten sein. Jedem Gitterpunkt ist damit ein diskret abgetasteter Dichtewert f_{ijk} zugeordnet. Dieser bildet zusammen mit den Koordinaten, welche mit den Indizes i, j, k verknüpft sind, ein 4-Tupel $(\mathbf{x}_{ijk}, f_{ijk})$. Damit stellt der Datensatz folgende Abtastung von ϕ dar:

$$\phi(\mathbf{x}_{ijk}) = f_{ijk}. \tag{2.2}$$

Im Folgenden werden nun die wichtigsten Bereiche sowie Techniken vorgestellt, bei denen dreidimensionale Daten anfallen und eine visuelle Weiterverarbeitung stattfindet.

Computertomographie (CT)

Die Computertomographie (altgr. *tome* „Schnitt“ und *graphein* „schreiben“) ist ein Verfahren das aus der klassischen Röntgentechnik hervorgegangen ist. Auch hierbei wird der Körper mittels Röntgenstrahlen „durchleuchtet“, die Aufnahmetechnik ist aber weitaus komplizierter und erfordert rechnergestützte Methoden.

Bei der CT rotiert die Röntgenröhre auf einer festen Achse um den Körper, dabei wird eine Vielzahl von Aufnahmen aus unterschiedlichen Richtungen gemacht. Mit diesen Aufnahmen werden durch numerische Algorithmen Querschnittsbilder des Körpers rekonstruiert. Diese Methode hat gegenüber der klassischen Röntgentechnik den Vorteil, dass sich die einzelnen Organe im Querschnittsbild nicht überlagern (Abbildung 2.2). Die Gesamtheit dieser Schichten bildet durch einfaches Übereinanderlegen einen 3D-Datensatz des Körpers.

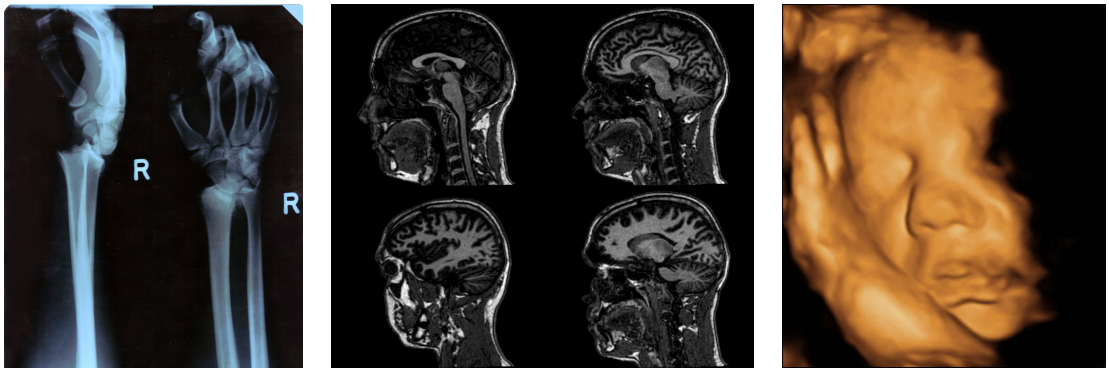


Abbildung 2.2: *Links:* Zwei Bilder eines gebrochenen Handgelenkes, aufgenommen mit der klassischen Röntgentechnik. Die Knochen überlagern sich, da nur eine Aufnahme für die Projektion auf die Ebene genutzt werden kann. *Mitte:* MRT-Schichtbilder eines menschlichen Schädels. Diese Art von Bildern wird durch computergestützte Methoden berechnet. *Rechts:* Visualisierung eines Volumendatensatzes der durch 3D-Ultraschall generiert wurde.

Magnetresonanztomographie (MRT)

Bei der Magnetresonanztomographie werden ebenfalls Querschnittsbilder erstellt. Anstatt den Körper mit Röntgenstrahlen zu „durchleuchten“, werden bei der MRT kurzzeitig bestehende, sehr starke magnetische Felder genutzt. Diese Felder bewirken eine Änderung des Eigendrehimpulses (Kernspin) der Atomkerne im Körper. Beim Wechsel des magnetischen Feldes fallen diese Atomkerne wieder auf ihren normalen Kernspin zurück und geben dabei Energie in Form von elektromagnetischen Wellen ab. Durch geeignete Wahl der Felder werden vornehmlich die Kerne von Wasserstoffatomen zum Aussenden der Elektromagnetischen Wellen angeregt. Da alle Gewebearten eine unterschiedliche Menge an Wasser enthalten, variieren auch ihre Relaxationszeiten. Diese Zeiten werden über das Empfangen der ausgesandten Wellen bestimmt und liefern Informationen über die Gewebezusammensetzung. Auch hier werden mit numerischen Algorithmen Querschnittsbilder berechnet, die durch Übereinanderlegen einen 3D-Datensatz bilden.

3D Ultraschall

Beim Ultraschall, auch Sonografie oder Echografie genannt, werden Schallimpulse in Frequenzen oberhalb des menschlichen Hörens in das Gewebe ausgesandt. Das Gewebe absorbiert, reflektiert und streut nun diese Schallwelle je nach Beschaffenheit. Das Echo des Schallimpulses liefert Informationen über die Gewebeart. Ein 2D-Ultraschallbild entsteht durch einen Schwenk des Schallkopfes. Die Vielzahl der mit dem Echo-Impuls-Verfahren gemessenen Informationen liefert das Querschnittsbild in der so abgetasteten Ebene.

Für ein 3D-Ultraschallbild schwenkt die Sonde mehrere nebeneinander liegende Ebenen ab, um Daten in der dritten Dimension zu erhalten. Dies liefert eine Menge von Querschnittsbildern die einen 3D-Volumendatensatz repräsentiert. Werden solche 3D-Daten in nahezu Echtzeit erstellt, spricht man auch von einem 4D-Ultraschall, die vierte Di-

mension entspricht hierbei der zeitlichen Änderung der Datenwerte. Um diese 4D-Daten nutzbar zu machen, sind effiziente Methoden zur Visualisierung in Echtzeit gefordert.

Die Daten liegen auf einem kegelförmigen Gitter, da die Schallimpulse beim Eindringen in den Körper divergieren (siehe [Sum04]). Eine geeignete Transformation kann diese in reguläre (quaderförmige) Datensätzen umwandeln.

Simulation

Bei vielen computergestützten Simulationen fallen Volumendaten an, die eine visuelle Weiterverarbeitung erfordern. Bei der Simulation werden mit Hilfe physikalischer Gesetze numerische Berechnungen auf einem diskretisierten Gebiet durchgeführt. Die Anwendungsgebiete sind sehr weitläufig. Sie liegen in Bereichen der Klima- und Wetterforschung, bei der Untersuchung chemischer Prozesse oder in CAE-Berechnungen des Ingenieurwesens. CFD (Computational Fluid Dynamics) ist ein Teilbereich, in dem Strömungssimulationen von Flüssigkeiten und Gasen untersucht werden. Diese Berechnungen erzeugen Volumendaten, welche grafisch dargestellt werden können.

Besonders Echtzeitsimulationen spielen eine immer wichtigere Rolle in Anwendungen aus dem CAE-Bereich und stellen höchste Anforderungen an eine interaktive Visualisierung variierender Datensätzen. Auch bei Computerspielen wird bereits physikalisch realistische Simulation von Rauch, Feuer und Wasser genutzt (siehe beispielsweise [CLT08]). Diese Simulationen basieren auch auf variierenden Volumendaten. Die Echtzeitfähigkeit und hochwertige visuelle Darstellung stehen hier im Vordergrund.

Mathematische Modelle

Implizite Oberflächen beschreiben auf einfache Weise Geometrien (Kugeln, Ebenen, Zylinder, etc.), die den Raum in ein Inneres und ein Äußeres trennen. Mit Hilfe Bool'scher Operationen, die Schnittbildung, Differenz und Vereinigung realisieren, lassen sich beliebig komplexe Bauteile konstruieren. Anwendungen liegen im CAE/CAD-Bereich, wo diese mathematischen Modelle zum *Construtive Solid Modeling* genutzt werden. Die so zusammengesetzten impliziten Funktionen bilden die Volumenfunktion ϕ, aus der durch Abtastung ein Volumendatensatz für die Visualisierung erstellt werden kann.

Signed-Distance-Functions können direkt als Funktion ϕ genutzt werden. Sie beschreiben für jeden Punkt im Raum den geringsten Abstand zu einer vorgegebenen Fläche. Dieser Abstand ist vorzeichenbehaftet, je nach Orientierung der Oberflächennormalen. Auf diese Art lassen sich aus beliebigen Polygonnetzen oder parametrischen Flächen Volumendaten verschiedener Auflösung erstellen. (siehe Anhang Abbildung A.7)

2.2 Rekonstruktion

Jede Volumenvisualisierung setzt eine mathematische Rekonstruktion der Funktion ϕ im betrachteten dreidimensionalen Gebiet voraus. Basierend auf den diskreten Datenwerten f_{ijk} spricht man dabei von interpolierenden oder approximierenden Methoden. Eine interpolierende Rekonstruktion durchläuft alle Datenwerte, sprich $\phi(\mathbf{x}_{ijk}) = f_{ijk}$. Eine

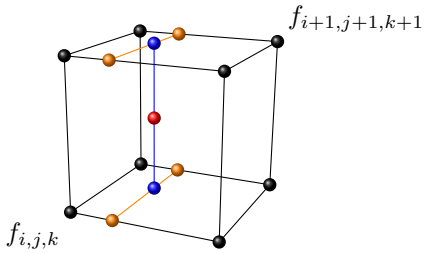


Abbildung 2.3: Trilineare Interpolation für die rot markierte Position. Dazu werden die 8 nächstgelegenen Datenwerte benötigt. Zunächst werden die Werte an den 4 orange markierten Positionen durch lineare Interpolation der zwei nächstgelegenen Datenwerte bestimmt. Die Werte an den blau markierten Positionen werden daraufhin durch Interpolation zwischen je zwei orangenen Werten ermittelt. Den resultierenden roten Wert erhält man durch Interpolation der beiden blauen Werte.

approximierende Rekonstruktion erfüllt dies nicht, nähert den Datenwerten aber unter geeigneten Gütekriterien an.

Zwei naheliegende Methoden für die Rekonstruktion von ϕ ist das Verfahren des *Nächsten Nachbarn* sowie das *Trilineare Interpolieren*.

Bei dem Verfahren des Nächsten Nachbarn kommt es zu einer Voxelisierung. Dies entspricht der Sichtweise aus Abbildung 2.1 (*links* und *Mitte*). Der Wert für ϕ ergibt sich aus dem Wert des am nächsten gelegenen Gitterpunktes. Diese sehr einfache Art der Rekonstruktion erzeugt unstetige Übergänge.

Das trilineare Interpolieren wird in Abbildung 2.3 dargestellt. Für die Rekonstruktion werden jeweils die 8 nächsten gelegenen Datenwerte herangezogen, um den Wert für ϕ an dem hier rot markierten Punkt zu bestimmen. Dabei wird nacheinander in alle drei Richtungen linear interpoliert. Die Übergänge sind im gesamten Volumen stetig. Das trilineare Interpolieren ist ein Standardverfahren zur Rekonstruktion und wird wegen seiner Einfachheit und Effizienz bei der Implementierung in vielfältigen Bereichen angewandt. Nachteilig sind die unstetigen Übergänge der Ableitungen. Trilineares Interpolieren basiert auf Polynomen mit drei Veränderlichen (x,y,z) und totalem aber unvollständigen Grad 3 (höchster Term: xyz, nicht genutzte Terme: x^3, x^2y, etc.). Demnach ist das Verfahren nicht linear sondern kubisch. Aufgrund des einfachen Multiplizierens der linearen Polynombasen spricht man auch von linearen Tensor-Produkt-Splines.

Tensor-Produkt-Splines höherer Ordnung ermöglichen glatte Rekonstruktionen von ϕ. Aufgrund der einfachen Erweiterbarkeit auf die benötigen drei Dimension und der teilweise geforderten C^1- oder C^2-Stetigkeiten finden hier meist kubische Polynome Anwendung. Kubische Polynome mit einer Veränderlichen benötigen 4 Stützstellen. Für trikubische Tensor-Produkt-Splines werden die am nächsten gelegenen 64 Datenwerte (4^3) benötigt. Interpolierende trikubische Tensor-Produkt-Splines (Catmull-Rom) ermöglichen C^1-stetige Rekonstruktionen. Besteht kein Anspruch auf Interpolieren, oder ist sogar ein Glätten der Daten gewünscht, so können approximierende trikubische Tensor-Produkt-Splines (B-Splines) auch C^2-Stetigkeit garantieren. Trikubische Tensor-Produkte basieren auf Polynomen mit totalem aber unvollständigen Grad 9 (höchster Term: $x^3y^3z^3$).

In dieser Arbeit wird eine Rekonstruktion verwendet die totalen sowie *vollständigen*

Grad 3 besitzt und dabei C^1-Stetigkeit garantiert. Darüberhinaus werden nur 27 Datenwerte benötigt, was das Verfahren lokal macht. Der vollständige niedrige Polynomgrad wird entlang aller Richtungen im Volumen gleich ausgenutzt, was eine effiziente und exakte Visualisierung von Isoflächen mittels Ray-Casting ermöglicht. Diese Rekonstruktion von ϕ basiert auf trivariaten Splines in Bernstein-Bézier-Form.

2.3 Direkte Volumenvisualisierung und Isoflächen

Bei der Visualisierung von Volumendaten gibt es zwei grundsätzlich zu unterscheidende Ansätze: Zum Einen die direkte Volumenvisualisierung, bei der die Informationen des gesamten sichtbaren Volumens dargestellt werden, zum Anderen die Visualisierung mittels Isoflächen. Bei diesem Ansatz werden Oberflächengeometrien gebildet, welche nur einen bestimmten Teil der Informationen des Datensatzes repräsentieren.

Beim direkten Darstellen der Daten ist es nötig, eine effiziente und visuell hochwertige Näherung des sogenannten *Volumen Rendering Integrals* zu bestimmen. Das Integral beschreibt die Aufsummierung der Materialfarbwerte entlang eines Sichtstrahls unter Berücksichtigung des Absorptionsverhaltens bzw. der Transparenz des Materials. Das Integral wird durch verschiedene Auswertungsverfahren und unter Berücksichtigung von Lichtquellen für jedes Pixel bestimmt [EHK*06a, EHK*04]. Abbildung 2.4 *oben* zeigt zwei Techniken zur direkten Volumenvisualisierung.

Andererseits wird bei Isoflächen nur ein Teil der im Datensatz enthaltenen Informationen dargestellt. Abbildung 2.4 (*unten*) zeigt zwei unterschiedliche Isoflächen aus demselben Datensatz. Die Isofläche (griech. *isos* für „gleich“) ist die höherdimensionale Erweiterung zur Isolinie, wie man sie beispielsweise als Höhenlinien aus Landkarten kennt. Eine Isolinie beschreibt die Menge aller Punkte, an denen ein gleicher Wert (Isowert) vorliegt. Diese Punkte bilden im zweidimensionalen Raum Linien. Beim Anheben der Daten in die dritte Dimension bilden diese Punkte Flächen.

Dem Isowert $c \in \mathbb{R}$ ist die Isofläche

$$I_c := \{\mathbf{v} \in \mathbb{R}^3 \mid \phi(\mathbf{v}) = c\} \tag{2.3}$$

zugeordnet.

Das menschliche Gehirn ist darauf trainiert, Oberflächen unter Einwirkung von Lichtquellen zu betrachten. Nur wenige Materialien in unserem Alltag sind teilweise bzw. vollständig durchsichtig (einige Flüssigkeiten, Glas, Nebel und Rauch). Selbst gedanklich bilden wir nur die Oberflächen von Objekten, die innere Beschaffenheit ist für unsere Vorstellung meist irrelevant. Die Knochen, das Herz oder andere Organe und Gewebearten stellen wir uns so als feste Einheiten umschlossen von einer Oberfläche vor. Diese Tatsache motiviert dazu, Verfahren zur Gewinnung von Isoflächen zu entwickeln, um mit deren Hilfe Anomalien, räumliche Anordnungen und eine große Anzahl anderer medizinischer Untersuchungen zu ermöglichen. Dabei werden höchste Anforderungen an die Rekonstruktion und Visualisierung gestellt, um die innere Beschaffenheit mit minimaler Anzahl an Artefakten darzustellen. Verfahren zu untersuchen sowie weiterzuentwickeln, welche diese Anforderungen erfüllen ist Ziel der vorliegenden Arbeit.

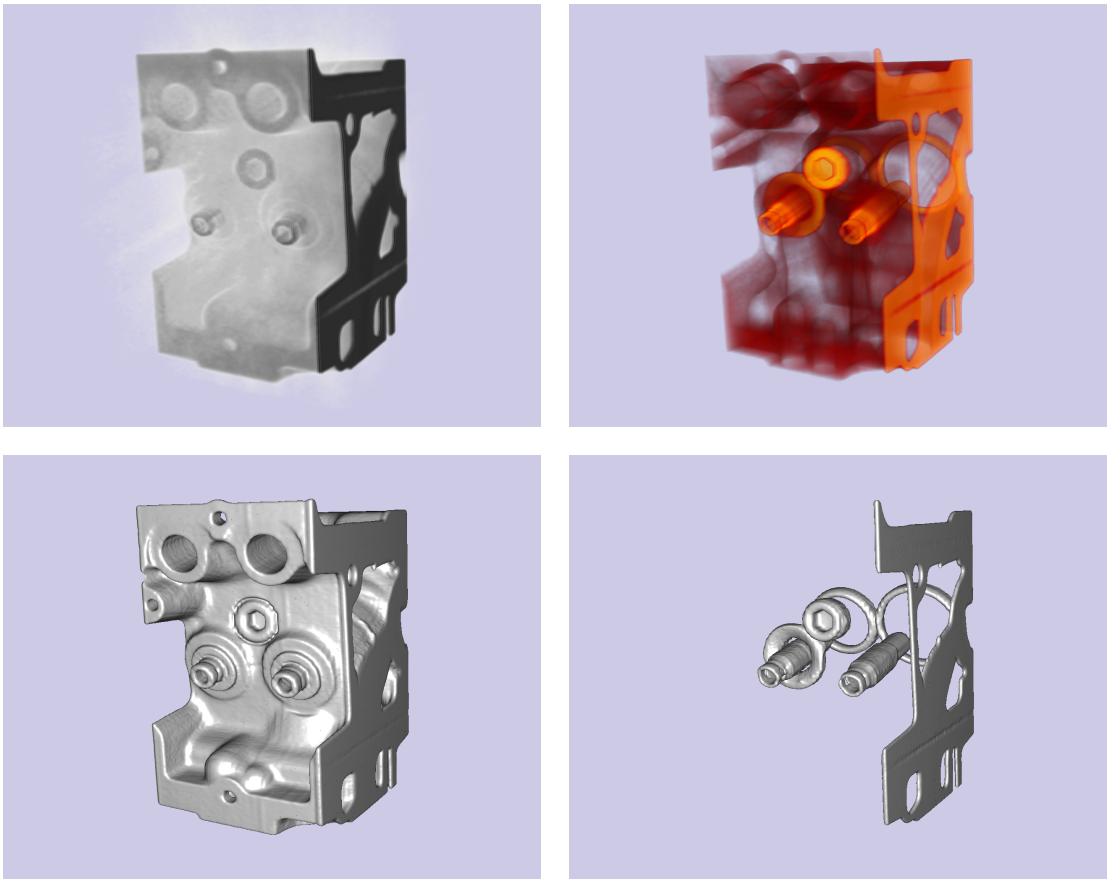


Abbildung 2.4: *Oben:* Direkte Volumenvisualisierung. *Unten:* Visualisierung mittels Isoflächen.

2.4 Variierende Isoflächen

Bei der Visualisierung variierender Isoflächen unterscheidet man zwischen zwei Anwendungsfällen: zum Einen das Darstellen von unterschiedlichen Isowerten durch Benutzerinteraktion in einem gleichbleibenden Datensatz, zum Anderen die Visualisierung *einer* Isofläche in einem variierenden Datensatz. Diese beiden Fälle werden nun vorgestellt.

Ein gleichbleibender 3D-Datensatz, wie er zum Beispiel bei einem CT- oder MRT-Scan vorliegt, bedarf interaktiver Nachbearbeitung. Diese benötigt computergestützte Verfahren, um schnell und unter hoher visueller Qualität die entscheidenden Informationen sichtbar zu machen. Bei der Darstellung von Isoflächen ist das Finden des entscheidenden Isowertes gefragt, um die ideale Grenzfläche von einzelnen Organen, der Knochenstruktur oder Teilen der Muskulatur anzuzeigen. Die Änderung des aktuellen Isowertes erfordert das Anpassen der dargestellten Fläche in Latenzzeiten von maximal 100 bis 200 Millisekunden, um interaktives Arbeiten mit Bildraten von mindestens 5 bis 10 Bildern pro Sekunde zu ermöglichen. Computer-Applikationen, die mit Hilfe von Isoflächen Informationen darstellen, benötigen aber neben dem schnellen Wechsel des

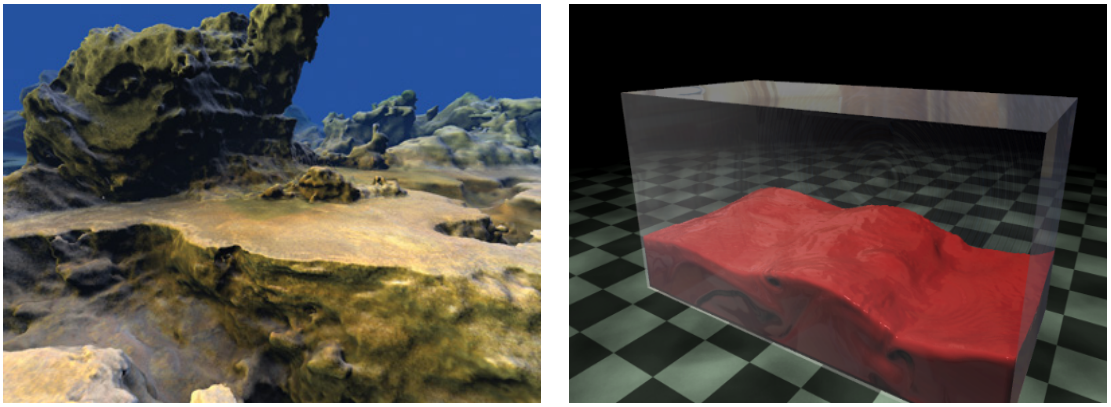


Abbildung 2.5: Isoflächen finden bei der Visualisierung in vielfältigen Bereichen Anwendung: *Links:* Eine Landschaft wird hier mittels Isofläche beschrieben, diese basiert auf impliziten Funktionen sowie 3D-Noise-Texturen. Illustration: Ryan Geiss [Gei08]. *Rechts:* Diese Echtzeit-Simulation einer zähen Flüssigkeit auf einen 3D-Gitter verwendet zur Darstellung Isoflächen. Illustration: Keenan Crane [CLT08].

Isowertes weitere Arbeitstechniken: gleichzeitiges Darstellen verschiedener Grenzflächen (auch unter teilweiser Transparenz), Einstellen verschiedener Schnittebenen und Kombinationen mit direkter Volumenvisualisierung. Diese Techniken sind erst bei interaktiv reagierenden Programmen und unter hoher visueller Qualität effizient nutzbar. Diese Anforderungen liegen natürlich nicht nur im humanmedizinischen Bereich, auch industrielle Qualitätskontrollen oder naturwissenschaftliche Simulationen benötigen interaktive Verfahren zur Darstellung der vom Anwender gewählten Isoflächen.

Variierende Datensätze können in Form von Messungen oder Simulationen erstellt werden. Im Bereich der Messung ist der 4D-Ultraschall als technisches Messinstrument bekannt. Die Pränataldiagnostik nutzt den 4D-Ultraschall, um Echtzeit-Aufnahmen des Fötus zu erstellen. Für die Visualisierung muss hier die Isofläche zu einem festen Isowert im variierenden Datensatz dargestellt werden. Dabei muss die Rekonstruktion mit sehr kurzen Latenzzeiten erfolgen, um zu jeder Änderung des Datensatzes die aktuelle Isofläche anzuzeigen. In der Medizin ermöglicht die Grenzflächendarstellung die Untersuchung des ungeborenen Fötus, seiner Organe und der Anatomie.

Viele Simulationen, die in unterschiedlichen Forschungsgebieten genutzt werden, generieren zeitlich variierende Volumendatensätze. Die Isoflächendarstellung ist eine mögliche Technik, die variierenden Daten anzuzeigen. So stellt die Isofläche beispielsweise Grenzflächen im zeitlich variablen Potentialfeld eines Moleküls oder Gebiete mit gleicher Temperatur bzw. gleichem Luftdruck in einer 3D-Wettersimulation dar. Auch in der Computerspielindustrie, welche der Hauptantriebsfaktor der rasanten Weiterentwicklung von Grafikkarten ist, werden Isoflächen genutzt. Spieleentwickler setzten immer mehr auf realistische Simulationen. Dabei spielen bei der Darstellung von Wasser Grenzflächen eine entscheidende Rolle [CLT08]. Isoflächen können auch komplette Landschaften repräsentieren [Gei08] und bei Echtzeitfähigkeit der Rekonstruktion die Landschaftsoberfläche jederzeit variieren oder neu erschaffen (siehe Abbildung 2.5).

Kapitel 3

Isoflächen - Techniken und Stand der Forschung

Bei der Wahl eines Algorithmus zur Darstellung von Isoflächen müssen folgende Fragen geklärt werden: Welche Eigenschaften (Approximationsordnung, Glattheitsbedingungen) garantiert die Rekonstruktion der Fläche? Wie effizient kann die so rekonstruierte Fläche visualisiert werden?

Ist darüber hinaus ein häufiger Wechsel des Isowertes durch Benutzereingabe gefordert, oder liegt der Fläche ein variierender Datensatz zu Grunde, dann kommt eine entscheidende Frage hinzu: Wie schnell kann die Fläche zu einem neu eingestellten Isowert oder bei Variation des Datensatzes rekonstruiert werden?

3.1 Visualisierungsmethoden

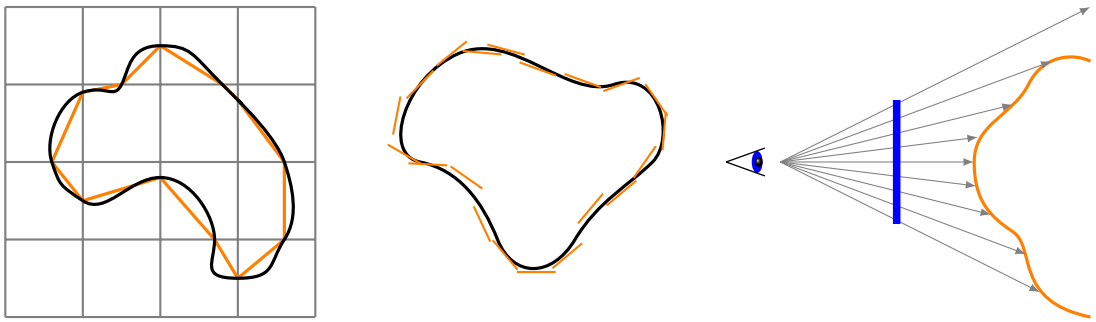


Abbildung 3.1: Drei grundlegende Visualisierungsmethoden für Isoflächen. *Links:* Polygonale Methode. *Mitte:* Punktbasierte Methode. *Rechts:* Das Ray-Casting Prinzip. Illustration nach [vKvdBT07].

Fast alle Algorithmen, die im Laufe der Zeit für die Darstellung von Isoflächen entwickelt und vorgestellt wurden, basieren auf einer der in Abbildung 3.1 dargestellten Methoden. Bei der polygonalen Methode wird die tatsächliche Isofläche durch Konstruktion eines Polygonnetzes (meist Dreiecke) angenähert. Der wohl bekannteste Vertreter dieser Methode ist das Marching-Cubes-Verfahren [LC87]. Hierbei wird das Volumen zunächst in Würfel zerlegt, wobei jeder Würfel so gewählt wird, dass seine Eckpunkte auf 8 Datenwerte im Volumendatensatz fallen. Anhand dieser Werte steht fest, ob ein Eckpunkt innerhalb oder außerhalb der Grenzfläche liegt und welche der 12 Würfelkanten von der

Isofläche geschnitten werden. Diese Schnittpunkte werden durch einfaches lineares Interpolieren zwischen den Datenwerten bestimmt und dann mittels Dreiecken zu einer Fläche verbunden. Abbildung 3.2 zeigt, wie das Marching-Cubes Verfahren die Isofläche innerhalb eines Würfels durch Dreiecke annähert. Das Verfahren kann sehr effizient mittels Lookup-Tabellen und vorberechneten Datenstrukturen (Octtree) implementiert werden. Viele Varianten von Marching Cubes wurden im Laufe der Zeit vorgestellt [TPH98] mit dem Ziel, die Effizienz zu steigern [DZTS08], höherqualitative Polygonnetze zu generieren [The02] und sich der tatsächlichen Isofläche besser anzunähern [Nie04, KBSS01].

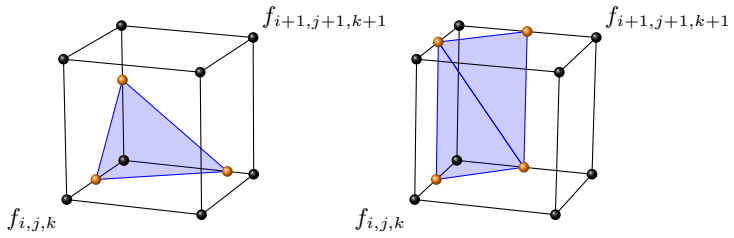


Abbildung 3.2: Beim Marching-Cubes-Verfahren wird das Volumen in Würfel unterteilt. Innerhalb jedes Würfels wird die Isofläche mit einer Reihe von Dreiecken angenähert.

Ein anderer Typ der polygonalen Methode sind Grenzverfolgungs-Algorithmen. Hier wird, beginnend von einer Position auf der Isofläche, eine Oberflächenverfolgung durchgeführt. *Marching-Front*-Algorithmen [Har98, KFU*09] nutzen dieses Prinzip, ausgehend von einem Startdreieck, um eine Triangulierung sukzessiv fortzuführen. Auswertung mehrerer Positionen von $\phi(\mathbf{x})$ und $\nabla\phi(\mathbf{x})$ liefert ein lokales Approximationspolynom der Isofläche an der aktuell wachsenden Front. Mit Hilfe des Polynoms wird ein nach Qualitätskriterien optimal passendes Dreieck hinzugefügt. Schritt für Schritt wächst das Netz an verschiedenen Fronten und generiert mit Hilfe von Operationen, die Lücken schließen und Fronten zusammenführen, ein Polygonnetz der Grenzfläche. Dabei enstehen Dreiecksnetze von sehr hoher Qualität. Problematisch sind die sehr langen Konstruktionszeiten, die teilweise Instabilität sowie die Tatsache, dass nicht automatisch alle Isoflächen zu einem Isowert trianguliert werden.

Vorteil der polygonalen Methode ist eine effiziente Visualisierung über die Grafikkarte, da diese für Polygone optimiert ist. Die Qualität der enstehenden Netze ist hauptsächlich von der Rekonstruktion von $\phi(\mathbf{x})$ und $\nabla\phi(\mathbf{x})$ abhängig. Nachteil aller polygonalen Methoden ist, dass die Fläche durch lineare Elemente angenähert wird, welche sich nur bedingt an eine glatte Oberfläche annähern können. Dieses Problem wird gerade bei der Silhouette und bei nah betrachteten Bereichen der Fläche deutlich.

Eine grundlegend andere Methode, um Isoflächen zu visualisieren, ist ein punktbasierter Ansatz [WH94]. Hier wird durch verschiedene Methoden eine große Anzahl von Punkten mit den zugehörigen Normalen auf der Isofläche bestimmt. Diese Punkte können nun entweder direkt oder als Flächenelemente visualisiert werden, die der Normalen entsprechen orientiert sind.

Beim Ray-Casting-Prinzip [PSL*98] werden für jedes Pixel des zu berechnenden Bildes Sichtstrahlen generiert, die mit der Isofläche auf Schnitt getestet werden. Auf diese

Art wird die Fläche unter Berücksichtigung des aktuellen Betrachterstandpunktes sowie der Perspektive abgetastet. Ist der Schnittpunkt gefunden, werden mit der Oberflächennormale Lichtberechnungen durchgeführt. Beim Ray-Casting-Verfahren, im Gegensatz zum Ray-Tracing-Verfahren, erfolgt aus Performancegründen kein weiteres Verfolgen der Strahlen, um Reflektionen oder verdeckte Lichtquellen in die Lichtberechnungen einfließen zu lassen. Den meisten Ray-Casting- und Ray-Tracing-Methoden geht ein Präprozess voraus, der geeignete Datenstrukturen erstellt, um einen großen Teil der auf Schnitt zu testenden Elemente zu reduzieren.

3.2 Parallelisierung - GPU Nutzung

Bei der durch Grafikkarten unterstützten Visualisierung werden die parallel arbeitenden Streaming-Prozessoren genutzt. Dies geschieht bei der polygonalen Methode über die Standard-Pipeline. Es ist die Grundidee des GPU-basierten Ray-Casting, den Volumendatensatz als 3D-Textur abzuspeichern. Mittels Fragmentshader-Programmen werden pro Pixel Strahlen durch das Volumen geschickt. Dies kann dadurch realisiert werden, dass die Vorderseiten des Volumenquaders als Polygone gerastert werden. Mit der aktuellen Kameraposition können Sichtstrahlen bestimmt werden, die mit Hilfe der 3D-Textur den Schnittpunkt mit der Isofläche bestimmen und Lichtberechnungen durchführen. Im Laufe der Zeit wurden hochwertige Ray-Casting-Verfahren vorgestellt [HSS*05, SGS06].

