

**Technische Universität Kaiserslautern**

**Fernstudium Medizinische Physik**

**Masterarbeit**

**Vergleich von Volumenmodellen**

**von**

**Kolja Kähler**

**Betreuer:**

**Herr Dr. Heiko Hengen**

Ich versichere, dass ich diese Masterarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum

Unterschrift

---

---

# Inhaltsverzeichnis

<b>1. Einführung</b>	<b>1</b>
1.1. Explizite und implizite Flächen . . . . .	2
1.2. Konvertierungen . . . . .	3
<b>2. Das reguläre Voxelgitter</b>	<b>5</b>
2.1. Datenstruktur . . . . .	5
2.2. Konvertierung von Dreiecksnetzen . . . . .	6
2.2.1. Ergebnisse . . . . .	7
2.3. Grundlagen der Visualisierung . . . . .	8
2.3.1. Rekonstruktion . . . . .	10
2.3.2. Transferfunktion . . . . .	10
2.3.3. Lichttransport im Volumen . . . . .	10
2.3.4. Das Volumen–Rendering–Integral . . . . .	11
2.3.5. Diskretisierung des Integrals . . . . .	12
2.3.6. Lokales Beleuchtungsmodell . . . . .	12
2.4. Hardwareunterstützte Renderingverfahren . . . . .	14
2.4.1. Transferfunktion mittels Textur–Lookup . . . . .	14
2.4.2. 3D–Texturen . . . . .	15
2.4.3. Compositing texturierter Polygone . . . . .	16
2.4.4. Berechnung von Gradienten für lokale Beleuchtung . . . . .	17
2.4.5. Ergebnisse . . . . .	18
2.5. Berechnung und Darstellung von Schnitten . . . . .	20
2.5.1. Hardwarebeschleunigtes Selektionsvolumen . . . . .	21
2.5.2. Ergebnisse . . . . .	22
2.6. Zusammenfassung . . . . .	23
<b>3. Distanztransformation</b>	<b>25</b>
3.1. Verfahren zur Erzeugung von Distanztransformationen . . . . .	26
3.1.1. Inkrementelle Propagierung . . . . .	26
3.1.2. Generalisierte Voronoidiagramme . . . . .	27
3.1.3. Vereinfachte Scankonvertierung . . . . .	28
3.2. Implementierung eines modifizierten Algorithmus . . . . .	32
3.3. Ergebnisse . . . . .	34
3.3.1. Schnitte zwischen Distanzfeldern . . . . .	36
3.4. Zusammenfassung . . . . .	37

---

<b>4. Tetraedisierung</b>	<b>39</b>
4.1. Verfahren zur Erzeugung von Tetraedisierungen . . . . .	39
4.2. Isosurface Stuffing . . . . .	41
4.2.1. Ablauf der Tetraedisierung . . . . .	42
4.2.2. Parameterwahl . . . . .	42
4.2.3. Implementierung der Gitterdatenstrukturen . . . . .	43
4.2.4. Implementierung der Stempel-Tabelle . . . . .	45
4.3. Ergebnisse . . . . .	46
4.4. Zusammenfassung . . . . .	48
<b>5. Zusammenfassung</b>	<b>51</b>
5.1. Ausblick . . . . .	52
<b>Abbildungsverzeichnis</b>	<b>53</b>
<b>Tabellenverzeichnis</b>	<b>55</b>
<b>Literaturverzeichnis</b>	<b>57</b>
<b>A. GPU-Programmierung: Ein Überblick</b>	<b>63</b>



# 1 Einführung

Die zahlreichen Darstellungsweisen dreidimensionaler Objekte in der Computergrafik lassen sich in zwei große Klassen einteilen: Oberflächen- und Volumenrepräsentationen. Die Art der vorliegenden Daten und der Verwendungszweck bestimmt die Wahl des passenden Modells. Oberflächenmodelle bilden die Form eines Objektes durch ihre äußere Hülle ab. Neben der weitverbreiteten Darstellung durch eine Dreiecksnetz sind auch Flächen höherer Ordnung, beispielsweise NURBS-Flächen im CAD-Bereich, gebräuchlich. Effiziente Datenstrukturen und moderne Grafikhardware erlauben eine extrem schnelle Darstellung detaillierter triangulierter Flächen schon auf preiswerten PCs und Workstations.

In der medizinischen Bildgebung hingegen dominieren Volumendaten, wie sie aus tomografischen Verfahren anfallen. Auch in der wissenschaftlichen Visualisierung sind skalare Volumendatensätze vertreten, beispielsweise zur Darstellung gasförmiger Phänomene wie Wolken oder Rauch, oder zur Modellierung von Flüssigkeiten. Die Parameter eines Volumenmodells beziehen sich dabei nicht unbedingt auf die Geometrie eines Objektes: In einem Klimamodell können die Volumenelemente für Temperatur oder Luftfeuchtigkeit stehen. Zwei der meistverbreiteten Volumenmodelle basieren auf einer Diskretisierung über ein Gitter: Zum einen das reguläre Voxelgitter, das für Tomografiedaten typisch ist. Zum anderen das tetraedische Gitter, das besonders für Finite-Elemente-Techniken bevorzugt eingesetzt wird. Mit den verschiedenen Aspekten der Handhabung von Volumendaten – also Erzeugung, Speicherung, Modellierung, Analyse, Manipulation, Anzeige, Animation – befasst sich die Volumengrafik (*volume graphics*), ein eigenes Teilgebiet der Computergrafik [CKY00].

Trotz aktueller Bemühungen, Volumendarstellungen als universelle Grafikprimitiven zu verwenden [PF01], werden die verschiedenen Flächen- und Volumendarstellungen auf absehbare Zeit gleichberechtigt koexistieren. Da viele Algorithmen *entweder* auf Flächen *oder* auf Volumenmodellen arbeiten, ist es daher oft notwendig, einen gegebenen Datensatz von der einen in die andere Form zu konvertieren. Es stellt sich hierbei zunächst die Frage, wie sich die Unterschiede und Zusammenhänge zwischen Flächen- und Volumenrepräsentationen ausdrücken lassen. Eine gängige Betrachtungsweise setzt bei der Darstellung der Oberfläche eines in volumetrischer Form vorliegenden Objektes an.

## 1.1 Explizite und implizite Flächen

Allgemein kann man die Volumendarstellung eines Objektes als Skalarfeld  $f(\mathbf{p})$ ,  $\mathbf{p} \in \mathbb{R}^3$ , auffassen, das Punkten im dreidimensionalen Raum jeweils einen Wert zuordnet. Unabhängig von der konkreten Struktur des Volumenmodells ist eine Oberfläche des Objektes dann *implizit* gegeben, d.h. als die Menge  $\{\mathbf{p} | f(\mathbf{p}) = 0\}$  der Punkte, an denen die Funktion  $f$  den Wert 0 annimmt. Für einen gegebenen Punkt  $\mathbf{p}$  kann in einer impliziten Darstellung also durch Auswertung von  $f(\mathbf{p})$  direkt festgestellt werden, ob er auf einer solchen Isofläche liegt. Wie eng benachbart zwei Punkte auf der Oberfläche sind, ist hingegen nicht ohne weiteres zu bestimmen.

Demgegenüber ist eine *explizite* Flächenrepräsentation, etwa in Form eines Dreiecksnetzes, parametrisch: eine Funktion  $f : \mathbb{R}^2 \mapsto \mathbb{R}^3$  bildet eine Parameterebene  $\Omega$  in den dreidimensionalen Raum ab. Bei einem Dreiecksnetz ist diese Abbildung stückweise linear. Benachbarte Punkte auf der Fläche sind auch in der Parameter Ebene benachbart. Es ist durch Variation der Parameter von  $f$  leicht, die Fläche “abzuschreiten”. Ob ein beliebiger Punkt  $\mathbf{p} \in \mathbb{R}^3$  allerdings auf der Fläche liegt oder nicht, ist nicht trivial zu klären.

Ein wesentlicher Unterschied zwischen den beiden Darstellungen besteht also in der Art der direkt verfügbaren Information. Je nach Anwendungszweck erweist sich daher die eine oder die andere Repräsentation als vorteilhaft. In [PSOP01] werden beispielsweise interaktive Techniken vorgestellt, die sowohl auf Volumendaten in der Form von Voxeligittern, also auch auf triangulierten Oberflächen arbeiten, etwa zur Messung von Abständen auf im Volumenmodell segmentierten Flächen.

Algorithmen zur Glättung von Flächen, Analyse von Oberflächenkrümmungen, sowie Modellierungs- und Animationsverfahren, werten oft Nachbarschaftsinformation auf der Oberfläche aus. Meist werden topologische Mannigfaltigkeiten vorausgesetzt, in der Regel in der Form eines geschlossenen, orientierten Dreiecksnetzes. Die Repräsentation von geometrischen Objekten in volumetrischen Datenstrukturen ist hingegen vorteilhaft bei der Durchführung von Schnittoperationen, oder der Klassifikation von Raumpunkten als innen, außen, oder auf der Oberfläche liegend. Auch Techniken der Formbearbeitung und Simulationsverfahren machen sich die innere Struktur eines Volumenmodells zu Nutze [DEL<sup>+</sup>99, OH99, BNC99].

## 1.2 Konvertierungen

Die Überführung einer parametrischen Fläche in ein Volumenmodell entspricht einer Darstellung in impliziter Form durch eine geeignete Funktion. Zur Formulierung eines Volumens als implizite Funktion sind vielerlei Repräsentationen gebräuchlich, darunter algebraische Flächen [BBB<sup>+</sup>97], Radialbasisfunktionen [CBC<sup>+</sup>01], tetraedische Gitter, und Voxelgitter. Die Funktionswerte werden oft durch die Distanz zur ursprünglichen Fläche definiert [Kau87]. Umgekehrt werden zur Erzeugung einer parametrischen Fläche aus einem Volumenmodell Punkte auf der impliziten Fläche gefunden und beispielsweise zu einem Polygonnetz verbunden [Blo88].

Ausgehend von dieser allgemeinen Problemstellung, betrachtet die vorliegende Masterarbeit die Verbindungen zwischen Dreiecksnetzen als Oberflächenrepräsentation, und Volumendarstellungen in Form regulärer Voxelgitter oder tetraedischer Gitter. Insbesondere werden Algorithmen zur Visualisierung von Volumendaten, Konvertierung der verschiedenen Repräsentationen in die anderen Formen, sowie zur Durchführung von booleschen Operationen (Schnitte und Selektionen) vorgestellt. Eine weitere Volumenrepräsentation, das diskretisierte Distanzfeld, nimmt eine zentrale Stellung ein, indem es die Verbindung zwischen den anderen Modellformen herstellt. Einen Schwerpunkt bei der Wahl der demonstrierten Algorithmen bildet die Implementierung unter Verwendung aktueller programmierbarer Grafikkartenhardware. Anhang A gibt eine kurze Einführung in diese spezielle Art der Programmierung paralleler Algorithmen.



## 2 Das reguläre Voxelgitter

Die einfachste und mit Abstand am weitesten verbreitete Volumenrepräsentation ist das reguläre Voxelgitter (*uniform voxel grid*), dessen Diskussion daher den Ausgangspunkt dieser Arbeit bilden soll.

### 2.1 Datenstruktur

Das Voxelgitter repräsentiert diskrete, quantisierte Abtastpunkte in einem dreidimensionalen Skalarfeld  $f : \mathbb{R}^3 \mapsto \mathbb{R}$ . Im Rechner wird das Feld als dreidimensionales Array abgebildet, dessen Indizes den orthogonalen Raumrichtungen entsprechen. Ein Element dieses Arrays bildet ein *Voxel* (*volume element*). Ein Voxel kann als ein Würfel mit Kantenlängen im Gitterpunktabstand interpretiert werden. Im Folgenden wird die Konvention übernommen, den Mittelpunkt der Voxel auf die Gitterpunkte (also die vorhandenen Messwerte) zu legen. Werte im Raum zwischen den Punkten werden durch eine geeignete Interpolationsvorschrift ermittelt.

Ein typischer Volumendatensatz eines Röntgen-CTs repräsentiert die Absorptionskoeffizienten der durchstrahlten Materie in Form von Hounsfield Units, die bei menschlichem Gewebe typischerweise im ganzzahligen Bereich -2000 bis +4000 liegen [GF05]. In der Kernspintomografie entsteht ein Bild durch die Messung elektromagnetischer Strahlung, die von Protonen nach Anregung in einem externen Magnetfeld ausgeht. Die Signale sind üblicherweise mit 12 Bit, entsprechend 4096 Graustufen, aufgelöst. Ein binäres Datenvolumen, das beispielsweise eine segmentierte Struktur abbildet, könnte prinzipiell mit einem einzigen Bit pro Voxel dargestellt werden. Für die schnellere Verarbeitung im Rechner werden jedoch im Allgemeinen auf ganze Bytes ausgerichtete Strukturen verwendet.

Der große Vorteil des regulären Gitters liegt im einfachen und direkten Zugriff auf die Elemente. Wesentlicher Nachteil ist der enorme Speicherbedarf: Ein Volumenmodell mit einer Auflösung von  $512 \times 512 \times 512$  Elementen mit jeweils 16 Bit belegt 128MB. Einer Steigerung der Auflösung sind dadurch enge Grenzen gesetzt: eine Verdoppelung in jeder Dimension hätte bereits einen Speicherbedarf von  $2 \times 1024^3 = 2\text{TB}$  zur Folge, und ist damit auf einem aktuellen PC nicht mehr handhabbar.

Operationen auf dem Volumen müssen unter Umständen jedes einzelne Voxel durchlaufen, was lange Laufzeiten der Algorithmen zur Folge hat. Selbst wenn

das gesamte Volumenmodell in den Hauptspeicher passt, sind direkte Hauptspeichierzugriffe auf aktuellen PC–Architekturen um Größenordnungen langsamer, als wenn die Daten im CPU–Cache vorliegen. Diese Caches sind in der Größe auf wenige MB beschränkt, daher ist bei großen Arraystrukturen eine große Anzahl von Fehlzugriffen (*cache misses*) zwangsläufig.

## 2.2 Konvertierung von Dreiecksnetzen

Ein in Form eines geschlossenen Dreiecksnetzes vorliegendes geometrisches Modell kann in ein Voxelgitter konvertiert werden. Diesen Vorgang bezeichnet man als Voxelisierung (*voxelization*). Hierzu ist es notwendig, zu jedem Punkt des Raums, in dem die Geometrie eingebettet ist, eindeutig zwischen ”innen” und ”außen” unterscheiden zu können. Ein solcher Innen/Aussen–Test kann beispielsweise durch den vorzeichenbehafteten Abstand eines Punktes im Raum zum ihm nächstgelegenen Dreieck dienen [JS00]. Diese Berechnung ist allerdings sehr aufwendig – wie im folgenden Kapitel über Distanzfelder besprochen – und für die reine binäre Klassifikation auch unnötig.

Einfacher und vielversprechender ist der Ansatz, die Voxelisierung als Erweiterung der Scankonvertierung von Dreiecksmodellen in den Bildspeicher zu betrachten. Tatsächlich ist der Begriff *3D–Scankonvertierung* für diese Art der Voxelisierung gängig. Im zweidimensionalen Fall füllt die Scankonvertierung die Fläche eines in die Bildebene projizierten Dreiecks mit Pixeln im Raster des Bildspeichers. Die Ausführung dieses Vorgangs mit hoher Geschwindigkeit ist die Domäne der seit einigen Jahren allgegenwärtigen Grafikkarten mit ”3D–Unterstützung”. Der Gedanke liegt nahe, diesen Mechanismus auf das 3D–Szenario zu übertragen [FC00].

Ein in [EHK<sup>+</sup>06] vorgestelltes Verfahren erzeugt ein Voxelgitter schichtweise aus einer Sequenz von planaren Schnitten durch ein Polygonmodell. Für jede Schicht wird ein Bild erzeugt, das die gewünschte binäre Zuordnung der Voxel dieser Schicht repräsentiert. Die hohe Geschwindigkeit der Darstellung von Dreiecksnetzen durch die Grafikkarte ermöglicht so eine schnelle Voxelisierung.

Zunächst wird eine orthogonale Projektion eingerichtet, die das Objekt genau in das quaderförmige Sichtvolumen einpasst. Der Algorithmus durchläuft dann vier Schritte für jede Schicht:

1. Eine Schnittebene (*clipping plane*) wird eingerichtet, die der Position und Ausrichtung der Schicht entspricht. Alle zwischen dieser Ebene und der Kamera liegenden Teile der Geometrie werden dadurch nicht dargestellt, so dass das Innere des Objektes sichtbar wird.