Auch die gesamte Rekonstruktion polygonaler Isoflächen wurde auf die Grafikkarte portiert [JC06, GJD05, Pas04]. Ein entscheidendes Kernstück aller GPU-Algorithmen ist die Geometrieerstellung. So wurde für ältere Grafikkarten die Geometrie noch als eine Art Rohling von der CPU erstellt [Pas04] und ab Shader-Modell 4.0 der Geometry-Shader verwendet [Gei08]. Bei einigen GPGPU-Varianten sowie CUDA [NVI09c] nutzte man eine Kombination aus *Prefix-Sum* und *Stream-Compaction* zur parallelen Geometrieerstellung. In [DZTS08] wurden sogenannte *Histogramm-Pyramiden* genutzt, um die Isoflächen-Rekonstruktion aus großen dünn besetzten Volumendaten zu beschleunigen. Das Marching-Cubes-Verfahren wurde in [Gei08] mittels Geometry-Shader genutzt, um Isoflächen von impliziten Funktionen zu triangulieren. Eine CUDA-Version von Marching-Cubes ist im NVidia CUDA-SDK enthalten [NVI09c]. Hier sind interaktive Rekonstruktionen von I_c auch bei großen Datensätzen möglich. Eine sehr ähnliche Version zu dem CUDA Marching-Cubes wurde als Referenzmethode für die Zeitmessungen in Kapitel 7 implementiert. ATI/AMD zeigt in [TS07, TSD07] eine parallele Version eines hybriden Marching Cubes/Marching Tetraeder Algorithmus [TPH98]. Auch hier lassen sich große Volumendatensätze interaktiv triangulieren.

3.3 Visualisierung mit trivariaten Splines

Alle oben vorgestellten Verfahren haben eines gemeinsam: Die Qualität der resultierenden Isofläche hängt maßgeblich von den Eigenschaften der Rekonstruktion von $\phi(\mathbf{x})$ und dem Gradienten $\nabla\phi(\mathbf{x})$ ab, welcher die Oberflächennormale bildet. Dies zum Anlaß, nutzten Zeilfelder und andere in [NRSZ05, SZ07] trivariate Splines auf Tetraeder-Partitionen

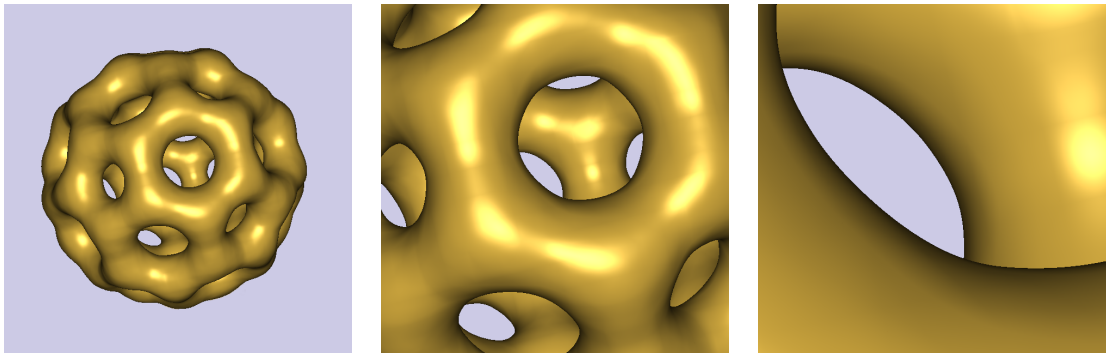


Abbildung 3.3: Automatisches *Level of Detail* der trivariaten Splines erlaubt beliebige Nahaufnahmen. Hier, die in dieser Arbeit verwendeten, kubischen C^1-Splines. Datensatz: Bucky 32^3.

für Rekonstruktionen von $\phi(\mathbf{x})$ und $\nabla\phi(\mathbf{x})$ mit garantierten Eigenschaften. Das entwickelte Verfahren erzielte glatte Silhouetten, automatisches *Level of Detail* und stetige Gradienten für hochwertige Lichtberechnungen (siehe Abbildung 3.3). Diese komplexen Splines benötigten bei der Visualisierung in [RZNS03, RZNS04] mit Hilfe eines CPU Ray-Casting-Verfahrens noch mehrere Sekunden pro Bild. In [KZ08] wurden die auf den Splines basierenden Isoflächen, GPU unterstützt, erstmalig interaktiv visualisiert. Hier wurde, genau wie in dieser Arbeit, eine Kombination aus *Cell-Projection* und Ray-Casting genutzt, indem die Tetraederelemente auf den Bildschirm projiziert werden, um mittels Sichtstrahl pro Pixel Schnitt- und Lichtberechnungen durchzuführen. Der Ansatz in [KZ08] benötigte trotz optimierter räumlicher Datenstrukturen einen zeitintensiven Präprozess von mehreren Sekunden, der nicht relevante Elemente aussortiert und Datenstrukturen erstellt. Dieser Präprozess mußte jedesmal neu durchgeführt werden, wenn ein Wechsel des Isowertes durch Benutzereingabe erfolgte.

Trivariate Splines auf massiv parallelen Streaming-Prozessoren

Diese Arbeit verwendet die Spline-Modelle aus [NRSZ05, SZ07] für die Rekonstruktion von $\phi(\mathbf{x})$, $\nabla\phi(\mathbf{x})$. Dabei werden alle Berechnungen in einem kombinierten CUDA- [NVI09b], OpenGL- [SWND05] und GLSL- [KBR, Ros06] Ansatz auf die Grafikkarte ausgelagert, um die enorme Rechenleistung zu nutzen und zeitintensiven Datentransfer zwischen CPU und GPU zu vermeiden. Die Geometrie der uniformen Tetraeder-Partition wird nicht wie bei [KZ08] in Form von Displaylisten abgelegt, sondern liegt in einem nur knapp 3 Kilobyte großen Vertex-Array und wird durch Geometry Instancing [Gol08] gezeichnet. Vertex- und Fragmentshader sind ähnlich zu dem Ansatz in [KZ08], profitieren aber enorm durch die Umstellungen auf Instancing und anderen Shader-Modell 4.0-Neuerungen. Zwei innovative Varianten zur Berechnung und Codierung der Spline-Koeffizienten werden vorgestellt: eine speicherminimale Variante, die sämtliche Koeffizienten unter nur leichten Bildrate-Einbußen *on-the-fly* bestimmt sowie eine bildratenoptimierte Variante, die bei jedem Isowertwechsel parallele Vorberechnungen durchführt.

Kapitel 4

Trivariate Splines in BB-Form

Für die mathematische Beschreibung eines kontinuierlichen Volumens werden Polynome mit drei Veränderlichen (trivariate Polynome) benötigt. Eine mögliche Basis für diese Polynome sind Monome. In dieser Arbeit wird hingegen die auf den baryzentrischen Koordinaten basierende Bernstein-Bézier-Form (BB-Form) verwendet. Die BB-Form verfügt über viele Eigenschaften, die für effiziente Berechnungen und für eine qualitativ hochwertige Visualisierung von Nutzen sind.

Zu Beginn des Kapitels wird der univariate Fall der Bernstein-Bézier-Form vorgestellt, die Bézier-Kurven. Hier lassen sich die vielen nützlichen Eigenschaften der BB-Form einfacher verdeutlichen als im benötigten trivariaten Fall. In diesem Zusammenhang werden die für diese Arbeit wichtigsten Merkmale der BB-Form dargestellt: die Auswertung mit dem Algorithmus von de Casteljau, stetige Übergänge zweier Bézier-Kurven durch einfache Geometriebedingungen oder die Eigenschaft der konvexen Hülle. Es folgt eine kurze Präsentation der baryzentrischen Koordinaten bezüglich eines Tetraeders (der Simplex, der ein Volumen im $\mathbb{R}^3$ aufspannt). Mit dieser Grundlage wird die trivariate Erweiterung der Bernstein-Bézier-Form auf Tetraedern vorgenommen. Eine Tetraeder-Partition Δ von der Grundmenge $\Omega \subseteq \mathbb{R}^3$ wird vorgestellt, wobei Ω das zu visualisierende Volumen umfaßt. Zum Ende des Kapitels werden Spline-Funktionen zweiten und dritten Grades auf dieser Triangulierung präsentiert, die einen skalaren Volumendatensatz unter Glattheitsbedingungen approximieren.

Ein ausführlicher Einstieg in die multivariaten Bernstein-Bézier-Techniken ist in [LS07] zu finden.

4.1 Approximation und Interpolation

Interpolation beschreibt in der Mathematik eine Methode, neue Datenwerte in der Nähe von bekannten Werten zu bestimmen. Approximation steht in der Mathematik allgemein für eine näherungsweise, aber in Hinblick auf die Nutzbarkeit ausreichende Beschreibung.

In den verschiedenen Bereichen fallen diskrete Datenwerte durch Abtastung, Experimente oder Überlegungen an. Es gilt eine Polynomfunktion zu finden, welche diese Werte durchläuft beziehungsweise sich den Werten annähert. Dabei können diese Werte beliebige Daten repräsentieren. Sehr anschaulich lassen sich die Interpolations- und Approximationseigenschaften einer Funktion betrachten, deren Werte Punkte im Raum beschreiben. Hier lassen sich Eigenschaften wie Polynomgrad oder Stetigkeiten einfach nachvollziehen. Darum wird im folgenden Abschnitt die geometrische Interpretation der

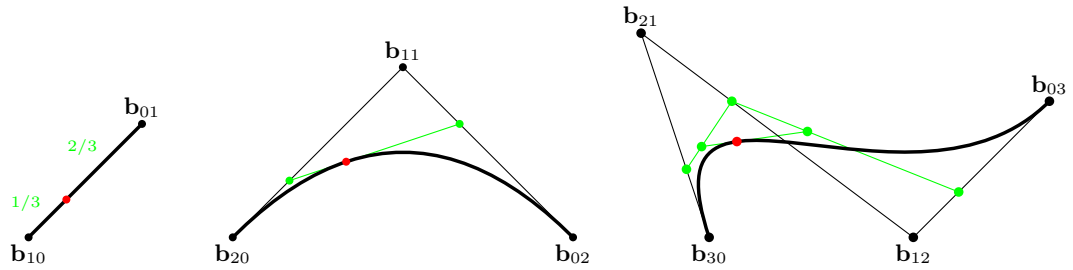


Abbildung 4.1: Bézier-Kurven unterschiedlichen Grades (*von links nach rechts*): Eine lineare, eine quadratische und eine kubische Bézier-Kurve verfügt über zwei, drei, bzw. vier BB-Koeffizienten $\mathbf{b}_{ij}$ (mit Grad $q = i + j$). Die geometrische Interpretation des de Casteljau-Algorithmus ist grün dargestellt, hier mit der beispielhaften Auswertung für $t = 1/3$. Der resultierende Punkt $\mathbf{p}(1/3)$ ist rot markiert.

Bernstein-Bézier-Form betrachtet. Später wird die BB-Form aber für die Approximation von Dichtewerten im Raum genutzt.

4.2 Bézier-Kurven

Um das Jahr 1960 entwickelten Pierre Bézier und Paul de Casteljau für die Automobilindustrie die intuitive Möglichkeit, polynomiale Freiformkurven und -flächen zu modellieren. Diese Bézier-Kurven sowie -Flächen bilden bis heute ein wichtiges Werkzeug in Bereichen wie Vektorgrafik, CAD und Animation.

Eine Bézier-Kurve des Grades q wird bestimmt durch $q+1$ Kontrollpunkte (Abb. 4.1). Diese Kontrollpunkte werden auch Bernstein-Bézier-Koeffizienten (BB-Koeffizienten) genannt und bestimmen den Verlauf der polynomialen Kurve. Mit Ausnahme des Start- und Endpunktes verläuft die Kurve nicht durch ihre Kontrollpunkte. Damit sind Bézier-Kurven approximierend. Die Bernstein-Form einer Bézier-Kurve des Grades q ist gegeben durch

$$\mathbf{p}(t) = \sum_{i+j=q} \mathbf{b}_{ij} B_{ij}^q(t) \quad \text{mit} \quad t \in [0,1], \quad i,j \geq 0, \tag{4.1}$$

wobei $\mathbf{b}_{ij} \in \mathbb{R}^2, \mathbb{R}^3$ die $q+1$ Kontrollpunkte und B_{ij}^q die Bernstein-Polynome (nach Sergei N. Bernstein) des Grades q sind:

$$B_{ij}^q(t) = \frac{q!}{i!j!} \lambda_0^i(t) \lambda_1^j(t), \quad i + j = q. \tag{4.2}$$

λ_0, λ_1 sind die baryzentrischen Koordinaten bezüglich des Start- und Endpunktes. Sie ermöglichen eine positionsunabhängige Definition der Polynombasis. Bei der obigen Parametrisierung von t auf das Interval $[0,1]$ vereinfachen sich diese zu $\lambda_0 = 1 - t$ und $\lambda_1 = t$. Eine allgemeinere Form mit $t \in [a,b]$ ist in [PBP02] zu finden.

Die Bernstein-Polynome sind die Gewichtsfunktionen der Bézier-Kurve bezüglich ihrer Kontrollpunkte. Abbildung 4.2 zeigt die Polynome für lineare, quadratische und kubische Bézier-Kurven. Die Bernstein-Polynome nehmen ihr Maximum für die Stellen $t = i/q$ an,

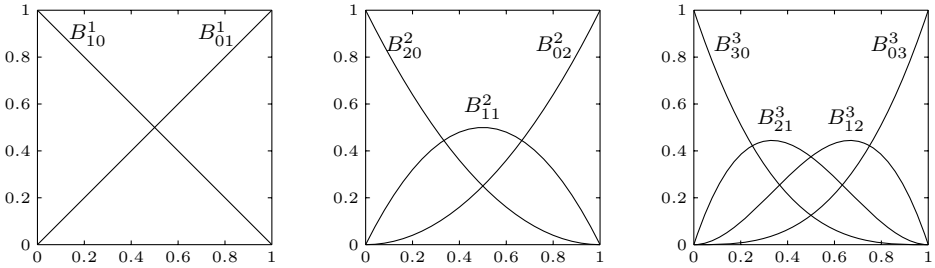


Abbildung 4.2: *Von links nach rechts:* Die Bernstein-Polynome für $q = 1$, $q = 2$ und $q = 3$. Sie sind die Trägerfunktionen der Bézier-Kurve und bilden eine Basis des polynomialen Vektorraumes.

mit $i = 0, \ldots, q$ im Intervall $[0, 1]$. Diese Stellen werden auch *Grundpunkte* (engl.Domain-Points) genannt. An ihnen hat der jeweilige Kontrollpunkt maximalen Einfluß auf den Verlauf der Kurve.

Algorithmus von de Casteljau (univariat)

Die rekursive Relation der Bernstein-Polynome

$$B^q_{ij} = \lambda_0 B^{q-1}_{i-1,j} + \lambda_1 B^{q-1}_{i,j-1}, \quad \text{für alle} \quad i + j = q, \tag{4.3}$$

welche als direkte Konsequenz aus Definition (4.2) folgt, ermöglicht eine Auswertungsmethode der Bézier-Kurve die als *Algorithmus von de Castejau* bezeichnet wird. Der Algorithmus benötigt keine explizite Berechnung der Bernstein-Polynome B^q_{ij}, denn hierbei wird mittels sukzessiver Linearkombination der Kontrollpunkte die Bézier-Kurve $\mathbf{p}(t)$ für den Parameterwert $z \in \mathbb{R}$ ausgewertet. Zunächst benötigt man die baryzentrischen Koordinaten $\lambda_0(z) = 1 - z$ und $\lambda_1(z) = z$, welche einmalig bestimmt werden. Die Startkoeffizienten $\mathbf{b}^{[0]}_{ij} = \mathbf{b}_{ij}$ werden mit den Kontrollpunkten initialisiert. Der Algorithmus berechnet nun schrittweise die Koeffizienten $\mathbf{b}^{[m]}_{ij}$ für $m = 1, \ldots, q$ mit der Vorschrift

$$\mathbf{b}^{[m]}_{ij} = \lambda_0 \mathbf{b}^{[m-1]}_{i+1,j} + \lambda_1 \mathbf{b}^{[m-1]}_{i,j+1}, \quad \text{für} \quad i + j = q - m. \tag{4.4}$$

Mit der letzten Rekursionsstufe liegt der ausgewertete Punkt $\mathbf{p}(z) = \mathbf{b}^{[q]}_{00}$ vor. Abbildung 4.3 verdeutlicht das schrittweise Vorgehen des Algorithmus, welches einem Dreiecksschema entspricht. Die geometrische Interpretation dieses Verfahrens ist in Abbildung 4.1 farblich angedeutet.

Der Algorithmus liefert neben dem Punkt $\mathbf{p}(z)$ mehr Informationen, es werden simultan auch die q Ableitungen an der Stelle z bestimmt. Diese können direkt aus dem Dreiecksschema konstruiert werden. So liegt beispielsweise die erste Ableitung mit der vorletzten Rekursionsstufe vor, die Punkte $\mathbf{b}^{[q-1]}_{10}$ und $\mathbf{b}^{[q-1]}_{01}$ bilden die Tangente der Kurve in z. Die höheren Ableitungen können auf ähnliche Weise konstruiert werden (siehe [PBP02]).

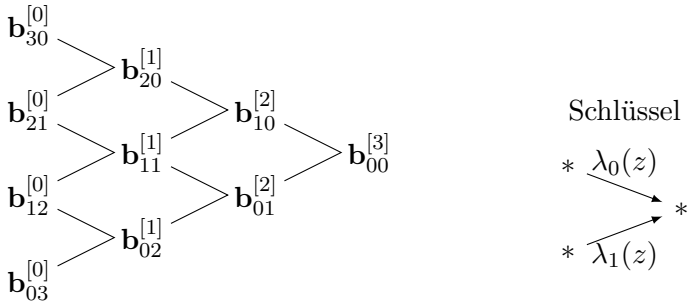


Abbildung 4.3: Das Vorgehen des de Casteljau-Algorithmus entspricht einem Dreiecksschema (hier für eine kubische Bézier-Kurve). Die Anzahl der Koeffizienten wird in jedem Schritt um eins reduziert. Der am Ende vorliegende Koeffizient bildet den Funktionswert $\mathbf{p}(z) = \mathbf{b}_{00}^{[q]}$.

Dieses Prinzip wird später auch im trivariaten Fall genutzt. Mit Hilfe der trivariaten Version des de Casteljau-Algorithmus werden die Polynome ausgewertet und darüber hinaus die Gradienten des skalaren Dichtefeldes ϕ bestimmt, um Lichtberechnungen der Isofläche durchzuführen.

Eigenschaft der konvexen Hülle

Die Bernstein-Polynome verfügen über zwei weitere Eigenschaften, welche entscheidend für die geometrische Interpretation der Kurve sind:

$$B_{ij}^q(t) \in [0,1] \quad \text{für} \quad t \in [0,1] \quad \text{und} \quad \sum_{i+j=q} B_{ij}^q = 1. \tag{4.5}$$

Aus den Gleichungen (4.1) und (4.5) kann gezeigt werden, dass die Kurve innerhalb der konvexen Hülle, die durch ihre Stützstellen aufgespannt wird, verläuft. In Abbildung 4.4 sind die konvexen Hüllen von kubischen Bézier-Kurven farblich angedeutet.

Die Eigenschaft der konvexen Hülle wird später auch im trivariaten Fall ausgenutzt. Konkret wird getestet, ob das Volumen eines Tetraederelementes einen eingestellten Isowert enthält oder nicht. Dies geschieht durch direkten Vergleich des Isowertes mit den Bernstein-Bézier-Koeffizienten.

Stetigkeiten und Spline-Kurven

Ein Nachteil der Bézier-Kurve (aller polynomialer Kurven) ist es, dass der Grad für jeden neu hinzugefügten Kontrollpunkt steigt. Um längere Kurven zu erzeugen, werden in der Praxis Bézier-Kurven niedrigen Grades zusammengesetzt. Verfügen die Übergänge der Kurven über Stetigkeitsbedingungen, so spricht man von Spline-Kurven. Sind die Übergänge stetig, dann verfügt die Spline-Kurve über C^0-Stetigkeit. Eine C^1-Spline-Kurve liegt vor, wenn darüber hinaus auch die erste Ableitung in jedem Übergang stetig ist. Bei stetiger Krümmung, sprich stetiger zweiter Ableitung, liegt C^2-Stetigkeit vor.

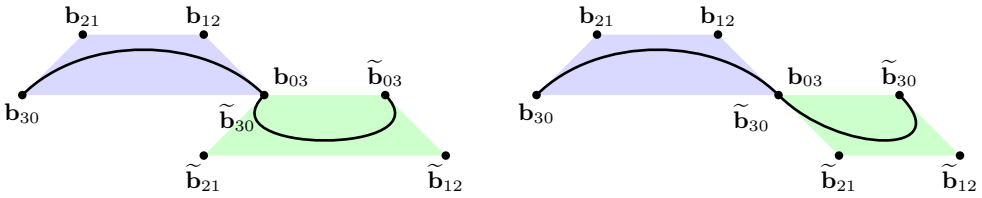


Abbildung 4.4: Stetigkeiten bei dem Übergang zweier kubischer Bézier-Kurven. *Links:* C^0-stetiger Übergang, da der Endpunkt $\mathbf{b}_{03}$ der ersten Kurve mit dem Anfangspunkt $\widetilde{\mathbf{b}}_{30}$ der zweiten Kurve übereinstimmt. *Rechts:* Ein C^1-Übergang, d.h. gleiche Steigung, da hier darüber hinaus $\mathbf{b}_{12}$, und $\widetilde{\mathbf{b}}_{21}$ auf einer Linie mit $\mathbf{b}_{03}$, $\widetilde{\mathbf{b}}_{30}$ liegen und den gleichen Abstand zum Übergang haben. Weiterhin lässt sich die Eigenschaft der konvexen Hülle (hier farblich dargestellt) erkennen: Jede Bézier-Kurve verläuft innerhalb der konvexen Hülle, welche durch die Kontrollpunkte (BB-Koeffizienten) aufgespannt wird.

Bei Spline-Kurven, welche aus zusammengesetzten Bézier-Kurven bestehen, kann man diese C^r-stetigen Übergänge durch einfache geometrische Anordnung der Kontrollpunkte erreichen. Abbildung 4.4 zeigt dies für C^0- und C^1-Stetigkeit. In [PBP02] sind Regeln zur Modellierung von C^r-Stetigkeit zu finden.

Möchte man mittels einer Spline-Kurve, bestehend aus Bézier-Kurven niedrigen Grades, eine große Menge von Punkten approximieren, so benötigt man eine gewisse Anzahl an Elementen, um bestimmte Approximationseigenschaften zu garantieren. Das Element im univariaten Fall ist die Bézier-Kurve, die auf einer Strecke definiert ist. Das Element im bivariaten Fall ist das Bézier-Dreieck und im trivariaten Fall sind es Tetraederelemente, auf denen die Polynome definiert sind. Dieses Konzept der stetigen Übergänge zwischen zwei Elementen durch einfaches Setzen der Koeffizienten wird in dieser Arbeit auch bei den trivariaten Elementen genutzt, um C^0- sowie C^1-Stetigkeit im gesamten Volumen zu garantieren.

4.3 Bézier-Dreiecke

Der bivariate Fall der Bernstein-Bézier-Form wird im Folgenden nur kurz, der Vollständigkeit halber, aufgeführt. Bézier-Dreiecke (Abb. 4.5) sind nicht zu verwechseln mit Bernstein-Bézier-Tensor-Produkt-Flächen. Letztere entstehen nämlich durch einfaches Multiplizieren zweier univariater Basen. Bézier-Dreiecke basieren auf den baryzentrischen Koordinaten bezüglich eines Dreiecks. Damit lassen sich diese auf zwei unabhängige Variablen parametrisieren und können somit Oberflächen im Raum beschreiben. Eine Anwendung finden Bézier-Dreiecke zum Beispiel bei den *Curved PN Triangles* [VPBM01]. Diese nutzen die bivariate BB-Form zur Verfeinerung eines bereits bestehenden, niedrig aufgelösten Dreiecknetzes, um glattere Formen zu erhalten (ATI TruForm [ATI01]).

Für die mathematische Beschreibung der Isoflächen in dieser Arbeit wird nicht die bivariate Form genutzt. Der vorgestellte Algorithmus beschreibt das komplette Volumen mittels trivariater Polynome und visualisiert die Isofläche durch Strahlschnitte. Es zeigt sich, dass die Polynome entlang eines Strahls zu univariaten Bézier-Kurven reduziert werden, für die der Schnitt mit der Isofläche sehr effizient bestimmt werden kann.

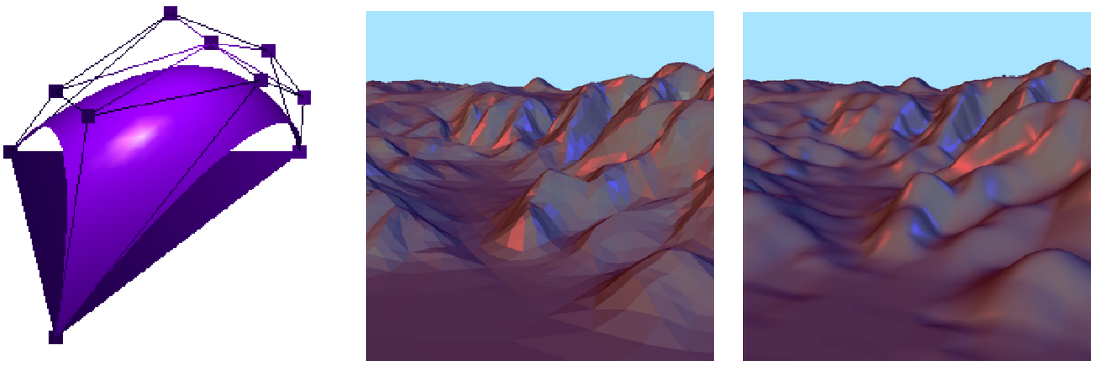


Abbildung 4.5: Die bivariate Bernstein-Bézier-Form basiert auf den baryzentrischen Koordinaten eines Dreiecks. Zusammengesetzte Bézier-Dreiecke können glatte Oberflächen beschreiben. *Links:* Kubisches BB-Patch. *Mitte:* Dreiecksnetze sind lineare, bivariate Splines mit stetigen Übergängen. *Rechts:* Eine glatte Spline-Oberfläche bestehend aus kubischen Bézier-Dreiecken. Illustration: Frank Zeilfelder.

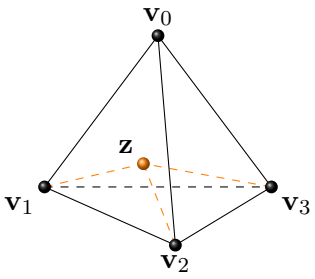


Abbildung 4.6: Baryzentrische Koordinaten bezüglich eines Tetraeders beschreiben jeden Punkt im Raum durch eine Linearkombination der 4 Eckpunkte. Die Summe der vier baryzentrischen Koordinaten ist 1, dadurch sind die Koordinaten eindeutig. Die geometrische Interpretation der Koordinate λ_0 ist das Volumenverhältnis des Tetraeders $[\mathbf{z}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ zum Tetraeder $[\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$. Die anderen drei Koordinaten besitzen eine dem entsprechende Interpretation.

4.4 Baryzentrische Koordinaten

Baryzentrische Koordinaten dienen in der Linearen Algebra dazu, Positionen bezüglich einer Strecke, eines Dreiecks oder eines höherwertigen Simplex zu beschreiben. Für die Parametrisierung der Bézier-Kurve und des Bézier-Dreiecks wurden bereits baryzentrische Koordinaten genutzt. Diese verwendeten die Strecke beziehungsweise das Dreieck als Simplex.

Beim Übergang zu trivariaten Bernstein-Bézier-Techniken werden die Koordinaten jedes Punktes $\mathbf{z} \in \mathbb{R}^3$ bezüglich der vier Eckpunkte eines Tetraeders $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ benötigt. Dies läßt sich durch eine Linearkombination ausdrücken:

$$\mathbf{z} = \sum_{i=0}^{3} \lambda_i \mathbf{v}_i. \tag{4.6}$$

Die vier Linearfaktoren λ_i bilden die baryzentrischen Koordinaten und legen jeden Punkt im Raum eindeutig fest. Zusätzlich gelten eine Reihe von Eigenschaften, z.B. die Zerlegung der Eins:

$$\sum_{i=0}^{3} \lambda_i(\mathbf{z}) = 1, \quad \forall \mathbf{z} \in \mathbb{R}^3 \tag{4.7}$$

sowie die Positivität im Inneren des Tetraeder-Elementes:

$$\lambda_i(\mathbf{z}) \geq 0, \quad \forall \mathbf{z} \in T. \tag{4.8}$$

Durch die Linearkombination (4.6) und die Zerlegung der Eins (4.7) ist gegeben, dass baryzentrische Koordinaten bezüglich eines nicht degenerierten Tetraeders ebenso wie kartesische Koordinaten über drei Freiheitsgrade verfügen. Die Transformation einer baryzentrischen Koordinate $\lambda = (\lambda_0, \lambda_1, \lambda_2, \lambda_3)^T \in \mathbb{R}^4$ in eine kartesische Koordinate $\mathbf{z} \in \mathbb{R}^3$ kann über das Gleichungssystem

$$\begin{pmatrix} \mathbf{z} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \\ 1 & 1 & 1 & 1 \end{pmatrix} \lambda \tag{4.9}$$

durchgeführt werden. Durch einfaches Umstellen dieses Gleichungssystemes kann auch λ bezüglich des Tetraeders T zu jedem Punkt $\mathbf{z}$ im Raum bestimmt werden:

$$\lambda(\mathbf{z}) = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \\ 1 & 1 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{z} \\ 1 \end{pmatrix}. \tag{4.10}$$

4.5 Trivariate BB-Form

Für die Rekonstruktion des skalaren Dichtefeldes ϕ aus Kapitel 2 werden Polynome mit drei Veränderlichen benötigt. Durch das Erweitern auf insgesamt drei unabhängige Variablen kann nun, ganz analog zur Bézier-Kurve und zum Bézier-Dreieck, die trivariate BB-Form formuliert werden. Dazu werden die Bernstein-Polynome bezüglich der baryzentrischen Koordinaten eines Tetraederelementes benötigt. Gegeben sei ein Tetraeder T und mit λ_0, λ_1, λ_2, λ_3 die assoziierten Funktionen der baryzentrischen Koordinaten. Dann ist die trivariate Bernstein-Polynombasis des Grades q bezüglich T definiert mit

$$B^q_{ijk\ell} = \frac{q!}{i!j!k!\ell!} \lambda_0^i \lambda_1^j \lambda_2^k \lambda_3^\ell \quad \text{mit} \quad i+j+k+\ell = q. \tag{4.11}$$

Weiterhin sei $B^q_{ijk\ell} = 0$ bei negativen Indizes i, j, k, ℓ. Da sich λ_0, λ_1, λ_2 und λ_3 im kartesischen Koordinatensystem durch lineare Polynome beschreiben lassen, bildet jedes $B^q_{ijk\ell}$ ein Polynom des Grades q. Es kann gezeigt werden (siehe hierzu [LS07]), dass alle Monome $\{x^\nu y^\mu z^\kappa\}_{0 \leq \nu+\mu+\kappa \leq q}$ in dem von $\mathcal{B}^q = \{B^q_{ijk\ell}\}_{i+j+k+\ell=q}$ aufgespannten Raum enthalten sind und weiterhin die Anzahl an Basisfunktionen in $\mathcal{B}^q$ gleich der Dimension $\binom{q+3}{2}$ von $\mathcal{P}_q$ ist. Somit bilden die Bernstein-Polynome genau wie die Monome eine Basis des polynomialen Vektorraumes:

$$\mathcal{P}_q := \text{span}\{\ x^\nu y^\mu z^\kappa\ :\ \nu, \mu, \kappa \geq 0,\ \nu+\mu+\kappa \leq q\}. \tag{4.12}$$

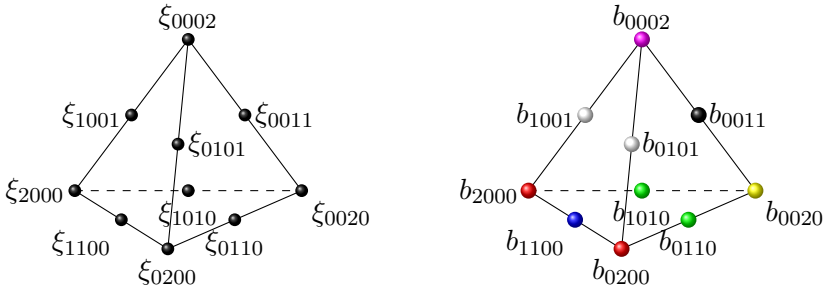


Abbildung 4.7: Die Gundpunkte (engl. Domain-Points) $\xi_{ijk\ell}$ und Bernstein-Bézier-Koeffizienten $b_{ijk\ell}$ eines quadratischen trivariaten Polynoms. Dabei beschreiben die Grundpunkte $\xi_{ijk\ell}$ die Position und die Koeffizienten $b_{ijk\ell}$ den Wert der Stützstelle. Im Gegensatz zu Monomen haben die Polynomkoeffizienten bei der BB-Form eine geometrische Interpretation. Die BB-Koeffizienten sind hier farblich kodiert, um später einen besseren Überblick über die komplexen Spline-Modelle zu bekommen.

Mit den Bernstein-Polynomen kann nun die für diese Arbeit benötigten trivariaten Polynome auf Tetraederelementen beschrieben werden durch

$$p(\lambda) = \sum_{i+j+k+\ell=q} b_{ijk\ell} B^q_{ijk\ell}(\lambda), \tag{4.13}$$

wobei $B^q_{ijk\ell}$ die obigen Bernstein-Basispolynome bezüglich T sind. Die Stützstellen $b_{ijk\ell}$ werden als die Bernstein-Bézier-Koeffizienten von p bezeichnet und die zu ihnen assoziierte Menge von Grundpunkten ist definiert mit:

$$\mathcal{D}_{q,T} := \left\{ \xi^T_{ijk\ell} := \frac{iv_0 + jv_1 + kv_2 + \ell v_3}{q} \right\}_{i+j+k+\ell=q}. \tag{4.14}$$

Die Koeffizienten $b_{ijk\ell}$ des Polynoms $p(\lambda)$ sind somit mit den jeweiligen Grundpunkten $\xi_{ijk\ell}$ für $i + j + k + \ell = q$ verknüpft. Dies ist in Abbildung 4.7 für den quadratischen Fall ($q = 2$) verdeutlicht. Die Grundpunkte $\xi_{ijk\ell}$ beschreiben die Position und $b_{ijk\ell}$ den Wert der Stützstelle. Ein Vorteil dieser Festlegung ist dadurch gegeben, dass die $\xi_{ijk\ell}$ nicht explizit gespeichert werden müssen und somit das Polynom eindeutig durch die vier Eckpunkte des Tetraeders und den $\binom{q+3}{3}$ Koeffizienten $b_{ijk\ell}$ bestimmt ist. Die Anzahl der Koeffizienten ist nur vom Grad q abhängig, wie in Abbildung 4.8 verdeutlicht wird.

Analog zum univariaten Fall werden die Koeffizienten der Endpunkte, sprich die 4 Eckpunkte des Tetraeders, interpoliert. Das bedeutet konkret $p(\lambda = (1, 0, 0, 0)^T) = b_{q000}$ für den Eckpunkt $\mathbf{v}_0$, mit gleichartigen Ausdrücken für die drei anderen Eckpunkte. Diese Eigenschaft wird in Abschnitt 4.7 genutzt, um ein approximierendes Spline-Modell zu erzeugen, bei dem die diskreten Dichtewerte aus dem Volumendatensatz auf einen Teil der Ecken der Tetraeder fallen.

Ein entscheidender Vorteil der Bernstein-Bézier-Form ist die positions- und orientierungsunabhängige Definition der Polynome. Dies liegt an der Parametrisierung der Basispolynome bezüglich der baryzentrischen Koordinaten des Tetraeders. Die Lage der Tetraedereckpunkte ist zunächst irrelevant für die Auswertung des Polynoms. Erst bei

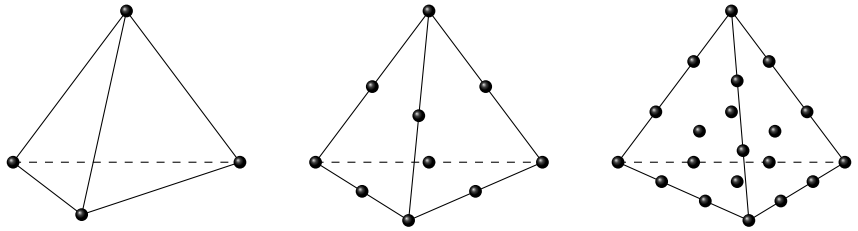


Abbildung 4.8: Tetraederelemente, auf denen trivariate Polynomfunktionen in BB-Form definiert sind, besitzen $\binom{q+3}{3}$ Stützstellen. *Links:* Lineare Elemente besitzen 4 Stützstellen. *Mitte:* Quadratische Elemente besitzen 10 Stützstellen. *Rechts:* Kubische Elemente haben 20 Stützstellen, wobei hier jeweils 1 Koeffizient in dem Baryzentrum der 4 Dreiecksflächen liegt.

der Umrechnung der baryzentrischen Koordinaten in anderen Koordinatensysteme kommen Position und Orientierung des Tetraeders zur Geltung. Die positions- und orientierungsunabhängige Beschreibung nutzt der Algorithmus gem. Kapitel 5 an vielen Stellen aus. Viele Berechnungen können so nur anhand der Koeffizienten durchgeführt werden, da die Lage der Eckpunkte des Tetraeders nicht benötigt werden.

Algorithmus von de Casteljau (trivariat)

Entsprechend zum de Casteljau-Algorithmus für Bézier-Kurven lassen sich trivariate Polynome des Grades q auf sehr ähnliche Weise auswerten. Auch hier werden insgesamt q Schritte benötigt, um $p(\lambda)$ aus (4.13) zu bestimmen. Die explizite Berechnung der Bernstein-Polynome braucht auch hierbei nicht zu erfolgen. Der Algorithmus basiert auf der einfachen rekursiven Relation der Bernstein-Polynome

$$B^q_{ijk\ell} = \lambda_0 B^{q-1}_{i-1,j,k,\ell} + \lambda_1 B^{q-1}_{i,j-1,k,\ell} + \lambda_2 B^{q-1}_{i,j,k-1,\ell} + \lambda_3 B^{q-1}_{i,j,k,\ell-1}, \tag{4.15}$$

die direkt aus der Definition von $B^q_{ijk\ell}$ in (4.11) folgt.

Für die in Kapitel 5 beschriebene Implementierung stellt der Algorithmus von de Casteljau ein wichtiges Werkzeug dar. Gegeben sei ein Auswertungspunkt $\mathbf{z} \in \mathbb{R}^3$. Für diesen werden zunächst die baryzentrischen Koordinaten

$$\lambda(\mathbf{z}) = (\lambda_0(\mathbf{z}), \lambda_1(\mathbf{z}), \lambda_2(\mathbf{z}), \lambda_3(\mathbf{z}))^T, \tag{4.16}$$

bezüglich des aktuellen Tetraeders einmalig bestimmt. Dies kann entsprechend Relation (4.10) geschehen. Die Startkoeffizienten $b^{[0]}_{ijk\ell} = b_{ijk\ell}$ werden mit den Werten der Kontrollpunkte initialisiert. Der Algorithmus berechnet nun sukzessive die Koeffizienten $b^{[m]}_{ijk\ell}$ für $m = 1, \ldots, q$ nach der Vorschrift

$$b^{[m]}_{ijk\ell} = \lambda_0 b^{[m-1]}_{i+1,j,k,\ell} + \lambda_1 b^{[m-1]}_{i,j+1,k,\ell} + \lambda_2 b^{[m-1]}_{i,j,k+1,\ell} + \lambda_3 b^{[m-1]}_{i,j,k,\ell+1} \tag{4.17}$$

mit $i + j + k + \ell = q - m$.

Der gesuchte Polynomwert $p(\lambda)$ liegt als Koeffizient $b^{[q]}_{0000}$ in der letzten Rekursionsstufe vor. Wie im univariaten Fall liefert der Algorithmus aber weitaus mehr als den

Polynomwert. Im vorletzten Schritt, sprich $b_{ijk\ell}^{[q-1]}$ mit $i+j+k+\ell=1$, liegen die Richtungsableitungen von p in $\mathbf{z}$ vor. Diese Ableitungen bilden den Gradienten der skalaren Funktion ϕ, welcher für die Lichtberechnungen in Kapitel 5.2.8 benötigt wird. Auch die höheren Ableitungen können direkt aus den berechneten Koeffizienten des de Casteljau-Algorithmus konstruiert werden. So kann beispielsweise die Stärke der Krümmung für die Visualisierung herangezogen werden (siehe [EHK*06b]).

Der in Kapitel 5 vorgestellte Algorithmus nutzt eine aus dem CAGD-Bereich bekannte Technik namens Blossoming [Sei93]. Blossoming generalisiert den de Casteljau-Algorithmus, indem die Argumente in den einzelnen de Casteljau-Schritten variieren können. Die trivariaten Polynome werden entlang von Sichtstrahlen auf univariate Polynome (Bézier-Kurven) gleichen Grades reduziert. Mit Blossoming kann die Reduzierung der Freiheitsgrade entlang der Sichtstrahlen genutzt werden, um die benötigten de Casteljau-Schritte zu minimieren.

4.6 Splines auf Tetraeder-Partitionen

Die Rekonstruktion der skalaren Funktion $\phi : \mathbb{R}^3 \to \mathbb{R}$ aus Kapitel 2 kann nun, basierend auf einem Tetraedernetz, erreicht werden, wobei auf jedem Tetraederelement ein trivariates Polynom definiert ist. Das Netz muss hierfür den zu visualisierenden Bereich lückenlos umfassen. Verfügen die Übergänge zwischen den Tetraedern weiterhin über Stetigkeitsbedingungen, spricht man von Splines auf Tetraeder-Partitionen.

Eine geeignete Definition einer Tetraeder-Partition ist in [LS07] zu finden. Die Definition besagt, dass eine Sammlung $\Delta := \{T_i\}_{i=1}^N$ von Tetraedern im $\mathbb{R}^3$ als Tetraeder-Partition eines polygonalem Gebietes $\Omega := \bigcup_{i=1}^N T_i$ bezeichnet wird, wenn jedes Tetraederpaar in Δ höchstens über einen gemeinsamen Knoten, eine gemeinsame Kante oder eine gemeinsame Dreiecksfläche verfügt. Weiterhin wichtig ist der Zusatz, dass die Partition *schälbar* ist. Eine Tetraeder-Partition Δ ist schälbar, wenn sie aus einem einzigen Tetraeder besteht oder aus einer schälbaren Partition $\widetilde{\Delta}$ konstruiert werden kann, indem ein weiterer Tetraeder hinzugefügt wird, welcher mit $\widetilde{\Delta}$ über eine, zwei oder drei gemeinsame Dreiecksflächen verfügt.

Für die Rekonstruktion wird eine Partition benötigt, welche den auf einem 3D-Gitter liegenden Volumendatensatz (das zu visualisierende Gebiet) umschließt. Im weiteren Verlauf der Arbeit bezieht sich $\Omega \in \mathbb{R}^3$ auf die Grundmenge, die den quaderförmigen Datensatz umfasst.

Trivariate Splines

Nun können die benötigten Spline-Räume wie in [LS07] definiert werden. Angenommen, Δ sei eine Tetraeder-Partition der begrenzten Grundmenge $\Omega \in \mathbb{R}^3$, weiterhin sei $\mathcal{P}_q$ der Raum der trivariaten Polynome des Grades q, dann ist der assoziierte Raum der C^r-stetigen polynomialen Splines des Grades q mit $0 \le r \le q$ auf Δ definiert mit:

$$\mathcal{S}_q^r(\Delta) := \{s \in C^r(\Omega) \ : \ s|_T \in \mathcal{P}_q, \text{ für alle } T \in \Delta\}. \tag{4.18}$$

In dieser Arbeit wird die in (4.13) beschriebe BB-Form von s verwendet:

$$s|_T = \sum_{i+j+k+\ell=q} b_{ijk\ell}\ B_{ijk\ell},\ T \in \Delta. \tag{4.19}$$

Die Grundpunkte assoziiert mit T werden genau wie in (4.14) definiert und somit ergibt sich die Menge der Grundpunkte von Δ als Vereinigung

$$\mathcal{D}_{q,\Delta} := \bigcup_{T\in\Delta} \mathcal{D}_{q,T}, \tag{4.20}$$

wobei ein Grundpunkt nur einmal hinzugefügt wird, auch wenn er zu mehreren Tetraedern assoziiert ist. Viele der so assoziierten BB-Koeffizienten fallen somit auf die selben Grundpunkte und müssen über den gleichen Wert verfügen. Es zeigt sich, dass mit dieser Definition bereits C^0-Stetigkeit gewährleistet ist, was im Folgenden genauer betrachtet wird.

Stetigkeiten

Ein großer Vorteil der BB-Form ist die einfache Festlegung von Glattheitsbedingungen zwischen zwei Polynomen der Splines auf benachbarten Tetraedern. Dies geschieht ähnlich wie im univariaten Fall durch Setzen der Koeffizienten, die auf oder nahe bei der gemeinsamen Dreiecksfläche liegen. Im Folgenden werden zwei nicht degenerierte Tetraeder $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$, $\widetilde{T} = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \widetilde{\mathbf{v}}_3]$ betrachtet, die über eine gemeinsame Dreiecksfläche $T \cap \widetilde{T} = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2]$ verfügen. Die BB-Form von $s|_T = p$ und $s|_{\widetilde{T}} = \widetilde{p}$ ist mit den korrespondierenden BB-Koeffizienten $b_{ijk\ell}$ und $\widetilde{b}_{ijk\ell}$ bestimmt. Der Übergang ist C^0-stetig über die gesamte Dreiecksfläche, wenn

$$b_{ijk0} = \widetilde{b}_{ijk0}, \quad \text{für} \quad i+j+k = q, \tag{4.21}$$

gilt. Wegen der Vereinigung der Grundpunkte (4.20) sind die trivariaten Splines (4.18) C^0-stetig. Solche C^0-stetigen Splines finden in vielen Bereichen Anwendung. So bilden sie zum Beispiel die Grundlage für die Finite Elemente Methode, die im Ingenieurwesen Verwendung findet [Web08]. Für die grafische Datenverarbeitung sind darüberhinaus oft höhere stetige Differenzierbarkeiten erforderlich. Gerade bei der Visualisierung von Oberflächen fallen unstetige Übergänge der ersten Ableitung durch sprunghafte Farbverläufe auf. Die C^1-Stetigkeit von s über der Dreiecksfläche $T \cap \widetilde{T}$ liegt genau dann vor, wenn

$$b_{ijk1} = \widetilde{b}_{i+1,j,k,0}\lambda_0(\widetilde{\mathbf{v}}_3) + \widetilde{b}_{i,j+1,k,0}\lambda_1(\widetilde{\mathbf{v}}_3) + \widetilde{b}_{i,j,k+1,0}\lambda_2(\widetilde{\mathbf{v}}_3) + \widetilde{b}_{i,j,k,1}\lambda_3(\widetilde{\mathbf{v}}_3) \tag{4.22}$$

mit $i+j+k = q-1$ gilt. Damit ergeben sich bei quadratischen Elementen 3 Gleichungen und bei kubischen Elementen 6 Gleichungen der obigen Form, die erfüllt sein müssen. Abbildung 4.9 zeigt die Koeffizienten, die bei quadratischen Elementen involviert sind, um C^0- sowie C^1-Stetigkeit zu gewährleisten.

Bei wenigen Elementen ist es einfach und geometrisch anschaulich, diese glatten Übergänge zu konstruieren. Komplizierter wird es aber, eine ganze Partition mit garantierten

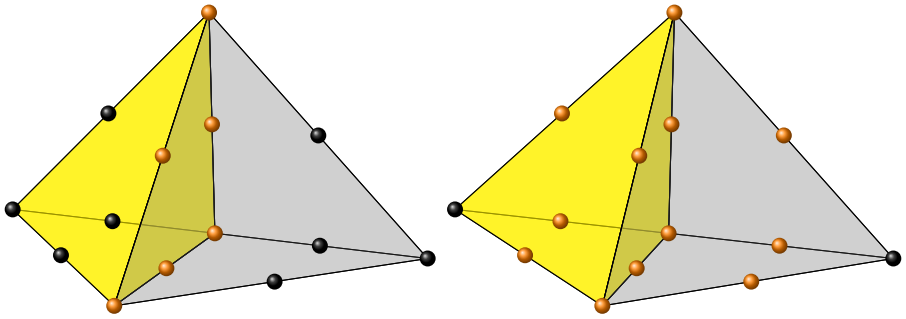


Abbildung 4.9: C^0- und C^1-Bedingungen beim Übergang zweier quadratischer Tetraederelemente mit gemeinsamer Dreiecksfläche involvieren eine gewisse Anzahl an Koeffizienten (hier orange dargestellt). *Links:* Ein C^0-Übergang erfordert gleiche Werte für die 6 Koeffizienten auf der gemeinsamen Dreiecksfläche. *Rechts:* Ein C^1-stetiger Übergang involviert zusätzlich alle Koeffizienten angrenzend zur Dreiecksfläche.

Eigenschaften zu entwickeln, da hier simultan die Bedingungen aller Übergange eingehalten werden müssen. Bei hohem Grad der Elemente ist dies noch ohne weiters möglich, da viele der Koeffizienten nur in wenigen Bedingungen involviert sind.

Für eine hochqualitative Visualisierung werden C^1-stetige Splines $\mathcal{S}_q^1$ von möglichst niedrigem Grad q benötigt. Erst der niedrige Grad ermöglicht eine effiziente Auswertung der Polynome. Durch den niedrigen Grad verfügen die Elemente über wenige Freiheitsgrade, um die Gleichungen (4.21) und (4.22) auf der gesamten Partition simultan zu erfüllen. Dies macht es schwer, bei möglichst wenigen Elementen, ein geeignetes Spline-Modell zu entwickeln. In Kapitel 4.7 werden zwei Spline-Modelle vorgestellt, von denen das quadratische Modell fast alle und das kubische Modell alle Gleichungen erfüllt und dadurch hohe Approximationseigenschaften besitzt.

(Minimal) Bestimmender Satz

Ein wichtiges Prinzip der verwendeten Splines ist der *Bestimmende Satz*. Durch die Glattheitsbedingungen verlieren die Splines an Freiheitsgraden. Dies zeigt sich darin, dass nicht alle BB-Koeffizienten frei wählbar sind. Liegt eine Menge von Koeffizienten Γ vor und lassen sich die restlichen Koeffizienten aus Bedingungen wie in (4.21) und (4.22) eindeutig bestimmen, dann ist Γ ein Bestimmender Satz. Somit gilt auch

$$\Gamma \subseteq \mathcal{D}_{q,\Delta}. \tag{4.23}$$

Für einen Spline $\mathcal{S}_q^0(\Delta)$ bildet die Menge $\mathcal{D}_{q,\Delta}$ einen Bestimmenden Satz. Liegt mindestens eine weitere Glattheitsbedingung vor, so gibt es einen Bestimmenden Satz, der weniger Koeffizienten enthält. Ein Bestimmender Satz ist minimal, wenn es keinen anderen Satz mit weniger Koeffizienten gibt.

Wie in Kapitel 5.2 zu sehen ist, bietet das Prinzip des Bestimmenden Satzes die Möglichkeit einer sehr effizienten Implementierung der Splines. Dort wird gezeigt das anhand des Bestimmenden Satzes die Eigenschaft der konvexen Hülle auf sehr wenige Operationen reduziert werden kann.

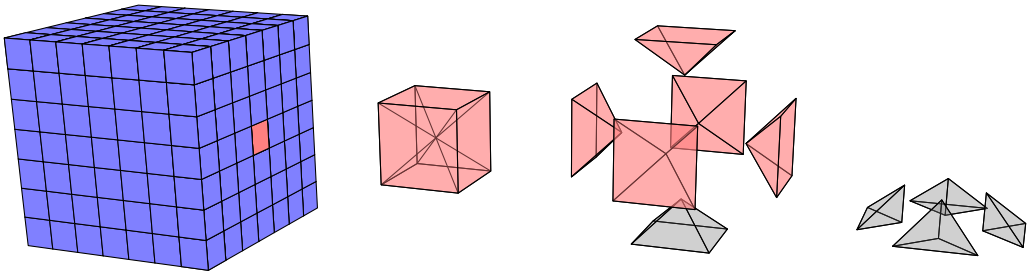


Abbildung 4.10: Eine Typ-6 Partition Δ_6 erhält man, indem jeder Quader Q in 6 Pyramiden geteilt wird. Weiterhin wird jede dieser Pyramiden in 4 Tetraeder zerlegt. Alle der so enstehenden 24 Tetraeder sind kongruent zueinander und haben im Mittelpunkt von Q einen gemeinsamen Eckpunkt.

Typ-6 Tetraeder-Partition

Die Spline-Modelle, die für diese Arbeit implementiert wurden, basieren auf der sogenannten *Typ-6 Tetraeder-Partition*, die im Weiteren als Δ_6 bezeichnet wird. Diese Partition wird aus einer Quader-Partition $\Diamond$ konstruiert, wie sie in Abbildung 4.10 zu sehen ist. Dabei wird jeder Quader $Q \in \Diamond$ in 24 kongruente Tetraeder unterteilt. Hierfür wird zunächst jeder Quadermittelpunkt $\mathbf{v}_Q$ mit den 8 Eckpunkten von Q verbunden. Dabei entstehen 6 Pyramiden, die jeweils eine der Außenflächen von Q als Grundfläche besitzen. Jede der Pyramiden wird nun in 4 Tetraeder zerteilt, so dass diese Tetraeder einen gemeinsamen Eckpunkt in $\mathbf{v}_Q$ und in der Mitte der Grundfläche der Pyramide haben. Ihren Namen trägt die Partition, da man mittels 6 Schnittebenen den Würfel in seine 24 Tetraeder zerschneiden kann.

In Abbildung 4.11 wird gezeigt, wie sich die Grundpunkte der Splines für die Typ-6 Partition organisieren lassen. Die Punkte liegen in Ringen $\mathcal{R}_\nu,\ \nu = 0, \ldots, q$ um den Mittelpunkt $\mathbf{v}_Q$ des Quaders, wobei $\mathcal{R}_0$ aus einem Grundpunkt besteht und genau mit dem Mittelpunkt $\mathbf{v}_Q$ übereinstimmt. Die Grundpunkte des äußeren Ringes $\mathcal{R}_q$ liegen alle auf dem Rand des Quaders.

4.7 Volumendaten approximierende Splines

Im Folgenden werden konkret zwei Spline-Schemata vorgestellt, welche einen Volumendatensatz unter Glattheitsbedingungen approximieren. Die skalaren Werte f_{ijk} aus den Volumendaten sind assoziiert mit den Gitterpositionen $\mathbf{x}_{ijk} = (h_x\ i, h_y\ j, h_z\ k)^T \in \mathbb{R}^3$ für $i = 1, \ldots, n_x,\ j = 1, \ldots, n_y$ und $k = 1, \ldots, n_z$. Dabei liegt jeder Volumendatenwert genau auf dem Mittelpunkt eines Quaders Q aus $\Diamond$, demnach $\mathbf{x}_{ijk} = \mathbf{v}_{Q_{ijk}}$. Die Skalierung (h_x, h_y, h_z) und die Größe (n_x, n_y, n_z) des Gitters können beliebig gewählt werden, da in der Praxis meistens keine isotropen Datensätze vorliegen.

Mit der obigen Assoziation der Volumendatenwerte zur Partition ist die Geometrie aller Tetraeder der Typ-6 Partition festgelegt. Approximierende Splines $\mathcal{S}$ auf Δ_6 werden nun eindeutig bestimmt durch direktes Setzen der Bernstein-Bézier-Koeffizienten. Hier wird das Prinzip des Bestimmenden Satzes ausgenutzt und jeweils nur ein Quader $Q \in \Diamond$ mit seinen 24 Tetraedern betrachtet. Diese Unterteilung in Quader erfolgt auch bei der

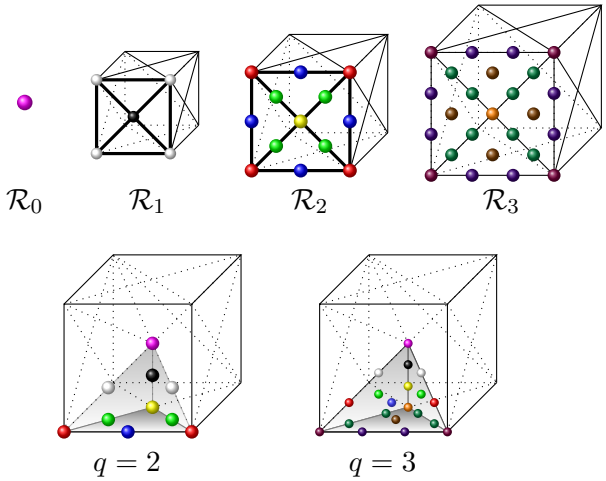


Abbildung 4.11: Die Grundpunkte sowie ihre assoziierten Koeffizienten von Splines in Bernstein-Bézier-Form auf einer Typ-6 Tetraeder-Partition lassen sich in Ringen organisieren. Dabei bestehen Splines des Grades $q = 2$ aus den Ringen $\mathcal{R}_0$, $\mathcal{R}_1$ und $\mathcal{R}_2$. Kubische Splines verfügen zusätzlich über alle Grundpunkte des Ringes $\mathcal{R}_3$. Die beiden unteren Bilder zeigen die farbliche Kodierung der Koeffizienten eines ausgezeichneten Tetraeders $T \in \Delta_6$. Beim quadratischen Element sind dies 10 und beim kubischen Element 20 Koeffizienten.

Implementierung, die in Kapitel 5 vorgestellt wird. Ausreichend ist die Festlegung der Koeffizienten b_ξ des Bestimmenden Satzes $\Gamma_{q,Q}$, welche sich jeweils aus einer gewichteten Summe eines lokal begrenzten Stempels des Datensatzes ergeben:

$$b_\xi = \sum_{\mathbf{x}_{ijk} \in \Theta_\xi} \omega_{ijk} \; f_{ijk}. \tag{4.24}$$

Hier ist Θ_ξ eine Teilmenge der Volumendaten, die sehr nahe bei ξ liegen. Dabei sind ω_{ijk} feste Anteile, mit denen die Werte f_{ijk} gewichtet werden.

Die Koeffizienten der Bestimmenden Sätze pro Quader Q beider Spline-Modelle berechnen sich aus einem Stempel, der die am nächsten gelegenen 27 Datenwerte benötigt. Für einen Quader Q mit dem Mittelpunkt $\mathbf{x}_{ijk}$ werden der Datenwert f_{ijk} sowie alle 26 um ihn liegenden Werte benötigt. In Abbildung 4.12 ist dieser 27er-Stempel verdeutlicht.

Eine große Schwierigkeit ist das Bestimmen der Gewichte ω sowie des Bereiches Θ_ξ, die gemeinsam den Stempel für den Volumendatensatz bilden, damit Approximationseigenschaften sowie die sehr wichtigen Glattheitsbedingungen garantiert werden. Weiterhin wichtig für eine hochwertige und effiziente Implementierung sind: kleine Datenstempel, Symmetrie, garantierte Approximationsordnung und Stabilität. Dies alles liefern die quadratischen Super-Splines und die kubischen C^1-Splines.

Quadratische Super-Splines

In dieser Arbeit wird ein quadratisches Spline-Schema genutzt, welches von Zeilfelder und anderen in [RZNS03] erstmalig vorgestellt wurde und in [NRSZ05], [RZNS04] und

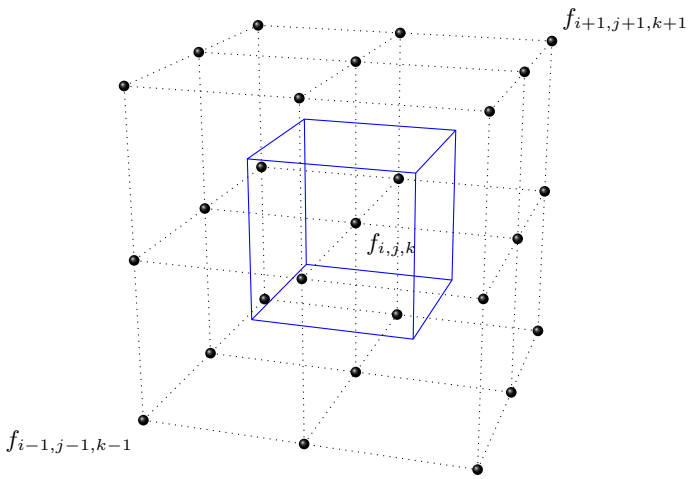


Abbildung 4.12: Die 27 benötigten Volumendatenwerte zur Berechnung der BB-Koeffizienten des Bestimmenden Satzes $\Gamma_{q,Q}$ sind schwarz markiert. Der Wert f_{ijk} liegt genau im Mittelpunkt des Quaders Q, die anderen 26 Werte liegen auf einem Quader um Q herum.

[SZH*05] bereits erfolgreich verwendet wurde. Diese quadratischen Splines basieren auf der Typ-6 Partition und besitzen C^0-Stetigkeit im gesamten Volumen. Zusätzlich verfügen sie über C^1-Stetigkeit an großen Teilen der Übergänge der Tetraederelemente, daher folgt auch ihre Bezeichnung „super“ (näheres dazu ist [LS07] zu entnehmen). Genauer gesagt sind die Übergänge aller Seitenflächen der Quader Q aus $\Diamond$ glatt. An den Bereichen, an denen die Freiheitsgrade im Inneren von Q nicht ausreichten, um C^1-Stetigkeiten zu gewährleisten, wurden einige der nötigen Koeffizientenrelationen gelockert. Dies geschah durch Mitteln der Glattheitsbedingungen, um Symmetrie zu garantieren. Somit könnte man dieses Spline-Modell auch als quadratische C^0 Super-Splines bezeichnen.

Der Raum der quadratischen Super-Splines bezüglich Δ_6 ist definiert mit:

$$\begin{aligned}\mathcal{S}_2(\Delta_6) = \{s \in C^0(\Omega) \ : \ & s|_T \in \mathcal{P}_2, \text{ für alle } T \in \Delta_6, \text{ und} \\ & s \text{ ist glatt in } v, \text{ für alle Knoten } v \text{ von } \Diamond\}.\end{aligned}$$

In dieser Arbeit wird die in (4.19) beschrieben BB-Form der Splines auf Δ_6 verwendet. Jeder Spline $s \in \mathcal{S}_2$ hat somit folgende Darstellung:

$$s|_T = \sum_{i+j+k+\ell=2} b_{ijk\ell}\ B^2_{ijk\ell},\ T \in \Delta_6. \tag{4.25}$$

Quadratische Polynome auf Tetraedern in BB-Form verfügen über 10 Koeffizienten. Aufgrund der Partition resultieren wegen Überschneidungen statt 240 lediglich 65 Koeffizienten pro Quader Q. Die Super-Splines besitzen einen Bestimmenden Satz $\Gamma_{2,Q}$ von 20 Koeffizienten. Diese liegen auf den acht Eckpunkten sowie den zwölf Mittelpunkten der Kanten des Quaders. Der Bestimmende Satz bildet sich demnach aus einer Teilmenge der Koeffizienten des zweiten äußeren Ringes:

$$\Gamma_{2,Q} \subset \mathcal{R}_2 \quad \text{und} \quad |\Gamma_{2,Q}| = 20. \tag{4.26}$$

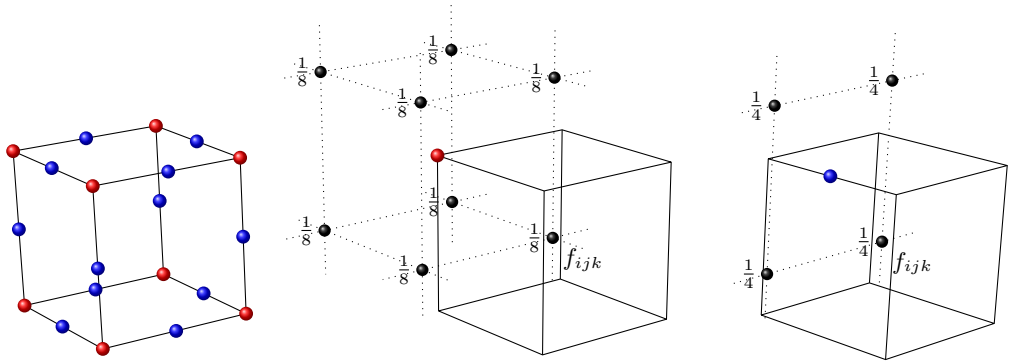


Abbildung 4.13: Gewichteter Stempel zur Berechnung des Bestimmenden Satzes $\Gamma_{2,Q}$ der quadratischen Super-Splines. *Links:* Der Bestimmende Satz besteht aus den Koeffizienten der 8 roten Eckpunkte und der 12 blauen Kantenmittelpunkte von Q. *Mitte:* Die roten Koeffizienten berechnen sich jeweils aus den acht angrenzende Volumendatenwerten. *Rechts:* Die blauen Koeffizienten benötigen jeweils die am nächsten gelegenen 4 Werte aus dem Datensatz.

Abbildung 4.13 zeigt, wie sich die 20 Koeffizienten aus dem 27er-Datenstempel berechnen.

Die verbleibenden 45 Koeffizienten lassen sich aufgrund der Bedingungen aus (4.21) und (4.22) über den Bestimmende Satz $\Gamma_{2,Q}$ jederzeit berechnen. Die geschieht lediglich durch Mitteln von Koeffizienten aus $\Gamma_{2,Q}$ und ist in Abbildung 4.14 bildlich dargestellt.

Kubische C^1-Splines

Zusätzlich zum obigen quadratischen Schema wurde für diese Arbeit auch ein kubisches Spline-Schema implementiert. Die verwendeten kubischen Splines wurden in [SZ07] erstmalig von Zeilfelder und Sorokina beschrieben und in [KZ08] erstmalig für interaktive Visualisierungen verwendet. Diese Splines verfügen über C^1-Stetigkeit im gesamten Volumen. Sie basieren genau wie die verwendeten quadratischen Splines auf der Typ-6 Partition.

Der Raum der kubischen C^1-Splines in Bezug auf Δ_6 ist definiert mit

$$\mathcal{S}_3(\Delta_6) = \{s \in C^1(\Omega) \ : \ s|_T \in \mathcal{P}_3, \text{ für alle } T \in \Delta_6\}, \tag{4.27}$$

wobei $\mathcal{P}_3$ den Raum der trivariaten Polynome 3. Grades aufspannt.

Auch hier wird die stückweise Bernstein-Bézier-Form für jeden Spline $s \in \mathcal{S}_3$ verwendet

$$s|_T = \sum_{i+j+k+\ell=3} b_{ijk\ell}\ B^3_{ijk\ell},\ T \in \Delta_6. \tag{4.28}$$

Kubische Polynome in BB-Form bezüglich eines Tetraeders benötigen 20 Stützstellen. Aufgrund Überschneidungen der Grundpunkte resultieren hier 175 Koeffizienten pro Quader. Die kubischen C^1-Splines verfügen über einen Bestimmenden Satz $\Gamma_{3,Q}$, bestehend aus 56 Koeffizienten, welche alle auf dem äußeren drittem Ring liegen:

$$\Gamma_{3,Q} \subset \mathcal{R}_3 \quad \text{und} \quad |\Gamma_{3,Q}| = 56. \tag{4.29}$$

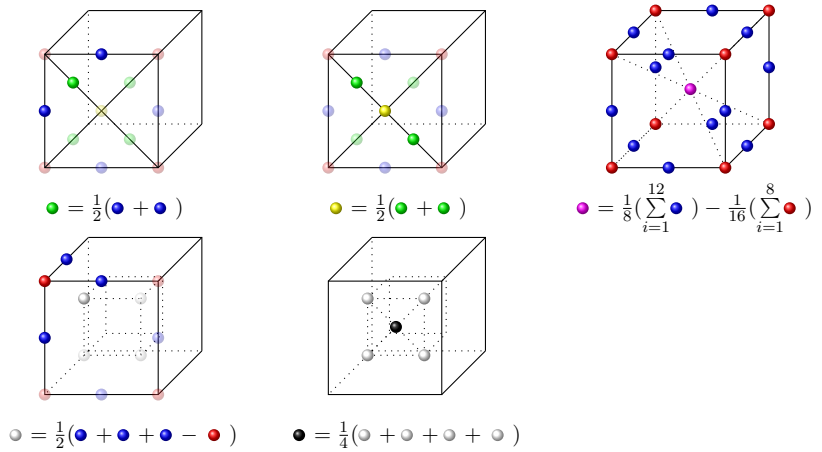


Abbildung 4.14: Aus dem Bestimmenden Satz $\Gamma_{2,Q}$ der quadratischen Super-Splines lassen sich die verbleibenden 45 Koeffizienten durch wiederholtes Mitteln effizient berechnen. So ergeben sich die grünen Koeffizienten durch Mitteln der beiden am nächsten gelegenen blauen Koeffizienten, die gelben durch Mitteln zweier grüner. Die Regeln zu Bestimmung der Koeffizienten des inneren Ringes $\mathcal{R}_1$ sind *unten links* sowie *unten Mitte* dargestellt. Der zentrale Koeffizient wird durch eine gewichtete Summe des gesamten Bestimmenden Satzes bestimmt.