2. Ein Bild der Dreiecke, deren *Rückseiten* sichtbar sind, wird erzeugt (*front–face culling*).
3. In einem zweiten Durchgang wird ein Teil der zuvor erzeugten Pixel wieder entfernt, indem nun nur die *Vorderseiten* in der Hintergrundfarbe dargestellt werden (*back–face culling*). Der *Z–Buffer–Inhalt* aus dem zweiten Schritt wird beibehalten, so dass nur weiter vorne liegende Dreiecke das vorhandene Bild überschreiben.
4. Der Bildspeicherinhalt enthält nun einen Querschnitt des Polygonmodells. Er wird in das Voxelgitter kopiert, und der Algorithmus beginnt erneut mit der nächsten Schicht.

In Abbildung 2.1 ist der Ablauf schematisch dargestellt.

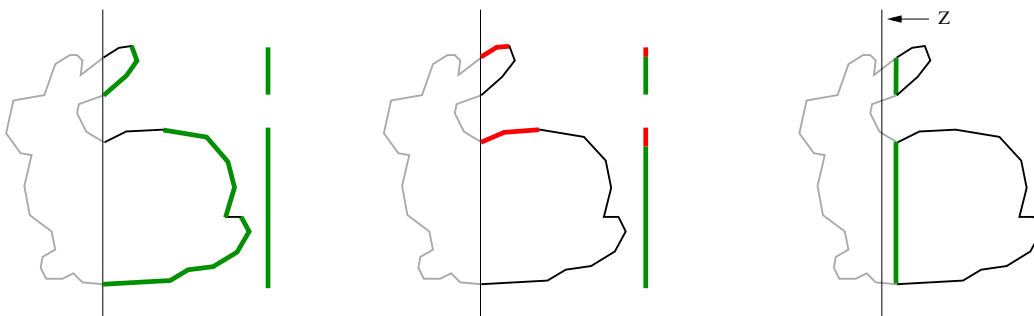


Abbildung 2.1.: 2D–Schema des Ablaufs des Voxelisierungsalgorithmus. Links: Rendering der rückseitigen Polygone (grün) hinter der Clipping–Ebene. Mitte: Korrektur durch Rendering der nach vorne zeigenden Polygone (rot). Rechts: Die Schnittebene wird in *Z*–Richtung verschoben.

### 2.2.1 Ergebnisse

Der vorgestellte Voxelisierungsalgorithmus wurde auf einem Apple iMac unter Mac OS X 10.5, 2,93 Ghz Intel Core 2 Duo, mit 4 GB RAM und einer NVIDIA GeForce GT 130 mit 512MB VRAM implementiert. Softwareseitig erfolgte die Implementierung in C++ (gcc 4.0) unter Verwendung von OpenGL 2.0. Alle weiteren im Rahmen dieser Arbeit vorgenommenen Implementierungen und Messergebnisse basieren ebenfalls auf diesem System.

Tabelle 2.1 gibt die Laufzeiten für die Konvertierung einiger Dreiecksnetze in reguläre Voxelgitter wieder. Die Zeiten werden im Wesentlichen durch die Größe des Bildspeichers bestimmt, dessen Inhalt in die Voxeldatenstruktur zurückkopiert werden muss. Das kleinste Modell mit nur 80 Dreiecken gibt diese minimale Laufzeit wieder. Mit zunehmend komplexeren Modellen fällt der wachsende

Aufwand für die Scankonvertierung der einzelnen Dreiecke pro Schicht stärker ins Gewicht. Abbildung 2.2 zeigt die Ausgangsgeometrien und die Ergebnisse der Voxelisierung.

Der Algorithmus ist schnell und sehr einfach zu implementieren. Wesentliche Voraussetzung für korrekte Ergebnisse ist jedoch ein orientiertes, geschlossenes Dreiecksnetz. Im Vorgriff auf Abschnitt 2.4.2 besteht eine Optimierungsmöglichkeit für eine weitere Beschleunigung darin, die einzelnen Schichten nicht in den Hauptspeicher zurückzukopieren, sondern direkt auf der Grafikkarte in die auch zur Darstellung verwendete 3D-Textur zu schreiben.

## 2.3 Grundlagen der Visualisierung

Für die Visualisierung von Voxelgittern steht eine reiche Auswahl an Verfahren zur Verfügung, die sich hinsichtlich Geschwindigkeit, Speicherbedarf, Qualität der Darstellung, und Hardwareanforderungen unterscheiden. Die Zweiteilung in oberflächen- und volumenorientierte Algorithmen zeigt sich auch bei den Visualisierungsmethoden: Ein Volumen kann zunächst in ein Oberflächenmodell umgewandelt werden, das dann etwa als Dreiecksnetz visualisiert wird. Die Darstellung eines Volumens durch Extraktion einer Oberfläche in ein Polygonnetz ist hervorragend geeignet, um Details dieser Oberfläche wiederzugeben. Das Bild beinhaltet allerdings keine Informationen über die innerhalb des Volumens liegenden Daten.

Die sogenannte direkte Darstellung (*direct rendering*) bildet hingegen über eine *Transferfunktion* die skalaren Volumeninhalte in den Farbraum ab, und erlaubt so unter anderem die Wahl des dargestellten Wertebereichs. Volumenmodelle werden häufig unter Verwendung von Transparenzen dargestellt, was unter Umständen das Verständnis der zugrundeliegenden Geometrie erschwert. Daher ist eine möglichst schnelle Darstellung, die das interaktive Wechseln der Ansicht erlaubt, wünschenswert. Trotz des größeren Rechenaufwands können Volumen

Tabelle 2.1.: Zeiten für die Voxelisierung von Polygonmodellen verschiedener Komplexität bei drei unterschiedlichen Zielauflösungen.

Modell	# $\Delta$	$64^3$	$128^3$	$256^3$
torus	80	0,06s	0,11s	0,33s
schädel	6621	0,07s	0,12s	0,34s
bunny	69660	0,11s	0,18s	0,46s
armadillo	345944	0,39s	0,75s	1,72s
happy buddha	1087716	1,2s	2,31s	4,73s



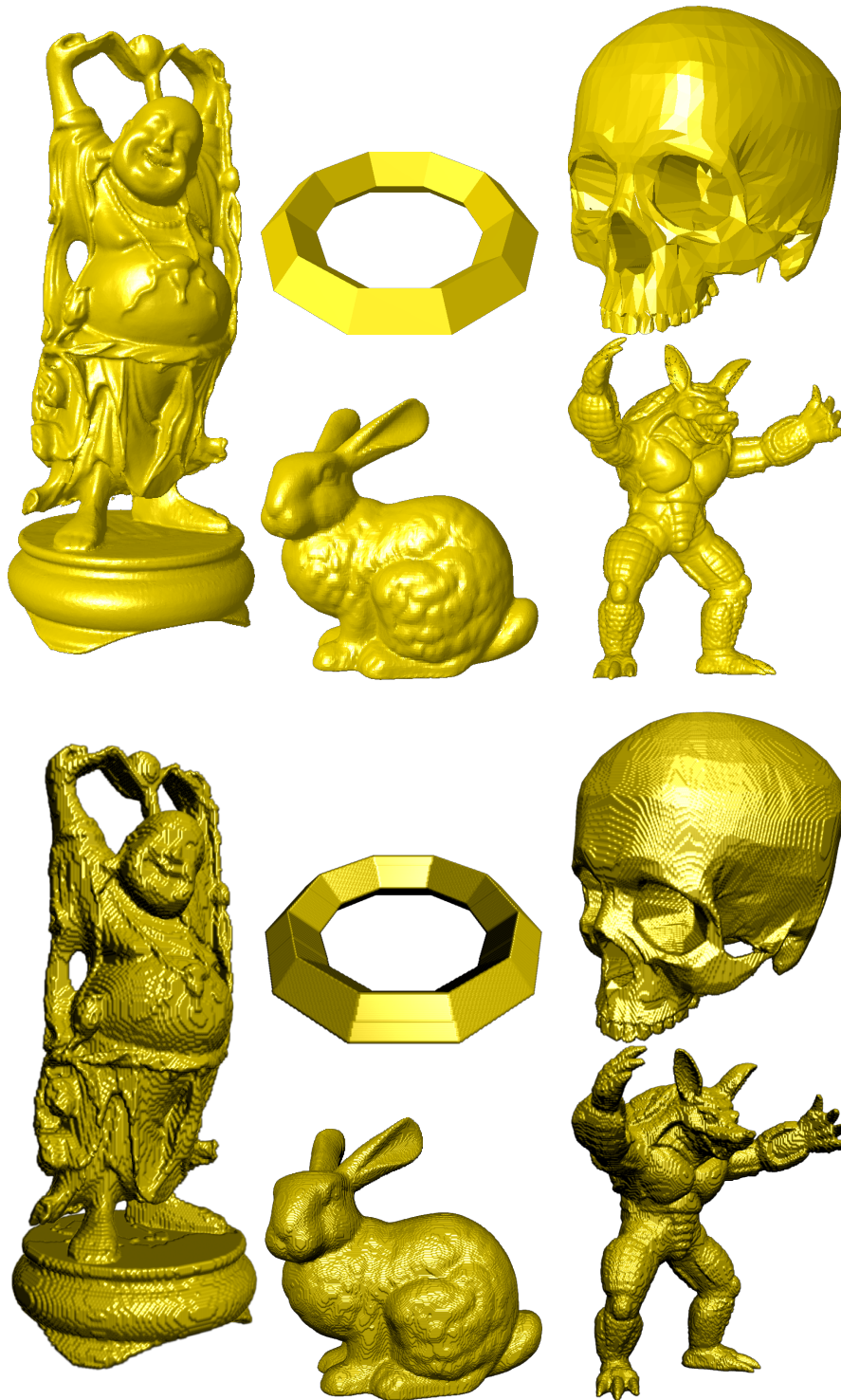


Abbildung 2.2.: Die in dieser Arbeit verwendeten Polygonmodelle, sowie in der Auflösung  $256^3$  erzeugte Voxelisierungen. Die Modelle bunny, armadillo, und happy buddha stammen aus dem Stanford 3D Scanning Repository (<http://www-graphics.stanford.edu/data/3Dscanrep>).

mit Unterstützung aktueller Grafikhardware bei genügend hohen Bildraten direkt dargestellt werden. Im Folgenden werden die Grundlagen der direkten Darstellung von Voxelgittern besprochen.

### 2.3.1 Rekonstruktion

Die Werte im Voxelgitter repräsentieren gleichmäßig verteilte, diskrete Abtastpunkte des Skalarfeldes an einzelnen Punkten im Raum. Für die Visualisierung stellt sich die Frage der Rekonstruktion der Werte zwischen den Gitterpunkten. Allgemein betrachtet werden die Datenpunkte hierzu mit einem geeigneten Rekonstruktionsfilter gefaltet. In der Praxis stellt der verwendete Filter einen Kompromiss zwischen Qualität und Rechenaufwand dar, der zumeist zugunsten des einfachsten Filters ausfällt, der noch akzeptable Ergebnisse bringt. Für Echtzeitvisualisierungen ist daher die trilineare Interpolation zwischen benachbarten Voxeln, die auf aktuellen Rechnern weitestgehend durch Hardware unterstützt wird, der dominierende Filter.

### 2.3.2 Transferfunktion

Vor der Visualisierung bedürfen die im Voxelgitter vorliegenden Daten zunächst der *Interpretation*: Die Daten entsprechen beispielsweise Dichtewerten, Klassifizierungen aus einer vorangegangenen Segmentierung, oder beliebigen Messwerten. Der Wertebereich muss auf entsprechende optische Materialeigenschaften abgebildet werden, die dann in ein Bild übersetzt werden. Diese Abbildung nennt man Transferfunktion. In der Regel werden die Eingangswerte auf RGBA-Farbwerte abgebildet, also Punkte im RGB-Farbraum mit zugeordneter Transparenz. Die Veränderung der Transferfunktion erlaubt beispielsweise, bestimmte Schichten besonders hervorzuheben oder auszublenden, sowie den Kontrast eines dargestellten Wertebereiches festzulegen.

### 2.3.3 Lichttransport im Volumen

Aus der Übersetzung mittels der Transferfunktion resultiert eine Beschreibung der optischen Eigenschaften des im Volumen eingeschlossenen Objektes. Für die Bilddarstellung wird die Interaktion des Lichts auf dem Weg durch das Medium bis ins Auge des Betrachters modelliert. Die üblichen Modelle sind stark vereinfachte Annäherungen an die physikalische Realität und beschränken sich in der Regel auf drei Arten der Interaktion zwischen Licht und partizipierendem Medium:

**Emission:** Das Material emittiert Licht, ist also selbstleuchtend.

**Absorption:** Das Material absorbiert eintreffendes Licht, beispielsweise durch Umwandlung in Wärme.

**Streuung:** Das Licht wird am Material gestreut, d.h. die Photonen ändern ihre Richtung. Man unterscheidet zwischen inelastischer Streuung, bei der die Wellenlänge des Lichtes verändert wird, und elastischer Streuung, bei der dies nicht geschieht.

Die Gesamtmenge abgegebener Lichtenergie entlang eines Strahls wird durch die Strahlungsdichte (*radiance*)  $L$  genauer bezeichnet. Sie ist eine zentrale radiometrische Größe der Computergrafik, da sie mit der von einem optischen System wie dem Auge empfangenen Strahlungsleistung korreliert, und damit mit der empfundenen Helligkeit eines Objektes.

$$L = \frac{d\Phi}{d\Omega dA \cos \alpha},$$

mit der in den Raumwinkel  $d\Omega$  abgestrahlten Strahlungsleistung  $d\Phi$ , sowie einem Flächenelement  $dA$ , dessen Normale im Winkel  $\alpha$  zur betrachteten Strahlrichtung liegt.

### 2.3.4 Das Volumen–Rendering–Integral

Vernachlässigt man den Einfluss der Streuung am Medium, erhält man das *Emissions–Absorptions–Modell*, dass in der Volumengrafik sehr häufig verwendet wird. Ist in jedem Punkt des Volumens der Absorptionskoeffizient  $\kappa$  sowie der Emissionskoeffizient  $q$  bekannt, ist die Änderung der Strahlungsdichte auf einem Wegstück  $s$  entlang eines Strahls durch eine einfache Differentialgleichung beschreibbar:

$$\frac{dL(s)}{ds} = -\kappa(s)L(s) + q(s). \quad (2.1)$$

Die Integration dieser Gleichung für einen Lichtstrahl, der am Punkt  $s_0$  in ein Volumen ein- und am Punkt  $s_1$  wieder austritt, ergibt das Volumen–Rendering–Integral:

$$L(s_1) = L_0 e^{-\int_{s_0}^{s_1} \kappa(t) dt} + \int_{s_0}^{s_1} q(s) e^{-\int_s^{s_1} \kappa(t) dt} ds. \quad (2.2)$$

### 2.3.5 Diskretisierung des Integrals

Das Volumen–Rendering–Integral ist in der Regel nicht analytisch lösbar, so dass numerische Techniken zur Approximation der Lösung verwendet werden. Eine übliche Näherung ist die Annahme gleichlanger Wegstücke  $\Delta x$  und die Näherung des Integrals durch die Riemann–Summe der resultierenden Segmente. Ein Verfahren zur numerischen Lösung ist die Raycasting–Methode: Die Werte werden entlang der Sichtstrahlen zum Betrachter hin integriert [UK88]. Aus der Betrachtung der differentiellen Formulierung 2.1 ergibt sich hierbei folgende einfache Vorschrift zur inkrementellen Berechnung der akkumulierten Strahlungsdichte  $L_N$  eines einzelnen Lichtstrahls:

1. Initialisiere  $L_0 = q_0$ .
2.  $L_{i+1} = q_i \Delta x - \kappa_i \Delta x L_i$  für alle  $i = 1, \dots, N - 1$ .
3.  $L_N$  enthält die resultierende akkumulierte Strahlungsdichte.

Dieses Vorgehen kann auf den Mechanismus abgebildet werden, mit dem in der Bildverarbeitung zwei Bilder überblendet werden (*compositing*). Die Konstante  $\Delta x$  wird eliminiert, und die Emissionskoeffizienten  $q_i$  werden auf RGB–Farbtupel  $c_i$  abgebildet. Die  $\kappa_i$  können durch den Alpha–Kanal (Deckkraft) als  $\alpha_i$  repräsentiert werden. Die Compositing–Operation verknüpft einen bereits im Bildspeicher vorhandenen Farbwert  $c_{dst}$  mit einem weiteren hereinkommenden Wert  $c_{src}$ . Die Standardregel

$$c_{dst} = (1 - \alpha_{src})c_{dst} + c_{src},$$

wie sie beispielsweise in OpenGL spezifiziert ist, beschreibt dann den bei sukzessivem Überblenden mit den  $c_i$  und  $\alpha_i$  denselben Renderingprozess.