Dieser Bestimmende Satz berechnet sich auch bei den verwendeten kubischen Splines aus einem 27er-Stempel. Abbildung 4.15 zeigt die Gewichtung der angrenzenden Volumendatenwerte zur Berechnung von $\Gamma_{3,Q}$.

Die verbleibenden 119 Koeffizienten lassen sich auch bei den kubischen Splines wegen der einzuhaltenden Glattheitsbedingungen aus dem Bestimmenden Satz $\Gamma_{3,Q}$ jederzeit berechnen. Diese Regeln sind in Abbildung 4.16 für Teile des dritten und zweiten Ringes zusammengefaßt. Die verbleibenden Koeffizienten berechnen sich genau wie beim quadratischen Schema, wie in Abbildung 4.14 zu sehen ist.

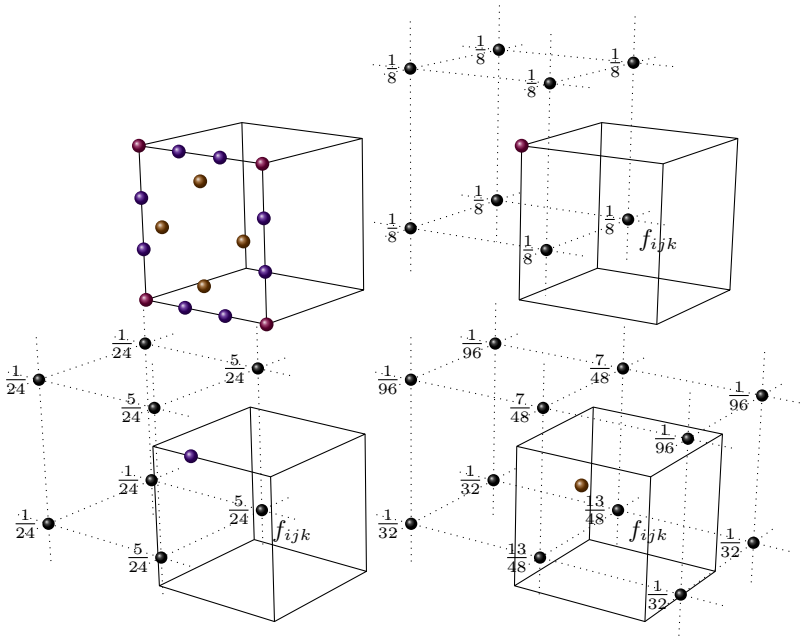


Abbildung 4.15: Der Bestimmende Satz $\Gamma_{3,Q}$ der verwendeten kubischen Splines besteht aus 56 Koeffizienten. Sie liegen alle auf dem äußeren Ring und sind *oben links* für die vordere Seite von Q in Magenta, Lila und Braun dargestellt. Die magenta- sowie die lilafarbenen Koeffizienten benötigen jeweils 8 Datenwerte des 27er-Stempels. Die braunen Koeffizienten ergeben sich durch eine gewichtete Summe von insgesamt 12 Datenwerten.

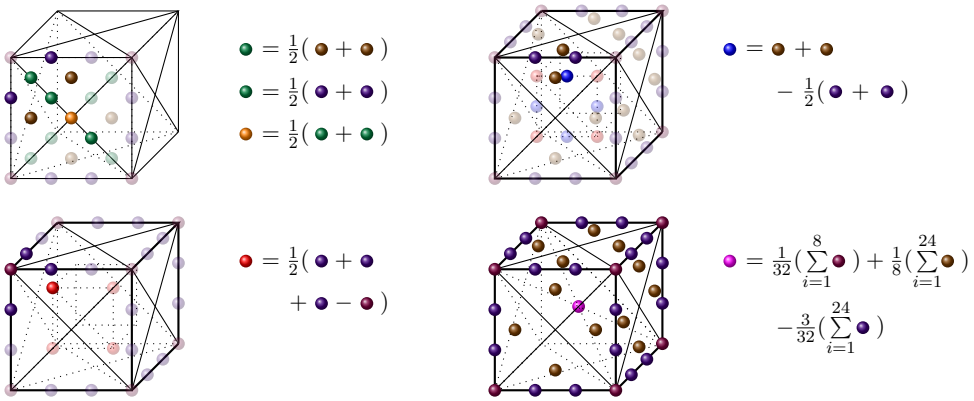


Abbildung 4.16: Aus dem Bestimmenden Satz $\Gamma_{3,Q}$ lassen sich die verbleibenden 119 Koeffizienten des verwendeten kubischen Spline-Schemas jederzeit bestimmen. Dies geschieht durch wiederholtes Mitteln der Koeffizienten. Die Abbildung *oben links* zeigt die Regeln zur Bestimmung der fehlenden Koeffizienten des äußeren dritten Ringes $\mathcal{R}_3$. *Oben rechts* sowie *unten links* sind die benötigten Mittelungen zur Bestimmung der roten sowie der blauen Koeffizienten des Ringes $\mathcal{R}_2$ dargestellt. Der zentrale Koeffizient berechnet sich wie *unten rechts* beschrieben. Die fehlenden Koeffizienten berechnen sich genau wie bei dem quadratischen Spline-Schema, wie in Abbildung 4.14 bereits gezeigt.

Kapitel 5

Ein massiv paralleler Algorithmus

Das Entwicklungspotential der Informatik liegt in der Parallelität [ABC*06]. Die Taktfrequenzen von Prozessoren können nur noch schwer gesteigert werden und ein Zuwachs an Rechenleistung ist nur durch zunehmende Parallelisierung der Prozessoren möglich. Die Transistoranzahl aktueller Grafikchips (GPUs) übersteigt die der CPUs [And09]. Die Anzahl an Gigaflops von GPUs übertrifft aktuelle 4-Kern-Prozessoren um das Zehnfache [NVI09a] (Abbildung 5.1). Grafikkarten haben einen Entwicklungsvorteil, denn die Grafikpipeline basierte schon immer auf parallel arbeitenden Streaming-Prozessoren.

Die Entwicklung von Software für aktuelle Prozessoren, welche über 2, 4 oder 8 Kerne verfügen, ist mittels handgestrickter Threads, die gemeinsame Ressourcen teilen, gerade noch möglich. Doch in einigen Jahren, wenn Prozessoren über hunderte von Kernen verfügen, müssen neue Programmierkonzepte bereitstehen, um die weiterhin stark wachsende Rechenleistung verfügbar zu machen. Große Hard- und Softwarehersteller bereiten sich auf diese Tatsache vor und stellen teilweise schon jetzt Plattformen zur Verfügung. Intel arbeitet an einem Programmiermodell namens Ct [INT07, INT09], um die Multicore-Prozessoren der kommenden Generationen nutzbar zu machen. AMD stellt mit dem Stream-Computing-SDK [AMD09] eine Programmierumgebung zur Verfügung, um die parallel arbeitenden Prozessoren von ATI-Grafikkarten für allgemeine Berechnungen zugänglich zu machen. NVidia bietet mit der CUDA-Technologie eine Möglichkeit, die enorme Rechenleistung der Grafikkarte als Coprozessor zur Untersützung der CPU zu nutzen. Auch die Entwicklung eines offenen, plattformunabhängigen Standards für parallele Programmierung unter dem Namen OpenCL (Open Computing Language [Khr09]) wird vorangetrieben.

In dieser Arbeit wird CUDA im Zusammenspiel mit OpenGL verwendet, um eine interaktive Isoflächen-Rekonstruktion aus Volumendaten zu ermöglichen.

5.1 Compute Unified Device Architecture (CUDA)

Die Rechenleistung von Grafikkarten wird schon länger für nicht grafikspezifische Berechnungen genutzt [OJL*07, GPG]. Mit der CUDA-API sowie der Einführung des G80-Chips stellte NVidia die Architektur der Grafikkarte um und ermöglichte somit eine einfachere Programmierung der parallel arbeitenden Streaming-Prozessoren für allgemeine Berechnungen. Die GPU wird bei CUDA als Device bezeichnet und stellt einen Coprozessor dar, welcher unterstützend zum Host (CPU) arbeitet. Das Device verfügt über seinen eigenen Speicher und führt die Berechnungen mittels CUDA-Threads durch,

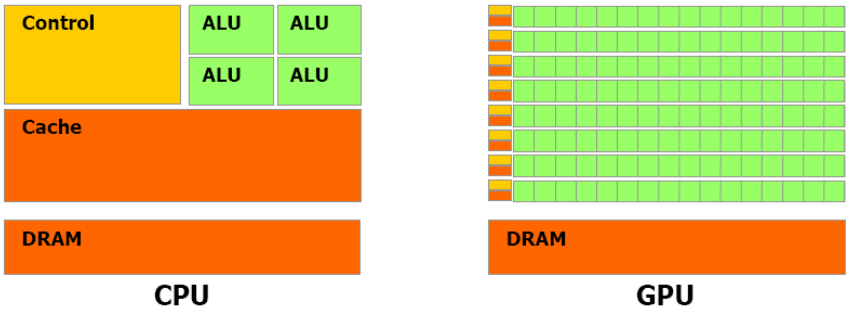


Abbildung 5.1: Bei der CPU wird ein Großteil der Transistoren für Datenflusskontrolle und Cache verwendet. Bei der Grafikkarte werden die Transistoren hauptsächlich für die reine Datenverarbeitung genutzt. Dadurch besitzen GPUs deutlich mehr Gigaflops als CPUs. Grafikkarten sind für hochparallele, berechnungsintensive Anwendungen konzipiert. Illustration: NVidia [NVI09a].

welche im Gegensatz zu CPU-Threads extrem leichtgewichtig sind. Die Applikation wird in datenparallele Portionen zerlegt und durch sogenannte Kernel-Aufrufe in Form von tausenden von CUDA-Threads parallel ausgeführt.

Die GPU besteht aus Multiprozessoren, welche jeweils eine SIMD-Architektur (Single Instruction, Multiple Data) besitzen. Die NVidia Grafikkarte GTX 280, welche für die Zeitmessungen in Kapitel 7 genutzt wurde, verfügt zum Beispiel über 240 Prozessoren, wobei jeweils acht Stück einen der 30 Multiprozessoren bilden. Bei jedem Taktzyklus führt ein Multiprozessor die gleichen Instruktionen für eine Gruppe von Threads aus. Diese Threads sind in *Blöcken* organisiert und werden intern zusätzlich in sogenannte *Warps* unterteilt. Ein Multiprozessor ist zuständig für sämtliche Berechnungen eines Blocks. Dabei springt er zwischen den Warps hin und her, um die Latenzzeiten der Lese- und Schreibzugriffe der einzelnen Threads zu kompensieren. Die Threads des Blockes können über einen Shared Memory, der schnelle Zugriffszeiten wie die der Prozessorregister besitzt, Daten austauschen und somit kommunizieren.

Die Berechnungen eines Kernels werden durch viele dieser Blöcke durchgeführt, um alle Multiprozessoren auszulasten. Diese Blöcke organisieren sich zu einem *Grid* (siehe Abbildung 5.2). Damit liegt ein Konzept vor, bei welchem ein einziger Kernel-Aufruf tausende von Threads startet. Für die vollständige Auslastung der Ressourcen ist nicht der Programmierer verantwortlich, denn die CUDA-Runtime übernimmt die Verteilung der Prozessorarbeit je nach Device, indem möglichst viele Blöcke parallel berechnet werden. Damit ist eine gute Skalierung auch für einen älteren Code auf zukünftigen GPUs gewährleistet.

Zusammenspiel mit OpenGL

Eine sehr effiziente Möglichkeit, Daten für eine OpenGL-Anwendung bereitzustellen, sind *Buffer-Objects*. Ein Buffer-Object ist zunächst nicht mehr als ein Array von Bytes, das im Grafikkartenspeicher liegt. Ein Vertex-Buffer-Object (VBO) beschreibt die Geometrie und zugehörige Daten wie Farbwerte oder Texturkoordinaten. Ein Pixel-Buffer-Object

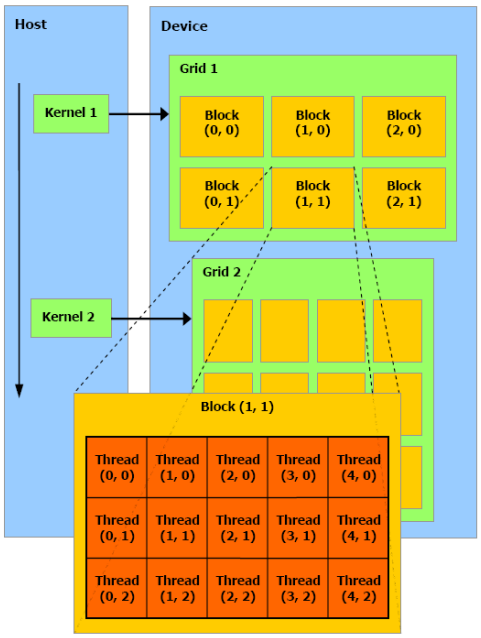


Abbildung 5.2: Bei CUDA müssen die Berechnungen in datenparallele Teilstücke zerlegt werden. Diese Berechnungen werden durch Kernel-Aufrufe gestartet und in Form von tausenden leichtgewichtigen Threads parallel durchgeführt. Jeder Thread führt das gleiche Programm auf unterschiedlichen Daten aus. Mittels einer eindeutigen Thread- und Block-ID erkennt der Thread, auf welchen Daten er operiert. Illustration: NVidia [NVI09a].

(PBO) bietet Beschleunigung für Pixel-Befehle und enthält reine Pixel-Daten, die beliebige Werte repräsentieren können. Um die Daten in OpenGL zu verwenden, wird ein Buffer-Object, welchem zuvor eine eindeutige ID zugeordnet wurde, für eine Reihe von OpenGL-Befehlen aktiviert. Die nun folgenden Befehle verwenden die Daten des Buffer-Objects. Der Grafikkartentreiber verwaltet die Objekte und bietet beispielsweise Möglichkeiten, besonderes schnell Texturen aus PBOs zu erstellen. Dies nutzt der in diesem Kapitel beschriebene Algorithmus für den Datentransfer von CUDA nach OpenGL.

Mittels der CUDA-API ist es möglich, ein bereits bestehendes OpenGL-Buffer-Object über den CUDA-Adressraum anzusprechen. Zunächst muss das Buffer-Object in CUDA registriert werden. Ein sogenanntes Mapping liefert dann den Zeiger auf die Daten des Objekts. Mit diesem Zeiger haben CUDA-Threads direkten Lese- und Schreibzugriff auf die Daten des Buffer-Objects.

5.2 Der Algorithmus

Entscheidend für eine effiziente GPU-Implementierung ist ein minimaler Transfer von Daten zwischen Host und Device sowie das Einteilen der Arbeit, damit die Hardware zu jedem Zeitpunkt ausgelastet ist. Der folgende Algorithmus, dessen Vorgehen in Abbildung 5.3 dargestellt ist, ermöglicht dies durch Auslagerung sämtlicher Berechnungs-

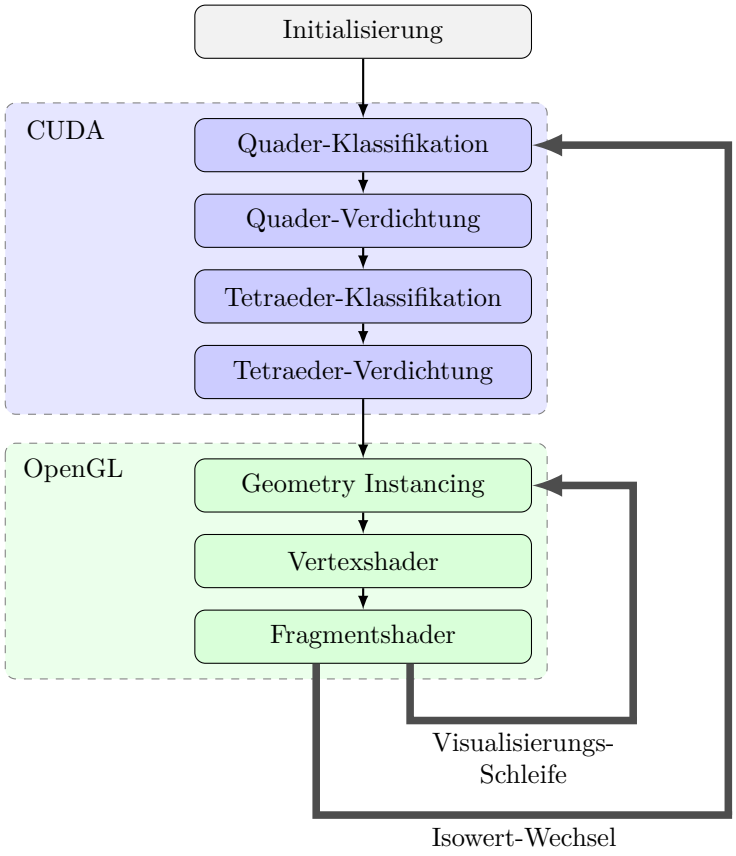


Abbildung 5.3: Das Diagramm zeigt den Aufbau des entwickelten Algorithmus. Die Visualisierungs-Schleife sorgt für eine interaktive Darstellung des aktuell eingestellten Isowertes. Erfolgt ein Wechsel des Isowertes, werden durch eine Reihe von CUDA-Kernels sämtliche Vorberechnungen für die Visualisierung mittels Instancing durchgeführt.

schritte auf die Grafikkarte.

Bei diesem parallelen Algorithmus ist CUDA zuständig für die schnelle Bestimmung der Tetraederelemente, welche Teile der Isofläche I_c enthalten. Dazu werden zunächst alle Quader $Q \in \Diamond$ daraufhin geprüft, ob Teile der Fläche I_c enthalten sind. Dies ist mit dem Bestimmenden Satz $\Gamma_{q,Q}$ und Ausnutzen der Eigenschaft der konvexen Hülle möglich. Die so deutlich reduzierte Menge von Quadern wird anschließend in Tetraeder unterteilt. Nur diese Tetraederelemente werden auf mögliches Enthalten der Isofläche I_c überprüft. Es zeigt sich (siehe Kapitel 7), dass durchschnittlich 11,8 der 24 Tetraeder pro Quader Teile der Isofläche enthalten. Von den Tetraedern, die I_c enthalten, wird der linear kodierter Quaderindex in einer Textur abgespeichert, wobei für jeden der 24 Tetraedertypen eine eigene Textur erstellt wird. Mittels *Geometry Instancing* werden die Hüllen der einzelnen Tetraeder gezeichnet bzw. gerastert. Vertex- und Fragmentshader-Programme visualisieren die Isofläche durch Ray-Casting (Abbildung 5.4). Die Bernstein-Bézier-Koeffizienten werden im Vertexshader mit Hilfe der 3D-Volumendatentextur *on-the-fly* bestimmt. Abschnitt 5.4 stellt eine für hohe Bildraten optimierte Algorithmus-Variante vor, die nur

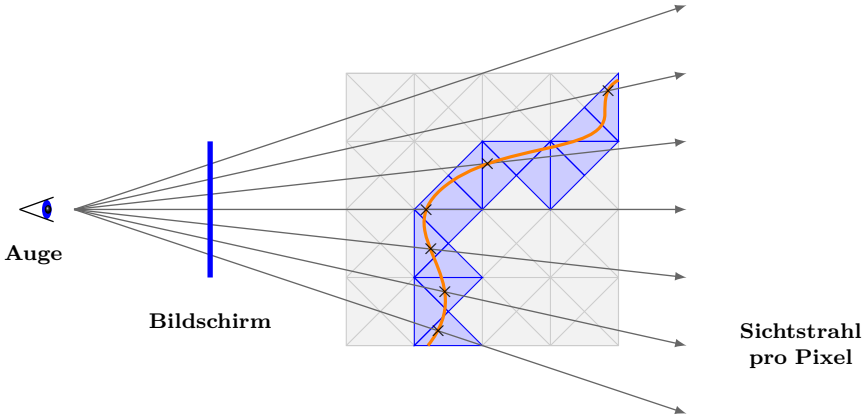


Abbildung 5.4: Vereinfachte Betrachtungsweise des verwendeten Ray-Castings: Nur jene Tetraeder (hier als blaue Dreiecke dargestellt), welche die Isofläche I_c enthalten, werden mit den Sichtstrahlen auf Schnitt getestet.

minimal längere Rekonstruktionszeiten benötigt. Hier werden für alle Quader, welche I_c enthalten, die Bestimmenden Sätze $\Gamma_{q,Q}$ von einem CUDA-Kernel vorberechnet. So kann im Vertexshader mit wenigen Texturzugriffen der Satz $\Gamma_{q,Q}$ geladen und die fehlenden Koeffizienten bestimmt werden.

Da die quadratischen Super-Splines und die kubischen C^1-Splines sehr ähnlich aufgebaut sind, beinhaltet der Algorithmus kaum Unterschiede für die beiden Spline-Modelle. Selbstverständlich liegt ein Unterschied im Polynomgrad und der damit verbundene Anzahl an Koeffizienten. Es werden beispielsweise für das Ausnutzen der Eigenschaft der konvexen Hülle bei den kubischen Splines mehr Koeffizienten herangezogen und für den Casteljau-Algorithmus ein weiterer Schritt benötigt. Die einzig unterschiedliche Behandlung der Splines wird im Fragmentshader durchgeführt. Bei den quadratischen Super-Splines wird der Schnittpunkt mit der Isofläche entlang der Sichtstrahlen exakt als Nullstellen-Problem gelöst. Bei den kubischen Splines wird der Schnittpunkt aus Gründen der Performance mittels des Newton-Verfahrens ermittelt.

Effizienz durch Shiften

Im Verlauf des Algorithmus werden die 3D-Gitterpositionen der Quader Q_{ijk} auf eindimensionale Werte kodiert. Der CUDA-Befehlssatz sowie GLSL-Shader-Programme unterstützen seit Einführung des Shader-Modell 4.0 [LB08] Integer-Datentypen und ermöglichen in Hardware realisierte bitweise Operationen (Shift, AND, OR, etc.). Um dies effizient auszunutzen, vergrößert der Algorithmus (nur) für die Kodierung den darzustellenden Datensatz in jeder Dimension auf die nächste Zweierpotenz. Somit kann jeder Gitterpunkt mit wenigen Shift-Operationen und bitweisen Oder-Operationen auf einen ganzzahligen Wert abgebildet werden. Dies bringt erhebliche Speichereinsparungen mit sich.

5.2.1 Initialisierung

CUDA bietet verschiedene Möglichkeiten, Daten als Texturen abzulegen und diese durch hardwareunterstützte Befehle zu verwenden. Anstatt einer auf CUDA-Arrays basierenden 3D-Textur wird für den Volumendatensatz eine eindimensionale Textur im linearen Speicher verwendet. Die Textureinheiten, welche für die Filterung der Texturdaten und für unterschiedliche Addressierungsmethoden verantwortlich sind, werden nicht benötigt. Der Volumendatensatz wird in den Grafikspeicher geladen. Dabei bleibt die Bitbreite (8 Bit, 16 Bit, 32 Bit-Float) und somit auch die Größe des Datensatzes bestehen. Das Umrechnen auf Float ist nicht nötig. Dadurch wird Speicherplatz gespart. CUDA bietet bei Texturzugriffen ein hardwareunterstütztes Umrechnen auf normalisierte, im Intervall $[0, 1]$ liegende Fließkommazahlen an. Damit nutzt der Algorithmus die datensatzunabhängige Darstellung aller im aktuellen Volumen enthaltenen Isoflächen:

$$I_c \in \Omega \quad \Rightarrow \quad c \in [0, 1]. \tag{5.1}$$

Der Algorithmus verwendet Pixel-Buffer-Objects, um Daten von CUDA nach OpenGL weiterzugeben. Die benötigen PBOs werden einmalig erstellt, was Performance Vorteile bringt. Alle Lookup-Tabellen werden auf den GPU-Speicher transferiert und nehmen nur marginalen Speicherplatz ein. Der Vertexshader benötigt zur Berechnung der BB-Koeffizienten Zugriff auf den Volumendatensatz. Zur Zeit ist es leider (noch) nicht möglich, dieselbe Textur in OpenGL-Shadern sowie in CUDA zu nutzen. Dies hat den Grund, dass der Volumendatensatz ein zweites Mal als 3D-OpenGL-Textur im Grafikkartenspeicher abgelegt wird. Der Vertexshader der Algorithmus-Variante, welche in Abschnitt 5.4 vorgestellt wird, benötigt keinen Zugriff auf den Volumendatensatz. Hier muss der Datensatz nur einmalig im Grafikkartenspeicher liegen.

5.2.2 Quader-Klassifikation

Die Partition der verwendeten Spline-Modelle unterteilt das Volumen zunächst in Quader Q_{ijk}. Im ersten Schritt des Algorithmus wird überprüft, welche Quader Teile der Isofläche enthalten. Um dies zu entscheiden, wird die Eigenschaft der konvexen Hülle auf den gesamten Quader angewandt.

Sind alle Bernstein-Bézier-Koeffizienten des Tetraeders kleiner oder größer als der Isowert, so steht aufgrund der Eigenschaft der konvexen Hülle fest, dass der Isowert nicht in dem eingeschlossen Volumen vorkommt. Der Bestimmende Satz $\Gamma_{q,Q}$ des Quaders legt die Koeffizienten der 24 Tetraeder fest: alle Koeffizienten werden durch Mitteln der Koeffizienten von $\Gamma_{q,Q}$ bestimmt. Aufgrund der Regeln zur Mittelung (siehe Abbildung 4.14 und 4.16) kann keiner der Tetraeder-Koeffizienten kleiner beziehungsweise größer als die Koeffizienten aus $\Gamma_{q,Q}$ werden. Das Berechnen der Koeffizienten aller 24 Tetraeder ist nicht notwendig, um die Eigenschaft der konvexen Hüllen auf dem gesamten Quader anzuwenden.

Bei der Umsetzung wird ein CUDA-Kernel zur Klassifikation der Quader verwendet. Der Kernel startet für jeden Quader einen Thread, welcher für die Berechnungen eines Quaders zuständig ist. Der Thread liest den benötigen 27er-Datenstempel aus der 1D-Textur und berechnet daraus im quadratischen Fall die 20 und im kubischen Fall die 56

Koeffizienten des Bestimmenden Satzes. Der kleinste und der größte Koeffizient werden mit einer Schleife über die Koeffizienten bestimmt und mit dem aktuellen Isowert c verglichen. Liegt der Isowert nicht zwischen beiden Koeffizienten, so liegt kein Teil der Fläche I_c im Quader.

Jeder Thread führt diese Berechnungen nun unabhängig voneinander aus und schreibt eine Flag (0 oder 1) in ein zuvor reserviertes, lineares Integer-Array $\mathbf{Q}_{class}$, welches für jeden Quader über einen Eintrag verfügt. Ist I_c im Quader enthalten, schreibt der Thread eine 1, ist I_c nicht enthalten, eine 0. Das dabei entstehende, unsortierte Array $\mathbf{Q}_{class}$ muss weiterverarbeitet werden, um eine komprimierte Liste der Quader zu erhalten, die zur Grenzfläche beitragen. Dieser Schritt wird im folgenden als *Quader-Verdichtung* bezeichnet.

5.2.3 Quader-Verdichtung mit paralleler Präfixsumme

Die Allgemeine Präfixreduzierung (oder Scan-Operation) [HSJ08] verwendet einen binär assoziierten Operator $\oplus$ mit der Identität I und eine Liste von n Elementen

$$[a_0, a_1, \ldots, a_{n-1}],$$

als Ergebnis liefert sie eine neue Liste:

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

Im Fall der Präfixsumme (der Addition für den Operator $\oplus$) und der Liste

$$[1\ 3\ 5\ 0\ 1\ 7\ 9],$$

liefert die Präfixsumme beispielsweise folgende Ergebnisliste:

$$[0\ 1\ 4\ 9\ 9\ 10\ 17].$$

Die Präfixsumme ist ein gutes Beispiel für einen grundsätzlich sequentiell erscheinenden Algorithmus, für den es aber auch effiziente parallele Varianten gibt [Ble89, Ble90]. Die Scans mit ihren verschiedenen binären Operatoren (Minimum, Maximum, Multiplikation, bitweise Vergleiche, etc.) bilden Grundbausteine für viele parallele Algorithmen, die unter anderem sortieren, komprimieren oder hierarchische Datenstrukturen erstellen [SHZO07]. Der Algorithmus nutzt eine parallele Version der Präfixsumme zur Verdichtung der Ergebnisse aus der Quader-Klassifikation. Die verwendete Implementierung [HOS*09] stammt aus der CUDPP-Bibliothek (CUDA Data Parallel Primitives Library), welche im CUDA-SDK enthalten ist. Die implementierte Variante besitzt, genau wie der sequentielle Algorithmus, einen Aufwand von $O(n)$, nutzt dabei aber die parallele Rechenleistung.

Aus der unsortierten Liste $\mathbf{Q}_{class}$ der Klassifikation muss eine Liste der Quader erstellt werden, die zur Isofläche I_c beitragen. Dazu wird eine parallele Präfixsumme mit $\mathbf{Q}_{class}$ als Eingabe durchgeführt. Durch die simple Addition der Nullen und Einsen wird eine Ergebnisliste $\mathbf{Q}_{scan}$ erstellt, welche die Indizes der Speicherpositionen für die Verdichtung

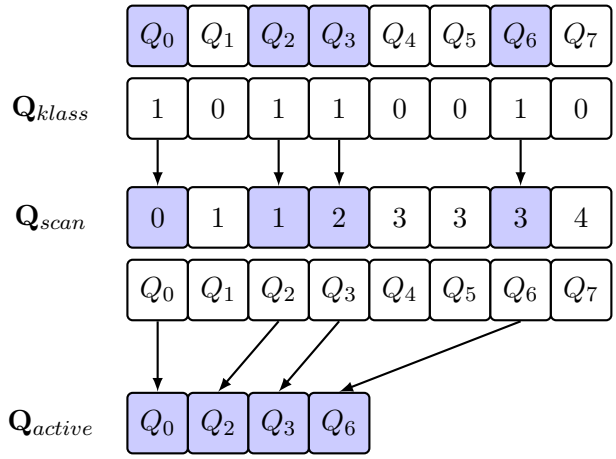


Abbildung 5.5: Die Quader-Verdichtung erfordert im ersten Schritt eine Präfixsumme, um die Zieladressen für den Komprimierungs-Kernel zu bestimmen. Die Präfixsumme hat als Eingabe das Array $\mathbf{Q}_{class}$ und liefert als Ergebnis $\mathbf{Q}_{scan}$. Im zweiten Schritt wird der Komprimierungs-Kernel gestartet. Dieser schreibt für alle Quader, die I_c enthalten, den Quaderindex an die Position, die durch die Präfixsumme berechnet wurde. Illustration nach [HSJ08].

liefert. Weiterhin kann die Anzahl der aktiven Quader a direkt abgelesen werden. Sie bildet sich aus der Summe des letzten Eintrages von $\mathbf{Q}_{scan}$ und $\mathbf{Q}_{class}$, da eine exklusive Präfixsumme genutzt wird.

Nun steht die Anzahl aktiver Quader fest und ein Array $\mathbf{Q}_{active}$ der Größe a kann reserviert werden. Für die Verdichtung wird ein Komprimierungs-Kernel mit einem Thread pro Quader $Q \in \Diamond$ gestartet. Der Thread kontrolliert, ob sein ihm zugewiesener Quader die Fläche enthält, folglich $\mathbf{Q}_{class}$ eine 1 an der entsprechenden Position hat. Nur wenn dies der Fall ist, wird von dem Thread der linear codierte Index des Quaders in das Array $\mathbf{Q}_{active}$ geschrieben, und zwar genau an die Position, die mittels der Präfixsumme in $\mathbf{Q}_{scan}$ bestimmt wurde. Dieses Vorgehen wird in Abbildung 5.5 verdeutlicht.

5.2.4 Tetraeder-Klassifikation

Die Tetraeder-Klassifikation ermittelt aus der deutlich reduzierten Quaderliste $\mathbf{Q}_{class}$ diejenigen Tetraederelemente, welche I_c enthalten. Da der Algorithmus für ein effizientes Nutzen des Instancing getrennte Listen für jeden der Tetraedertypen benötigt, werden 24 Arrays $\mathbf{T}_{class,i}$ der Größe a für die Ergebnisse des Tetraeder-Klassifikations-Kernels reserviert.

Für die Klassifikation aller Tetraeder ist ein einziger Kernel zuständig. Der Kernel startet für jeden aktiven Quader einen Thread. Der Thread liest den linear codierten Quaderindex aus dem Array $\mathbf{Q}_{active}$, um zu bestimmen, für welchen Quader Q_{ijk} die 24 Tetraeder klassifiziert werden. Anhand des linearen Quaderindex wird die Gitterposition bestimmt und der 27er-Datenstempel aus der Volumendatentextur geladen. Aus den 27 Datenwerten werden alle 65 bzw. 175 BB-Koeffizienten berechnet, wobei effizient vorgegangen wird, indem häufige Faktoren nur einmalig bestimmt werden. Für die Tetraeder

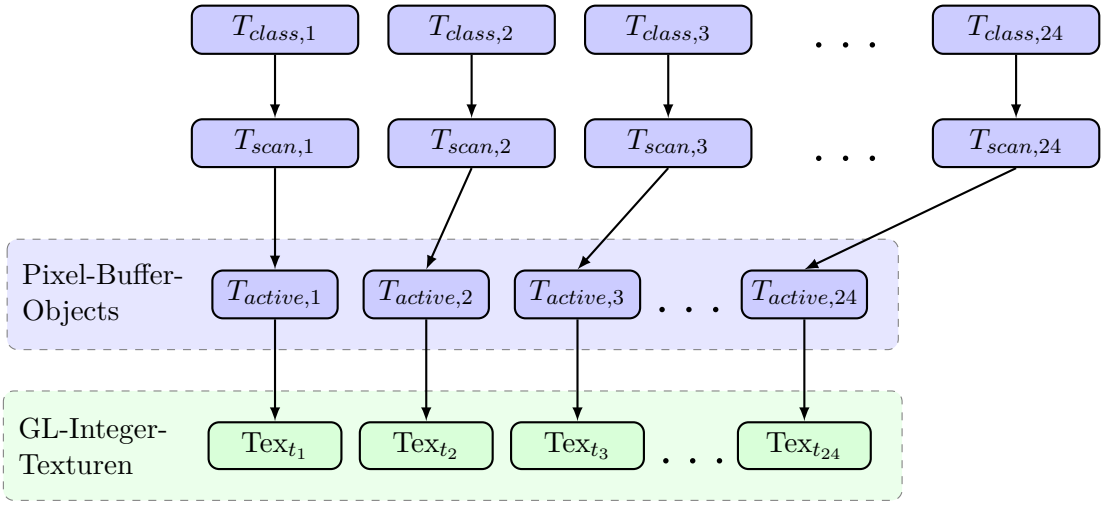


Abbildung 5.6: Zu jedem der 24 Arrays $\mathbf{T}_{class,i}$ wird mit Hilfe einer parallelen Präfixsumme das Array $\mathbf{T}_{scan,i}$ erstellt, welches die Adressen für die Verdichtung liefert. Die Ergebnisse der Verdichtung werden direkt in die Pixel-Buffer-Objects geschrieben. Mit den PBOs wird für jeden der 24 Tetraedertypen eine OpenGL-Integer-Textur erstellt. Jeder Eintrag der Textur Tex_{t_i} entspricht einem Tetraeder des Typs i und enthält als Wert den linear codierten Quaderindex.

$T_1, T_2, \ldots, T_{24}$ wird nun der Reihe nach die Eigenschaft der konvexen Hülle genutzt, um zu testen, ob I_c in T_i enthalten ist. Je nach Spline-Modell werden so jeweils 10 bzw. 20 der Koeffizienten benötigt. Welche der 65 bzw. 175 Koeffizienten den aktuellen Tetraeder T_i festlegen, wird mittels einer Lookup-Tabelle, welche in einer linearen Textur abgelegt ist, bestimmt. Ist c kleiner oder größer als alle Koeffizienten von T_i, so wird eine 0, ansonsten eine 1, in $\mathbf{T}_{class,i}$ an die Position des Threads geschrieben. Die 24 Arrays $\mathbf{T}_{class,i}$ müssen anschließend komprimiert werden.

5.2.5 Tetraeder-Verdichtung mit paralleler Präfixsumme

Die Verdichtung jedes der 24 Arrays $\mathbf{T}_{class,i}$ erfolgt sehr ähnlich zur Quader-Verdichtung über eine Präfixsumme und einem Komprimierungs-Kernel. Der einzige Unterschied liegt darin, dass der jeweilige Komprimierungs-Kernel die Ergebnisse direkt in das Datenarray des entsprechenden Pixel-Buffer-Objects schreibt. Dazu wird vor dem Aufruf des Kernels der PBO in den CUDA-Adressraum *gemapped*, um einen Zeiger auf seine Daten zu erhalten. Die Tetraeder-Verdichtung ist in Abbildung 5.6 dargestellt.

Für alle 24 Tetraedertypen T_i führt der Algorithmus eine parallele Präfixsumme mit $\mathbf{T}_{class,i}$ als Eingabeliste durch. Die Präfixsumme liefert das Array $\mathbf{T}_{scan,i}$ mit den Adresspositionen für den Komprimierungs-Kernel. Die Anzahl an aktiven Tetraedern t_i des jeweiligen Typs ergeben sich aus der Summe des letzten Eintrages von $\mathbf{T}_{class,i}$ und $\mathbf{T}_{scan,i}$. Ein Komprimierungs-Kernel, der für jeden der Tetraedertypen aufgerufen wird, startet einen Thread pro aktiven Quader. Der Thread prüft, ob die Isofläche in T_i enthalten ist, folglich $\mathbf{T}_{class,i}$ eine 1 enthält. Nur wenn dies der Fall ist, schreibt er den linear codierten

Quaderindex in ein Array $\mathbf{T}_{active,i}$ an die Position, welche über die Präfixsumme ermittelt wurde. Hierbei ist $\mathbf{T}_{active,i}$ das Datenarray des entsprechenden Pixel-Buffer-Objects, das von OpenGL kontrolliert wird.

Aus den Pixel-Buffer-Objects $\mathbf{T}_{active,i}$ werden nun OpenGL-Texturen Tex_{t_i} erstellt, um die benötigten Daten für die Shader-Programme bereitzustellen. Der Algorithmus verwendet 2D-Texturen, da 1D-Texturen auf herkömmlicher, aktueller Hardware auf 8192 Einträge begrenzt sind. 2D-Texturen erlauben es dem Algorithmus, auch große Datensätze darzustellen. Der OpenGL-Treiber übernimmt das Kopieren der Daten vom PBO zur Textur. Dies geschieht asynchron, somit muss nicht auf das vollständige Erstellen der Textur gewartet werden. Jeder Eintrag der Textur Tex_{t_i} entspricht einem Tetraeder des Typs i und enthält als Wert den linear codierten Quaderindex. Mit den Texturen liegen nun alle Daten zur Visualisierung der aktuellen Isofläche I_c mittels Instancing vor.

5.2.6 Geometry Instancing

Bei grafischen Anwendungen ist es oft nötig, ein aus Polygonen bestehendes Objekt wiederholt für dasselbe Bild zu zeichnen. Dabei verfügt jede Instanz des Objektes über dieselbe Geometrie, unterliegt aber unterschiedlichen Translationen, Rotationen, Skalierungen oder greift auf verschiedene Texturen zu. Um Speicherplatz zu sparen und ein programmierbares Design zu erhalten, nutzten Software-Entwickler schon immer Techniken, um viele solcher Geometrieinstanzen zu einem sogenannten *Batch* zusammenzufassen [Car05]. Je nach verwendeter Technik können so auch die Anzahl an Zeichenaufrufen, welche einen starken Einfluß auf die Performance haben [Wlo03], reduziert werden. Auch Grafik-APIs bieten solche Techniken an (beispielsweise Pseudo-Instancing [NVI]). Lange Zeit gab es aber keine effiziente Möglichkeit, Objekte mit sehr wenigen Polygonen, die über gleiche Geometrie verfügen, aber andere unterschiedliche Parameter besitzen, mehrere tausend mal pro Bild zu visualisieren.

Seit Shader-Modell 4.0, welches in DirectX 10.0 [Bly06] und ab OpenGL 2.0 über Extensions verfügbar ist, unterstützen Grafikkarten hardwarebasiertes Instancing. Für die Zeichenaufrufe der Geometrie nutzt der Algorithmus das als Extension einzubindende, hardwarebasierte OpenGL-Geometry-Instancing [Gol08]. Diese Extension ermöglicht das Zeichnen vieler Instanzen einer Geometrie mit einem einzigen Zeichenaufruf und stellt eine *Instance-ID*-Variable im Vertexshader-Programm bereit, um pro Instanz Berechnungen durchzuführen. Dieses echte Hardware-Instancing bringt dem Algorithmus zwei sehr große Vorteile: die Anzahl von OpenGL-Funktionsaufrufen wird deutlich reduziert (24 Aufrufe pro Bild) und die Geometrie der 24 Tetraeder muss nur einmal im Grafikkartenspeicher vorliegen.

Der Algorithmus legt die Geometrie von jedem der 24 Tetraedern in Form eines Triangle-Strips in einem Vertex-Array im Grafikkartenspeicher ab. Zu jedem Typ wird ein Vertex-Buffer-Object (VBO) assoziiert. Der umschließende Quader Q mit den 24 Tetraedern ist auf den Einheitswürfel $[-1/2, 1/2]^3$ skaliert. Die endgültige Translation und Skalierung in Objektkoordinaten erfolgt im Vertexshader. Jeder Eckpunkt $\mathbf{v}$ im Vertex-Array enthält neben den x-, y- und z-Koordinaten noch einen vierten Eintrag $i = 0, 1, 2, 3$, welcher festlegt, um welchen der 4 Eckpunkte $\mathbf{v}_i$ des Tetraeders es sich

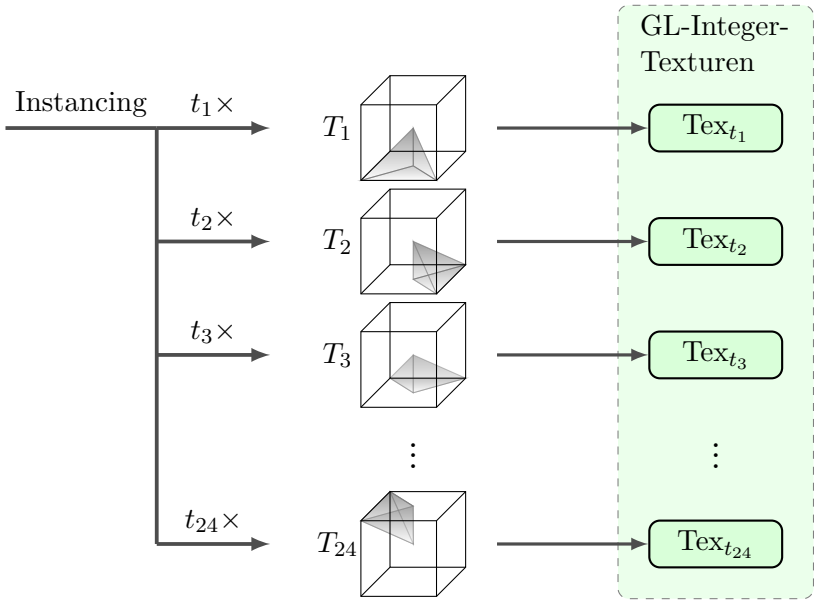


Abbildung 5.7: Mit insgesamt 24 Instancing-Zeichenaufrufen werden die Dreiecke aller Tetraeder, welche I_c enthalten, gezeichnet. Aus der Tetraeder-Komprimierung ist die Anzahl an Tetraedern t_i jedes Typs bekannt. Die OpenGL-Texturen Tex_{t_i} enthalten die Indizes der Quader, aus denen der Vertexshader die benötigten Translationen der Eckpunkte berechnet.

handelt.

Für jedes Bild der Visualisierungs-Schleife werden die Triangle-Strips aus den Vertex-Buffer-Objects nacheinander mit einer Instancing-Variante des OpenGL-Befehls *glDrawArrays* gezeichnet. Dieser Befehl (*glDrawArraysInstanced*) verfügt über einen zusätzlichen Parameter, welcher die Anzahl an Instanzen festlegt. Auf diese Art werden mit 24 Aufrufen und den Parametern t_i die Dreiecke aller Tetraeder gezeichnet (siehe Abbildung 5.7). Der Vertexshader hat Zugriff auf die Instance-ID, die intern von 0 bis $t_i - 1$ hochzählt und jedem Eckpunkt den aktuellen Tetraeder zuordnet.

5.2.7 Vertexshader

In der Standard-Grafikpipeline ist der Vertexshader für die Transformation der Eckpunkte der Dreiecke sowie für die Lichtberechnung pro Eckpunkt zuständig. Auch in diesem Algorithmus werden in einem Vertexshader-Programm (eigentlich in sehr vielen, siehe Abschnitt 5.3) die Eckpunkte $\mathbf{v}_i$ der Dreiecke transformiert. Zusätzlich werden die benötigten Bernstein-Bézier-Koeffizienten berechnet und Teile der Sichtstrahl-Parameter bestimmt, welche mittels der Rastereinheiten unter linearer Interpolation an die einzelnen Fragmente (Pixel) weitergereicht werden.

Das Geometry Instancing stellt im Vertexshader-Programm die Instance-ID-Variable bereit. Mit der fortlaufenden ID wird festgelegt, zu welchem Tetraeder der aktuelle Eckpunkt gehört, indem Texturkoordinaten berechnet werden, die jeder Instance-ID einen

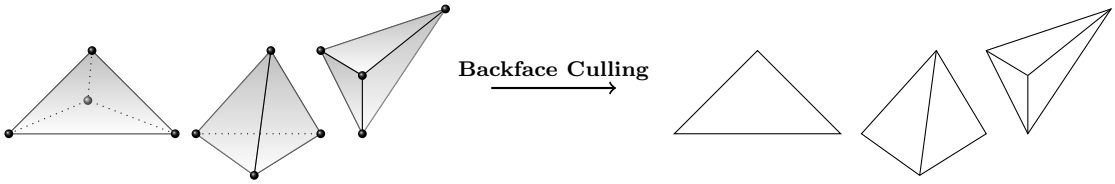


Abbildung 5.8: Die Tetraederelemente, welche I_c enthalten, werden durch das Zeichnen der vier Dreiecksflächen visualisiert. Backface-Culling sorgt für ein Aussortieren der nicht sichtbaren Außenflächen, je nach aktueller Perspektive und Orientierung. Somit werden nur eine, zwei oder drei der insgesamt vier Dreiecksflächen gezeichnet.

eigenen Eintrag aus Tex_{t_i} zuordnen. Der Wert des Eintrag liefert nach wenigen Shift- und bitweisen Oder-Operationen den 3D-Index des Quaders Q_{ijk}, zu dem der aktuelle Eckpunkt gehört. Damit steht auch die benötigte Translation $\mathbf{v}_Q = (i, j, k)^T$ fest.

Auf die Eckpunktkoordinaten $\mathbf{v}_i$ wird die Translation $\mathbf{v}_Q$ addiert und mit der konkatenierten *Modelview*- und *Projection*-Matrix, welche auch die entsprechende Skalierung beinhaltet, multipliziert. Die resultierenden homogenen Clipping-Koordinaten werden an den nicht programmierbaren Teil der Pipeline weitergegeben. Das Clippen der Dreiecke sowie Backface-Culling (Abbildung 5.8), sprich das Aussortieren derjenigen Dreiecke, die dem Betrachter nicht zugewandt sind, übernimmt die Standard-Pipeline.

Um im Fragmentshader die univariaten Polynome von $s|_T$ entlang von Sichtstrahlen zu bestimmen, werden weitere Daten benötigt. Dazu müssen die baryzentrischen Koordinaten von zwei Punkten auf dem Strahl vom Auge durch $\mathbf{v}$ ermittelt werden. Der Algorithmus verwendet den Punkt $\mathbf{v}$ sowie den Punkt $\bar{\mathbf{v}} = \mathbf{v} + (\mathbf{v} - (\mathbf{e} - \mathbf{v}_Q))/||\mathbf{v} - (\mathbf{e} - \mathbf{v}_Q)||$, wobei $\mathbf{e}$ die Augpunktkoordinaten und $\mathbf{v}_Q$ die Quader-Translation in Weltkoordinaten sind. Die baryzentrischen Koordinaten von $\mathbf{v}$ bezüglich T lassen sich aus dem Index $i = 0, 1, 2, 3$ ablesen, welcher als vierter Eintrag neben den x,y,z Koordinaten im Vertex-Array vorliegt. Um $\lambda(\bar{\mathbf{v}})$ zu bestimmen, muss, wie in Gleichung (4.10) in Kapitel 4.4 beschrieben, eine Multiplikation mit einer 4×4-Matrix durchgeführt werden. Aufgrund der Typ-6 Partition existieren 24 solcher Matrizen. Diese sind vorberechnet und werden dem jeweiligen Shader-Programm bereitgestellt (siehe Abschnitt 5.3). Die baryzentrischen Koordinaten von $\mathbf{v}$ und $\bar{\mathbf{v}}$ werden schließlich als *varying*-Variable an den Rasterisierungsprozess weitergereicht.

Die 10 bzw. 20 Bernstein-Bézier-Koeffizienten des aktuellen Tetraeders werden *on-the-fly* aus dem Volumendatensatz berechnet. Es zeigt sich, dass hierzu nur 23 der 27 Werte aus dem Datenstempel benötigt werden. Welche der 23 Datenwerte dies sind, ist vom Tetraedertyp abhängig. Aufgrund der Symmetrie von $\Gamma_{q,Q}$ können über Permutationen der Werte alle 24 Fälle beschrieben werden. Der entsprechende Vertexshader liest die 23 Werte aus der 3D-Textur. Eine Möglichkeit, die Tetraeder-Koeffizienten zu bestimmen, wäre es nun, die benötigten Teile des Bestimmenden Satzes zu berechnen und die Koeffizienten über die Regeln aus Kapitel 4.7 zu bestimmen. Der Algorithmus verwendet eine optimierte Variante, welche mehrmalig vorkommenden Faktoren nur einmalig bestimmt. So können viele Rechenoperationen eingespart werden. Die 10 bzw. 20 Bernstein-Bézier-

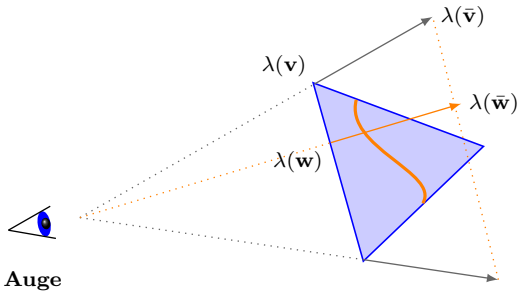


Abbildung 5.9: Für jeden Tetraedereckpunkt $\mathbf{v}$, welcher den Vertexshader durchläuft, wird ein Punkt $\bar{\mathbf{v}}$ bestimmt und die baryzentrischen Koordinaten $\lambda(\mathbf{v})$, $\lambda(\bar{\mathbf{v}})$ ermittelt, die dann als *varying*-Variable weitergeben werden. Bei der Rasterisierung werden diese baryzentrischen Koordinaten über das Dreieck interpoliert. Das Fragmentshader-Programm erhält die interpolierten Koordinaten $\mathbf{w}$, $\bar{\mathbf{w}}$, die den Sichtstrahl für den aktuellen Pixel parametrisieren.

Koeffizienten werden als *varying*-Variable an den Fragmentshader weitergereicht. Die Bestimmung der BB-Koeffizienten ist neben der Anzahl an aktiven Tetraedern der limitierende Faktor für die Bildraten der Visualisierung. Bei einer Implementierung muss darauf geachtet werden, die Shader-Instruktionen auf ein Minimum zu reduzieren (siehe Kapitel 7).

5.2.8 Fragmentshader

Die Funktion des Fragmentshader ist es, für jedes Fragment f einen Punkt $\mathbf{w}_c \in \mathbb{R}^3$ mit $s(\mathbf{w}_c) = c$ zu bestimmen, der am nächsten zum Augpunkt liegt und sich auf dem Sichtstrahl $\mathbf{r}$ befindet. Als weitere Bedingung muss der Punkt im Tetraeder liegen. Ist solch ein gültiger Schnittpunkt $\mathbf{w}_c$ mit der Isofläche I_c gefunden, wird die Oberflächennormale $\mathbf{n}_c$ in $\mathbf{w}_c$ bestimmt, um Lichtberechnungen durchzuführen. Liegt kein Schnittpunkt vor, wird f frühzeitig verworfen. Für die Sichtbarkeit (Verdeckungsberechnung) des aktuellen Pixels wird der *Z-Buffer*-Algorithmus verwendet.

Alle Dreiecke des Tetraeders, welche dem Betrachter zugewandt sind, werden in Bildschirmkoordinaten transformiert und gerastert. Die Rasterisierung liefert jedem resultierenden Fragment f die baryzentrischen Koordinaten von $\mathbf{w}$ und $\bar{\mathbf{w}}$ und legt damit den Sichtstrahl $\mathbf{r}(t) = \mathbf{w} + t(\bar{\mathbf{w}} - \mathbf{w}),\ t \in \mathbb{R}$ fest. Das trivariate Polynom s des Grades q wird entlang des Sichtstrahl auf ein univariates Polynom $p(t)$ gleichen Grades reduziert. Um den ersten Schnittpunkt von $\mathbf{r}$ mit der Isofläche zu finden (sprich ein möglichst kleines $t_c \geq 0$ zu bestimmen), so dass $s(\mathbf{r}(t_c)) = c$ gilt, wird eine geeignete Darstellung des univariaten Polynoms $p(t) = s(\mathbf{r}(t))$ mit $t \geq 0$ benötigt. Der Algorithmus nutzt *Blossoming* [Sei93], um die BB-Form von $p(t) = \sum_{i+j=q} b_{ij} B_{ij}^q(t)$ mit $t \in [0, 1]$ zu bestimmen. Blossoming beinhaltet den de Casteljau Algorithmus und generalisiert diesen so, dass die Argumente in den einzelnen de Casteljau Schritten variieren können. Blossoming nutzt so die Reduzierung der Freiheitsgrade von $p(t)$ zur Minimierung der Anzahl an nötigen de Casteljau-Schritten.

Aus der Rasterisierung ergeben sich für jedes Fragment f die beiden baryzentrischen

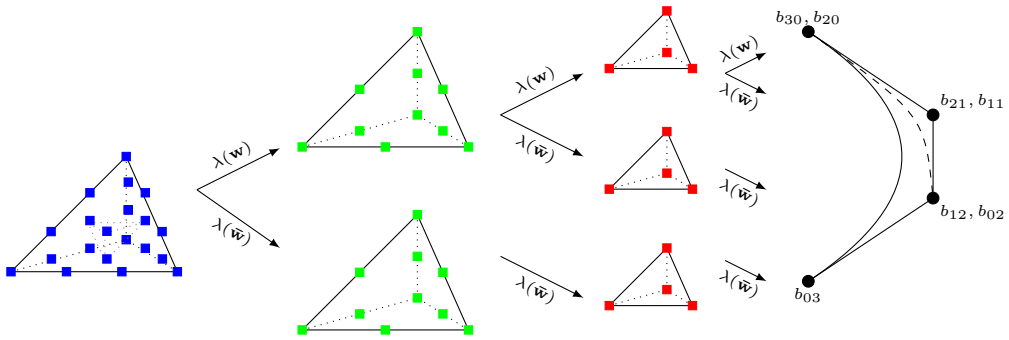


Abbildung 5.10: Der Algorithmus nutzt trivariates Blossoming, um das univariate Polynom entlang von Sichtstrahlen $\mathbf{r}(t) = \mathbf{w} + t(\bar{\mathbf{w}} - \mathbf{w}),\ t \in \mathbb{R}$ zu bestimmen. Blossoming generalisiert den de Casteljau Algorithmus, indem die Argumente in den einzelnen de Casteljau Schritten variieren können. So wird die Reduzierung der Freiheitsgrade des trivariaten Polynoms entlang des Sichtstrahls genutzt, um Berechnungen einzusparen. Die Abbildung zeigt das Schema bei den kubischen Splines.

Koordinaten $\lambda(\mathbf{w})$ und $\lambda(\bar{\mathbf{w}})$. Der Vertexshader liefert die BB-Koeffizienten $b_{ijk\ell}$, die als Start-Koeffizienten $b^{[0]}_{ijk\ell}$ für das Blossoming dienen. Diese Start-Koeffizienten sind in Abbildung 5.10 für $q = 3$ blau markiert. Die Abbildung zeigt die Schritte des de Casteljau-Algorithmus sowie die variierenden Argumente $\lambda(\mathbf{w})$ und $\lambda(\bar{\mathbf{w}})$. Im ersten Schritt werden die hier grün dargestellten Koeffizienten $b^{[1]}_{ijk\ell}(\mathbf{w})$ und $b^{[1]}_{ijk\ell}(\bar{\mathbf{w}})$ bestimmt. Dazu werden zwanzig de Casteljau-Operationen (lineare Interpolationen) durchgeführt. Jede dieser elementaren de Casteljau-Operationen entspricht dem inneren Produkt von zwei Vektoren aus dem $\mathbb{R}^4$, was auf der GPU direkt als Vektor-Instruktion durchgeführt werden kann. Mit insgesamt zwölf de Casteljau-Operationen werden im nächsten Schritt die in der Abbildung rot dargestellten Koeffizienten $b^{[2]}_{ijk\ell}$ bestimmt. Dabei werden die Koeffizienten $b^{[1]}_{ijk\ell}(\mathbf{w})$ für beide Argumente $\lambda(\mathbf{w})$, $\lambda(\bar{\mathbf{w}})$ genutzt. Nach dem letzten Schritt (welcher vier Operationen benötigt) stehen die 4 Koeffizienten b_{ij} fest, welche das univariate Polynomen 3. Grades entlang des Sichtstrahls festlegen. Bei dem quadratischen Spline-Modell wird analog vorgegangen und ein Schritt weniger benötigt. In Abbildung 5.10 würde dementsprechend mit den oberen grünen Koeffizienten gestartet werden und das resultierende quadratische Polynom über die drei Koeffizienten b_{20}, b_{11} und b_{02} festgelegt sein.

Um t_c zu bestimmen muss nun ein Nullstellenproblem gelöst werden:

$$p(t) - c = 0 \tag{5.2}$$

Da $p(t)$ ein quadratisches bzw. kubisches Polynom mit einer Veränderlichen ist, existieren für das Finden der Nullstellen von $p(t) - c$ stabile, analytische Formeln in Monomform. Bei den quadratischen Splines wird eine solche Formel zum Finden von t_c genutzt. Bei den kubischen Splines werden hierfür zu viele Fallunterscheidungen benötigt. Der Algorithmus verwendet zum Finden aller Nullstellen für $t \in [0, 1]$ das numerische Newton-

Verfahren:

$$t_{n+1}^{(\ell)} = t_n^{(\ell)} + p(t_n^{(\ell)})/p'(t_n^{(\ell)}), \quad n \in \mathbb{N}_0. \tag{5.3}$$

Um die für Vektor-Operationen optimierte GPU voll auszunutzen, wird mit den drei Startwerten $t_0^{(0)} = 0$, $t_0^{(1)}$ und $t_0^{(2)} = 1$ begonnen. Für $t_0^{(1)}$ wird der Wendepunkt gewählt. Falls kein Wendepunkt in $[0, 1]$ liegt, wird $t_0^{(1)} = 0{,}5$ gesetzt. Visuell zeigt sich, dass nach 5 Iterationen die Nullstellen mit ausreichender Genauigkeit bestimmt sind.

Die baryzentrische Koordinate $\lambda(\mathbf{w}_c^{(\ell)})$ kann aufgrund $\mathbf{w}_c^{(\ell)} = \mathbf{w} + t_c^{(\ell)}(\bar{\mathbf{w}} - \mathbf{w})$ über lineare Interpolation zwischen $\lambda(\mathbf{w})$ und $\lambda(\bar{\mathbf{w}})$ bestimmt werden. Für $t_c^{(\ell)}$ wird der kleinste, positive Wert aus $\{t_c^{(0)}, t_c^{(1)}, t_c^{(2)}\}$ genommen, welcher den am nächsten gelegenen Schnittpunkt mit der Isofläche I_c parametrisiert. Zusätzlich muss $\mathbf{w}_c$ im Tetraeder liegen, was durch $\lambda_\nu(\mathbf{w}_c^{(\ell)}) \geq 0$, $\nu = 0, \ldots, 3$ sichergestellt wird. Ist kein solcher Schnittpunkt gefunden, wird das Fragment verworfen.

Die Normale der Isofläche an einer Position $\mathbf{x}$ bildet sich aus dem normierten Gradienten der Dichtefunktion ϕ:

$$\mathbf{n}(\mathbf{x}) = \frac{\nabla\phi(\mathbf{x})}{\|\nabla\phi(\mathbf{x})\|}. \tag{5.4}$$

Um Lichtberechnungen der Fragmente durchzuführen (d.h. Phong-Shading), wird die Oberflächennormale $\mathbf{n}_c$ in $\mathbf{w}_c$ bezüglich $s_{|_T}$ benötigt:

$$\mathbf{n}_c = \nabla s(\mathbf{w}_c) = (\frac{\partial s}{\partial x}(\mathbf{w}_c), \frac{\partial s}{\partial y}(\mathbf{w}_c), \frac{\partial s}{\partial z}(\mathbf{w}_c))^T. \tag{5.5}$$

Zur Berechnung von $\mathbf{n}_c$ werden die Bernstein-Bézier-Koeffizienten $b_{ijk\ell}^{[q-1]}(\mathbf{w}_c)$ mit $i + j + k + \ell = 1$ des vorletzten de Casteljau Schrittes benötigt (siehe Kapitel 4). Diese Koeffizienten können direkt durch die lineare Interpolation von $b_{ijk\ell}^{[q-1]}(\mathbf{w})$ und $b_{ijk\ell}^{[q-1]}(\bar{\mathbf{w}})$ sowie dem Strahlparameter t_c bestimmt werden. Mit den Ableitungen $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, $\frac{\partial}{\partial z}$ und der Hilfe von Relationen der Ableitungen in Richtung der Kanten des Tetraeders kann die Normale bestimmt werden (siehe [RZNS04]). Dazu werden die inneren Produkte des Vektors $b_{ijk\ell}^{[q-1]}(\bar{\mathbf{w}}_\mathbf{c})$ mit den drei Vektoren $\lambda(\mathbf{x})$, $\lambda(\mathbf{y})$ und $\lambda(\mathbf{z})$ benötigt, wobei $\lambda(\mathbf{x})$, $\lambda(\mathbf{y})$ und $\lambda(\mathbf{z})$ die baryzentrischen Koordinaten der Einheitsvektoren $\mathbf{x} = (1, 0, 0)^T$, $\mathbf{y} = (0, 1, 0)^T$ und $\mathbf{z} = (0, 0, 1)^T$ bezüglich T sind. Die baryzentrischen Koordinaten der drei Einheitsvektoren bezüglich T sind vorberechnet und werden dem entsprechenden Fragmentshader als Funktion zugelinkt. Die drei inneren Produkte liefern die Einträge der Normale in Objekt-Koordinaten, welche noch normiert wird.

Mit $\mathbf{w}_c = \mathbf{w} + t_c^{(\ell)}(\bar{\mathbf{w}} - \mathbf{w})$ und $\mathbf{n}_c$ liegt die Position sowie die Normale des aktuellen Fragmentes in Objekt-Koordinaten vor. Damit können nun verschiedene Lichtberechnungsmodelle (siehe [AMHH08, Ros06]) angewandt werden. Für den Z-Buffer wird der über das aktuelle Dreieck interpolierte Tiefenwert verwendet, welcher für die Verdeckungsberechnung ausreicht.

5.3 Einsatz multipler Kernel und Shader

Für hohe Bildraten werden Shader-Programme mit möglichst wenigen Instruktionen benötigt. Da Grafikkarten darauf ausgelegt sind, zwischen vielen Shader-Programmen hin und her zu wechseln, ist es sinnvoll, die Geometrie zu gruppieren und angepasste Shader für diese einzelnen Gruppen zu nutzen. Aufgrund der Visualisierung mittels Instancing liegt eine vorteilhafte Unterteilung der Tetraeder vor. So können im Algorithmus für jeden der 24 Tetraedertypen optimierte Vertex- und Fragmentshader eingesetzt werden, welche keine konditionalen Bedingungen benötigen. Vor jedem der Instancing-Zeichenaufrufe wird dann zum jeweiligen Shader-Programm gewechselt.

Um Programiercode-Verdoppelung zu vermeiden und möglichst kurze, angepasste Programme zu erhalten, können bei Shader-Sprachen Teile des Codes zu Funktionen zusammengefasst werden. Diese Funktionen können separat compiliert und am Ende mit den verschiedenen Shader-Hauptprogrammen verlinkt werden. Auf diese Art sind verschiedene Lichtberechnungsmodelle, prozedurale Texturen und andere Effekte kombinierbar. Durch die vielen Kombinationen entstehen so schnell hunderte, und bei größeren Anwendungen wie Computerspielen tausende, von Shader-Programmen.

Auch bei CUDA werden sehr ähnliche Konzepte zur Effizienzsteigerung genutzt. Anstatt Shader-Programme zu laden, werden hier Kernel-Aufrufe verwendet. Einerseits ist es somit möglich konditionale Bedingungen aus dem Kernel zu nehmen und die Bedingungen durch unterschiedliche angepasste Kernel zu lösen. Andererseits können unabhängige Kernel direkt hintereinander als sogenannte Streams gestartet werden. Die CUDA-Runtime kann so die Latenzzeiten beim Kernel-Wechsel reduzieren, um die Hardware bestmöglich auszulasten.

Um die GPU-Rechenleistung voll auszunutzen, muss weiterhin die parallele Architektur berücksichtigt werden. Die Shader-ALUs sind *unified* ausgelegt und lassen sich je nach Bedarf als Vertex- oder Fragmentshader konfigurieren. Für die üblichen Berechnungen schaltet beispielsweise NVidia vier ALUs zusammen, die als Vertexshader einen Koordinatenvektor (x, y, z, w) und als Fragmentshader einen Pixel-Farbwert (r, g, b, alpha) verarbeiten. Somit kann es vorteilhaft sein, mehrere skalare Instruktionen zu Vektor-Operationen zusammenzufassen.

5.4 Algorithmus-Variante für hohe Bildraten

Dieser Abschnitt beschreibt eine Variante des obigen Algorithmus, welche deutlich höhere Bildraten erzielt und dabei nur etwas längere Rekonstruktionszeiten benötigt. Dies wird dadurch erreicht, dass die Bestimmenden Sätze $\Gamma_{q,Q}$ aller Quader Q die I_c enthalten vorberechnet werden und mittels OpenGL-Texturen den Vertexshader-Programmen bereitgestellt werden.

Bei der Tetraeder-Klassifikation werden ohnehin alle Koeffizienten der aktiven Quader berechnet. Somit müssen bei der Algorithmus-Variante hier lediglich die Bestimmenden Sätze in Pixel-Buffer-Objects abgespeichert werden. Bei den quadratischen Super-Splines werden hierzu zwei und bei den kubischen C^1-Splines vier PBOs reserviert. Die 20 Ko-

effizienten bei den quadratischen Splines werden auf die beiden PBOs mit 16 und 4 Koeffizienten unterteilt. Die 56 Koeffizienten beim kubischen Fall werden auf drei PBOs zu 16 Koeffizienten und einen PBO mit 8 Koeffizienten verteilt. Grund für diese Aufteilung ist, das Texturzugriffe im Vertexshader auf Vektoren mit vier Einträgen optimiert sind und außerdem Texturkoordinaten effizienter berechnet werden können.