### 2.3.6 Lokales Beleuchtungsmodell

Das beschriebene Emissions–Absorptionsmodell liefert Bilder, die visuell in etwa einer Röntgenaufnahme entsprechen. Da keinerlei Streuung an Flächen im Volumen berücksichtigt wird, lassen die Abbildungen kaum Details der Oberflächenbeschaffenheit erkennen. Die Aussagekraft der Abbildung kann bereits durch ein einfaches lokales Beleuchtungsmodell wie das Blinn–Phong–Modell stark verbessert werden. In erster Näherung kann ein solches Modell als Einbeziehung von Streuung an Volumenelementen betrachtet werden. Angenommen wird hierbei, dass das Licht einer externen Lichtquelle ungehindert jedes Volumenelement erreicht, so dass der Effekt allein durch lokale Betrachtung des Auftreffpunktes berechnet werden kann.

Das Blinn–Phong–Modell unterteilt den Beitrag der Beleuchtung eines Punktes in drei Komponenten:

**Ambiente Beleuchtung:** Simuliert gestreute, ungerichtete Hintergrundbeleuchtung.

**Diffuse Beleuchtung:** Die Helligkeit eines Flächenelementes nimmt mit dem Winkel zur Lichtquelle ab.

**Spekulare Beleuchtung:** Spiegelnde Oberflächen reflektieren eintreffendes Licht in einem je nach Oberfläche mehr oder weniger breiten Kegel, wodurch ein Glanzlicht auf der Oberfläche entsteht.

Die genannten Materialeigenschaften eines Volumenelementes seien in Form der RGB–Tupel  $k_a, k_d, k_s$  gegeben. Zusätzlich fließen die Farbe der gerichteten Lichtquelle  $l_d$  und der ambienten Beleuchtung  $l_a$  ein. Der Koeffizient  $s$  regelt die Größe des Glanzlichtes, und damit visuell die Reflektivität der Fläche. Vom betrachteten Volumenelement sei der Vektor  $\mathbf{v}$  zum Betrachter,  $\mathbf{l}$  zur Lichtquelle, und der Normalenvektor  $\mathbf{n}$  zur angenommenen Oberfläche gegeben. Die resultierende Farbe  $c$  ergibt sich dann zu:

$$c = k_a l_a + k_d l_d (\mathbf{l} \cdot \mathbf{n}) + k_s l_d \left( \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|} \right)^s.$$

Eine eingehendere Betrachtung dieses Modells finden sich in der einführenden Computergrafikliteratur [FvDFH90, Shi05].

Wesentlicher Bestandteil des Modells ist die Oberflächennormale am betrachteten Punkt  $\mathbf{p}$ . In einem Volumenmodell kann die Oberflächennormale durch den Gradienten des Skalarfeldes  $f$  nachgebildet werden:

$$\nabla f = \begin{pmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \\ \frac{\delta f}{\delta z} \end{pmatrix}$$

weist in die Richtung des maximalen Anstiegs der Funktion. Eine entsprechende Verteilung der Werte des Skalarfeldes vorausgesetzt – etwa mit dem Abstand zur Objektoberfläche nach außen hin ansteigende Werte – weist der Gradient also senkrecht von der Oberfläche nach außen. Daher entspricht die Normale in diesem Fall dem normierten Gradienten:

$$\mathbf{n}(\mathbf{p}) = \frac{\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|}.$$

## 2.4 Hardwareunterstützte Renderingverfahren

Einsatzmöglichkeiten für die aufkommende Hardwareunterstützung bei der Polygondarstellung in der Volumenvisualisierung wurden schon frühzeitig untersucht [WG91, LH91]. Heutzutage können die Möglichkeiten der Grafikhardware unter mehreren Aspekten für die Implementierung eines echtzeitfähigen Volumenrenderers genutzt werden: Einerseits für die Realisierung der Transferfunktion und die anschließende Abbildung der dreidimensionalen Geometrie, also die Lösung des Volumen–Rendering–Integrals. Zum Anderen können mit programmierbaren GPUs auch Gradientenberechnungen für lokale Beleuchtungsmodelle effizient umgesetzt werden.

### 2.4.1 Transferfunktion mittels Textur–Lookup

Die skalaren Werte im Volumendatensatz müssen erst durch Anwendung der Transferfunktion in Farbwerte übertragen werden. Hier gäbe es prinzipiell die Möglichkeit, die Transferfunktion in einem Vorverarbeitungsschritt auf sämtliche Volumenelemente anzuwenden. Nachteilig ist dabei der sehr viel höhere Speicherbedarf (jedes Volumenelement besteht dann aus einem in der Regel 32–bittigen RGBA–Wert). Zudem bedingt jede Änderung der Transferfunktion eine komplette Neuberechnung des Datenvolumens.

Ein eleganterer Weg besteht daher darin, die Transferfunktion in einem Fragment Shader (siehe Anhang A) erst während der Darstellung anzuwenden. Hierzu werden die aus dem Volumen stammenden Rohdaten als Indices in eine weitere, ein-dimensionale Textur aufgefasst, in der die resultierenden RGBA–Werte stehen. Unter Ausnutzung der Interpolation in der Textur–Hardware lässt sich so eine stückweise lineare Transferfunktion realisieren, wie in Abbildung 2.3 gezeigt.

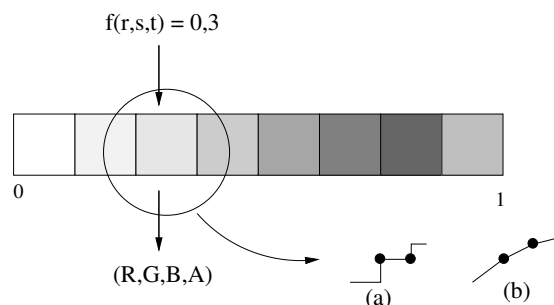


Abbildung 2.3.: Transferfunktion mittels 1D–Texture–Lookup. Der Wert aus dem Skalarfeld, auf das Intervall  $[0, 1]$  normiert, wird auf einen RGBA–Wert abgebildet. Die Interpolation zwischen benachbarten Texturelementen ist entweder stückweise konstant (a) oder stückweise linear (b).

## 2.4.2 3D-Texturen

Als logische Fortsetzung der zweidimensionalen Texturen unterstützen moderne Grafikkarten zunehmend auch 3D-Texturen: ihre Struktur entspricht der des Voxelgitters, wobei die drei Raumdimensionen auf Texturkoordinaten im Intervall  $[0, 1]$  abgebildet werden (Abbildung 2.4). Bei Verwendung von 16-Bit-Werten pro Datenelement belegt ein  $512^3$ -Modell 256MB Speicher, eine Datenmenge, die gut ausgestattete Grafikkarten komplett in ihrem dezidierten Speicher halten können. Analog zu zweidimensionalen Texturen können Grafikkarten eine trilineare Interpolation zwischen den Datenpunkten der Textur in Hardware durchführen.

Eine 3D-Textur kann jedoch nicht direkt für die Darstellung des dreidimensionalen Volumens verwendet werden. Vielmehr können den Eckpunkten eines Polygons je drei Texturkoordinaten zugeordnet werden, so dass das texturierte Polygon eine Schnittebene durch das Datenvolumen darstellt. Die Darstellung des Volumens erfolgt dann *schichtweise* durch Überblendung von texturierten Ebenen, unter Verwendung des in Abschnitt 2.3.5 beschriebenen Compositing-Ansatzes [EHK<sup>+</sup>06]. Der grundlegende Prozess der Visualisierung ist also wie folgt:

1. Das Voxelarray wird als 3D-Textur an die Grafikkarte übertragen.
2. Für jede Schicht des Datensatzes wird ein Polygon (typischerweise ein Rechteck) erzeugt, dem die 3D-Texturkoordinaten der entsprechenden Schicht zugewiesen sind.
3. Die Polygone werden in sortierter Folge, in der Regel von hinten nach vorne (*back to front*), dargestellt. Jede neue Schicht wird im Bildspeicher mit den bereits dargestellten überblendet.

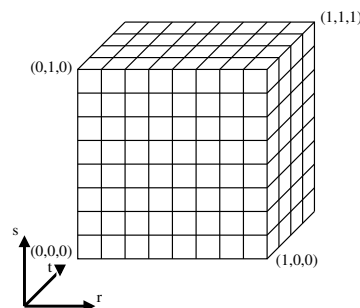


Abbildung 2.4.: 3D-Textur im  $(r,s,t)$ -Koordinatensystem.

### 2.4.3 Compositing texturierter Polygone

Für das Compositing–Verfahren ist es notwendig, die Sortierung der texturierten Rechtecke beim Rendering einzuhalten, beim Back–to–front–Ansatz muss also zuerst das am weitesten vom Betrachter entfernte Polygon dargestellt werden. Bei interaktiver Darstellung mit wechselnden Kameraeinstellungen ergeben sich jedoch ungünstige Perspektiven, in denen die texturierten Polygone nur von der Seite sichtbar sind, oder die Anordnung der Polygone in Sichtrichtung sogar invertiert wird, was eine fehlerhafte Darstellung zur Folge hat.

Eine einfache Lösung basiert auf der Verwendung dreier unabhängiger Stapel von texturierten Polygonen, orientiert entlang der Raumachsen, zwischen denen je nach Betrachtungsrichtung umgeschaltet wird (Abbildung 2.5). Als Umschaltkriterium wird die größte Komponente des Sichtvektors herangezogen. Bei Blickrichtung entlang der negativen Z-Achse wird also der entlang der Z-Achse orientierte Stapel vom kleinsten zum größten Z–Wert hin dargestellt.

Der Mehrbedarf an Speicher ist bei dieser Methode gering: Es fällt zusätzliche Geometrie in der Größenordnung von einigen Hundert Rechtecken an. Die Texturkoordinaten referenzieren jedoch immer die gleiche 3D–Textur, die also nicht dupliziert werden muss.

Die Stapel texturierter Polygone erzeugen zum einen zusätzliche Geometrie, zum anderen sind bei bestimmten Sichtwinkeln Artefakte an den Rändern der Texturen sichtbar. Der Wechsel zwischen den Texturstapeln erzeugt ein kurzes, aber wahrnehmbares Flackern, da die 3D–Textur nicht an genau identischen Punkten abgetastet wird. In [EHK<sup>+</sup>06] wird eine Methode vorgestellt, die mittels eines Vertex Shaders für jedes Bild einen am Sichtvektor ausgerichteten Polygonstapel

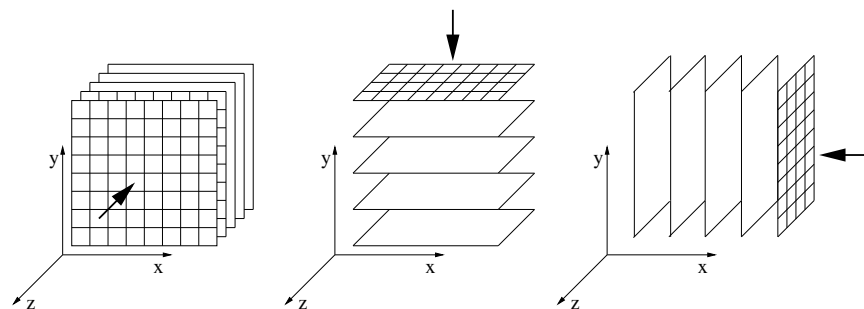


Abbildung 2.5.: Drei texturierte Polygonstapel, dargestellt im Weltkoordinatensystem mit  $(x,y,z)$ –Koordinaten, werden entsprechend der Hauptkomponente des Sichtvektors umgeschaltet.



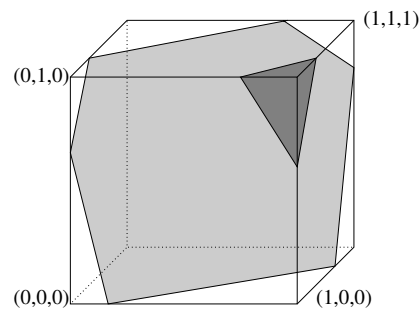


Abbildung 2.6.: Der Schnitt des Texturquaders mit einer Ebene senkrecht zur Sichtrichtung erzeugt ein Polygon mit drei (dunkles Polygon) bis sechs (helles Polygon) Seiten. Ein Vertex Shader erzeugt diese Schnittpolygone mit passenden Texturkoordinaten zur Laufzeit.

erzeugt. Der Volumenquader wird in den gewünschten Schichtabständen mit einer orthogonal zur Sichtrichtung angelegten Ebene geschnitten. Auf diese Weise entstehen Polygone mit drei bis sechs Eckpunkten (Abbildung 2.6), die im Shader mit den entsprechenden Texturkoordinaten belegt werden. Welche Kanten des Quaders jeweils mit der Ebene geschnitten werden müssen, kann durch geschickte Fallunterscheidungen in Tabellen nachgeschlagen werden, wodurch eine hohe Effizienz erreicht wird. Der Vertex Shader benötigt als Eingabe die Eckpunkte des Quaders, eine Beschreibung der Schnittebene in Hessescher Normalform, sowie den Index des am nächsten zum Betrachter gelegenen Eckpunktes (0 . . . 7), um die richtige Permutation der benötigten Schnittkanten zu bestimmen.

Ein Problem bei der Implementierung des Compositing-Schemas ist die begrenzte Genauigkeit des Bildspeichers, der üblicherweise nur 8 Bit pro Farbkanal besitzt. Da eine Compositing-Operation das Ergebnis immer wieder im Bildspeicher ablegt, führt die Wiederholung mit der Anzahl der Schichten zu wachsenden Quantisierungsfehlern. Bei Hundert Schichten kann dieser Fehler bis zu 40% des korrekten Wertes annehmen [MB05], mit visuell inakzeptablen Ergebnissen. Der Effekt lässt sich durch Wahl einer höheren Farbtiefe des Bildspeichers begrenzen. Unter OpenGL kann, entsprechende Hardware vorausgesetzt, statt in den Bildspeicher in eine Floating-Point-Textur gerendert werden, die Werte mit 32 Bit Genauigkeit darstellt. Die Quantisierung auf die Auflösung des Bildspeichers erfolgt dann erst abschließend zur Darstellung.

#### 2.4.4 Berechnung von Gradienten für lokale Beleuchtung

Für ein lokales Beleuchtungsmodell muss, wie bereits beschrieben, der lokale Gradient in jedem Punkt des Volumens berechnet werden. Der Gradient kann vorbe-

rechnet werden, was jedoch die zusätzliche Speicherung eines dreidimensionalen Feldes von Normalenvektoren bedeutet. Mit schneller programmierbarer Grafikhardware kann der Gradient aber auch zur Laufzeit in einem Fragment Shader approximiert werden. Hierzu bedient man sich der zentralen Differenzen

$$\nabla f(x, y, z) \simeq \frac{1}{2h} \begin{pmatrix} f(x+h, y, z) - f(x-h, y, z) \\ f(x, y+h, z) - f(x, y-h, z) \\ f(x, y, z+h) - f(x, y, z-h) \end{pmatrix}.$$

Die Schrittweite  $h$  der Differenzen ist ein einstellbarer Parameter für den Shader, und von der Auflösung des Voxelgitters beziehungsweise der 3D-Textur abhängig. Bei trilinearer Interpolation zwischen den Textur-elementen ergibt sich eine hinreichend gute Qualität bei vertretbaren Kosten für die Berechnung.

Innerhalb homogener Regionen des Volumens wird der Betrag des Gradienten sehr klein. Aufgrund der beschränkten numerischen Auflösung der Vektorkomponenten ergibt sich keine sinnvolle Richtungsinformation zur Berechnung einer Oberflächennormale, die daher stark schwanken kann. Der Betrag des Gradienten kann als Grad der "Oberflächennähe" herangezogen werden, um die Ergebnisse des lokalen Beleuchtungsmodells zu skalieren, oder unterhalb eines Schwellwertes die Berechnung ganz zu unterlassen.