Jeder Thread schreibt die 20 bzw. 56 Koeffizienten des Bestimmenden Satz $\Gamma_{q,Q}$ als 32Bit-Float in die zwei bzw. vier PBOs. Aus den PBOs werden OpenGL-Texturen Tex_{bc} erstellt. Damit der Vertexshader für den aktuellen Eckpunkt Zugriff auf den vorberechneten Bestimmenden Satz des Quader hat, wird eine zusätzliche Adresse benötigt. Die benötigte Adresse ist ein Index in die Textur Tex_{bc}. Dieser Index wird als zweiter Eintrag in die Texturen Tex_{t_i} geschrieben, was in den jeweiligen Tetraeder-Komprimierungs-Kernel passiert. Somit wird kein zusätzlicher Texturzugriff benötigt und ein Auslesen des Bestimmenden Satzes kann bei den quadratischen Super-Splines mit 5 Texturzugriffen und bei den kubischen C^1-Splines mit 14 Zugriffen erfolgen. Der jeweilige Vertexshader kann die benötigten Koeffizienten durch wiederholtes Mitteln aus den Koeffizienten des Bestimmenden Satz berechnen und an den Fragmentshader weiterreichen. Die Vertexshader dieser Algorithmus-Varianten benötigen weniger Texturzugriffe und weniger Instruktionen zum berechnen der BB-Koeffizienten, was deutlich höhere Bildraten ermöglicht.

Kapitel 6

Arbeitstechniken & illustrative Methoden

In diesem Kapitel werden eine Reihe bekannter Techniken vorgestellt, welche dabei helfen, die volumetrischen Daten zu analysieren. Dabei wird speziell auf die vorteilhaften Eigenschaften der genutzten trivariaten Splines eingegangen.

6.1 Interaktives Glätten

Bei der direkten Volumenvisualisierung wird mit Transferfunktionen gearbeitet. Diese Transferfunktionen werden über Teile der Dichteverteilung (und manchmal auch über Teile der Gradientenverteilung) gelegt und bilden damit auf Farbwerte sowie Transparenz ab. Wenn Konturen oder Annäherungen von Isoflächen mittels direkter Volumenvisualisierung dargestellt werden, können zeltförmige oder gaußförmige Transferfunktionen verwendet werden, um Peak-Funktionen anzunähern. So wird der Datensatz prinzipiell geglättet. Einen ähnlichen Effekt, der weichere Übergänge sowie glattere Flächen ermöglicht, kann bei der Isoflächen-Darstellung durch Glätten des Datensatzes erreicht werden. Hierbei wird der Datensatz verfälscht und wichtige Informationen gehen verloren. Zusätzlich muss stets berücksichtigt werden, dass jede Form der Vorverarbeitung (z.B. Glätten) und der Darstellung (Farbe, Skalierung) die Interpretation stark beeinflusst. Bei stark verrauschten Daten ist ein Glätten zwangsläufig nötig. Auch das Glätten von nicht verrauschten Daten kann sinnvoll sein, um einen besseren Überblick über die Geometrie zu bekommen.

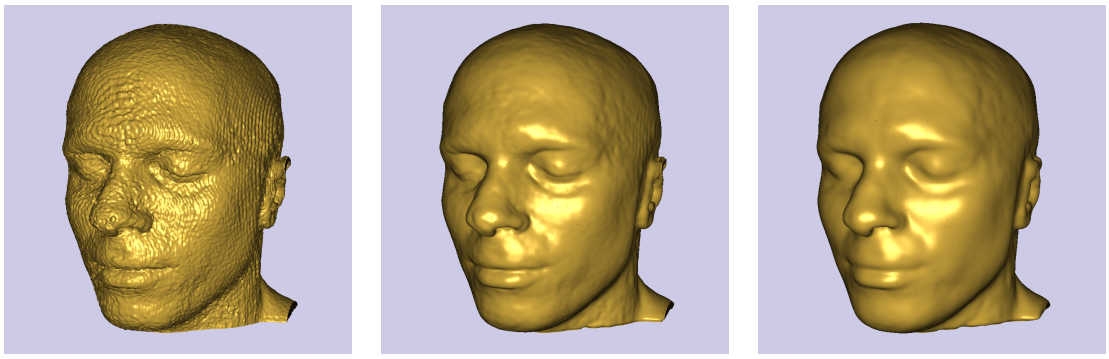


Abbildung 6.1: Ein verrauschter MRT-Datensatz der mit einem diskreten 3D-Gauß-Filter auf der Grafikkarte interaktiv geglättet wurde.

Abbildung 6.2: Einfaches Volumen-Clipping kann erreicht werden, indem Teile der Voxel auf Extremwerte gesetzt werden. Dies kann interaktiv über einen CUDA-Kernel geschehen. Die so durchtrennten Isoflächen werden automatisch geschlossen und die Kanten lokal abgerundet.

Bei dem in Kapitel 5 vorgestellten Algorithmus kann interaktives Glätten durch einen diskreten 3D-Gauß-Filter erreicht werden. Dazu wird ein CUDA-Kernel benötigt, der für jeden Eintrag in dem Volumendatensatz einen Thread startet. Der Thread liest die benachbarten Datenwerte (z.B. 3^3, 5^3, je nach verwendeten Filtermaske) und bestimmt damit den neuen, geglätteten Datenwert. Nach dem Synchronisieren der Threads können die neu bestimmten Werte zurückgeschrieben werden. Abbildung 6.1 zeigt die Isoflächen eines unterschiedlich stark geglätteten Datensatzes.

6.2 Volumen-Clipping

Volumen-Clipping steht für das Wegschneiden ausgewählter Volumenanteile, um das Verständnis der 3D-Daten zu unterstützen. Bei der Isoflächen-Darstellung bietet Volumen-Clipping die Möglichkeit, wichtige und ansonsten verdeckte Details des Datensatzes sichtbar zu machen. Bei der direkten Volumenvisualisierung können ähnliche Effekte über Transferfunktionen erreicht werden.

Ein sehr einfaches Clipping kann durch Abscheiden bzw. Verkleinern des Datensatzes erfolgen. Eine andere Möglichkeit ist, Teile der Voxel des Datensatzes auf Extremwerte zu setzen, sodass Ergebnisse gem. Abbildung 6.2 erzeugt werden. Damit können sehr einfache Geometrien, die sich aus Ebenen entlang der Volumenhauptachsen zusammensetzen, aus dem Volumen geschnitten werden. Da die Methoden der verwendeten Splines lokal sind (27er Stempel), werden hierbei saubere, glatte Schnittkanten rekonstruiert, welche den Datensatz nur minimal verfälschen. Auch komplexere Clipping-Geometrien können über die Quader-Klassifikation interaktiv Teile der Voxel auf Extremwerte setzen. Die erzielten Ergebnisse sind abhängig von der Voxel-Auflösung (Datensatzgröße).

Eine weitere Möglichkeit, interaktiv an komplexen Geometrien zu clippen, ist es, direkt im Fragmentshader-Programm Teile der Fragmente (Pixel) zu verwerfen. Dies kann

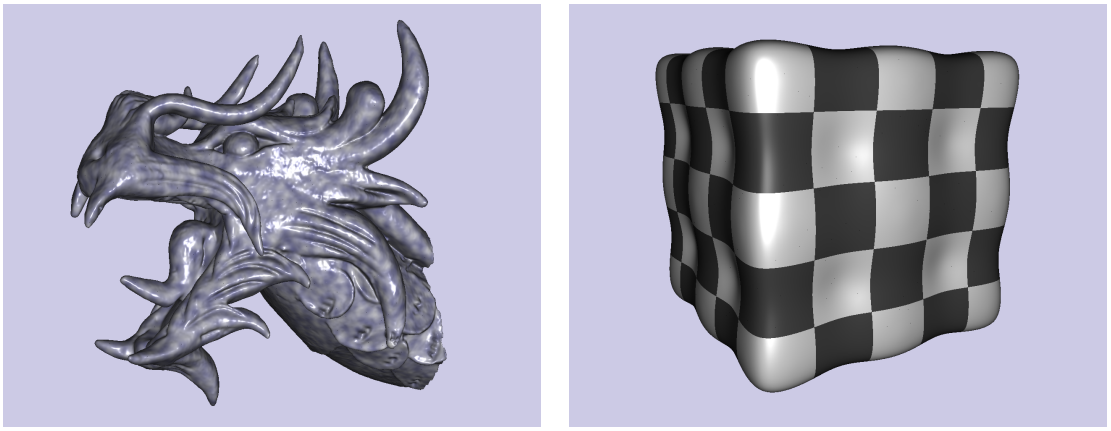


Abbildung 6.3: Prozedurale Texturen erzielen bei der Visualisierung mit kubischen C^1-Splines hochwertige Ergebnisse, da das verwendete Ray-Casting-Verfahren pro Pixel präzise 3D-Koordinaten bestimmt. *Links:* Eine steinähnliche 3D-Textur die auf zufällig generiertem Rauschen basiert (Improved-Perlin-Noise [Per02]). Der Datensatz (256^3) wurde mittels Signed-Distance-Functions aus einem Polygonnetz des Asian Dragons generiert. *Rechts:* Eine Isofläche der mit 64^3 Datenwerten abgetasteten impliziten Funktion: $x^{16} + y^{16} + z^{16} - \cos(7x) - \cos(7y) - \cos(7z)$ kombiniert mit einem prozeduralem 3D-Schachbrett.

effizient mittels analytischer Formeln geschehen. Zusätzlich können die gleichen analytischen Formeln bei der Quader-Klassifikation und gegebenenfalls bei der Tetraeder-Klassifikation genutzt werden, um die Anzahl aktiver Tetraeder zu reduzieren. Aufgrund des verwendeten Ray-Casting-Verfahrens werden präzise 3D-Koordinaten berechnet. So entstehen saubere (C^1-stetige) Schnittkanten (siehe Anhang Abbildung A.9). Problematisch sind die dadurch sichtbaren Rückseiten der Isoflächen.

6.3 Prozedurale Texturen

Die klassischen Texturen in der Computergrafik sind vorberechnete Bilder bestimmter Auflösung, die den darzustellenden Objekten zusätzliche Informationen zuschreiben. Dies können unter anderem Farbwerte, Beleuchtungswerte, Glanzverhalten oder Oberflächenstrukturen sein. Prozedurale Texturen beschreiben die gleichen Informationen durch mathematische Berechnungen. Sie können dadurch erst zur Laufzeit erzeugt werden, umgehen so das Problem der Skalierbarkeit und belegen marginalen Speicherplatz. Weiterhin benötigen prozedurale Texturen keine Filtertechniken, um den Abtastungsproblemen der Bildschirmauflösung entgegenzuwirken.

Die Visualisierung mit Isoflächen kann durch dreidimensionale, prozedurale Texturen unterstützt werden. Durch die zusätzlichen Informationen der Textur wirkt die Isofläche, als wäre sie aus einem großen Block herausgeschnitten worden. So entsteht bei einem Wechsel des Isowertes ein besserer Bezug zum vorherigen Isowert. Um prozedurale Texturen zu nutzen, werden Koordinaten benötigt. Eine natürliche Wahl für Texturkoordinaten ist die 3D-Position des jeweiligen Pixels. Hier haben die trivariaten Splines sehr entscheidende Vorteile. Aufgrund des niedrigen totalen Grades von 2 bzw. 3

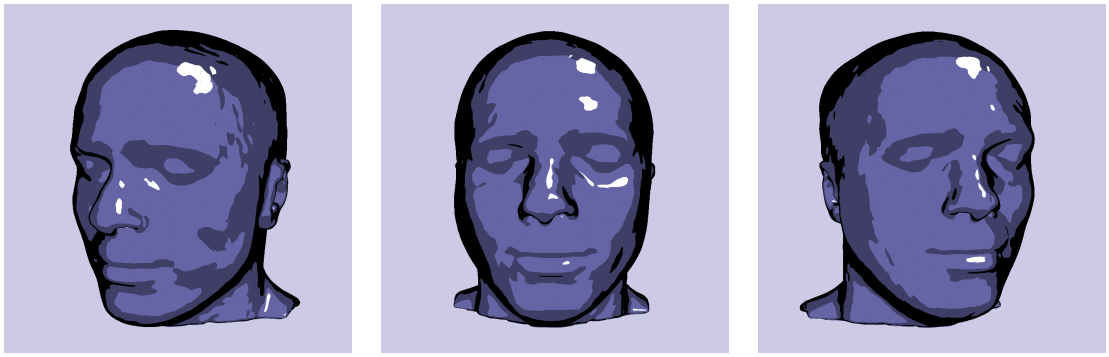


Abbildung 6.4: Cel-Shading (Toon-Shading) bei kubischen C^1-Splines.

kann der Schnittpunkt mit der Isofläche exakt bestimmt werden. Damit liegen direkt präzise Textur-Koordinaten vor. Zusätzlich werden bei dem verwendeten Algorithmus die Schnittberechnungen in Objektkoordinaten durchgeführt, welche ohne Umrechnung als Texturkoordinaten verwendet werden können. Abbildung 6.3 zeigt, welche qualitativ hochwertigen Ergebnisse aufgrund der C^1-stetigen Rekonstruktion von ϕ und dem präzisen Ray-Casting-Verfahren zu erzielen sind.

6.4 Nicht-fotorealistisches Rendering

Die Volumenvisualisierung stellt gegenüber dem Volumenrendering keinen Anspruch auf fotorealistische Darstellung. In manchen Anwendungen können gezielte Effekte gefragt sein, die die dargestellten Informationen zusätzlich abstrahieren. Diese Effekte werden als *Nicht-fotorealistisches Rendering* [SS02] bezeichnet. Cel-Shading (Contour Enhancing Lines), welches auch als Toon-Shading bezeichnet wird, erzielt eine zeichentrickähnliche Visualisierung. Statt die Schattierungen auf der Geometrie weich verlaufen zu lassen, werden nur drei oder vier Helligkeitsstufen verwendet. Ein sehr einfaches Modell nutzt dazu den Winkel zwischen der Oberflächennormale und dem Lichtvektor, um mit einer 1D-Textur auf die wenigen Farbwerte abzubilden. Abbildung 6.4 zeigt Ergebnisse der C^1-stetigen Rekonstruktion von ϕ und einem sehr simplen Cel-Shading, welche auf 4 Farbwerte abbildet. Neben den Oberflächennormalen können auch krümmungsinformationen herangezogen werden, welche sehr gute Ergebnisse erzielen können [KWTM03].

Kapitel 7

Numerische Ergebnisse

Jede 3D-Applikation möchte interaktives Arbeiten ermöglichen. Nicht nur um aktuelle Ergebnisse mittels Transformationen aus verschiedenen Perspektiven zu betrachten, sondern auch, um zügig optimale Werte für verschiedene Parameter zu finden. Bei einem Bild pro Sekunde ist Interaktivität kaum möglich. Die Eingabe von Parametern über Tastatur und Maus scheint in keinem Zusammenhang mit der Visualisierung zu stehen. Ab 6 Bildern pro Sekunde fängt ein Gefühl der Kontrolle an. Vielleicht kann man ab 15 Bildern pro Sekunde von einer Echtzeitanwendung sprechen.

In diesem Kapitel werden Zeitmessungen, die mit dem implementierten Algorithmus aus Kapitel 5 (beide Varianten) gemacht wurden, beschrieben, analysiert und diskutiert.

Details zur Implementierung & Plattform

Für die Programmierung der CPU, welche die kontrollierenden Strukturen des Algorithmus aus Kapitel 5 übernimmt, wurde die Sprache C++ verwendet. Alle Berechnungen auf der Grafikkarte wurden mit einer Kombination aus CUDA [NVI09b], OpenGL [KBR, SWND05] und der Shader-Sprache GLSL [Ros06] programmiert. Für das Erzeugen der OpenGL-Kontexte sowie für die Benutzerinteraktion wurde Qt [Nok] von Nokia Corporation (ehemals Trolltech) genutzt. Der Shadercode, der für die Berechnung der Bernstein-Bézier-Koeffizienten verantwortlich ist, wurde mit Hilfe von Template-C++ Klassen automatisch generiert. Dazu wurde die Symmetrie des Bestimmenden Satzes der Splines ausgenutzt und der Fall für einen einzelnen Tetraeder programmiert. Die Berechnungen wurden zusätzlich mit einem Computeralgebrasystem (Maple [Map09]) optimiert, um die Anzahl der Instruktionen zu reduzieren. Die Berechnungen der Koeffizienten für die einzelnen Tetraeder wurden über Permutationen zur Kompilierzeit gelöst. Der Dabei entstehende Shadercode wurde dann als Funktion kompiliert und dem entsprechenden Vertexshader hinzugelinkt (Abschnitt 5.3).

Alle Zeitmessungen wurden mit einem System bestehend aus Intel Core 2 Duo (2 CPUs a 3,16 GHz), 4 Gigabyte Arbeitsspeicher und Windows XP gemacht. Als Grafikkarte kam eine NVidia Geforce GTX 280 mit 1.024 Megabyte Arbeitsspeicher zum Einsatz. Der Grafikchip der GTX 280 verfügt über 240 Prozessoren, wobei jeweils 8 Stück einen der 30 Multiprozessoren bilden.

Silicium	Tooth	Foot	Bonsai	Dragon (SDF)
$(98 \times 34 \times 34)$	$(256 \times 256 \times 161)$	(256^3)	(256^3)	$(512 \times 256 \times 256)$

Abbildung 7.1: Die fünf Datensätze, welche für die Zeitmessungen in diesem Kapitel verwendet wurden. Die Abbildungen zeigen die Isoflächen sowie die verwendete Betrachtung der Isofläche, die bei einem 1.280×1.024-Viewport für die Zeitmessungen verwendet wurden.

7.1 Rekonstruktionszeiten

Die Rekonstruktionszeit ist die Zeit, die der CUDA-Präprozess benötigt, bevor die Visualisierungs-Schleife die aktuelle Isofläche anzeigen kann. Bei sehr kleinen Datensätzen, wie Bucky 32^3 oder Fuel 64^3 (siehe Abbildung A.1 auf Seite 64), benötigt der Algorithmus Rekonstruktionszeiten von ca. 40ms (Millisekunden), unabhängig von dem Spline-Modell oder der Algorithmus-Variante. Hier liegt die Grundzeit oder Mindestzeit der Implementation. Die Zeit von 40ms ist erforderlich wegen der vielen Kernel-Aufrufe (27 Stück und 25 Präfixsummen). Bei CUDA ist ein Kernel-Start ein grundsätzlich zeitintensiver Vorgang.

Für die Zeitmessungen wurden die in Abbildung 7.1 dargestellten Datensätze verwendet. Als kleinster der fünf Datensätze wurde Silicium ausgewählt, da ab dieser Datensatzgröße und Anzahl an aktiven Tetraedern beginnend, Zeiten über 40ms benötigt werden. Der Silicium-Datensatz [Bar] stammt aus einer Gitter-Simulation. Die Datensätze Tooth [Roe], Foot [Bar] und Bonsai [Bar] wurde durch CT-Scans generiert. Der Datensatz Dragon ist mittels Signed-Distance-Functions [Fuh07] aus einem Polygonnetz des Stanford Dragon berechnet worden. Größere Datensätze ($\geq 512^3$) wurden in die Zeitmessung nicht aufgenommen, da die hierbei entstehenden Isoflächen ohne Strategien für große Datensätze nicht interaktiv visualisiert werden können. Bei den fünf verwendeten Datensätzen (sowie den verwendeten Isowerten) enthalten, pro aktiven Quader, durchschnittlich 11,8 Tetraeder Teile der Isofläche. Diagramm 7.2 zeigt die gemessen Rekonstruktionszeiten die beide Algorithmus-Varianten benötigen.

Die gemessenen Zeiten sind von zwei Faktoren abhängig: der Datensatzgröße und die Anzahl aktiver Tetraeder, sprich derjenigen Tetraeder, die Teile der aktuellen Isofläche I_c enthalten. Überprüft wurde dies (unter anderem) mit syntetisch generierten Datensätzen der Marschnerlobb-Testfunktion [ML94] der Größen 64^3, 128^3 und 256^3, welche Rekonstruktionszeiten (bei den kubischen Splines) von 58ms, 85ms und 250ms benötigen. Die zeitintensivste Berechnung des Algorithmus ist die Quader-Klassifikation. So werden hierzu für den 64^3, 128^3 und 256^3 Datensatz Zeiten von 2ms, 14ms und 112ms benötigt. Dies zeigt einen linearen Zusammenhang zwischen der Anzahl an Voxel und der benötigten Klassifikationszeit. Weiterhin zeitaufwendig ist die Tetraeder-Klassifikation und das erstellen der OpenGL-Texturen Tex_{t_i}. Diese Berechnungen sind direkt abhängig

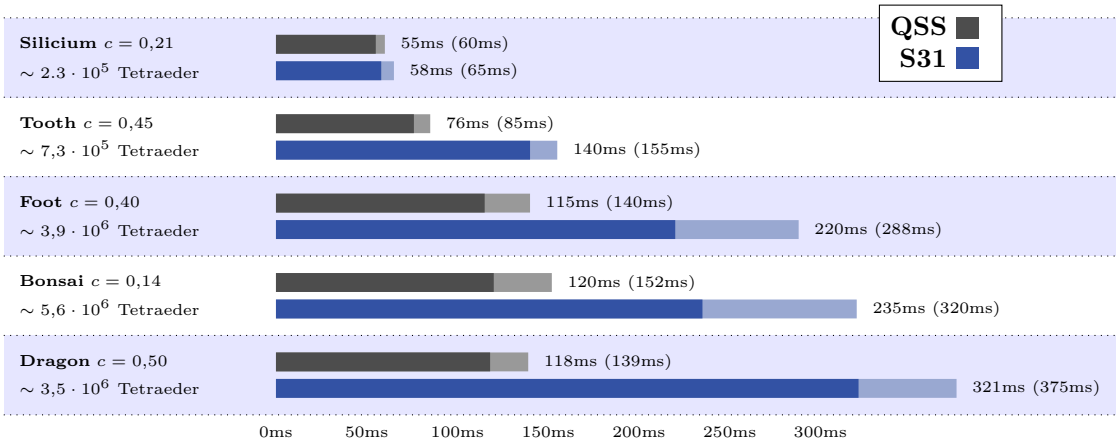


Abbildung 7.2: Die Rekonstruktionszeiten der quadratischen Super-Splines (QSS) und der kubischen C^1-Splines (S31) für die fünf ausgewählten Datensätze. Die im helleren Ton (■,■) dargestellten Zeiten beziehen sich auf die für hohe Bildraten optimierte Algorithmus-Variante.

von der Anzahl an aktiven Tetraedern. So liegen die Isoflächen bei den drei verschieden großen Datensätzen in $2{,}6 \cdot 10^5$, $13{,}5 \cdot 10^5$ und $57{,}7 \cdot 10^5$ Tetraederelementen. Die Tetraeder-Klassifikation benötigt Zeiten von 2ms, 8ms und 35ms. Das Erstellen der 24 Texturen Tex_{t_i} benötigt 2ms, 7ms und 30ms. Somit besteht auch zwischen der Anzahl der aktiven Tetraeder und der benötigten Rekonstruktionszeit ein linearer Zusammenhang.

7.2 Bildraten

Die Bildraten sind bei einem 1.280×1.024-Viewport gemessen worden. Die Isoflächen wurden dabei so vergrößert, dass der gesamte Viewport ausgenutzt, aber kein Teil der Fläche abgeschnitten wurde (siehe Abbildung 7.1). Bei den Bildraten erreichen kleine Datensätze, bei denen die Isoflächen typischerweise in wenigen Tetraederelementen liegen, sehr hohe Bildraten. So erreicht eine Isofläche beim Bucky-Datensatz 32^3 ($c = 0{,}35$) Bildraten von 65 fps und eine Isofläche beim Fuel-Datensatz 64^3 ($c = 0{,}13$) 145 fps. Die Anzahl der aktiven Tetraeder liegt hier bei $7{,}0 \cdot 10^4$ (Bucky) und $3{,}5 \cdot 10^4$ (Fuel). Die gemessenen Bildraten für beide Algorithmus-Varianten der 5 ausgewählten Datensätze sind in Abbildung 7.4 dargestellt.

Die erzielten Bildraten sind direkt abhängig von der Anzahl der aktiven Tetraeder. Ab einer gewissen Anzahl Tetraeder besteht hier ein linearer Zusammenhang. Die Visualisierung ist nicht limitiert durch den Fragmentshader. Dies zeigt sich daran, dass ein starkes Verkleinern des Objekts die Bildraten nur in Maßen erhöht. Der grundsätzlich limitierende Faktor ist die hohe Anzahl der Dreiecke und die damit verbundenen Berechnungen im Vertexshader. Problematisch sind hier die vielen Texturzugriffe und die vielen Instruktionen, die zum Berechnen der BB-Koeffizienten benötigt werden.

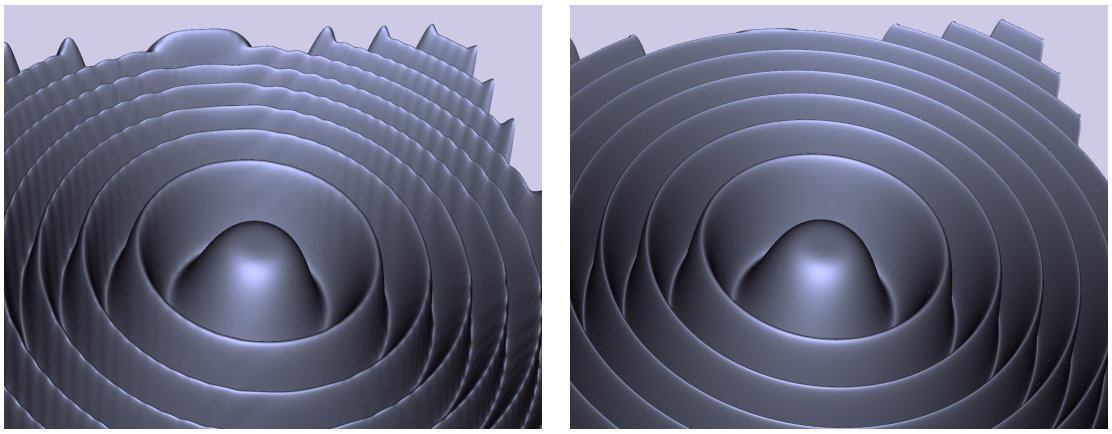


Abbildung 7.3: Die Marschnerlobb-Testfunktion rekonstruiert mit den kubischen C^1-Splines. Die Funktion wurde von Marschner und Lobb zur Evaluierung von 3D-Rekonstruktionsfiltern in [ML94] vorgestellt. *Links:* Abgetastet mit 64^3 Datenwerten. *Rechts:* Abgetastet mit 256^3 Datenwerten.

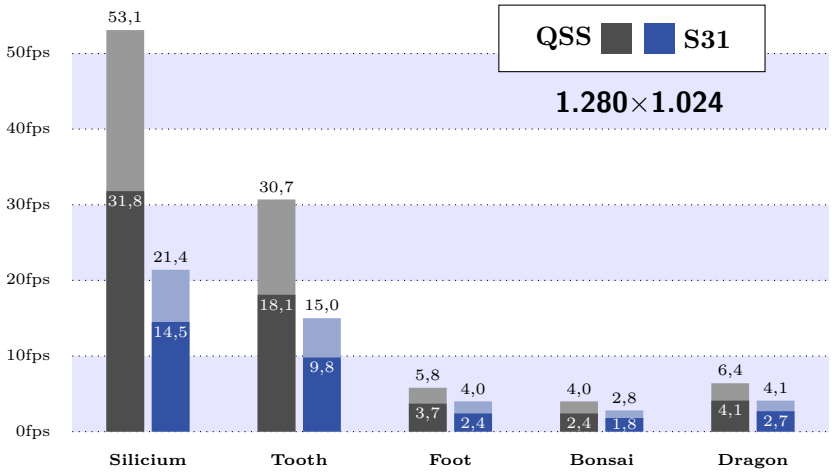


Abbildung 7.4: Die Bildraten für die quadratischen Super-Splines (QSS) und die kubischen C^1-Splines (S31). Die im helleren Ton (■,■) dargestellten Zeiten beziehen sich auf die für hohe Bildraten optimierte Algorithmus-Variante.

		Trivariate Splines					CUDA-Marching-Cubes			
	c	#Tetraeder	$\Gamma_{2,Q}$	$\Gamma_{3,Q}$	Tex_{t_i}	Δ_6	#Dreiecke	Eck-punkte	Nor-malen	Rekon-struktion
Silicium	0,21	$2{,}3 \cdot 10^5$	1,5	4,2	0,9	$2{,}3kb$	$3{,}9 \cdot 10^4$	1,7	1,7	$5ms$
Tooth	0,45	$7{,}3 \cdot 10^5$	4,8	13,4	2,8	$2{,}3kb$	$1{,}2 \cdot 10^5$	5,4	5,4	$23ms$
Foot	0,40	$3{,}9 \cdot 10^6$	25,5	71,6	14,9	$2{,}3kb$	$8{,}5 \cdot 10^5$	38,9	38,9	$37ms$
Bonsai	0,14	$5{,}6 \cdot 10^6$	35,7	100,1	21,4	$2{,}3kb$	$1{,}0 \cdot 10^6$	45,8	45,8	$40ms$
Dragon	0,50	$3{,}5 \cdot 10^6$	23,3	65,3	13,4	$2{,}3kb$	$6{,}0 \cdot 10^5$	27,5	27,5	$53ms$
Kilobyte [kb], Millisekunden [ms], andere Speicherangaben in Megabyte [mb]										

Abbildung 7.5: Vergleich des Speicherverbrauchs für die Geometrie bei den trivariaten Splines sowie eines mittels CUDA implementierten Marching-Cubes-Verfahrens.

7.3 Trivariate Splines vs. Marching-Cubes

In diesem Abschnitt wird der benötigte Speicherplatz für die Geometrie der trivariaten Splines für beide Algorithmus-Varianten betrachtet. Zusätzlich wird auf den Speicherbedarf eines mittels CUDA implementierten Marching-Cubes-Verfahrens eingegangen. Abbildung 7.5 zeigt den Speicherbedarf für die Geometrie der trivariaten Splines sowie des verwendeten CUDA-Marching-Cubes-Verfahrens für die fünf ausgewählten Datensätze. Die Rekonstruktionszeiten wurden bei einer sehr ähnlichen Implementierung zu dem Marching-Cubes-Verfahren aus dem CUDA-SDK gemessen [NVI09c]. Die Normalen werden hier mittels zentraler Differenzen approximiert. Das Verfahren nutzt Präfixsummen sowie *Stream-Compaction* zu parallelen Geometrie-Erstellung. Die Eckpunkte sowie die Normalen der berechneten Dreiecke werden hierbei in Vertex-Buffer-Objects gespeichert und mittels OpenGL gezeichnet. Der Speicherverbrauch der Geometrie für die fünf ausgewählten Datensätze setzt sich aus den benötigten Eckpunkten sowie den Normalen zusammen.

Der Speicherverbrauch für die Geometrie der trivariaten Splines setzt sich aus den Texturen Tex_{t_i} und der Tetraeder-Geometrie (Δ_6) zusammen, welche in Vertex-Buffer-Objects abgelegt sind. Die Tetraeder-Geometrie nimmt dabei nur marginalen Speicherplatz ein. Bei der Algorithmus-Variante für hohe Bildraten kommt der Speicherverbrauch für die Bestimmenden Sätze $\Gamma_{2,Q}$ bzw. $\Gamma_{3,Q}$ hinzu. Weiterhin wird hier doppelter Speicherplatz für die Texturen Tex_{t_i} benötigt (siehe Kapitel 5.4).

Der gezeigte Algorithmus benötigt nur ein Viertel des Speichers im Vergleich zum Marching-Cubes-Verfahren. Selbst bei der Algorithmus-Variante für hohe Bildraten liegt der Speicherbedarf der quadratischen Super-Splines noch leicht unter dem des Marching-Cubes-Verfahrens und bei den kubischen C^1-Splines ca. 60 Prozent darüber. Bei den Rekonstruktionszeiten ist das CUDA-Marching-Cubes-Verfahren gegenüber den quadratischen Super-Splines um den Faktor 3 schneller. Bei den kubische Splines liegt dieser Faktor bei 6.

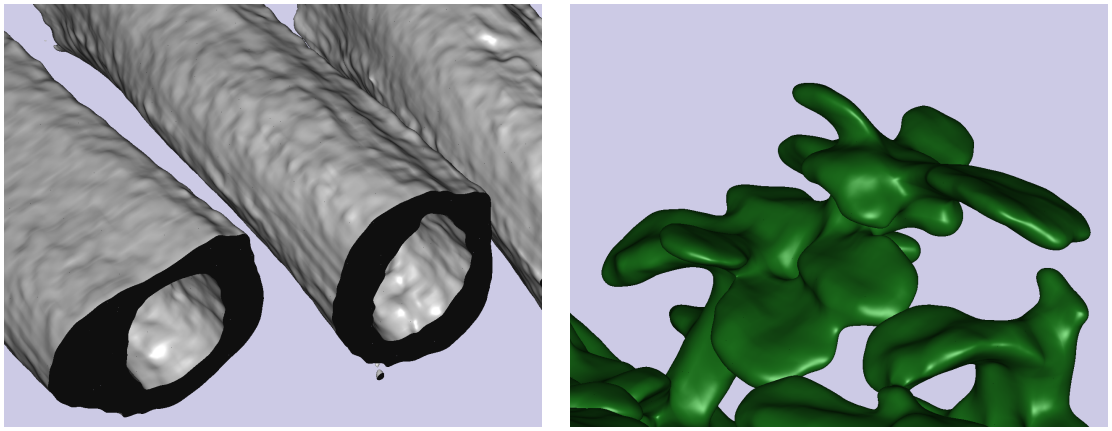


Abbildung 7.6: Die für die Zeitmessungen verwendeten Isoflächen des Foot- und des Bonsai-Datensatzes setzen sich aus 3,9 bzw. 5,6 Millionen Tetraedern zusammen. Trotz dieser sehr hohen Anzahl an aktiven Tetraedern, benötigt der Algorithmus Rekonstruktionszeiten von unter 235 Millisekunden. Die Visualisierung der so rekonstruierten Flächen erfolgt, trotz der benötigten 15, 6 bzw. 22,4 Millionen Dreiecke, mit 2 bis 6 Bildern pro Sekunden.

7.4 Diskussion und Bewertung

Bei kleinen Datensätzen (64^3, 128^3) erreicht der Algorithmus eine mindestens zehnfache Beschleunigung zu der CPU-Variante aus [KZ08]. Dies ist bereits eine signifikante Steigerung. In [KZ08] wurde der Präprozess für den Foot-Datensatz mit acht Sekunden angegeben (bei kubischen Polynomen). Hier werden durch vorberechnete, räumliche Datenstrukturen (Octree) die Tetraederelemente, welche I_c enthalten, bestimmt und die Geometrie-Kodierung sowie Zeichenaufrufe über Displaylisten realisiert. Bei dem GPU-Algorithmus aus dieser Arbeit benötigt der gesamte Präprozess für den Foot-Datensatz 220 Millisekunden. Dies ist 36-mal schneller als der CPU-Algorithmus. Bei dichten Datensätzen, welche Isoflächen mit sehr vielen Tetraederelementen enthalten, ist der Leistungszuwachs noch gravierender. So benötigt der Bonsai-Datensatz 16 Sekunden auf der CPU und 235 Millisekunden auf der GPU. Das entspricht einer 68-fachen Beschleunigung. Dies ist ein fundamentaler Unterschied, der einen Plattformwechsel von der CPU zur GPU lohnenswert macht. Weiterhin werden so neue Applikationen möglich, welche auf eine interaktive Rekonstruktion von hochqualitativen Isoflächen aus Volumendaten aufbauen.

Eine CUDA-Marching-Cubes-Implementierung ermöglicht Rekonstruktionszeiten die um einen Faktor 3 bis 6 schneller sind, was kein signifikanter Unterschied ist. Dabei benötigt der gezeigte Algorithmus aus Kapitel 5 zur Kodierung der Geometrie lediglich ein Viertel des Speichers gegenüber dem CUDA-Marching-Cubes. Selbst in der für hohe Bildraten optimierten Algorithmus-Variante wird nur minimal mehr Speicher benötigt. Die Ergebnisse machen deutlich, dass der vorgestellte Algorithmus eine echte Alternative zu Standard-Verfahren ist.

Kapitel 8

Schlussbetrachtung

In dieser Arbeit wurde gezeigt, dass ein interaktives Variieren von hochqualitativen Isoflächen durch Auslagerung aller Berechnungsschritte auf die Grafikkarte möglich ist. Der vorgestellte GPU-Algorithmus ermöglicht dies durch eine glatte Rekonstruktion der Volumenfunktion ϕ, basierend auf stückweise zusammengesetzten, trivariaten Polynomen des totalen sowie vollständigen Grades 2 bzw. 3. Dies ist bei Standardverfahren erst bei Polynomgrad 9 möglich (trikubische Tensor-Produkt-Splines). Die Polynome niedrigen Grades sowie die verwendete Bernstein-Bézier-Form bringen dem Algorithmus entscheidende Vorteile: Das Überprüfen, ob ein Isowert in einem Quader bzw. in einem Tetraederelement enthalten ist, kann über die Eigenschaft der Konvexen Hülle sehr effizient geschehen und problemlos parallelisiert werden. Weiterhin wird durch den niedrigen Grad sowie die BB-Form ein präzises und effizientes Ray-Casting ermöglicht. Mit dem Speicherbedarf und den erzielten Bildraten wurde zusätzlich bestätigt, dass Instancing ein ideales Werkzeug zur Visualisierung der strukturierten Tetraeder-Partition darstellt.

Der entwickelte Algorithmus kann direkt als Software-Komponente zur Visualisierung bei vielfältigen Anwendungen genutzt werden. Ein Beispiel wäre das Darstellen von Flüssigkeiten basierend auf interaktiven Gitter-Simulationen. Weiterhin könnte der Algorithmus zur Visualisierung von zusammengesetzten impliziten Funktionen genutzt werden, was Anwendung beim *Construtive Solid Modeling* in den Bereichen CAE/CAD findet.

Da die Sichtbarkeit über den Z-Buffer gelöst wird, ist eine Kombination mit anderen Visualisierungstechniken problemlos möglich. Auch Kombinationen mit direkter Volumenvisualisierung (beispielsweise einem Ray-Marching-Verfahren) können ohne großen Aufwand realisiert werden. Anwendung finden solche kombinierten Visualisierungstechniken bei der modernen medizinischen Diagnostik, in welcher immer mehr mit 3D-Scans (MRT, CT) gearbeitet wird. Die dabei generierten Datensätze werden mit der Zeit immer größer und stellen so höhere Anforderungen an die Visualisierung. Durch Parallelisierung aller Berechnungsschritte skaliert der gezeigte Algorithmus mit der schnell steigenden Anzahl an Streaming-Prozessoren und Textur-Einheiten kommender GPU-Generationen.

Ausblick

Die Möglichkeiten, die Techniken zur Echtzeitvisualisierung mit trivariaten Splines weiterzuentwickeln, sind vielfältig und vielversprechend. NVidia plant in naher Zukunft ein Zusammenspiel von CUDA und OpenGL über sogenannte Texture-Objects. Damit

würde der Umweg über die Pixel-Buffer-Objects wegfallen. Eine höhere Performance könnte auch durch den Einsatz des Geometry-Shaders erreicht werden. Zwar ist diese Shader-Variante noch sehr limitiert [NVI08], doch in einiger Zeit wird das Erstellen von Geometrie auf der GPU zum Standard werden. So wäre es möglich, für jeden aktiven Tetraeder lediglich einen Punkt an den Vertexshader mittels Instancing zu schicken und erst im Geometry-Shader die Tetraedergeometrie zu generieren. Damit müssten die BB-Koeffizienten des Tetraeders nur einmalig berechnet werden. Eine andere denkbare Idee wäre es, eine komplette Grafik-Pipeline in CUDA zu implementieren, um so den Umweg über die Dreiecke zu einzusparen. So könnten spezielle Algorithmen zum Rastern der Tetraeder bzw. der Quader entwickelt werden, um alle Redundanzen beim Bestimmen der Bernstein-Bézier-Koeffizienten zu beseitigen.

Ein interessantes Thema wäre es die direkte Volumenvisualisierung mit der Isoflächen-Darstellung zu kombinieren und dabei die verwendete Rekonstruktion von ϕ sowie die Bernstein-Bézier-Form zu nutzten. Ein sehr entscheidender Vorteil liegt klar auf der Hand: die trivariaten Polynome lassen sich aufgrund der Bernstein-Bézier-Form entlang von Sichtstrahlen effizient sowie analytisch integrieren. Solch ein Integrieren entlang von Strahlen ist die grundlegende Operation der direkten Volumenvisualisierung.

Zum Schluss bleibt eine Frage offen: Wozu möchte man die stetig wachsende Rechenleistung zukünftiger Grafikkarten nutzen? Einerseits könnten Geometrien weiterhin mit Dreiecken approximiert werden. Die steigende Rechenleistung kann in immer mehr Dreiecke investiert werden, bis schließlich jedes Dreieck auf sehr wenige oder nur einen Pixel gerastert wird und eine Parallelisierung des Fragmentshader sinnlos macht. Andererseits liegt die Möglichkeit darin, die Rechenleistung in Visualisierungsprimitive höherer Ordnung zu investieren. Aufgrund der höheren Approximationsordnung dieser Primitive können Geometrien mit erheblich weniger Elementen beschrieben werden. Weiterhin fallen die Abtastungsprobleme der Bildschirmauflösung weg und das automatische *Level of Detail* bringt zusätzliche, erhebliche Vorteile.

Anhang A

Abbildungen

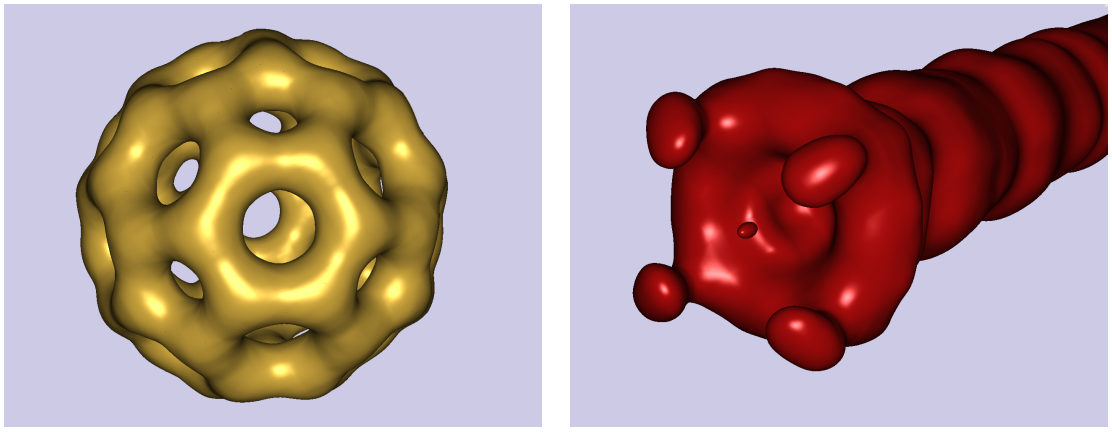


Abbildung A.1: Sehr kleine Datensätze wie Bucky 32^3 (*links*, $c = 0{,}35$) oder Fuel 64^3 (*rechts*, $c = 0{,}13$) benötigen Rekonstruktionszeiten von ca. 40 Millisekunden. Die so rekonstruierten Flächen liegen in $7{,}0 \cdot 10^4$ (Bucky) bzw. $3{,}5 \cdot 10^4$ (Fuel) Tetraederelementen und lassen sich mit Bildraten von 65 bzw. 145 Bildern pro Sekunde bei einer Auflösung von 1.280×1.024 visualisieren (siehe Kapitel 7.1 und 7.2).

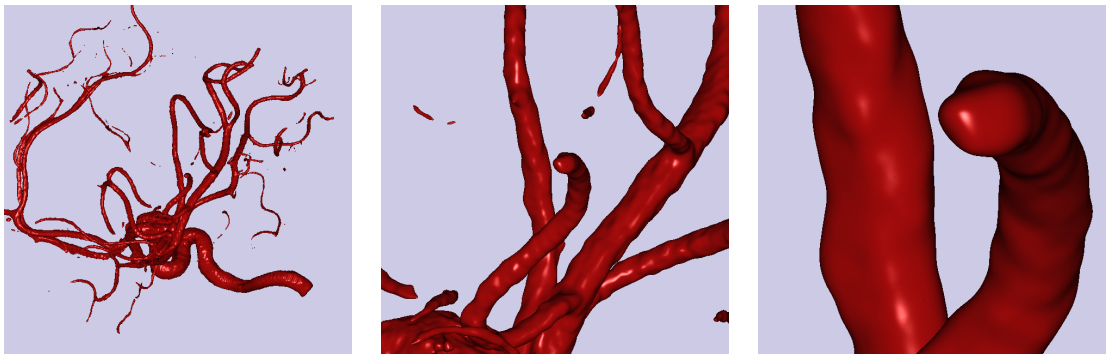


Abbildung A.2: Ein zweites Beispiel zur Abbildung 3.3 auf Seite 16.
Automatisches *Level of Detail* der trivariaten Splines erlaubt beliebige Nahaufnahmen. Hier, die in dieser Arbeit verwendeten, kubischen C^1-Splines. Datensatz: Aneurism (256^3).

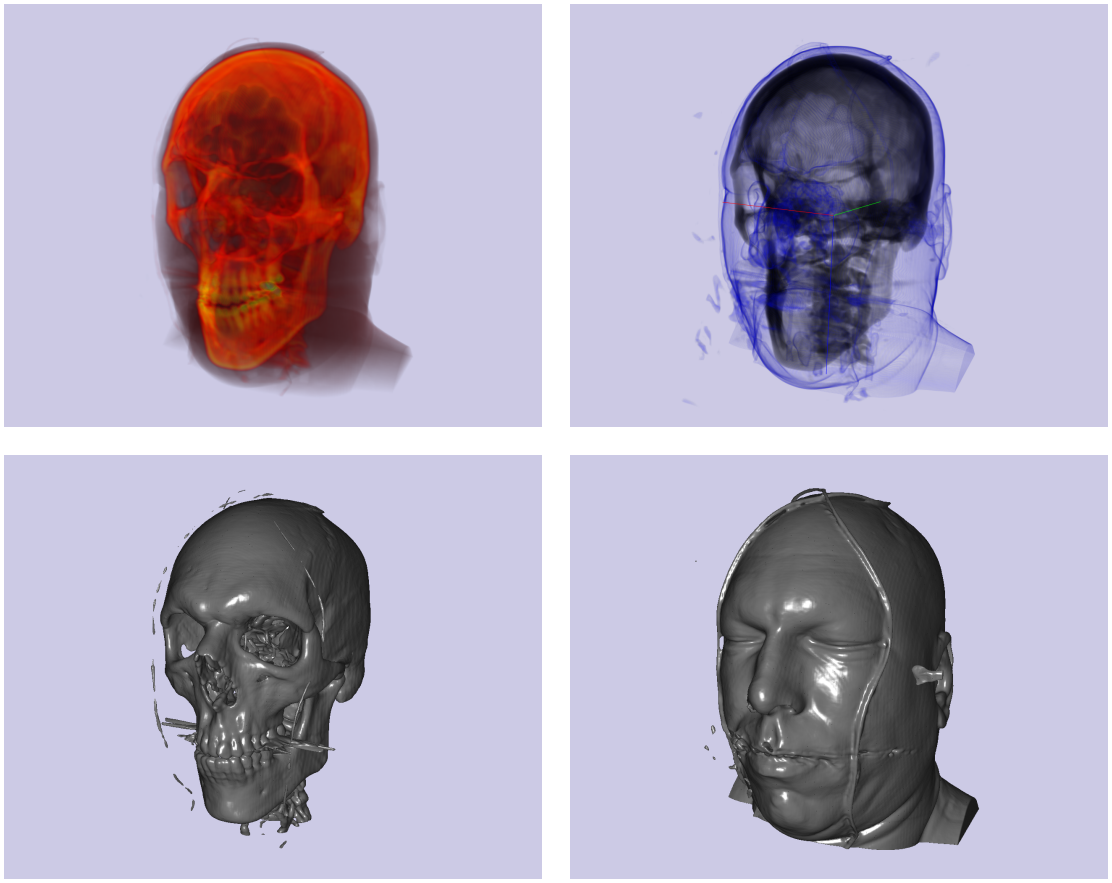


Abbildung A.3: Ein zweites Beispiel zur Abbildung 2.4 auf Seite 11.
Direkte Volumenvisualisierung *oben* (Ray-Marching-Verfahren) und Isoflächen-Visualisierung *unten* (quadratische Super-Splines). Datensatz: Vismale ($256 \times 256 \times 128$) Visible Human Project [NAT].

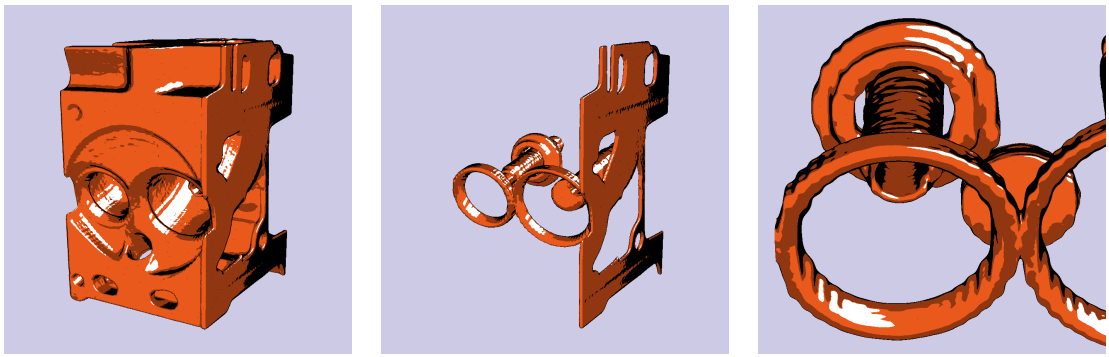


Abbildung A.4: Cel-Shading (siehe Kap. 6.4) bei kubischen C^1-Splines. Datensatz: Engine ($256 \times 256 \times 128$).

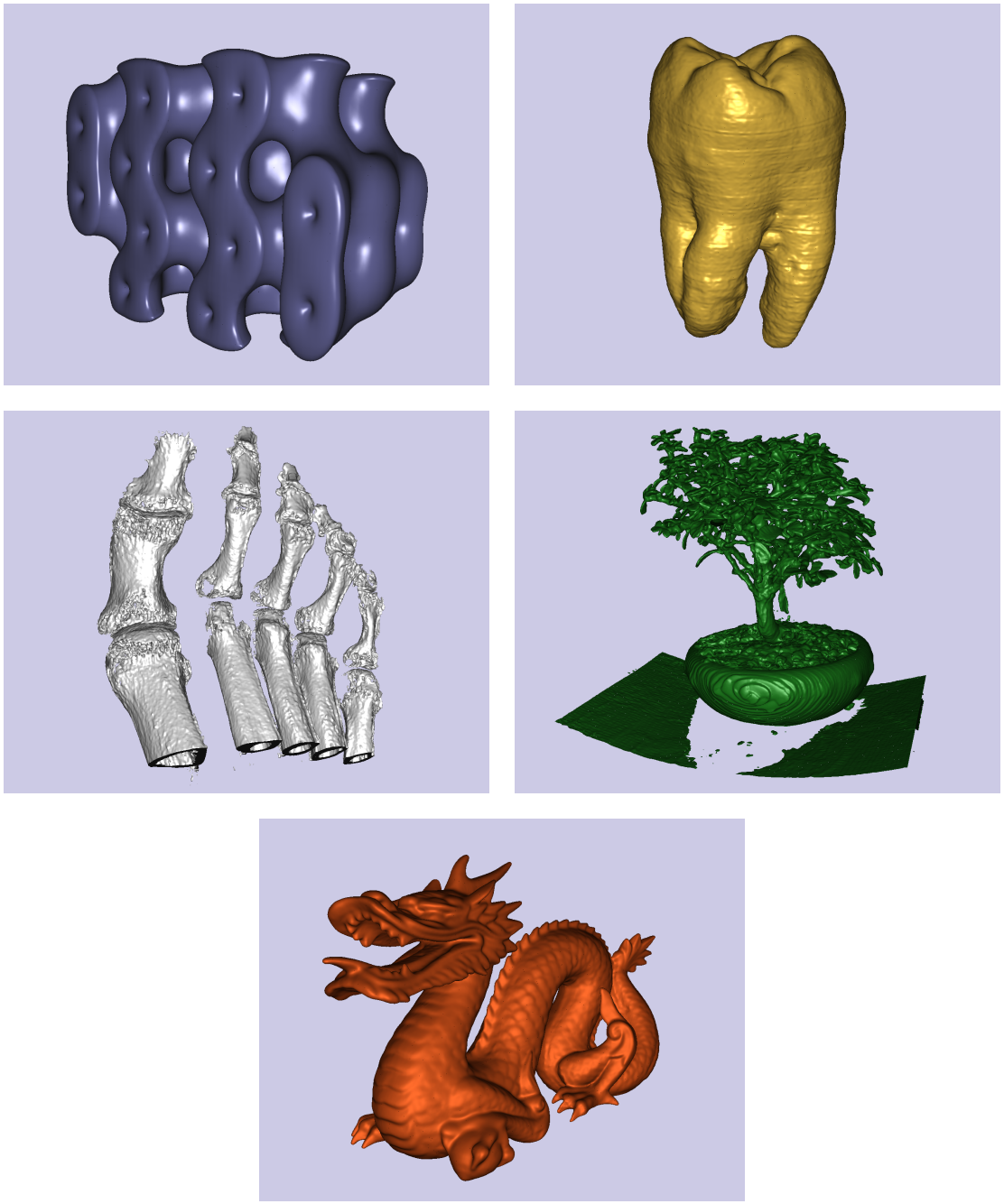


Abbildung A.5: Die fünf Datensätze, die bei den Zeitmessungen aus Kapitel 7 verwendet wurden.

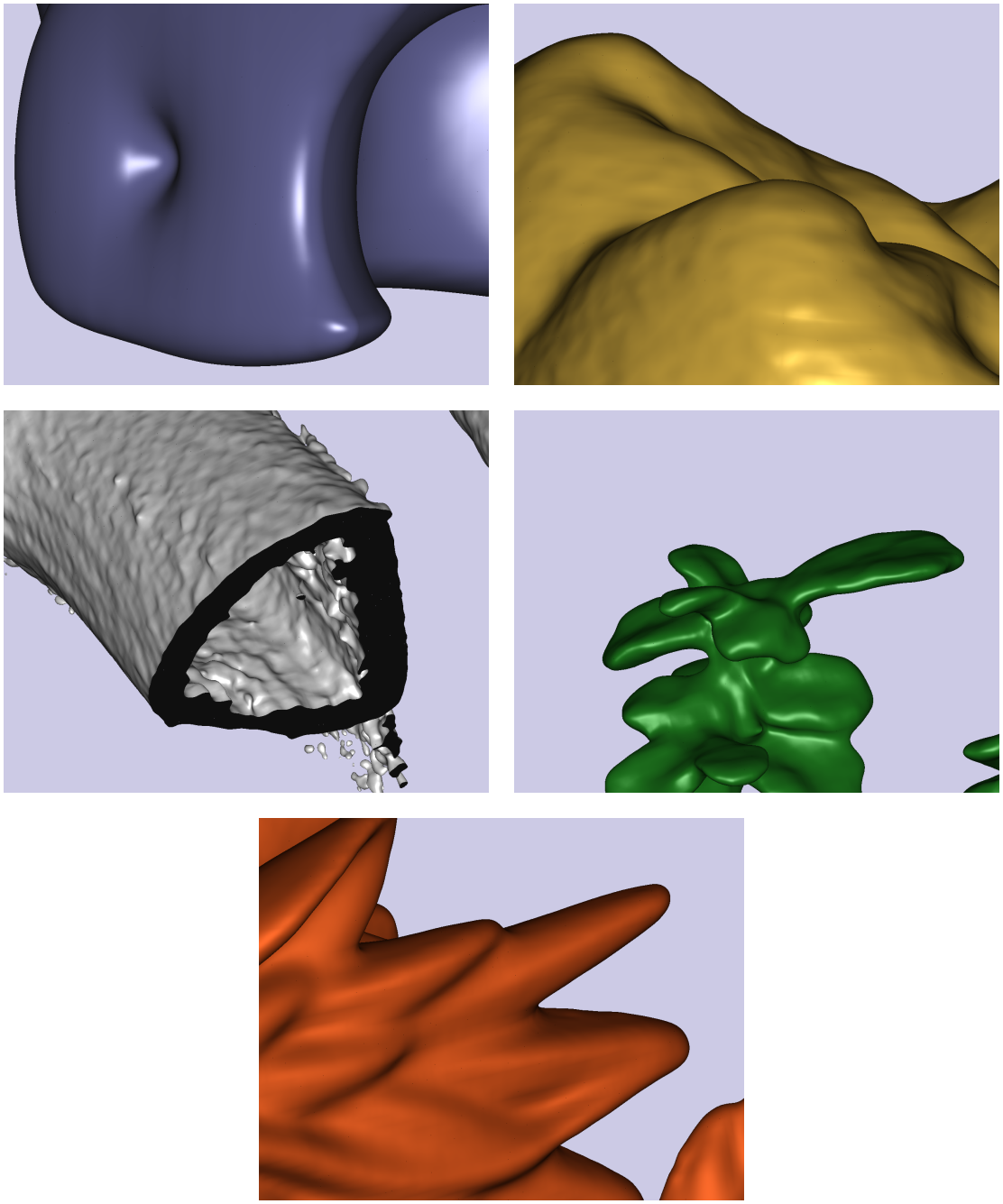


Abbildung A.6: Nahaufnahmen der fünf Datensätze aus Kapitel 7.

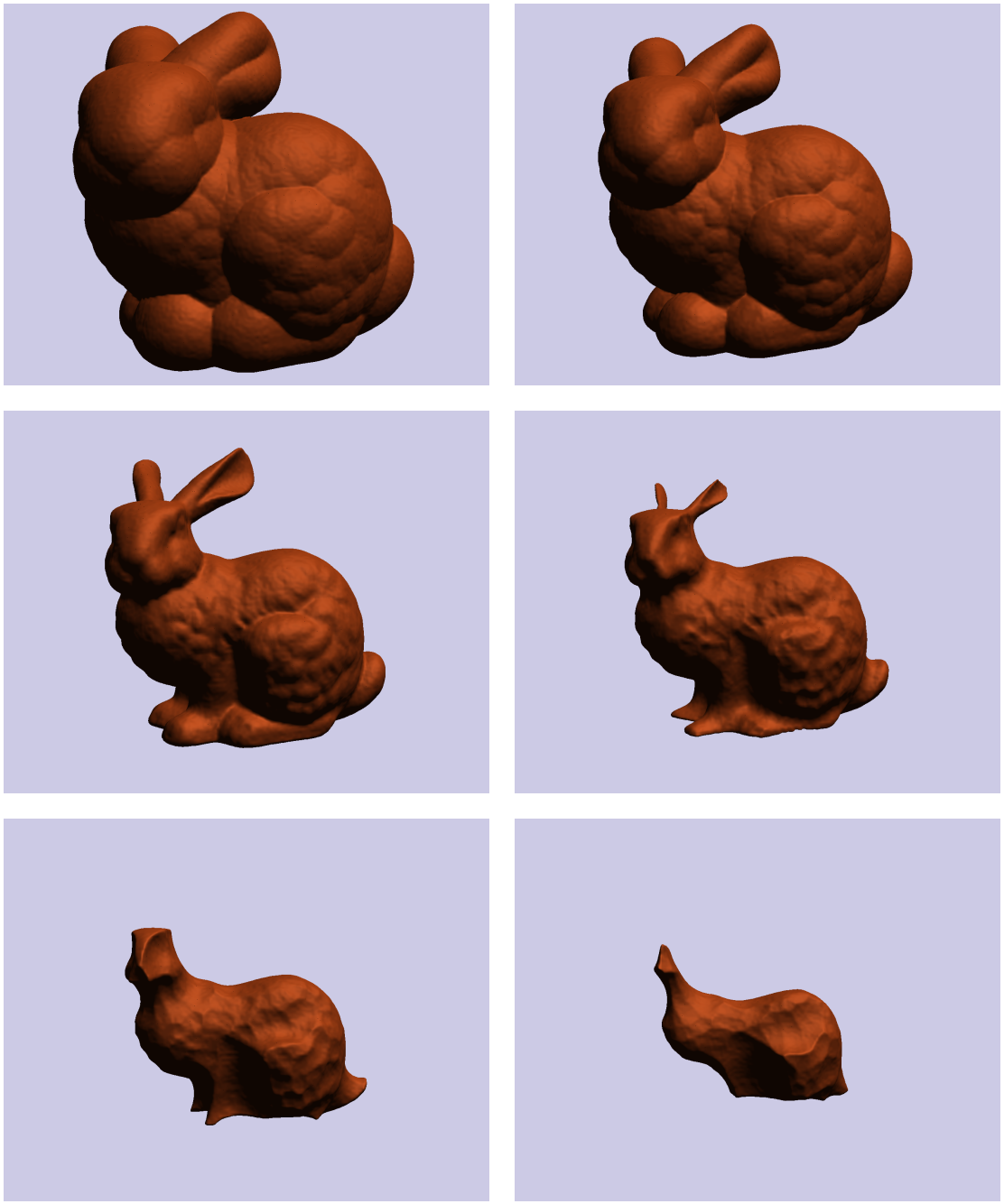


Abbildung A.7: Quadratische Super-Splines Visualisierung eines 128^3 Volumendatensatz der mittels *Signed-Distance-Functions* (siehe Seite 8) aus einem Polygonnetz des Stanford-Bunnys generiert wurde. Zwischen den verschiedenen Isowerten kann interaktiv mit ca. 12 Bildern pro Sekunden variiert werden (bei einer NVidia GTX 280).

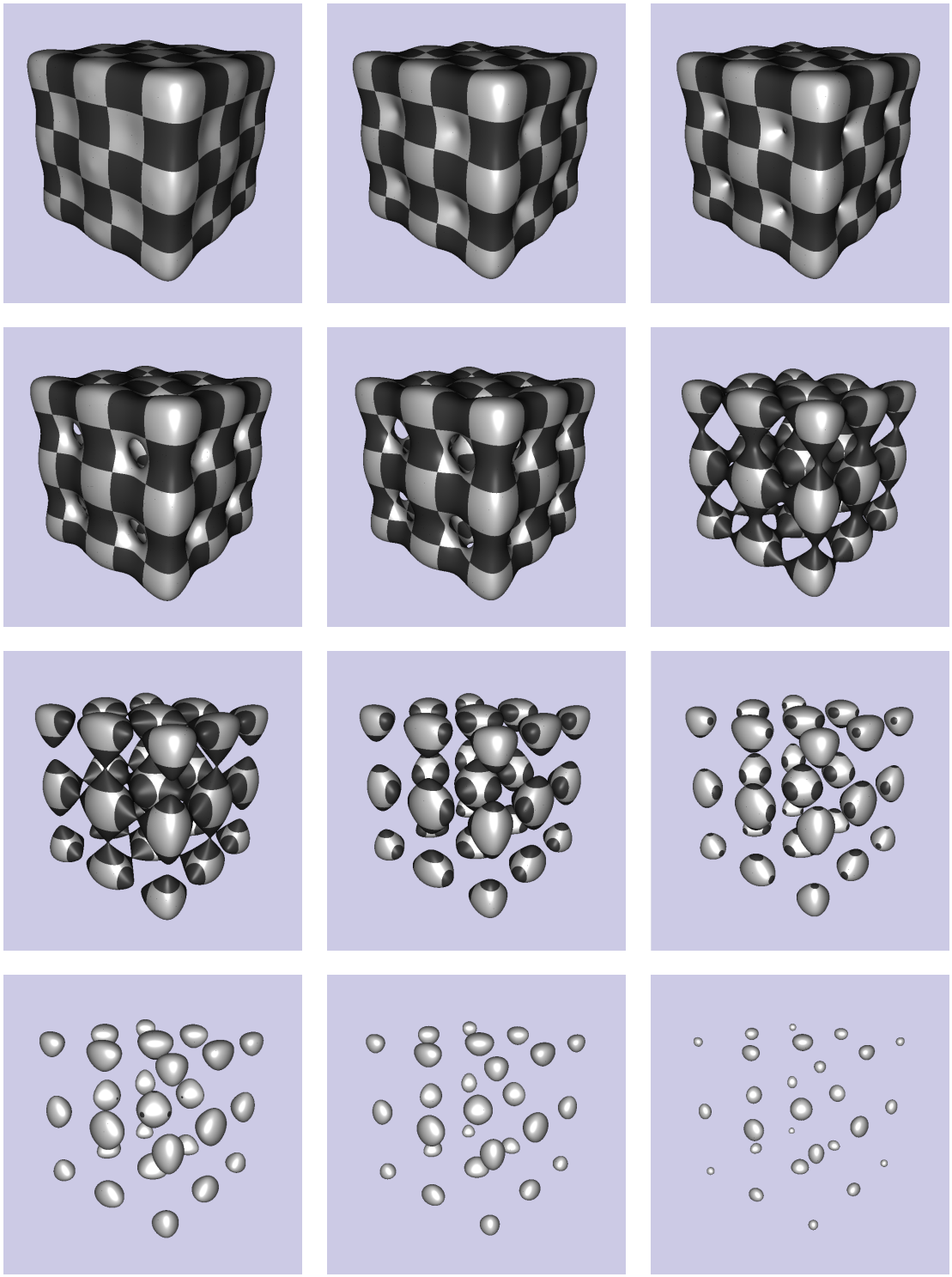


Abbildung A.8: Die Sequenz zeigt Isoflächen der mit 64^3 Datenpunkten abgetasteten impliziten Funktion $x^{16} + y^{16} + z^{16} - \cos(7x) - \cos(7y) - \cos(7z)$. Die Rekonstruktion (kubische C^1-Splines) ist interaktiv mit 15 Bildern pro Sekunde auf einer NVidia GTX 280.

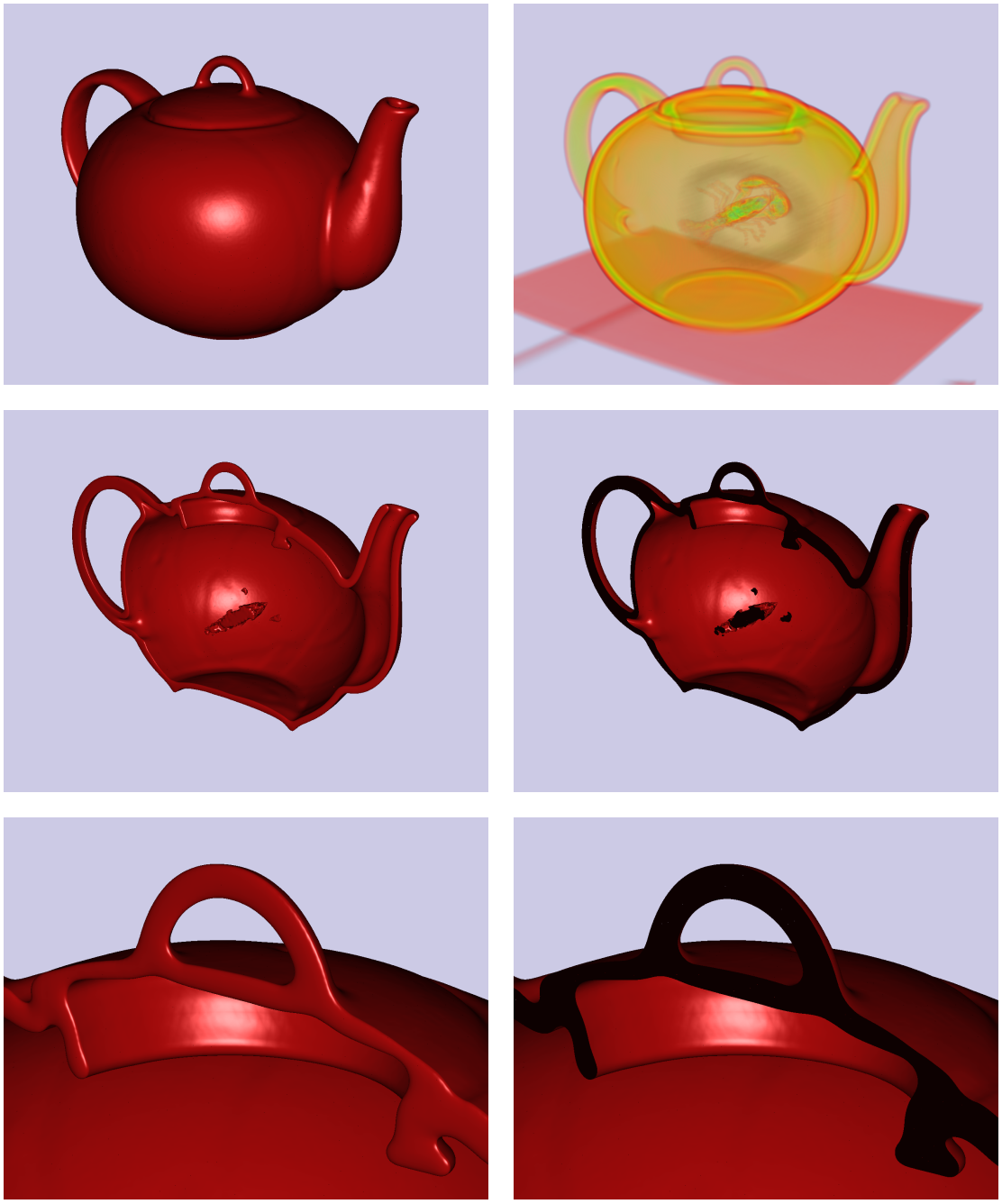


Abbildung A.9: Volumen-Clipping (siehe Kapitel 6.2) beim Boston-Teapot Datensatz ($256 \times 256 \times 178$). Die Abbildungen zeigen die kubischen C^1-Splines (außer *oben rechts*, welches mit einem Ray-Marching-Verfahren generiert wurde). Bei dem Volumen-Clipping *links* werden Teile der Voxel auf Extremwerte gesetzt. Bei dem Volumen-Clipping *rechts* werden im Fragmentshader-Programm Pixel anhand von analytischen Formeln verworfen.