### 2.4.5 Ergebnisse

Implementiert wurden die in den Abschnitten 2.4.1 bis 2.4.4 beschriebenen Verfahren zur Darstellung eines als 3D-Textur vorliegenden Voxelgitters durch Überblenden texturierter Polygone. Abbildung 2.7(a) zeigt die resultierende Darstellung ohne Oberflächenbeleuchtung. Obwohl die schnelle interaktive Manipulation dem Anwender durch Betrachten aus verschiedenen Blickwinkeln Rückschlüsse auf die Geometrie erlaubt, liefert das lokale Beleuchtungsmodell in Abbildung 2.7(b) auf einen Blick mehr Informationen. Mischformen, die den Blick ins Innere gestatten, sind durch entsprechende Wahl der Transferfunktion möglich (Abbildung 2.7(c)).

Die Transferfunktion ist für binäre Volumenmodelle, wie sie durch die Voxelisierung erzeugt werden, nur begrenzt nützlich. Bei Datensätzen mit größeren Wertebereichen kann sie verwendet werden, um beispielsweise verschiedene Gewebearten visuell voneinander abzugrenzen, wie Abbildung 2.8 zeigt.

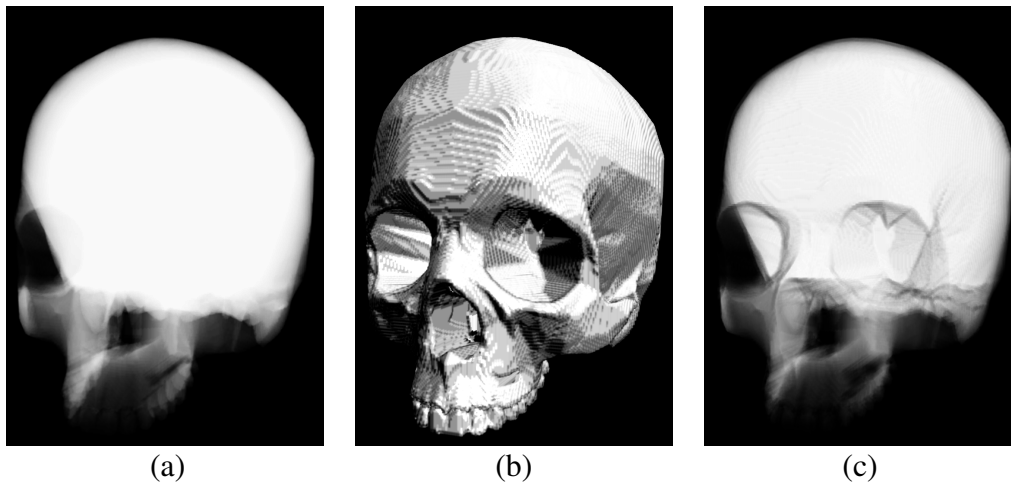


Abbildung 2.7.: Visualisierung eines Voxelmodells: (a) ohne lokale Beleuchtung (Emission / Absorption); (b) lokales Blinn/Phong-Beleuchtungsmodell (Streuung); (c) Kombination aus (a) und (b).

Inklusive der Berechnung der sichtvektororientierten Schnittpolygone im Vertex Shader wird eine Bildrate von 23 fps (*frames per second*) erreicht. Die Zuschaltung des lokalen Beleuchtungsmodells führt allerdings zu einer Halbierung dieser Bildrate auf 11 fps. Der Grund ist in der kostspieligen Auswertung der 3D-Textur zur Bestimmung des Gradienten zu sehen: In jeder dargestellten Schicht werden pro Pixel drei Differenzen entlang der Hauptachsen ermittelt, es werden also sechs zusätzliche Texturzugriffe notwendig.

Die trilineare Interpolation über das diskretisierte Volumen in Verbindung mit den

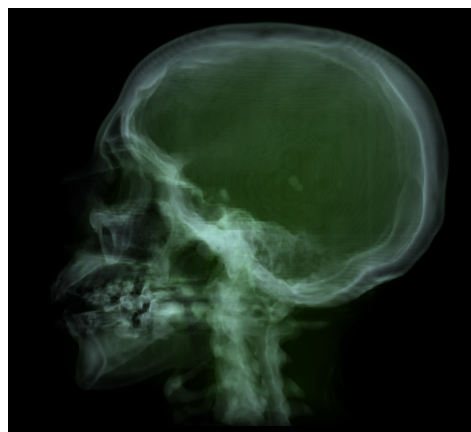


Abbildung 2.8.: Darstellung eines CT-Datensatzes mit unterschiedlicher Färbung in verschiedenen Wertebereichen der Transferfunktion. Der Datensatz stammt aus dem Stanford Volume Data Archive (<http://www-graphics.stanford.edu/data/voldata>).

finiten Differenzen kann nur eine ungenaue Annäherung an die "wirkliche" Oberflächennormale liefern. Aufwändigere Filterkernel, die größere Nachbarschaften und komplexere Interpolationsvorschriften implementieren – beispielsweise kubische B-Splines – erzielen deutlich bessere Oberflächendarstellungen. Eine derartige Implementierung bedingt eine Vorberechnung der Gradienten, was einen erhöhten Speicherbedarf zur Folge hat, andererseits eine wiederum höhere Bildrate erlaubt.

Es bleibt anzumerken, dass die numerische Integration durch das Überblenden mittels texturierter, planarer Polygone streng genommen nicht korrekt ist. Bei perspektivischer Projektion ist die Strecke, die ein Sichtstrahl zwischen zwei Texturerebenen zurücklegt, für verschiedene Sichtwinkel unterschiedlich lang. Die Compositingregel geht dagegen von einer konstanten Schrittweite aus. Dieses Problem kann durch Verwendung sphärischer Polygone vermieden werden, worauf in der Praxis jedoch wegen des vernachlässigbaren visuellen Effektes meist verzichtet wird.

## 2.5 Berechnung und Darstellung von Schnitten

Das Schneiden eines Voxelgitters mit einem binären Volumenmodell ist konzeptionell eine sehr einfache Operation. Das binäre Voxelgitter, *Selektionsvolumen* genannt, wird zum Maskieren des Datenvolumens verwendet. Im einfachsten Fall liegt es genau deckungsgleich über dem Datenvolumen, so dass korrespondierende Voxel direkt verknüpft werden können. Alle Voxel des Datenvolumens, die innerhalb des Selektionsvolumens liegen, werden "gelöscht", d.h. mit dem Wert 0 belegt.

Eine naive Implementierung in dieser Form besitzt gravierende Nachteile:

- Die Auflösung des Selektionsvolumens muss der des Datenvolumens angepasst sein.
- Für ein Selektionsvolumen in der Größe des Datenvolumens fällt entsprechender Speicherbedarf an.
- Eine Veränderung (beispielsweise Verschiebung oder Rotation) des zu maskierenden Bereiches bedingt zeitaufwendige Modifikationen des Selektionsvolumens.
- Zur Darstellung des modifizierten Datensatzes muss dieser erneut auf die Grafikkarte hochgeladen werden, was eine weitere Einschränkung der erzielbaren Geschwindigkeit zur Folge hat.

In der Regel wird das Selektionsvolumen kleiner als das Datenvolumen sein, und in Gestalt eines beispielsweise würfel- oder kugelförmigen "Werkzeugs" über die interessierenden Regionen des Datenvolumens geschoben. Daher definiert man zum Selektionsvolumen noch eine affine Transformation  $M$ , die eine Translation, Rotation, und Skalierung des Selektionsvolumens relativ zum Datenvolumen angibt. Dieses Vorgehen reduziert den Speicherbedarf und macht die direkte Veränderung des Selektionsvolumens im Hauptspeicher unnötig.

Die restlichen Schwierigkeiten lassen sich überwinden, wenn die Maskierungsoperation direkt auf die Grafikkarte verlegt wird, so dass Datenvolumen und Selektionsvolumen als 3D-Texturen vorliegen:

- Das Ausmaskieren jedes Voxels ist eine unabhängige Operation, so dass eine sehr schnelle, parallele Ausführung in einem Fragment Shader möglich ist.
- Der Zugriff auf die Texturen erfolgt in normalisierten Texturkoordinaten, bei automatischer Interpolation in der Hardware. Datenvolumen und Selektionsvolumen können also verschiedene Auflösungen besitzen.
- Es ist kein zeitraubender Transfer vom Hauptspeicher in den Speicher der Grafikkarte zur Darstellung jedes Einzelbildes notwendig.

Auf diese Weise lassen sich Bildraten erzielen, die hoch genug sind, um eine interaktive Handhabung und Darstellung von Ausschnitten des Datenvolumens zu ermöglichen. Die im Folgenden geschilderte Implementierung beruht auf dem in [EHK<sup>+</sup>06] vorgestellten Verfahren.

### 2.5.1 Hardwarebeschleunigtes Selektionsvolumen

Datenvolumen und Selektionsvolumen liegen als 3D-Texturen in potentiell verschiedener Auflösung vor. Eine affine Transformation  $M$  beschreibt die Translation, Rotation, und Skalierung des Selektionsvolumens relativ zum Datenvolumen. Die inverse Matrix bildet also einen Punkt  $\mathbf{p}_d$  im Datenvolumen auf den korrespondierenden Punkt  $\mathbf{p}_s$  im Selektionsvolumen ab:  $\mathbf{p}_s = M^{-1}\mathbf{p}_d$ .

Wegen des linearen Zusammenhangs ist es nicht notwendig, diese Transformation für jedes Voxel der Textur erneut auszuführen. Stattdessen wird schon im Vertex Shader, bei Erzeugung der Schnittpolygone, für jeden Polygonpunkt die Transformation vorgenommen und gespeichert. Im Fragment Shader liegt dann zu jedem Texturpunkt die zugehörige interpolierte Position  $\hat{\mathbf{p}}_s$  im Selektionsvolumen zum ebenfalls interpolierten Ort  $\hat{\mathbf{p}}_d$  im Datenvolumen vor.

Im Fragment Shader wird vor die Berechnung der Pixelfarbe die Selektionsoperation geschaltet:

1.  $d \leftarrow$  Wert aus dem Datenvolumen an  $\hat{p}_d$
2.  $s \leftarrow$  Wert aus dem Selektionsvolumen an  $\hat{p}_s$
3. Wenn  $s \neq 0$ , dann verwerfe das Pixel
4. Sonst: weitere Berechnungen der Pixelfarbe (Transferfunktion, Beleuchtung).

## 2.5.2 Ergebnisse

Implementiert wurde eine einfache interaktive Steuerung eines Selektionsvolumens, das relativ zum Datenvolumen verschoben und skaliert werden kann. Ein beliebiges binäres Voxelgitter kann als Selektionsvolumen verwendet werden. Abbildung 2.9 zeigt den Einsatz mit einem einfachen Würfel als Werkzeug auf einem MRT-Datensatz. Die Methode gestattet die Kombination von Volumenmodellen beliebiger Form, wie in Abbildung 2.10 gezeigt.

Die automatische Interpolation der 3D-Texturkoordinaten erlaubt es, Volumen mit nicht identischer Auflösung zu verwenden. Wie Abbildung 2.11 veranschaulicht, resultiert ein grobes Selektionsvolumen jedoch ebenso in Blockartefakten, wie ein unzureichend aufgelöstes Datenvolumen.

Die Zuschaltung des Selektionsmodells reduziert die Bildrate für die Darstellung eines Volumens der Auflösung  $256^3$  mit lokalem Beleuchtungsmodell von 11 fps (siehe Abschnitt 2.4.5) auf 3 fps. Die wesentliche Ursache liegt auch hier in der Laufzeitberechnung der Gradienten im Fragment Shader: Für jeden der sechs Texturzugriffe, die für die Differenzbildung stattfinden (siehe Abschnitt 2.4.4), muss zusätzlich die beschriebene Selektionsprozedur durchgeführt werden, um an der

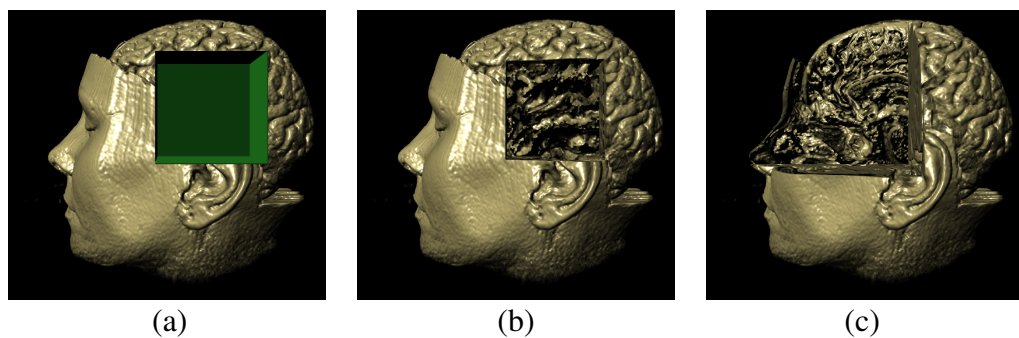


Abbildung 2.9.: Interaktive Selektion von Ausschnitten: (a) MRT-Datensatz mit interaktiv positioniertem, würfelförmigem Werkzeug als Selektionsvolumen; (b) Schnitt wird bei ausgeblendetem Werkzeug sichtbar; (c) anderer Ausschnitt mit höher skaliertem Werkzeug. Der Datensatz stammt aus dem Stanford Volume Data Archive (<http://www-graphics.stanford.edu/data/voldata>).

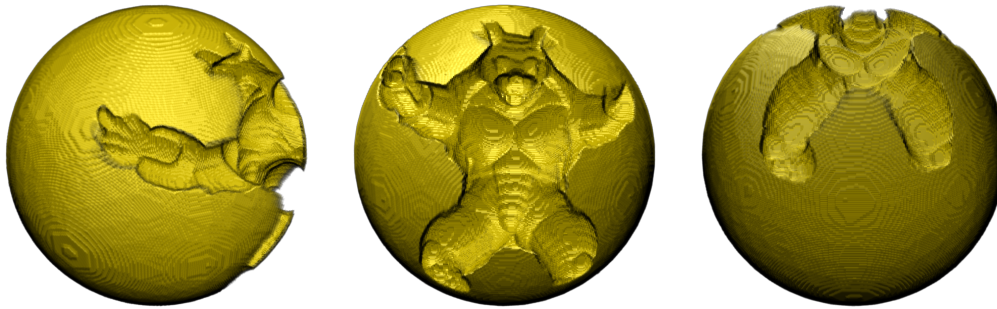


Abbildung 2.10.: Aus einer in der Auflösung  $256^3$  diskretisierten Kugel wurde das "armadillo"-Modell, ebenfalls in der Auflösung  $256^3$ , herausgeschnitten. Dargestellt sind drei verschiedene Ansichten des resultierenden Modells.

Schnittfläche mit dem Selektionsvolumen korrekte Oberflächennormalen zu erhalten. Die Anzahl der Texturzugriffe verdoppelt sich daher.

Um die Interaktivität aufrechtzuerhalten, bietet es sich an, komplexere Berechnungen nicht durchzuführen, während das Objekt bzw. die Sicht bewegt wird. Beispielsweise könnte die beschriebene Berechnung der Gradienten während der Bewegung das Selektionsvolumen nicht berücksichtigen, oder das gesamte Volumenmodell in einer niedrigeren Auflösung dargestellt werden.

## 2.6 Zusammenfassung

Die Repräsentation von Volumenmodellen besitzt viele Vorteile, die vor allem auf der Regularität der Datenstruktur beruhen. Grundlegende Algorithmen zur Darstellung des Volumens und zur Berechnung von Schnitten sind einfach zu implementieren. Da das Volumenmodell als Menge gleichartiger, voneinander un-

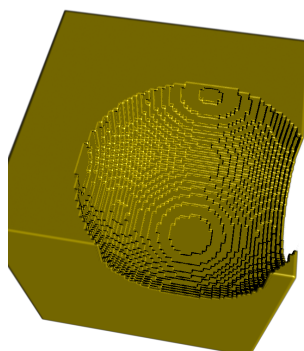


Abbildung 2.11.: Selektion mit einem in der Auflösung  $64^3$  voxelisierten Kugelmodell.

abhängiger Voxel vorliegt, bietet sich ein großes Potential zur Anwendung parallelisierter Algorithmen.

Problematisch insbesondere bei binären Voxelgittern — wie sie der dargestellte Voxelisierungsalgorithmus erzeugt — ist die Qualität der Geometrie bei niedrigen Auflösungen. Die visuelle Darstellung weist Aliasing-Effekte auf und wirkt “blockig”, wie in Abbildung 2.12 gut sichtbar. Die visuellen Artefakte sind hauptsächlich auf die mangelhafte Rekonstruktion der Oberflächennormalen zurückzuführen: Werden sie aus dem binären Voxelgitter berechnet, weisen sie grundsätzlich in die 6 Raumrichtungen. Glattere Normalen sind nur um den Preis geringerer Ortsauflösung zu erhalten. Eine Erhöhung der Auflösung des Modells andererseits ist nur in den durch Speicherplatz und Rechenzeit gegebenen engen Grenzen möglich. Wie eingangs erwähnt, stellen Modelle von etwa  $512^3$  Voxeln Auflösung die derzeitige Obergrenze dar.

Wesentlich bessere Resultate sind erzielbar, wenn die Diskretisierung kein binäres, sondern ein Grauwertvolumen erzeugt. Damit beschäftigt sich das folgende Kapitel.

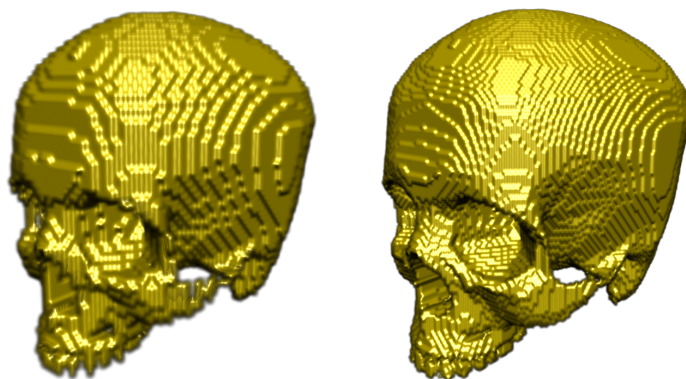


Abbildung 2.12.: Das voxelisierte ”schädel”-Modell in zwei weiteren Auflösungen  $64^3$  und  $128^3$ .



### 3 Distanztransformation

Aus dem vorangegangenen Kapitel wird deutlich, dass die in einem binären Voxelgitter enthaltene Information nicht für eine hochwertige Rekonstruktion von Oberflächen ausreicht. Insbesondere bereitet die Berechnung eines Gradienten im Volumen Schwierigkeiten. Die Diskretisierung eines Dreiecksmodells kann auf verschiedene Weise verbessert werden. Zum einen kann eine komplexere Abtastung der Oberfläche durchgeführt werden [WK93, WK94, SK99]. Eine attraktive Alternative, die weitere Anwendungen eröffnet, ist die Belegung des Voxelgitters mit *Abstandswerten*: An die Stelle der aus der Voxelisierung resultierenden, binären Innen/Außen-Information tritt eine Messung des vorzeichenbehafteten Abstands zur Oberfläche des Objekts. Für ein geschlossenes, orientiertes Dreiecksnetz besitzt diese Distanzfunktion auf einer Seite der Fläche ein positives Vorzeichen, auf der anderen ein negatives. Die gängige Konvention ist hier, dem Außenbereich das positive Vorzeichen zuzuweisen. Aus der Diskretisierung der Distanzinformation resultiert ein Grauwertvolumen mit einem linearen Gradienten in Richtung der Oberfläche. Die ursprüngliche Oberfläche der konvertierten Geometrie lässt sich dann aus der Abstandsfunktion als implizite Fläche  $f(\mathbf{p}) = 0$  rekonstruieren. Der Vorgang der Konvertierung einer Oberfläche in diese implizite Form wird als *Distanztransformation* bezeichnet.

Über die Gradientenbildung und Rekonstruktion der Oberfläche hinaus seien hier einige weitere interessante Anwendungen nur stichpunktartig aufgeführt, die von dieser Repräsentation profitieren können:

**Skelettierung:** Die Menge der innen liegenden Punkte mit einem gegebenen Abstand zur Oberfläche liegt direkt vor.

**Boolesche Operationen:** Aus impliziten Flächen lassen sich Differenz, Vereinigung und Schnittmenge bilden. Dies wird in Abschnitt 3.3.1 weiter ausgeführt.

**Morphing:** Aus zwei Distanzfeldern kann durch Interpolation der Distanzwerte die implizite Fläche des einen Volumens in die andere überführt werden [COLS98].

**Beschleunigtes Raytracing:** Die Distanzinformation kann genutzt werden, um größere Leerräume im Volumen zu überspringen.

**Kollisionsbehandlung:** Die Eindringtiefe einer virtuellen Probe in das Volumen kann direkt festgestellt werden.

In diesem Kapitel wird ein Algorithmus zur Berechnung eines diskreten Distanzfeldes vorgestellt, und dessen Implementierung diskutiert. Wegen der äußerst umfangreichen Literatur zu diesem Thema werden hauptsächlich mit diesem Algorithmus verwandte Verfahren vorgestellt, um dessen Einordnung zu ermöglichen. Ein ausführlicher Überblick über andere Verfahren zur Erzeugung von Distanztransformationen findet sich bei Jones *et al.* [JBS06].

### 3.1 Verfahren zur Erzeugung von Distanztransformationen

Die Methoden zur Generierung einer Distanztransformation unterscheiden sich grundlegend in der Wahl der Ausgangsdaten: Ausgehend von einer initialen Belegung des Distanzfeldes erzeugen iterative Verfahren durch Propagierung von Information zu den Nachbarvoxeln eine Näherungslösung. Der andere Ansatz ist die Berechnung der Abstandsinformation direkt aus der gegebenen Oberfläche. Dies bedeutet konzeptionell, zu jedem Punkt im Raum den nächstgelegenen Punkt der Oberfläche zu bestimmen, und den Abstand zu speichern. Dieses Vorgehen liefert exakte Lösungen, ist jedoch extrem rechenaufwändig. Auch mit einer effizienten Raumaufteilung mittels Octree berichten beispielsweise Jones und Satherly [JS00] Rechenzeiten von 377 Sekunden für 216000 Distanzwerte (entsprechend etwa einem Voxelgitter der Auflösung  $128^3$ ), ermittelt für ein Netz von 1500 Dreiecken.

#### 3.1.1 Inkrementelle Propagierung

Der wohl älteste Vertreter der inkrementellen Methoden ist die Berechnung der *chamfer distance transform* (CDT) [RP66, Bor86, COLS98]. Die zugrundeliegende Annahme ist, dass aus einer bestehenden Belegung von Gitterpunkten mit Distanzwerten die Distanzwerte der jeweiligen Nachbarpunkte bestimmt werden können. Dieses Verfahren kann auf zwei- oder dreidimensionalen Gittern ausgeführt werden, und ist zwar schnell, aber ungenau.

Neueren Datums ist die *fast marching method* (FMM) [Set99]: Prinzipiell berechnet diese Methode Abstandswerte als Ausbreitungszeit einer voranschreitenden Front, die sich in Normalenrichtung innerhalb eines Gitters ausbreitet. Hierzu wird eine diskretisierte Version der Eikonalgleichung  $\|\nabla T\| = \frac{1}{F}$  betrachtet, wobei  $F \geq 0$  die Geschwindigkeit der Front bezeichnet, und  $T(\mathbf{p})$  die Ankunftszeit der Front an einem Punkt  $\mathbf{p}$  ist. Die Gleichung setzt also die Ausbreitungsgeschwindigkeit der Front mit der benötigten Zeit zum Erreichen eines Punktes in Beziehung. Der Abstand von  $\mathbf{p}$  zu der Front zum Zeitpunkt 0 ist dieser Zeit proportional, so daß die FMM zur Berechnung von Distanzfeldern eingesetzt werden

kann. Das Standardverfahren der FMM ist nicht sehr präzise, und es existieren zahlreiche Varianten mit dem Ziel einer Steigerung der Genauigkeit [JBS06].

Eine weitere Spielart der Propagierung von Informationen bilden die vektorbasierten Verfahren. Diese speichern und propagieren einen Vektor von jedem Gitterpunkt zum jeweils nächstgelegenen Oberflächenpunkt [Mul03, SJ00], und berechnen erst im letzten Schritt aus dieser Information die Distanzwerte.

Allen Verfahren gemein ist die nur näherungsweise Lösung, die sich aus der initialen Diskretisierung und den sich akkumulierenden Ungenauigkeiten der inkrementellen Berechnung ergibt.

### 3.1.2 Generalisierte Voronoidiagramme

Um die direkte Berechnung eines Distanzfeldes aus einem gegebenen Dreiecksnetz zu beschleunigen, kann ein dreidimensionales generalisiertes Voronoidiagramm [HCK<sup>+</sup>99] berechnet werden. Ein Standard-Voronoidiagramm ist in der Ebene für eine endliche Menge von Punkten definiert: Es teilt die Ebene vollständig in Zellen ein, die jeweils alle Punkte beinhalten, die einem Voronoi-punkt am nächsten liegen. In einem *generalisierten* Voronoidiagramm werden Abstände nicht nur zu Punkten, sondern auch zu Objekten wie Liniensegmenten oder Flächenelementen im  $\mathbb{R}^3$  betrachtet. Damit kann der Raum um ein Dreiecksnetz partitioniert werden: Zu jedem Punkt, jeder Kante und jedem Flächenelement des Dreiecksnetzes existiert eine Voronoizelle, die jene Punkte im Raum beinhaltet, die näher an diesem Element liegen als an allen anderen. Ist eine solche Aufteilung einmal gefunden, lässt sich das Abstandsfeld relativ schnell durch Abstandsmessung zu nur jeweils diesem Element berechnen [BMW01, SPG03].

Um die Berechnung eines diskreten Voronoidiagramms zu beschleunigen, approximieren Hoff *et al.* [HCK<sup>+</sup>99] die Voronoizellen durch verschiedene Polyeder. Diese Polyeder lassen sich durch eine schichtweise 3D-Scankonvertierung, ähnlich der in Abschnitt 2.2 beschriebenen Voxelisierung, in das Distanzfeld abbilden. Ein Voronoidiagramm kann also mit Hilfe der Grafikhardware errechnet werden. Allerdings besitzen die Voronoizellen analytisch teilweise gekrümmte Flächen (Kegel und Hyperboloide), die aufwendig mit bis zu 100 Dreiecken für beispielsweise einen Kegel trianguliert werden müssen, um eine ausreichende Genauigkeit zu erzielen.

### 3.1.3 Vereinfachte Scankonvertierung

Eine Spezialisierung dieses Verfahrens, ausschließlich zum Zweck der Berechnung von Distanztransformationen, stellt die *characteristics / scan-conversion*-Methode (CSC) dar [Mau]. Dieser Algorithmus berechnet keine exakten Voronoi-Diagramme, sondern verwendet größere, einfachere Polyeder als Einhüllende der Voronoizellen, die zudem nur einen Maximalabstand  $d$  von der Oberfläche abdecken — es wird also kein vollständiges Distanzfeld berechnet. Bei der Scankonvertierung ergeben sich dementsprechend Überlappungen. An diesen Stellen wird die Abstandsfunktion mehrfach berechnet, das heißt, für einen Raumpunkt wird der Abstand zu zwei oder mehr Flächenelementen bestimmt. Der Konflikt wird in diesem Falls durch Übernahme des minimalen Abstandswertes gelöst. Mauch verwendet drei Arten von Polyedern: dreiseitige Prismen als Verlängerung von Dreiecken nach oben und unten, Keile auf den Kanten zwischen den Dreiecken, und  $n$ -seitige Kegel für Eckpunkte, die die Lücken zwischen den beiden erstgenannten Polyedern füllen.

Ein Variante, die insbesondere auf eine GPU-Implementierung abzielt, stellen Sigg *et al.* [SPG03] vor. Anstelle der drei verschiedenen Polyeder wird nur eine einzige geometrische Form, ein dreiseitiges Prisma über der Dreiecksfläche, konstruiert. Die Seitenflächen der Prismen verlaufen wie in Abbildung 3.1 dargestellt, entlang der Winkelhalbierenden zwischen den Dreiecken, so dass dort keine Lücken entstehen. Jedes Prisma wird weiterhin so vergrößert, dass es die Vertexnormalen (resultierend aus dem Durchschnitt der angrenzenden Flächennormalen) an den Eckpunkten einhüllt. Es ergibt sich auch hier eine geringfügige Überlappung der Primitiven. Da nur ein Prisma pro Dreieck verwendet wird, reduziert sich die Anzahl der zu schneidenden Polyeder auf weniger als ein Drittel im Vergleich zur CSC-Methode.

Auch dieser Algorithmus geht schichtweise vor: In jeder Schicht des diskreten Distanzfeldes wird für jedes der die Dreiecke einhüllenden Prismen ein Schnitt-

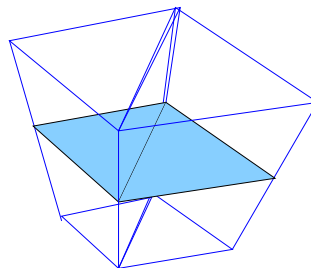


Abbildung 3.1.: Konstruktion der Prismen für benachbarte Dreiecke nach Sigg *et al.* [SPG03].

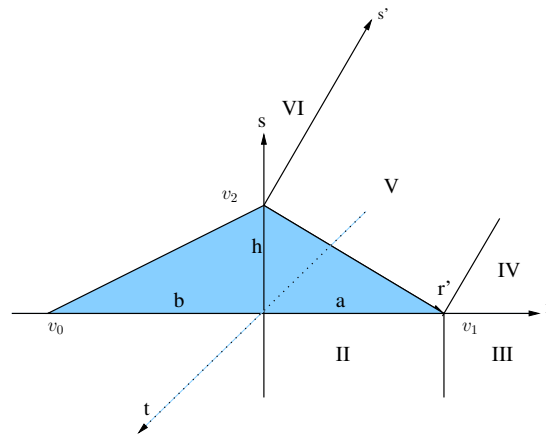


Abbildung 3.2.: Lokales Koordinatensystem eines Dreiecks mit den Achsen  $r$ ,  $s$ ,  $t$  zur Abstandsberechnung. Das Dreieck liegt in der Ebene  $t = 0$ .

polygon mit der Schicht berechnet. Dieses Polygon wird in den Bildspeicher gerendert, wobei ein Fragment Shader für jedes Pixel anstelle einer Farbe den Distanzwert zum zugehörigen Dreieck berechnet. Die Minimierung von mehrfachen Distanzwerten innerhalb überlappender Polygone wird mit Hilfe des Z-Buffer-Tests implementiert, der nur den kleinsten berechneten Wert in den Bildspeicher schreibt.

Da die Schichten in Z-Richtung sequentiell abgearbeitet werden, ist es nicht notwendig, für die Berechnung einer Schicht tatsächlich sämtliche Prismen zu betrachten: In einem Vorverarbeitungsschritt werden die Prismen nach ihren Z-Koordinaten in eine Liste sortiert, und für jeden Z-Schritt so ein aktiver Bereich der Liste bestimmt, der nur die Prismen enthält, die die aktuelle Ebene schneiden.

Für die Berechnung der Abstandswerte im Fragment Shader machen sich Sigg *et al.* die automatische lineare Interpolation von Werten zwischen den Eckpunkten eines Polygons, beispielsweise der Texturkoordinaten, zunutze. Für jedes Dreieck des zu konvertierenden Netzes wird ein lokales Koordinatensystem definiert, und jedem Eckpunkt eines Prismas die entsprechenden Koordinaten in diesem lokalen System als Texturkoordinaten zugewiesen. Beim Schnitt gegen die Distanzvolumenschicht wird nicht nur die Position im Weltkoordinatensystem, sondern auch diese Texturkoordinate interpoliert. Im Fragment Shader kommen zu jedem Pixel die linear interpolierten lokalen Koordinaten als Texturkoordinaten an. Eine einfache Vorschrift berechnet den Abstand des durch diese Koordinaten repräsentierten Punktes zum Dreieck. Auf diese Weise wird die teuerste Rechenoperation, die Berechnung der Länge des Distanzvektors in jedem Punkt des Distanzvolumens, auf die Grafikkarte verlegt und kann stark parallelisiert werden.

Die lokalen Koordinaten, wie in Abbildung 3.2 dargestellt, berechnen sich für ein Dreieck wie folgt:

- Die  $t$ -Achse ist die Flächennormale des Dreiecks.
- Sei die längste Kante des Dreiecks die Verbindung von Punkt  $v_0$  nach  $v_1$ . Dann ist der Ursprung  $o$  des Koordinatensystems die Projektion von  $v_2$  auf  $v_1 - v_0$ .
- Die  $r$ -Achse ist der Einheitsvektor in Richtung  $v_1 - o$ .
- Die  $s$ -Achse ergibt sich aus  $v_2 - o$ .