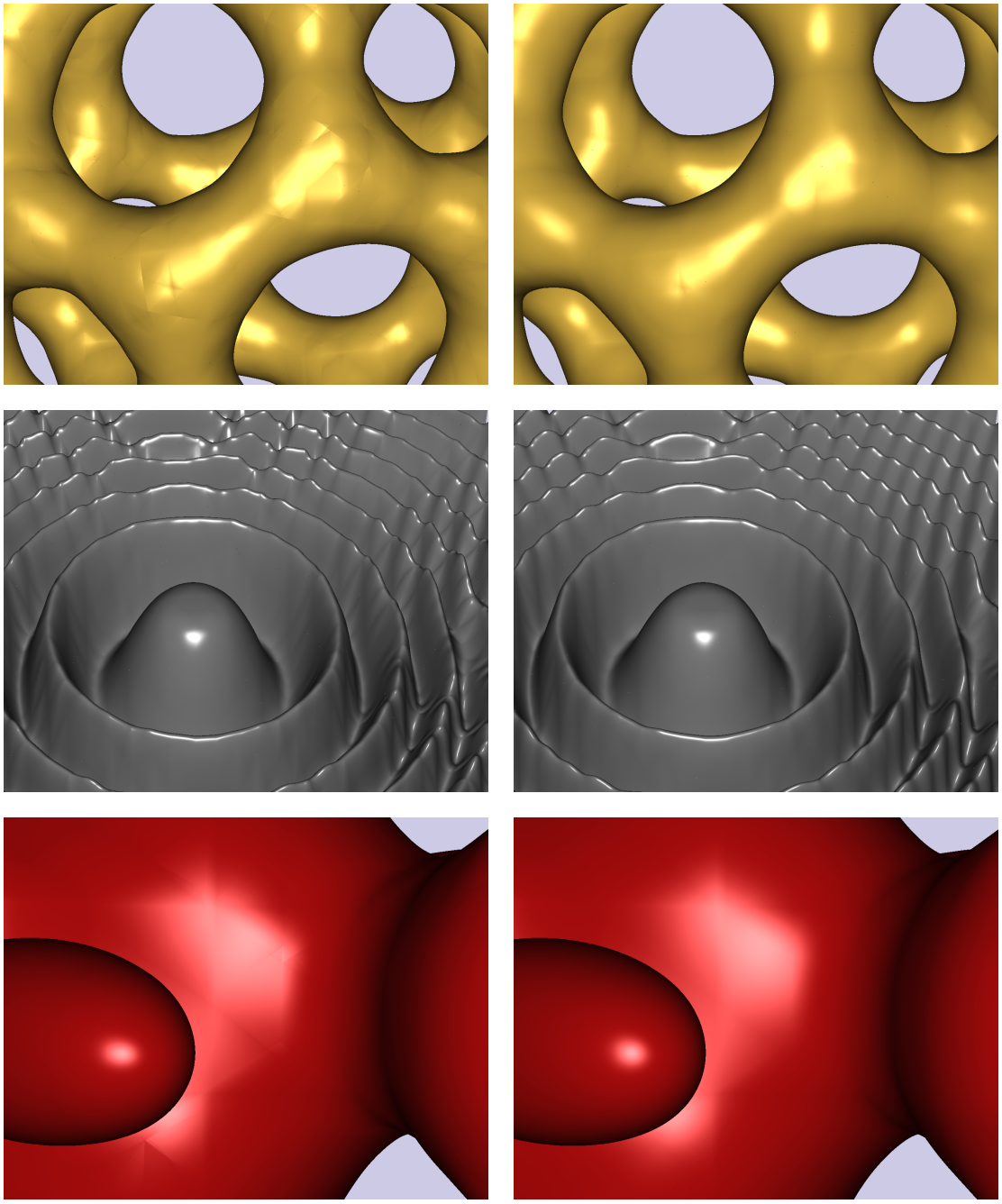


Abbildung A.10: Quadratische Super-Splines *links* und kubische C^1-Splines *rechts* und die Datensätze: Bucky 32^3, Marschnerlobb 41^3 und Fuel 64^3. Bei vielen Datensätzen reichen die Freiheitsgrade der quadratischen Super-Splines aus und ein Unterschied zu den kubische C^1-Splines ist nicht oder kaum festzustellen. Die Abbildungen *oben* zeigen Isoflächen sowie Datensätze, bei den denen ein Unterschied zu erkennen ist.

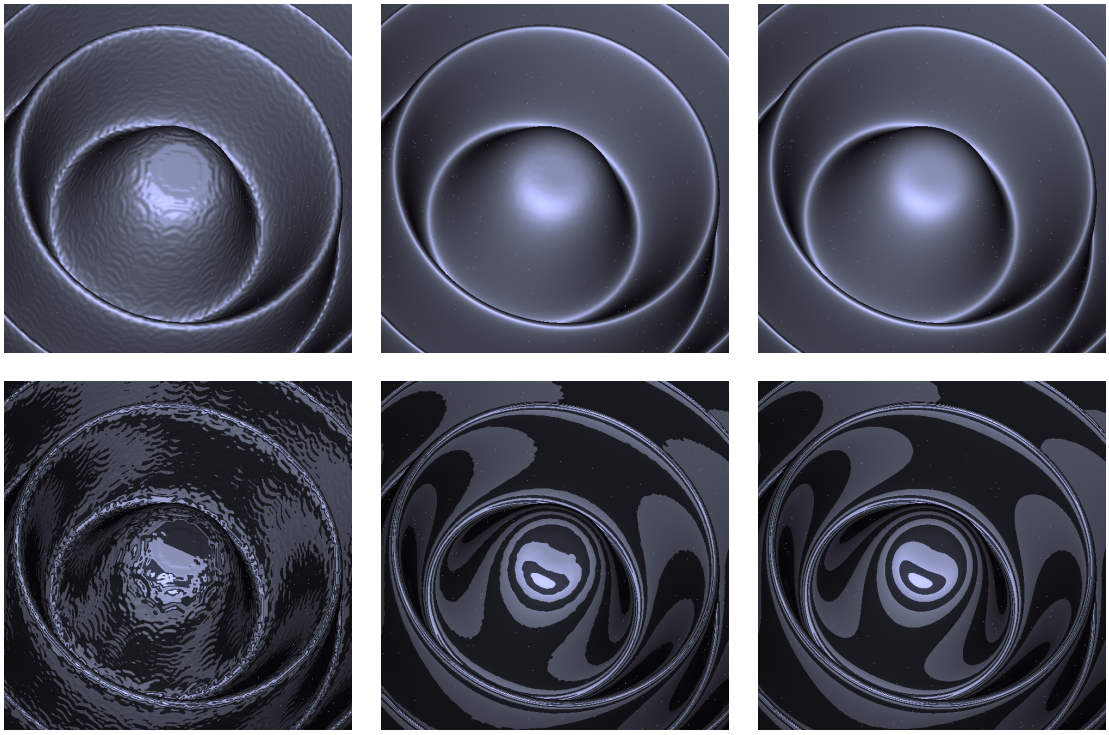


Abbildung A.11: Qualitative Rekonstruktionen, hier mittels kubischer C^1-Splines, sind bei geringer Bitbreite der Datensätze nicht möglich, da die skalaren Voxelwerte stark auf bzw. abgerundet werden. *Von links nach rechts:* 8, 12 und 16-Bit Auflösung der Marschnerlobb-Testfunktion (256^3) [ML94] und *unten* die dazugehörigen Reflektionslinien. Auch bei der im medizinischen Bereich häufig verwendeten 12-Bit Auflösung, sind leichte Dellen zu erkennen. Dieses Problem wird erst bei glatten Oberflächen und dicht abgetasteten Datensätzen sichtbar.

Abbildung A.12: Ein zweites Beispiel zur Abbildung 1.1 auf Seite 2.
Alle drei Abbildungen wurden aus demselben 8^3 Volumendatensatz, der eine implizite Torusfunktion abtastet, erstellt. *Links:* Marching Cubes Verfahren mit Flat-Shading. *Mitte:* Marching Cubes Verfahren mit Gouraud-Shading. Die Normalen wurden durch zentrale Differenzen approximiert. *Rechts:* Die in dieser Arbeit verwendeten kubische C^1-Splines.

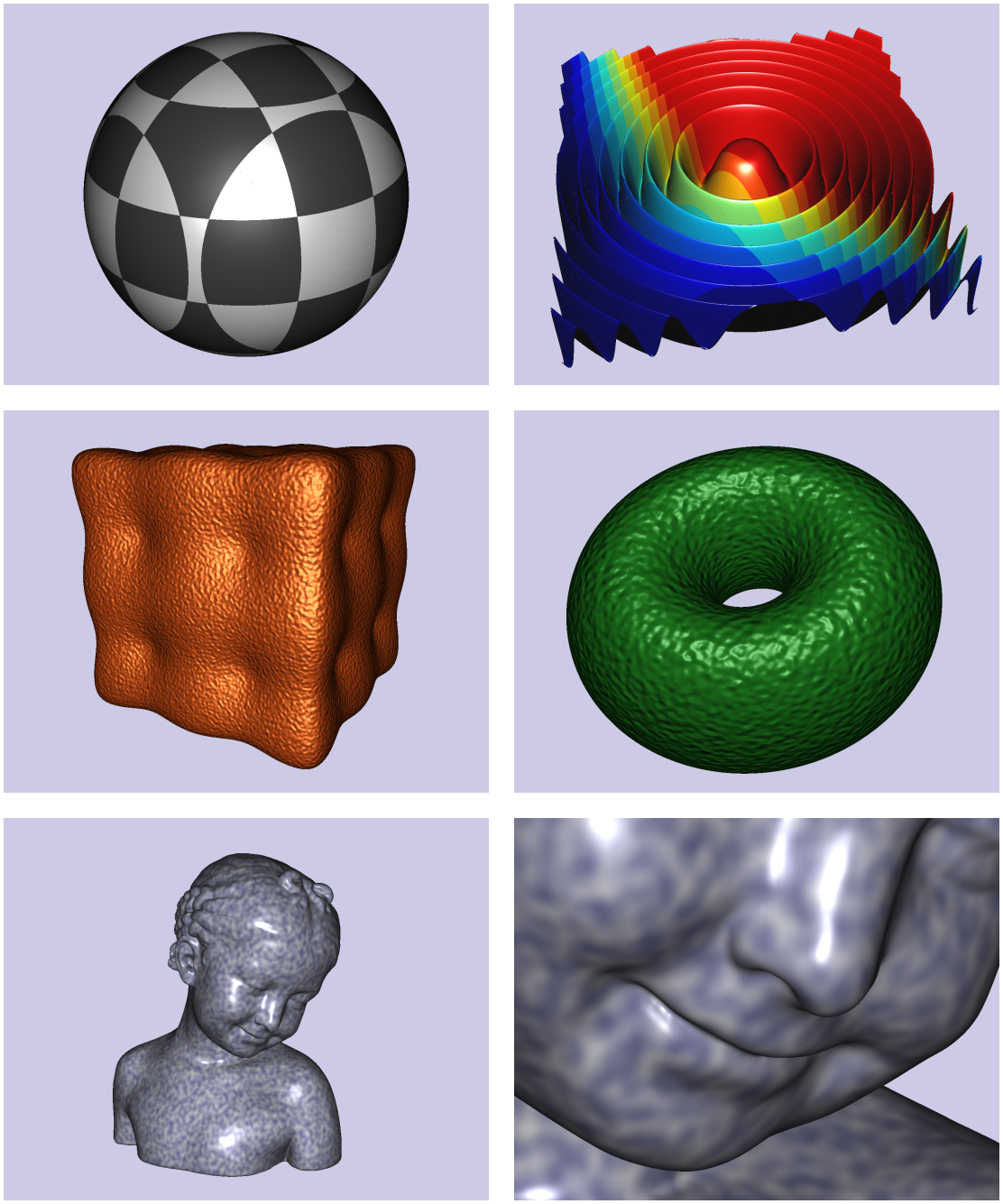


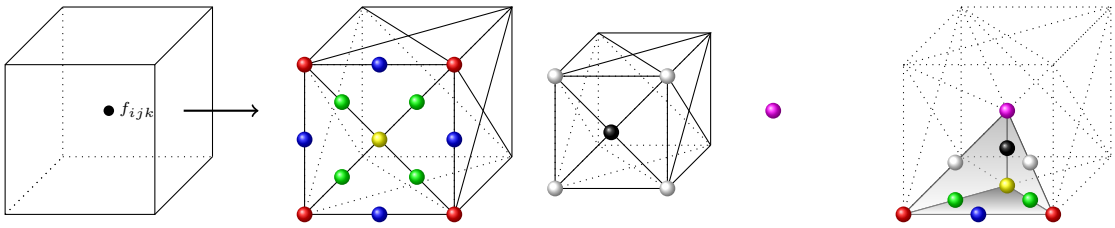
Abbildung A.13: Prozedurale Texturen (siehe Kapitel 6.3) erzielen bei der Visualisierung mit trivariaten Splines Ergebnisse hoher Qualität. Dies liegt an der glatten Rekonstruktion von ϕ sowie dem präzisen Ray-Casting-Verfahren. Auf diese Art werden bei den kubischen Splines pro Pixel C^1-stetige Texturkoordinaten bestimmt. *Oben:* Mittels einfacher, analytischer Formeln bestimmte prozedurale Texturen. *Mitte:* Prozedurales Bump-Mapping basierend auf C^2-stetigem Rauschen (Improved-Perlin-Noise [Per02]). *Unten:* Diese prozedurale Textur basiert ebenfalls auf zufällig generiertem C^2-stetigem Rauschen.

Anhang B

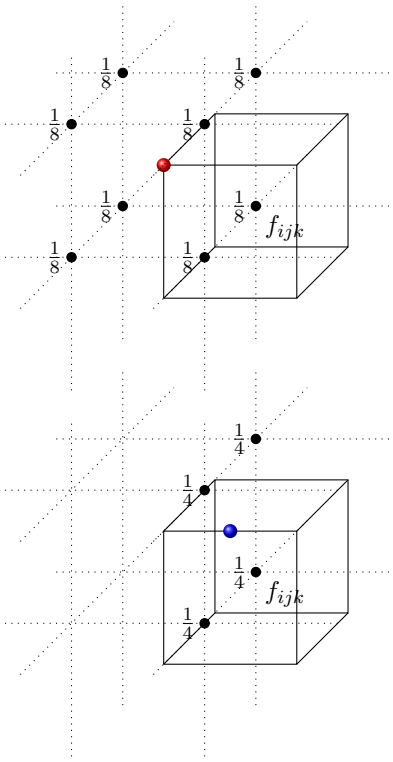
Trivariate Splines

B.1 Quadratische Super-Splines

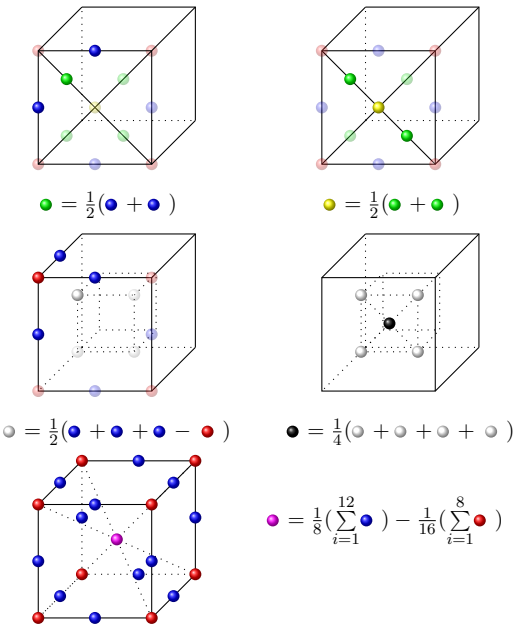
Schema



Bestimmender Satz

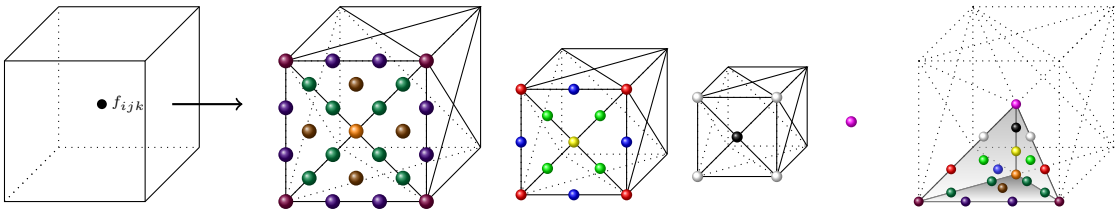


Verbleibende Koeffizienten

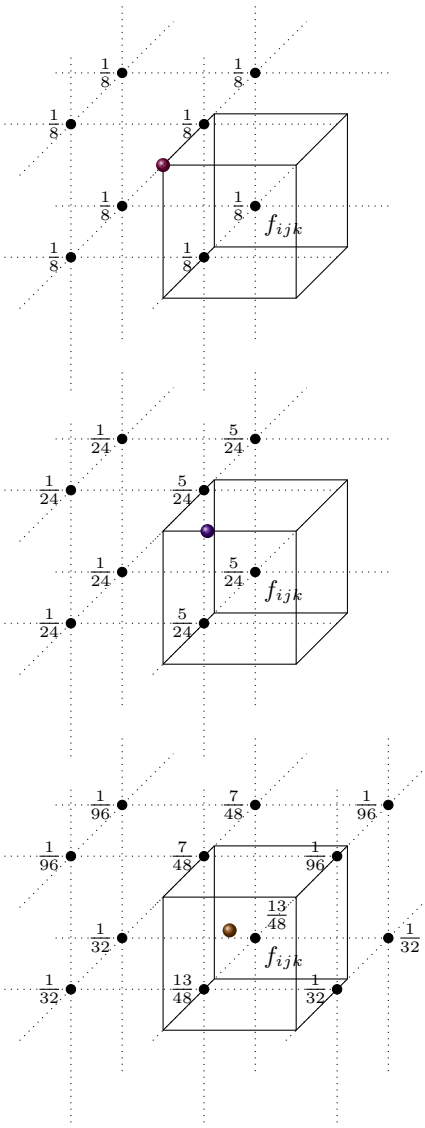


B.2 Kubische C^1-Splines

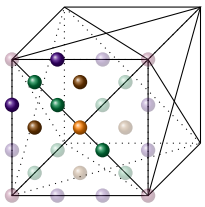
Schema



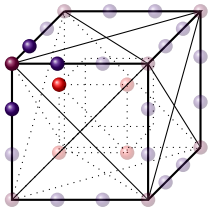
Bestimmender Satz



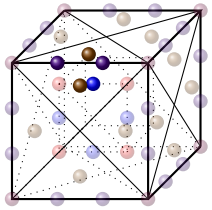
Verbleibende Koeffizienten



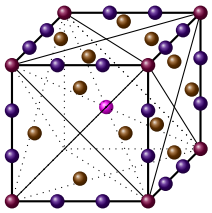
$= \frac{1}{2}(+)$
$= \frac{1}{2}(+)$
$= \frac{1}{2}(+)$



$= \frac{1}{2}(+ + -)$



$= + - \frac{1}{2}(+)$



$= \frac{1}{32}(\sum_{i=1}^{8}) + \frac{1}{8}(\sum_{i=1}^{24}) - \frac{3}{32}(\sum_{i=1}^{24})$

Anhang C

Symboltabelle

ϕ	skalares Dichtefunktion
f_{ijk}	diskreter Volumendatenwert
c	Isowert
I_c	Isofläche zum Isowert c
q	Grad eines Bernstein-Polynoms
λ_i	baryzentrische Koordinate
λ	Vektor der baryzentrischen Koordinaten
$\mathbf{b}_{ij}$	Koeffizienten für univariate Polynome in Bernstein-Bézier-Form
$\mathbf{b}_{ijk\ell}$	Koeffizienten für trivariate Polynome in Bernstein-Bézier-Form
B^q_{ij}	univariate Bernstein-Polynome des Grades q
$B^q_{ijk\ell}$	trivariate Bernstein-Polynome des Grades q
$\mathbf{v}, \mathbf{z}, \mathbf{x}$	Punkte im $\mathbb{R}^2$, $\mathbb{R}^3$
C^r	Stetigkeit der r-ten Ableitung bei Splines
T	Tetraederelement
Δ	Tetraeder-Partition
Q	Quader
$\Diamond$	Uniforme Quader-Partition
$\xi_{ijk\ell}$	Grundpunkt (engl. Domain-Point) eines trivariaten Polynoms in BB-form
$\mathcal{D}_{q,T}$	Menge aller Grundpunkte eines Polynoms des Grades q bezüglich T
$\mathcal{D}_{q,\Delta}$	Menge aller Grundpunkte der Partition Δ
Γ	Bestimmender Satz (engl. Determining Set)
$\Gamma_{q,Q}$	Bestimmender Satz für den Quader Q
Ω	Grundmenge im $\mathbb{R}^3$ (das zu visualisierende Gebiet)
Δ_6	Typ-6 Tetraeder-Partition der Grundmenge Ω
$\mathcal{S}^r_q(\Delta)$	Splineraum der C^r-stetigen Splines des Grades q
$s\|_T$	Splinefunktion bezüglich T
$\mathcal{P}_q$	Raum der trivariaten Polynome des Grades q

Literaturverzeichnis

[ABC*06] Asanovic K., Bodik R., Catanzaro B. C., Gebis J. J., Husbands P., Keutzer K., Patterson D. A., Plishker W. L., Shalf J., Williams S. W., Yelick K. A.: *The Landscape of Parallel Computing Research: A View from Berkeley.* Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[AMD09] AMD, Inc.: Offizielle webseite der ati stream technology. http://ati.amd.com/technology/streamcomputing/index.html, 2009.

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering 3rd Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2008.

[And09] Andermahr W.: Computerbase.de, diverse artikel. http://www.computerbase.de/artikel/hardware/grafikkarten/, 2009.

[ATI01] ATI Technologies Inc.: Truform whitepaper. http://ati.amd.com/products/pdf/truform.pdf, 2001.

[Bar] Bartz D.: University of tübingen, volume dataset repository. http://www.volvis.org/.

[Ble89] Blelloch G. E.: Scans as primitive parallel operations. *IEEE Trans. Comput. 38*, 11 (1989), 1526–1538.

[Ble90] Blelloch G. E.: *Prefix Sums and Their Applications.* Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[Bly06] Blythe D.: The direct3d 10 system. *ACM Trans. Graph. 25*, 3 (2006), 724–734.

[Car05] Carucci F.: Inside geometry instancing. In *GPU Gems II*, Pharr M., (Ed.). Addison-Wesley, 2005, pp. 47–67.

[CLT08] Crane K., Llamas I., Tariq S.: Real-time simulation and rendering of 3d fluids. In *GPU Gems III*, Nguyen H., (Ed.). Addison Wesley, 2008, pp. 633–675.

[DZTS08] Dyke C., Ziegler G., Theobalt C., Seidel H.-P.: High-speed marching cubes using histogram pyramids. *Computer Graphics Forum 27*, 8 (2008).

[EHK*04] Engel K., Hadwiger M., Kniss J. M., Lefohn A. E., Rezk-Salama C., Weiskopf D.: ACM SIGGRAPH 2004 course notes on Real-Time Volume Graphics, Number 28. http://old.vrvis.at/via/resources/course-volgraphics-2004/, 2004.

[EHK*06a] Engel K., Hadwiger M., Kniss J., C.Rezk-Salama, Weiskopf D.: *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006.

[EHK*06b] Engel K., Hadwiger M., Kniss J., C.Rezk-Salama, Weiskopf D.: *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006, ch. 14.4 Surface and Isosurface Curvature, pp. 368–380.

[Fuh07] Fuhrmann S.: Volume data generation from triangle meshes using the signed distance function. http://simonfuhrmann.de/uni/bthesis.html, 2007. Bachelor Thesis, TU-Darmstadt, Fachbereich Informatik.

[Gei08] Geiss R.: Generating complex procedural terrains using the gpu. In *GPU Gems III*, Nguyen H., (Ed.). Addison-Wesley, 2008, pp. 7–37.

[GJD05] Goetz F., Junklewitz T., Domik G.: Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics 2005* (2005).

[Gol08] Gold M.: GL_EXT_draw_instanced Specification. http://www.opengl.org/registry/specs/EXT/draw_instanced.txt, 2008.

[GPG] GPGPU: General-Purpose Computation on GPUs. http://gpgpu.org/.

[Har98] Hartmann E.: A marching method for the triangulation of surfaces. *The Visual Computer 14*, 3 (1998), 95–108.

[HOS*09] Harris M., Owens J., Sengupta A., Zhang Y., Davidson A.: Cudpp: Cuda data parallel primitives library. http://www.gpgpu.org/developer/cudpp/, 2009.

[HSJ08] Harris M., Sengupta S., J.D.Owens: Parallel prefix sum (scan) with cuda. In *GPU Gems III*, Nguyen H., (Ed.). Addison-Wesley, 2008, pp. 851–876.

[HSS*05] Hadwiger M., Sigg C., Scharsach H., Bühler K., Gross M. H.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum 24*, 3 (2005), 303–312.

[INT07] INTEL Corporation: Ct: A flexible parallel programming model for tera-scale architectures. http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf, 2007.

[INT09] INTEL Corporation: Offizielle ct webseite von intel. http://intel.com/go/Ct, 2009.

[JC06] Johansson G., Carr H.: Accelerating marching cubes with graphics hardware. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (New York, NY, USA, 2006), ACM, p. 39.

[KBR] Kessenich J., Baldwin D., Rost R.: The OpenGL Shading Language. http://www.opengl.org/documentation/glsl/.

[KBSS01] Kobbelt L. P., Botsch M., Schwanecke U., Seidel H.-P.: Feature sensitive surface extraction from volume data. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 57–66.

[KFU*09] Kalbe T., Fuhrmann S., Uhrig S., Zeilfelder F., Kuijper A.: A new projection method for point set surfaces. In *Eurographics 2009 (short paper)* (2009).

[Khr09] Khronos Group: Offizielle webseite von opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2009.

[KWTM03] Kindlmann G., Whitaker R., Tasdizen T., Möller T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. *Visualization Conference, IEEE* (2003), 67. http://www.cs.utah.edu/~gk/papers/vis03/.

[KZ08] Kalbe T., Zeilfelder F.: Hardware-accelerated, high-quality rendering based on trivariate splines approximating volume data. *Computer Graphics Forum 27*, 2 (2008), 331–340.

[LB08] Lichtenbelt B., Brown P.: GL_EXT_gpu_shader4 Specification. http://www.opengl.org/registry/specs/EXT/gpu_shader4.txt, 2008.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), vol. 21, ACM Press, pp. 163–169.

[LS07] Lai M.-J., Schumaker L. L.: *Spline Functions On Triangulations*. Cambridge University Press, 2007.

[Map09] Maplesoft, Waterloo Maple Inc.: Maple - ein computeralgebrasystem. http://www.maplesoft.com/, 2009.

[ML94] Marschner S. R., Lobb R. J.: An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 100–107.

[MSM05] Mattson T. G., Sanders B. A., Massingill B. L.: *Patterns For Parallel Programming*. Addison-Wesley, 2005.

[NAT] NATIONAL LIBRARY OF MEDICINE: Visible human project. http://www.nlm.nih.gov/research/visible/visible_human.html.

[Nie04] Nielson G. M.: Dual marching cubes. In *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 489–496.

[Nok] Nokia Corporation: Qt Software: Eine Programmbibliothek für plattformübergreifende Softwareentwicklung. http://www.qtsoftware.com/.

[NRSZ05] Nürnberger G., Rössl C., Seidel H.-P., Zeilfelder F.: Quasi-interpolation by quadratic piecewise polynomials in three variables. *CAGD, Vol. 22* (2005), 221–249.

[NVI] NVIDIA CORPORATION: Technical report - glsl pseudo instancing. http://developer.nvidia.com/page/opengl.html.

[NVI08] NVIDIA CORPORATION: Nvidia gpu programming guide geforce 8 and 9 series. http://developer.nvidia.com/object/gpu_programming_guide.html, 2008.

[NVI09a] NVIDIA Corporation: Cuda 2.1 programming guide. http://www.nvidia.de/object/cuda_develop_emeai.html, 2009.

[NVI09b] NVIDIA Corporation: Cuda compute unified device architecture. http://www.nvidia.de/cuda, 2009.

[NVI09c] NVIDIA Corporation: Cuda software development kit: Version 2.1. http://www.nvidia.com/object/cuda_get.html, 2009.

[OJL*07] Owens, John D., Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, Aaron E., Purcell, Timothy J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (March 2007), 80–113.

[Pas04] Pascucci V.: Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. In *In Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym* (2004), pp. 293–300.

[PBP02] Prautzsch H., Boehm W., Paluszny M.: *Bézier and B-Spline Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[Per02] Perlin K.: Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*

(New York, NY, USA, 2002), ACM, pp. 681–682. http://mrl.nyu.edu/~perlin/paper445.pdf.

[PSL*98] Parker S., Shirley P., Livnat Y., Hansen C., pike Sloan P.: Interactive ray tracing for isosurface rendering. In *In IEEE Visualization '98* (1998), pp. 233–238.

[Roe] Roettger S.: University of erlangen, the volume library. http://www9.informatik.uni-erlangen.de/External/vollib/.

[Ros06] Rost R. J.: *OpenGL Shading Language, 2nd Edition*. Addison-Wesley Professional, 2006.

[RZNS03] Rössl C., Zeilfelder F., Nürnberger G., Seidel H.-P.: Visualization of volume data with quadratic super splines. *IEEE Vis.* (2003), 393–400.

[RZNS04] Rössl C., Zeilfelder F., Nürnberger G., Seidel H.-P.: Reconstruction of volume data with quadratic super splines. *IEEE Trans. Vis. & CG 10*, 4 (2004), 397–409.

[Sei93] Seidel H.-P.: An introduction to polar forms. *IEEE Computer Graphics & Applications 13*, 1 (1993), 38–46.

[SGS06] Stoll C., Gumhold S., Seidel H.-P.: Incremental raycasting of piecewise quadratic surfaces on the gpu. *Symposium on Interactive Ray Tracing* (2006), 141–150.

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 97–106.

[SS02] Strothotte T., Schlechtweg S.: *Non-photorealistic computer graphics: modeling, rendering, and animation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Sum04] Sumanaweera T.: Applying real-time shading to 3d ultrasound visualization. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, pp. 693–707.

[SWND05] Shreiner D., Woo M., Neider J., Davis T.: *OpenGL Programming Guide, Version 2 (5th Edition)*. Addison-Wesley Professional, 2005.

[SZ07] Sorokina T., Zeilfelder F.: Local Quasi-Interpolation by cubic C^1 splines on type-6 tetrahedral partitions. *IMJ Numerical Analysis 27* (2007), 74–101.

[SZH*05] Schlosser G., Zeilfelder F., Hesser J., Rössl C., Nürnberger G., Männer R., Seidel H.-P.: Fast visualization by shear-warp on quadratic super-spline models using wavelet data decompositions. *Proc. IEEE Vis. 2005* (2005), 45–55.

[The02] Theisel H.: Exact isosurfaces for marching cubes. *Computer Graphics Forum 21*, 1 (2002), 19–31.

[TPH98] Treece G. M., Prager R. W., H.Gee A.: *Regularised marching tetrahedra: improved iso-surface extraction.* Tech. Rep. CUED/F-INFENG/TR 333, Speech, Vision and Robotics Group, Department of Engineering, Cambridge University, Sept. 1998. http://svr-www.eng.cam.ac.uk/reports/svr-ftp/treece_tr333.pdf.

[TS07] Tatarchuk N., Shopf J.: Real-time medical visualization with firegl, siggraph 2007, amd technical talk. http://ati.amd.com/developer/gdc/2007/MedicalVisualization.pdf, 2007.

[TSD07] Tatarchuk N., Shopf J., DeCoro C.: Real-time isosurface extraction using the gpu programmable geometry pipeline. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, pp. 122–137.

[vKvdBT07] van Kooten K., van den Bergen G., Telea A.: Point-based visualization of metaballs on a gpu. In *GPU GEMS III*, Nguyen H., (Ed.). Addison-Wesley, 2007, ch. 7, pp. 123–148.

[VPBM01] Vlachos A., Peters J., Boyd C., Mitchell J. L.: Curved pn triangles. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM, pp. 159–166.

[Web08] Weber D.: Trivariate bernstein-bézier-techniken für finite elemente zur interaktiven simulation von deformationen, 2008. Diplomarbeit, TU-Darmstadt, Fachbereich Informatik.

[WH94] Witkin A., Heckbert P.: Using particles to sample and control implicit surfaces. In *Computer Graphics (Proc. SIGGRAPH '94)* (1994), vol. 28.

[Wlo03] Wloka M.: Batch, batch, batch: What does it really mean? presentation at game developers conference 2003. http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf, 2003.