In diesem Koordinatensystem beschreiben die Längen  $a$ ,  $h$  und  $b$  das Dreieck vollständig. Zwei wesentliche Merkmale der Koordinatentransformation sind:

- Der Wechsel des Koordinatensystems beinhaltet keine Skalierung. Daher ist der Abstand  $D(r, s, t)$  eines Punktes im lokalen Koordinatensystem gleich dem Abstand zu dem ursprünglichen Dreieck in Modellkoordinaten.
- Das Dreieck liegt in der Ebene  $t = 0$ , so dass  $D(r, s, t) = \sqrt{D(r, s, 0)^2 + t^2}$ . Das Problem lässt sich also auf zwei Dimensionen reduzieren.

Eine Fallunterscheidung erfolgt nach den in Abbildung 3.2 aufgeführten Regionen I – VI, in die der Punkt  $(r, s, 0)$  fällt. Für  $r < 0$  können die Koordinaten an der  $s$ -Achse gespiegelt werden, so dass die linke Hälfte des Diagramms nicht mehr betrachtet werden muss. Pseudocode für die Berechnung der Distanz im Fragment Shader ist in Abbildung 3.1.3 wiedergegeben.

```
// Normalisierung auf r >= 0
if (r < 0)
    r = -r; a = b;

// Abstandsberechnung
if (s < 0)
    r = max(r-a, 0); // II, III
    d = sqrt(r^2 + s^2 + t^2);
else
    // Hilfskoordinatensystem
    lenSqr = a^2 + h^2;
    r' = (a*r - h*s + h^2) / lenSqr;
    s' = (h*r + a*s - a*h) / lenSqr;

    r' = max(-r', r'-1, 0); // IV, V, VI
    s' = max(s', 0); // I, V
    d = sqrt((r'^2 + s'^2) * lenSqr + t^2);
```

Abbildung 3.3.: Pseudocode für die Distanzberechnung im lokalen  $r, s, t$ -Koordinatensystem nach [SPG03].

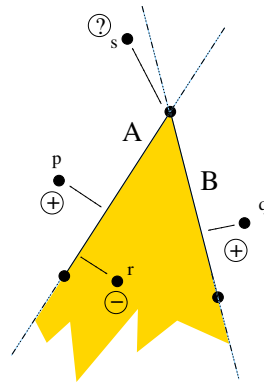


Abbildung 3.4.: Fehlerhaftes Vorzeichen bei der Distanzberechnung: die farbige Fläche liegt innerhalb eines durch die Polygone  $A$  und  $B$  berandeten Körpers. Das Vorzeichen der Distanzfunktion innerhalb ist negativ, ausserhalb positiv. Die Punkte  $p$ ,  $q$ ,  $r$  werden ihrem nächstgelegenen Polygon zugeordnet. Der Test gegen die Ebene des Polygons ergibt das korrekte positive ( $p$ ,  $q$ ) bzw. negative ( $r$ ) Vorzeichen. Der Punkt  $s$  ist einem gemeinsamen Eckpunkt von  $A$  und  $B$  zugeordnet. Je nachdem, welchem Polygon  $s$  zugerechnet wird, ergibt der Ebenentest das korrekte positive oder fälschlicherweise ein negatives Vorzeichen.

Sowohl Jones *et al.* [JBS06] als auch Erleben *et al.* [ED08] weisen auf Probleme bei der Bestimmung des korrekten Vorzeichens hin: Das Vorzeichen des Abstands eines Punktes zu einem Dreieck wird durch einen Test gegen dessen Ebene ermittelt. Im einen Halbraum ist das Vorzeichen positiv, im anderen negativ. Bei einigen geometrischen Konfigurationen, insbesondere bei spitzen Winkeln zwischen aneinandergrenzenden Dreiecken, kann auch bei korrektem Absolutwert der Distanz dieser Ebenentest ein falsches Ergebnis liefern. Ein Punkt wird dann fälschlich als im Körper liegend markiert, oder umgekehrt. Abbildung 3.4 illustriert einen solchen Fall. Diese Problemfälle treten auf, wenn der untersuchte Punkt nicht direkt über einer Dreiecksfläche liegt, sondern einem gemeinsamen Eckpunkt oder Kante im Dreiecksnetz am Nächsten liegt. Zur Lösung wird daher die Einführung von sogenannten Pseudonormalen [AB03] vorgeschlagen, die eine Gewichtung aus den Flächennormalen darstellen. Der Ebenentest wird dann gegen diese Normale durchgeführt.

Erleben *et al.* variieren das Verfahren von Sigg, indem anstelle eines Prismas für jedes Dreieck lediglich eine orientierte Boundingbox konstruiert wird, die so groß gewählt wird, dass mindestens der Abstand  $d$  zwischen jedem Punkt des Polygons und der Außenseite der Box liegt (vgl. Abbildung 3.5). Die Berechnung der Schnittpolygone wird nach einer Tetraedisierung der Boundingbox auf der CPU ausgeführt. Die Konstruktion eines solchen Quaders erlaubt einerseits den Verzicht auf Konnektivitätsinformation im Dreiecksnetz (bei Sigg *et al.* sind noch

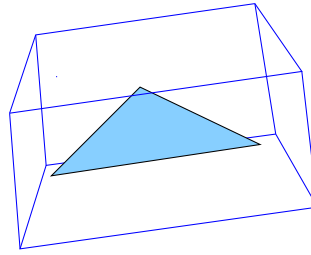


Abbildung 3.5.: Orientierte Boundingbox für ein Dreieck nach Erleben *et al.* [ED08].

gemeinsame Vertexnormalen und Winkelhalbierende zwischen Dreiecken erforderlich), erzeugt andererseits sehr große Überlappungen. Da Erleben *et al.* zudem die beschriebenen Pseudonormalen zur Vorzeichenberechnung verwenden, wird die Konnektivität im Dreiecksnetz letztendlich doch benötigt.

### 3.2 Implementierung eines modifizierten Algorithmus

Zur Berechnung des diskretisierten Distanzfeldes wurde ein Algorithmus implementiert, der die Verfahren von Sigg und Erleben mit der in Kapitel 2 erläuterten Voxelisierung kombiniert. Auf diese Weise wird für das gesamte Volumen ein Innen/Außen-Test durchgeführt, und damit das Vorzeichen der Abstandsfunktion festgelegt. Die absoluten Abstandswerte werden nach der beschriebenen Systematik berechnet, und anschließend mit dem Vorzeichen kombiniert. Wie bei Erleben *et al.* wird eine orientierte Boundingbox für jedes Dreieck verwendet. Anstatt diese in Tetraeder zu zerlegen, wird jedoch der in Abschnitt 2.4.3 beschriebene Vertex Shader zur Berechnung der Schnittpolygone verwendet. Dieser Berechnungsschritt konnte somit ebenfalls auf die GPU verlagert werden.

Mit einem Dreiecksnetz und der gewünschten Auflösung des diskretisierten Zielvolumens als Eingabe, läuft der Algorithmus dann in den folgenden Schritten ab:

1. Vorbereitung: Berechnung einer orientierten Boundingbox und des lokalen Koordinatensystems für jedes Dreieck.
2. Für jede Z-Schicht im zu füllenden Distanzvolumen:
  - a) Vorzeichenbestimmung durch binäre Voxelisierung. Das Ergebnis steht anschließend in einem Kanal des Bildspeichers.



- b) Für jede Boundingbox, die die aktuelle  $Z$ -Schicht schneidet: Berechnung des Schnittpolygons und Abstandsberechnung zum zugehörigen Dreieck. Der absolute Abstandswert wird in einem anderen Kanal des Bildspeichers abgelegt. Der  $Z$ -Buffer wird für die Minimierung mehrfach berechneter Abstandswerte in einem Punkt verwendet.
- c) Kombination des Vorzeichens mit dem absoluten Wert in jedem Punkt, Speicherung des Resultats im Distanzvolumen.

Wie im Abschnitt 3.1.3 beschrieben, wird in der Initialisierungsphase zunächst ein lokales  $r, s, t$ -Koordinatensystem erzeugt. Das Dreieck wird in diesem Koordinatensystem durch die drei Parameter  $a, b, h$  beschrieben. Mit Hilfe der Achsen dieses lokalen Koordinatensystems wird eine orientierte Boundingbox berechnet, die so groß gewählt wird, dass ein minimaler Abstand  $d$  der Seitenflächen zu jedem Punkt des Dreiecks eingehalten wird.

Für die spätere Berechnung der Schnittpolygone einer  $Z$ -Ebene mit der Boundingbox werden die Weltkoordinaten der Eckpunkte benötigt, sowie der Index des Eckpunktes mit dem größten  $Z$ -Wert (siehe Abschnitt 2.4.3). Schließlich wird noch der minimale und maximale  $Z$ -Wert jeder Boundingbox verwendet, um eine nach  $Z$ -Werten sortierte Liste zu erzeugen, und einen aktiven Abschnitt dieser Liste während der Iteration durch die Schichten zu verwalten. Diese Daten werden also für jedes Dreieck in der folgenden Struktur gespeichert:

```
struct OrientedBox
{
    Vec3f v[8]; // globale Eckpunktkoordinaten
    Vec3f rst[8]; // lokale Eckpunktkoordinaten
    float a, b, h; // Beschreibung des Dreiecks
    float minZ, maxZ; // abgedeckter Z-Bereich
    int maxZVertexIndex; // EckpunktIndex [0 - 7]
}
```

Alle aufwändigen Berechnungen finden direkt auf der GPU statt: Die Vorzeichenberechnung ist identisch zu der Voxelisierung aus Abschnitt 2.2, und kommt ohne Shader-Implementierung aus. Für die nachfolgenden Schritte der Berechnung der Schnittpolygone und der eigentlichen Distanzberechnung kommen ein Vertex Shader und ein Fragment Shader zum Einsatz. Der auf der CPU laufende Programmteil aktualisiert lediglich den aktiven Abschnitt der Boundingbox-Liste, und überträgt für jede aktive Boundingbox die in der oben beschriebenen Struktur abgelegten Daten an die Grafikkarte. Das im Bildspeicher abgelegte Resultat wird dann wieder im Hauptspeicher kombiniert.

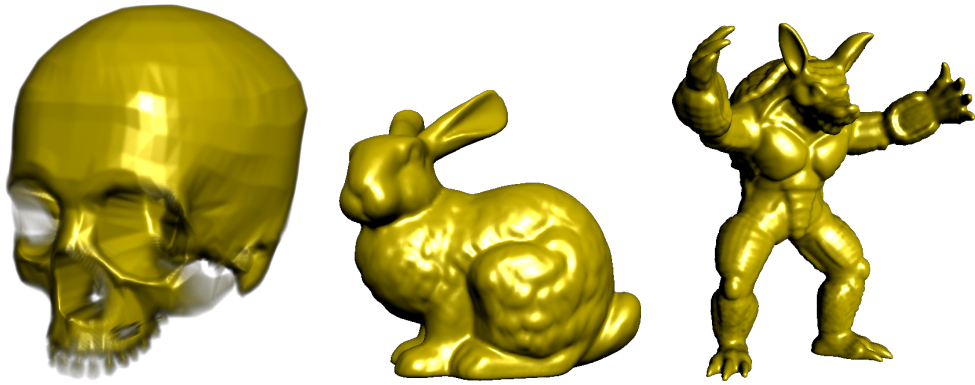


Abbildung 3.6.: Darstellung der Isofläche  $f = 0$  für Distanzfelder einiger Modelle.

Der Vertex Shader, der die Polygonschnitte berechnet, verwendet die Eckpunkte jeder Boundingbox in globalen Koordinaten, um den korrekten Schnitt mit der Z-Ebene zu berechnen. Jedes ausgegebene Polygon enthält jedoch zusätzlich die entsprechenden lokalen Dreieckskoordinaten der Schnittpunkte, die vom Fragment Shader zur Abstandsberechnung benötigt werden.

### 3.3 Ergebnisse

Der implementierte Algorithmus teilt den Großteil der Eigenschaften der Verfahrens von Sigg beziehungsweise Erleben. So werden exakte Abstandswerte bis zu einem maximalen Abstand  $d$  zum Dreiecksnetz berechnet, wobei die Distanzberechnung parallelisiert in einem Fragment Shader erfolgt.

Das modifizierte Verfahren besitzt den Vorteil eines garantiert korrekten Vorzeichens im gesamten Volumen, nicht nur in der Nähe der Oberfläche, was einen Innen/Außen-Test ermöglicht. Ferner werden als Eingabedaten nur die Geometrien einzelner Dreiecke benötigt, so daß keine Datenstrukturen benötigt werden, die Nachbarschaftsbeziehungen im Dreiecksnetz abbilden. Schließlich wurde auch die Berechnung der Schnittpolygone auf die GPU verlegt.

Abbildung 3.6 zeigt Darstellungen der Isoflächen einiger in ein Distanzfeld konvertierten Polygonmodelle verschiedener Komplexität. Gut sichtbar ist die verbesserte Qualität der Gradienten im Distanzfeld: Im Vergleich zu Abbildung 2.2 ist die Oberflächendarstellung deutlich artefaktfreier. In Abbildung 3.7 ist die Visualisierung von Isoflächen in verschiedenen Bereichen des Distanzfeldes illustriert. Die Variationen wurden allein durch Veränderung der im vorigen Kapitel vorgestellten Transferfunktion erzeugt.

Tabelle 3.1 zeigt die Berechnungszeiten der dargestellten Distanztransformationen. Alle Distanzfelder wurden bei einer Auflösung von  $256^3$  erzeugt. Die Werte beinhalten nur die Zeiten für die innere Schleife des Algorithmus, ohne die Zeiten für die Vorberechnung der Boundingboxes und das Kopieren der Resultate in den Hauptspeicher, die auch bei den größeren Modellen nur ca. 1% der Gesamtlaufzeit einnehmen.

Die Breite des Bereiches, in dem Distanzwerte berechnet werden, bestimmt die Größe der Boundingboxes für jedes Dreieck. Mit zunehmender Größe dieser Quader werden im Fragment Shader zum einen entsprechend der Fläche des Schnittpolygons mehr Distanzwerte berechnet, zum anderen steigt auch die Anzahl der sich gegenseitig durchdringenden Quader stark an. Dies führt dazu, dass bei einem Dreiecksnetz mit gegenüber der Boundingbox-Größe kleinen Dreiecken in jeder Schicht sehr viele Schnittpolygone berechnet werden müssen, und es zu massiven Mehrfachberechnungen von Distanzwerten für ein Pixel kommt. Daraus erklärt sich die starke Zunahme der Laufzeit nach rechts und unten in der Tabelle.

Der vielversprechendste Ansatz zur Optimierung der Laufzeit ist daher die Verwendung besser zugeschnittener Polyeder, was ohne die Einbeziehung von Nachbarschaftsinformationen aus dem Ursprungsdreiecksnetz nur sehr eingeschränkt möglich ist. Die hier gewählte Implementierung gewährleistet dagegen größere Robustheit bei einfacherer Implementierung, wenn eine geringe Laufzeit keine hohe Priorität hat.

Weiterhin werden in der aktuellen Implementierung die Geometrien der Boundingboxes jeweils neu an die Grafikkarte übertragen, bei den größeren Modellen oft pro Schicht jeweils mehrere Zehntausend. OpenGL bietet verschiedene

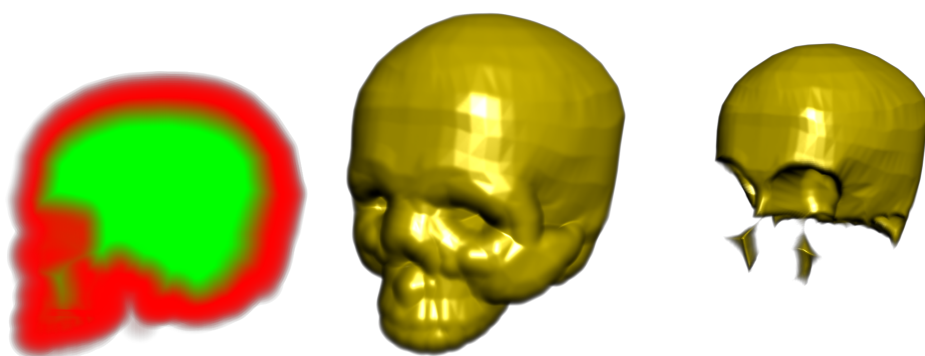


Abbildung 3.7.: Darstellung verschiedener Isoflächen. Links: Schnitt durch das Distanzfeld; Bereiche mit positivem Vorzeichen sind rot, mit negativem Vorzeichen grün dargestellt. Mitte: Isofläche im äußeren Bereich ( $f > 0$ ). Rechts: Isofläche im inneren Bereich ( $f < 0$ ).

Tabelle 3.1.: Zeiten für die Erzeugung eines diskretisierten Distanzfeldes für Dreiecksnetze verschiedener Komplexität, in Abhängigkeit von der Breite des berechneten Bandes in Prozent der Z-Ausdehnung des jeweiligen Modells. bei 256 Schichten und 10% belegt das Band an Dichtewerten also ca. 25 Voxel jeweils in positiver und negativer Entfernung von der Modelloberfläche.

Modell	# $\Delta$	1%	5%	10%
schädel	6621	3,83s	5,4s	6,7s
bunny	69660	9,00s	25,35s	42,29s
armadillo	345944	30,49s	113,84s	198,25s

Möglichkeiten, solche konstanten Daten direkt auf der Grafikkarte zu belassen, wodurch der Zeitaufwand für die wiederholte Übertragung entfallen könnte.

Schließlich wäre zur Berechnung eines vollständigen Distanzfeldes die Kombination mit einem iterativen Verfahren interessant. Die innerhalb eines Bandes exakt berechneten Distanzwerte können verwendet werden, um durch Propagierung den nicht erfassten Raum des Volumens zu füllen.

### 3.3.1 Schnitte zwischen Distanzfeldern

Sind zwei implizite Flächen als Distanzfelder gegeben, können durch Vereinigung, Differenz und Schnittmenge neue implizite Flächen gebildet werden [BBB<sup>+</sup>97]. Mit negativen Distanzwerten für den inneren Teil der durch  $f_1$  und  $f_2$  gegebenen Körper, und positiven Werten entsprechend ausserhalb, gilt:

$$\begin{aligned} f_1 \cup f_2 &= \min(f_1, f_2) \\ f_1 \cap f_2 &= \max(f_1, f_2) \\ f_1 \setminus f_2 &= f_1 \cap -f_2 = \max(f_1, -f_2). \end{aligned}$$

Anschaulich umfasst die Vereinigung der Volumina den Teil, der in mindestens einem der Körper negativ ist, die Schnittmenge nur den Teil, der in beiden Körpern negativ ist. Die Differenz bildet der Bereich, der innerhalb von  $f_1$ , aber ausserhalb von  $f_2$  liegt. Auf das in Abschnitt 2.5 vorgestellte Selektionsvolumen angewandt, ist die Ausmaskierung von Voxeln, die im Selektionsvolumen liegen, durch die Differenzbildung zu ersetzen. Abbildung 3.8 zeigt noch einmal das Herausschneiden einer Kugelfläche aus einem Quader. Beide Volumina liegen als Distanzfelder der Auflösung  $64^3$  vor. Im Unterschied zur Abbildung 2.11 werden die sehr viel glatteren Schnittflächen deutlich.

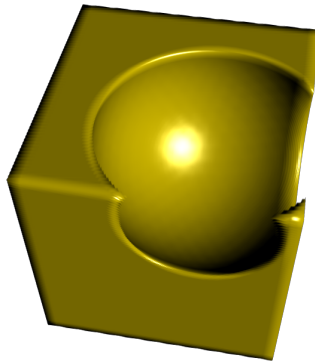


Abbildung 3.8.: Selektion auf Basis von Distanzfeldern: Kugelförmiger Ausschnitt aus einem Quader.

### 3.4 Zusammenfassung

Die direkte Konvertierung eines Dreiecksnetzes in ein diskretisiertes Distanzfeld kann die Qualität der Visualisierung durch die Verfügbarkeit lokaler Gradienten an der Oberfläche stark verbessern. Diese Repräsentation ermöglicht viele weitere Anwendungen, von denen in diesem Kapitel die Visualisierung von Isoflächen und eine verbesserte Selektion gezeigt wurden.

Strukturell ist das diskrete Distanzfeld identisch mit einem Grauwertvolumen als Voxelmodell, mit den bereits vorgestellten Vorteilen und Einschränkungen, beispielsweise hinsichtlich der maximalen Auflösung. Auch ist die Manipulation von Werten im Voxelgitter, etwa bei einer simulierten Deformation, aufgrund der kubisch mit der Auflösung wachsenden Anzahl der Elemente sehr aufwändig.

Weiterhin wurde bisher gezeigt, wie eine triangulierte Oberfläche in eine Volumendarstellung überführt werden kann. Nach einer Operation auf dem Volumen, wie etwa Schnitten oder einer Segmentierung, kann jedoch auch die Rückkonvertierung in eine triangulierte Oberfläche sinnvoll sein. Das nächste Kapitel befasst sich mit einer weiteren wichtigen Volumendarstellung auf Basis von Tetraedern, und zeigt darüber auch den Rückweg zu einer Oberflächenrepräsentation als Dreiecksnetz auf.



## 4 Tetraedisierung

Eine wichtige Volumenrepräsentation ist die Unterteilung in tetraederförmige Elemente. Finite-Elemente-Methoden (FEM), die für die Simulation beispielsweise mechanischer Deformation eingesetzt werden [CDM<sup>+</sup>02, Bat82], setzen in der Regel eine Tetraedisierung voraus. In medizinischen Anwendungen werden diese Methoden in der Operationsplanung und in virtuellen Trainingsumgebungen verwendet [BNC99]. Die Unterteilung des Volumens in Tetraeder stellt dann eine Diskretisierung von Materialeigenschaften über ein Volumen dar.

Im Gegensatz zur Voxelrepräsentation ist eine Tetraederdarstellung nicht zwangsläufig mit einem regelmäßigen Gitter verbunden, so dass eine adaptive, der jeweiligen Problemstellung und der Form des Objektes angemessene Unterteilung verwendet werden kann.

Da die Oberfläche eines Tetraedernetzes ein geschlossenes Dreiecksnetz bildet, kann aus dieser Repräsentation ohne weiteres eine Oberflächendarstellung generiert werden. Werden die Oberflächendreiecke mit Attributen wie Oberflächennormalen und Texturkoordinaten versehen, können sie direkt visualisiert werden. In der Betrachtung als volumetrische Erweiterung von Dreiecksnetzen sind viele geometrische Algorithmen, wie Subdivision und Netzreduktion auf Tetraedisierungen übertragbar [GH97, CDM<sup>+</sup>02].

### 4.1 Verfahren zur Erzeugung von Tetraedisierungen

Die Generierung von qualitativ hochwertigen Tetraedernetzen ist besonders problematisch, wenn scharfe Kanten an der Oberfläche von Objekten möglichst exakt nachgebildet werden müssen, ohne degenerierte Tetraeder zu erzeugen [BS89]. Die Literatur zu diesem Thema ist außerordentlich umfangreich und kann hier nicht vollständig abgehandelt werden. Eine Anzahl von Algorithmen erzeugt Tetraedisierungen über eine als implizite Funktion gegebene Geometrie. Diese sind hier von besonderem Interesse, da sich so die Verbindung zu den im vorangegangenen Kapitel besprochenen Distanzfeldern herstellen lässt.

Im Gegensatz zu Methoden, die eine Tetraedisierung ausgehend von einer existierenden triangulierten Oberfläche berechnen [Bak89, Loh88], ist für eine implizit gegebene Geometrie der Oberflächenverlauf im Allgemeinen nicht analytisch zugänglich. Daher wird dem Volumen, das die zu konvertierende Geometrie enthält, eine Struktur überlagert, auf der die implizite Funktion abgetastet

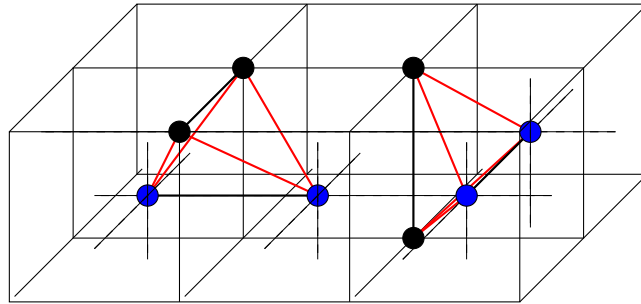


Abbildung 4.1.: Zwei Tetraeder im BCC–Gitter. Jeder Tetraeder verbindet je zwei Punkte des einen Gitters (schwarze Punkte), mit zwei des anderen, verschobenen Gitters (blaue Punkte, gestrichelte Linien) entlang der roten, diagonalen Kanten.

wird. Ein einfaches Beispiel ist die Erweiterung des bekannter *marching cubes*–Algorithmus [LC87] auf Tetraeder, wie sie von Bloomenthal [BBB<sup>+</sup>97] diskutiert wird. *Marching Cubes* extrahiert aus einem dreidimensionalen Gitter von Abtastwerten einer impliziten Funktion die Isofläche, auf der die Funktion einen bestimmten Wert (in der Regel 0) annimmt. Denkt man sich zwei Gitterpunkte mit innen beziehungsweise außen liegenden Abtastwerten durch Kanten verbunden, liegt auf dieser Kante ein Schnittpunkt mit der Isofläche. Der Verlauf der Fläche durch einen Gitterwürfel mit acht Eckpunkten ergibt 256 verschiedenen Möglichkeiten für “innen” und “außen”. Eine Tabelle enthält passende Triangulierungen für alle möglichen Durchgänge der Fläche durch den Gitterwürfel. Über das gesamte Gitter iteriert, ergibt sich eine Triangulierung der gesuchten Fläche. Auf ähnliche Weise kann jeder Gitterwürfel in fünf oder sechs Tetraeder unterteilt werden. Diese Methode generiert allerdings sehr viele Tetraeder oft sehr schlechter Qualität, wobei die Qualität in der Regel über den minimalen und maximalen dihedralen Winkel definiert wird, der im Netz auftreten kann.

Anstelle eines regulären Gitters können Tetraeder auch in einem Doppelgitter, dem sogenannten BCC–Gitter (*body centered cubic grid*) [Fuc98, Nay99] erzeugt werden. Das BCC–Gitter besitzt Knoten an den ganzzahligen Koordinaten und an den Mittelpunkten dieser Zellen:  $BCC = \mathbb{Z}^3 \cup (\mathbb{Z}^3 + (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}))$ . Die Delaunaytriangulation dieses Gitters besteht aus identisch geformten Tetraedern, wie in Abbildung 4.1 dargestellt. Die horizontalen Kanten dieser Tetraeder besitzen die Länge 1, die Diagonalen die Länge  $\sqrt{3}/2$ . Die Seitenflächen der Tetraeder sind über die dihedralen Winkel  $60^\circ$  und  $90^\circ$  verbunden.

In der Regel wird vorausgesetzt, dass die implizit repräsentierte Geometrie scharfe Ecken und Kanten besitzt, die in der Tetraedisierung exakt wiederzugeben sind. Diese Einschränkung kompliziert die Entwicklung eines robusten Algorithmus er-



heblich, und macht eine Garantie für hohe Tetraederqualität an jedem Punkt der Oberfläche nahezu unmöglich. Besser geformte Tetraeder lassen sich erzielen, wenn die Anforderung an beliebig genaue Repräsentation von Diskontinuitäten fallen gelassen wird, also mehr oder weniger "glatte" implizite Flächen rekonstruiert werden. Molino *et al.* [MBTF03] erzielen auf diese Weise in der Praxis gut geformte Tetraeder, geben aber keine Garantien.

Ein relativ neues Verfahren von Labelle und Shewchuck [LS07], ebenfalls auf dem BCC-Gitter basierend, bietet hingegen eine beweisbare Qualität der Tetraeder. Der Algorithmus, *isosurface stuffing* getauft, eignet sich besonders für glatte, aus Isoflächen generierte Objekte. Die Autoren versprechen eine robuste, einfache Implementierung, hohe Ausführungsgeschwindigkeit, sowie die erwähnt garantierte hohe Qualität der Tetraedisierung. Daher wurde dieser Algorithmus im Rahmen der vorliegenden Arbeit für die Implementierung ausgewählt und wird im Rest des Kapitels genauer vorgestellt.

## 4.2 Isosurface Stuffing

Isosurface Stuffing füllt das Innere einer Isofläche mit einem regelmäßigen Tetraedergitter. Nach Angabe der Autoren ist dies der erste Algorithmus, der für FEM-Simulationen geeignete Tetraedisierungen über komplexe Isoflächen garantieren kann. Der Algorithmus garantiert dihedrale Winkel entweder zwischen  $10,7^\circ$  und  $164,8^\circ$ , oder  $8,9^\circ$  und  $158,8^\circ$ , je nach Wahl der Parameter. Ähnlich wie Marching Cubes verwendet der Algorithmus eine kleine Anzahl vorberechneter Stempel für lokale Tetraedisierungen. Im Folgenden wird der Algorithmus skizziert, für die vollständige Diskussion der Details und für die Beweise der gegebenen Garantien sei auf [LS07] verwiesen. Stattdessen werden an dieser Stelle Details über die konkrete Realisierung der zentralen Datenstrukturen vorgestellt, auf die Labelle *et al.* nicht im Einzelnen eingehen.

Im Text ist – der Anwendung in dieser Arbeit entsprechend – von Distanzfunktionen und Distanzfeldern die Rede. Der Algorithmus funktioniert im Allgemeinen jedoch nicht nur auf Distanzfunktionen, sondern auf beliebigen impliziten Funktionen  $f : \mathbb{R}^3 \mapsto \mathbb{R}$ , bei denen Raumpunkte innerhalb der Geometrie ein positives, und außerhalb ein negatives Vorzeichen besitzen. Diese Vorzeichenkonvention dreht die in den bisherigen Kapiteln verwendete Zuweisung um. Die Originaldarstellung der Autoren wird hier dennoch beibehalten.

### 4.2.1 Ablauf der Tetraedisierung

Wie erwähnt, verwendet der Algorithmus ein BCC–Gitter. Auf Basis der regelmäßigen, raumfüllenden Tetraedisierung dieser Struktur wird das Innere der Isofläche in vier Schritten gefüllt. Der Ablauf ist ähnlich dem Marching Cubes–Verfahren, mit dem wesentlichen Unterschied des zusätzlichen ”Verschieben”–Schrittes.

**Klassifikation:** Die Distanzfunktion wird an den Gitterpunkten des BCC–Gitters ausgewertet. Jeder Gitterpunkt wird als negativ (außen) oder positiv (innen) klassifiziert.

**Schnittpunktbestimmung:** Für jede horizontale, vertikale, und diagonale Kante des Gitters, die je einen positiven und negativen Endpunkt besitzt, wird der Schnittpunkt mit der Isofläche bestimmt und auf der Kante gespeichert.

**Verschieben:** Von jedem Gitterpunkt gehen 4 horizontale, 2 vertikale, und 8 diagonale Kanten aus. Für jeden Gitterpunkt wird jede dieser 14 Kanten auf einen nahegelegenen Schnittpunkt überprüft. Die genaue Definition, ab wann ein Schnittpunkt als ”nah” betrachtet wird, ist der wesentliche Parameter des Verfahrens, der die Winkelgarantien beeinflusst. Gegebenenfalls wird der Gitterpunkt zum nächstgelegenen dieser Schnittpunkte verschoben. Da der Punkt nun genau auf der Oberfläche liegt, wird seine Klassifikation von ”positiv” oder ”negativ” auf ”Null” geändert. Etwaige weitere Schnittpunkte auf den angrenzenden Kanten sind nun ungültig und werden verworfen.

**Tetraedergenerierung:** Jeder BCC–Tetraeder mit mindestens einem positiven Eckpunkt wird mit einem der tabellierten Stempel gefüllt. Im einfachsten Fall (kein Schnittpunkt entlang der Kanten) wird nur ein Tetraeder aus den – potentiell verschobenen – Gitterpunkten erzeugt. Schneidet die Oberfläche ein oder mehrere Kanten des Tetraeders, wird er in bis zu drei Tetraeder unterteilt.

### 4.2.2 Parameterwahl

In dritten Schritt des Algorithmus werden Gitterpunkte zu den auf angrenzenden Kanten liegenden Schnittpunkten verschoben. Die Wahl des Schwellwertes, ab wann ein Schnittpunkte als ”nahegelegen” betrachtet wird, bestimmt die in der resultierenden Tetraedisierung vorkommenden dihedralen Winkel. Die Autoren geben für die kurzen horizontalen und vertikalen, sowie die langen diagonalen Kanten je eigene Schwellwerte an, die mit  $\alpha_{short}$  beziehungsweise  $\alpha_{long}$  bezeichnet sind. Eine Tabelle mit verschiedenen Werten für  $\alpha_{short}$  und  $\alpha_{long}$  ist in [LS07] aufgeführt.

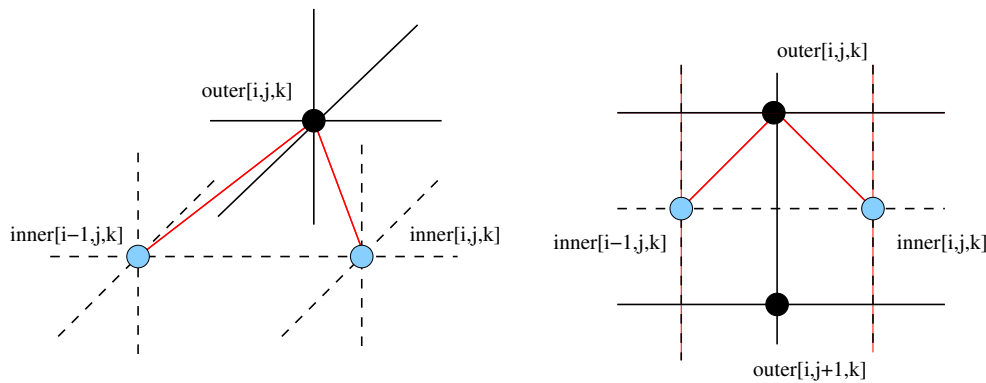


Abbildung 4.2.: Abbildung der geometrischen Anordnung auf die Indexbeziehung zwischen äußeren (schwarze Punkte) und inneren Gitterpunkten (blaue Punkte). Links: perspektivische Ansicht im dreidimensionalen Gitter. Rechts: von oben. Innerhalb jedes Arrays sind Punkte durch schwarzen Kanten horizontal und vertikal verbunden, zwischen den Arrays liegen diagonale rote Kanten.

### 4.2.3 Implementierung der Gitterdatenstrukturen

Die beiden ineinander verschachtelten, und gegeneinander verschobenen Gitter werden als dreidimensionale Arrays repräsentiert. Das innere Gitter (mit den konzeptionell halbzahligen Koordinaten) hat in jeder Raumrichtung eine um eins kleinere Dimension:

```
struct BccGrid
{
    BccGridPoint outerGrid[xDim, yDim, zDim];
    BccGridPoint innerGrid[xDim-1, yDim-1, zDim-1];
}
```

Die Verknüpfung zwischen den beiden Gittern ist damit durch einfache Beziehungen zwischen den Koordinaten gegeben. Die horizontalen und vertikalen "schwarzen Kanten" verbinden Einträge innerhalb eines Arrays. Diagonale "rote Kanten" verbinden ein Array mit dem anderen. Eine diagonale Kante wird beispielsweise durch die beiden Punkte  $outerGrid[x, y, z]$  und  $innerGrid[x, y, z]$  beschrieben. Abbildung 4.2 illustriert diese Beziehung.

Da ein Gitterpunkt durch Verschieben neu positioniert werden kann, wird in jedem Gitterpunkt dessen aktuelle Position eingetragen. Diese wird mit den ganzbeziehungsweise halbzahligen Gitterkoordinaten initialisiert. Weiterhin wird das Vorzeichen der Distanzfunktion in jedem Punkt gespeichert: positiv, negativ, oder Null (genau auf der Oberfläche).

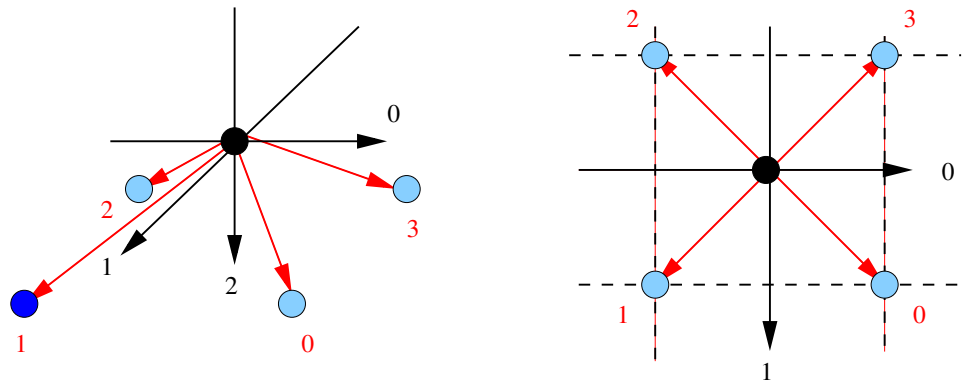


Abbildung 4.3.: Zuordnung der Halbkanten in der BccGridPoint-Struktur. Links: perspektivische Ansicht des dreidimensionalen Arrays. Rechts: von oben. Die Parametrisierung eines Schnittpunktes von 0 bis 1 verläuft immer in Pfeilrichtung.

Entlang der roten und schwarzen Kanten müssen etwaige Schnittpunkte mit der Fläche verwaltet werden. Dies kann effizient mit einer Halbkantenstruktur erfolgen: Jeder Gitterpunkt enthält Schnittpunkte zu genau der Hälfte seiner Nachbarn, also für drei schwarze und vier rote gerichtete Kanten. Der Schnittpunkt ist der Parameter zwischen der Position des Start- und Endknotens,  $t \in [0, 1]$ . Die vollständige Datenstruktur eines Gitterpunktes stellt sich somit wie folgt dar:

```
struct BccGridPoint
{
    float position[3];
    byte sign; // -1, 0, 1
    float blackEdge[3];
    float redEdge[4];
}
```

Die Numerierung der Kanten in dieser Struktur ist in Abbildung 4.3 dargestellt.

Gitterpunkte am Ende einer Halbkante besitzen also selbst keine Information über die Kante. Um den Schnittpunkt in Gegenrichtung zu bestimmen, wird über die Beziehung zwischen den Gitterkoordinaten der Kantenstartpunkt gesucht. Aus dem auf der Halbkante gespeicherten Parameter  $t$  wird dann der gesuchte Parameter als  $1 - t$  ermittelt. Sei beispielsweise ein Schnittpunkt mit der Fläche zwischen den äußeren und inneren Gitterpunkten  $(i, j, k)$  vorhanden, mit  $t = 0, 3$ , wie in Abbildung 4.4 dargestellt

Der Wert für den Schnittpunkt vom Punkt `innerGrid[i, j, k]` auf dem inneren Gitter entlang der entgegengesetzten Halbkantenrichtung ergibt sich also zu  $(1.0 - \text{outerGrid}[i, j, k].\text{redEdge}[0])$ .

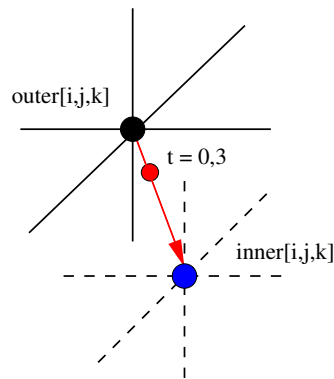


Abbildung 4.4.: Zuordnung eines Parameterwertes auf einer diagonalen Halbkante.

Diese Datenstruktur ist ein guter Kompromiss zwischen Kompaktheit durch Vermeidung von Redundanz, und schnellem Zugriff auf alle erforderlichen Informationen. In der üblichen Fließkommadarstellung mit 32 Bit benötigt jeder Gitterpunkt 41 Byte, bei 4-Byte-Alignment 44 Byte. Bei einem äußeren Gitter der Größe  $100^3$  werden insgesamt  $(100^3 + 99^3) \times 44 = 86693156$  Byte, also rund 82 MB Speicher belegt.

Wenn ein Gitterpunkt nicht verschoben wird, ist seine Position implizit durch die Gitterkoordinaten gegeben. Für sehr große Gitter lässt sich daher der Speicherbedarf auf Kosten der Zugriffsgeschwindigkeit optimieren, indem die nicht an allen Gitterpunkten benötigten Positionen und Schnittpunktinformationen ausgelagert werden. Die `BccGridPoint`-Struktur enthält dann nur noch einen Verweis auf die bei Bedarf gespeicherte Information.

#### 4.2.4 Implementierung der Stempel-Tabelle

In [LS07] sind insgesamt zwölf verschiedene Stempel für die Tetraedisierung einer Gitterzelle entsprechend den verschiedenen Konfigurationen der vier beteiligten Gitterpunkte aufgeführt. Jeder Stempel kodiert ein bis drei Tetraeder, die entweder die Gitterpunkte selbst, oder die Schnittpunkte auf den Kanten verbinden. Durch Permutation der drei möglichen Vorzeichen der vier Eckpunkte ergeben sich  $3^4 = 81$  mögliche Konfigurationen von Eckpunkten, für die der Algorithmus eine Tetraedisierung ausgeben soll.

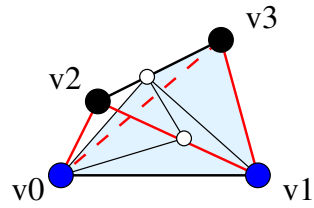
Die drei Vorzeichen +,-,0 lassen sich in jeweils 2 Bit kodieren, so dass durch einfache Bitoperationen ein Index in eine Tabelle für jede gegebene Vorzeichenkombination wie folgt durchgeführt werden kann:

```
enum sign { zero = 0x00, positive = 0x01, negative = 0x02 }

int tableIndex(sign s0, sign s1, sign s2, sign s3)
{
    return (s0 << 6) | (s1 << 4) | (s2 << 2) | s3;
}
```

Der Index besitzt somit 8 Bit, daher wird sinnvollerweise ein Array mit 256 Einträgen angelegt, von denen jedoch nur 81 Einträge belegt sind.

Jeder Tabelleneintrag kodiert eine variable Anzahl Tetraeder in Form von jeweils vier Punkten, die entweder die gegebenen vier Gitterpunkte oder Schnittpunkte entlang der Kanten darstellen. Wird jeder Eckpunkt durch ein Bit repräsentiert, können die Schnittpunkte als Bitmaske ausgedrückt werden. Ein Eintrag in der Tabelle für den Stempel Nr. 8 aus [LS07] mit zwei Tetraedern kann dann folgendermaßen angelegt werden:



```
enum vertex { v0 = 0x01, v1 = 0x02,
              v2 = 0x04, v3 = 0x08 };
table[tableIndex(zero, positive,
                 negative, positive)] =
{
    2, // Anzahl Tetraeder
    v0, v1 | v2, v2 | v3, v1, // #1
    v0, v3, v1, v2 | v3      // #2
}
```

Die zwölf angegebenen Stempel werden auf diese Weise handkodiert. Die übrigen 69 Tabelleneinträge können automatisch durch Permutierung dieser Einträge erzeugt werden.

### 4.3 Ergebnisse

Abbildung 4.5 zeigt eine Tetraedisierung des "schädel"-Modells in verschiedenen Auflösungen, hergestellt aus einem Distanzfeld der Auflösung  $256^3$ . Die exakten Distanzwerte werden nur in der Größenordnung der BCC-Gitterabstände um die Oberfläche herum verwendet. Im übrigen Volumen benötigt der Algorithmus nur das Vorzeichen der impliziten Funktion. Die Tetraeder, und somit auch die Dreiecke an der Oberfläche, besitzen die durch das Verfahren garantierten guten Winkleigenschaften und sind annähernd gleich groß.

Sichtbar wird ein generelles Problem bei der Rekonstruktion von dünnen Schichten aus dem Distanzfeld: Die initiale Abtastung erfolgt auf diskreten Gitterpunkten. Teile des Dichtefeldes, die kleiner als dieses Raster sind, werden bei der Rekonstruktion nicht erkannt. Dem kann nur durch eine Erhöhung der Abtastrate

abgeholfen werden, wodurch in unkritischen Regionen jedoch wesentlich mehr Tetraeder als notwendig erzeugt werden. Erst das rechte Modell in Abbildung 4.5 zeigt eine genügend feine Abtastung, die die dünneren Teile des Schädelknochens vollständig erfasst – bei annähernd 1,7 Millionen Tetraedern. Eine wesentliche Optimierung besteht daher in einer flexibleren Tetraederstruktur. Labelle *et al.* stellen eine adaptive Variante ihres Algorithmus vor, die mittels eines hierarchischen Octrees im Inneren des Volumens größere Tetraeder erzeugt, an der Oberfläche jedoch die reguläre, feinste Auflösung beibehält.

Die einfachen, regulären Datenstrukturen und der Marching Cubes–ähnliche Ansatz, die Tetraedisierung einer Zelle in einer Tabelle nachzuschlagen, sorgen für eine sehr hohe Ausführungsgeschwindigkeit. Die in Abbildung 4.5 dargestellten Tetraedisierungen wurden in nur 0,16s, 0,55s, und 2,42s berechnet. Diese Zeiten beinhalten auch die Auswertung der Dichtefunktion, wofür zwischen den Gitterpunkten der exakte Schnittpunkt mit der Fläche durch Bisektion ermittelt wird. Im diskreten Dichtefeld fällt zudem bei jeder Auswertung eine trilineare Interpolation zwischen den Voxeln an.

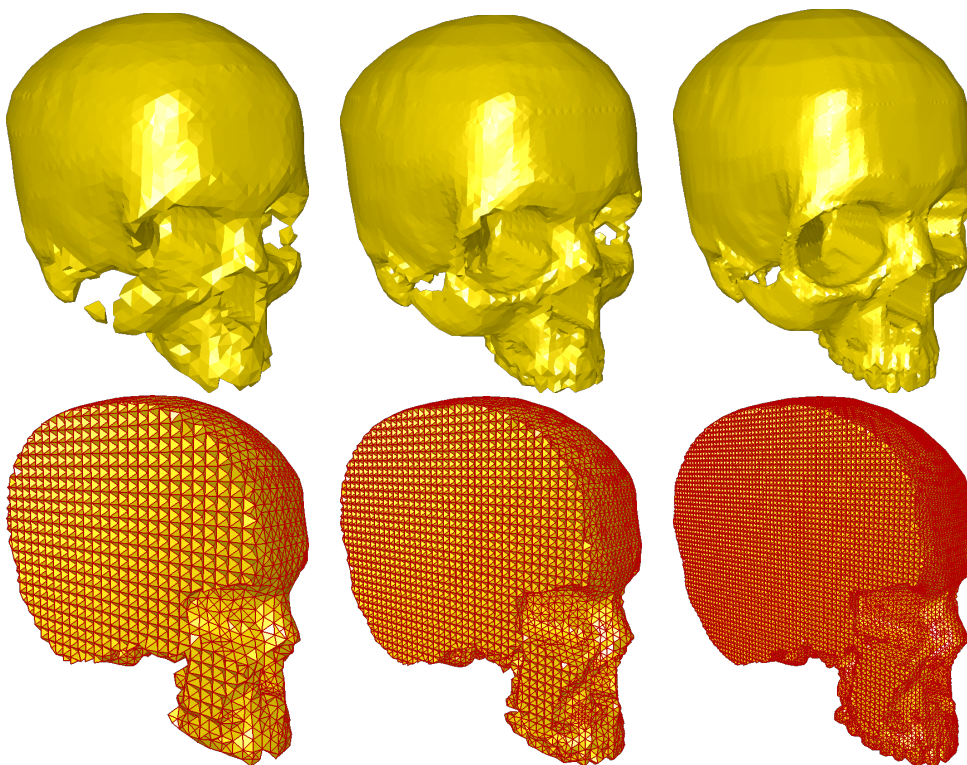


Abbildung 4.5.: Tetraedisierung des aus dem "schädel"-Modell erzeugten Distanzfeldes in drei Auflösungen. Von links nach rechts: 106392, 365976, und 1648902 Tetraeder.

Scharfe Objektkanten werden nicht gut rekonstruiert, worauf Labelle *et al.* auch hinweisen. Möglich erscheint eine Nachbearbeitung des tetraedischen Netzes durch Detektion scharfer Objektkanten und Neuordnung der Flächen zwischen benachbarten Tetraedern, wie in [KBSS01] für Dreiecksnetze demonstriert.

#### 4.4 Zusammenfassung

Die im vorigen Kapitel demonstrierten Schnitte von Objekten durch Differenzbildung der Distanzfelder können mit diesem Verfahren in eine tetraedische, und somit auch wieder in eine Dreiecksnetzform überführt werden. Insbesondere glatte Schnittflächen werden dabei gut rekonstruiert. Der Isosurface–Stuffing–Algorithmus lässt sich auch direkt zur Erzeugung von Triangulierungen einsetzen, wenn keine inneren Tetraeder erzeugt werden, und von den an der Oberfläche liegenden Tetraedern nur die Dreiecke ausgegeben werden, die als direkt auf der Fläche liegend markiert sind. Es ist auf diese Weise also möglich, in Oberflächenrepräsentation vorliegende Objekte in der Volumendarstellung auf einfache Weise zu manipulieren, und aus dem Ergebnis wieder eine Oberfläche zu erzeugen. Abbildung 4.6 zeigt exemplarisch diesen Ablauf.



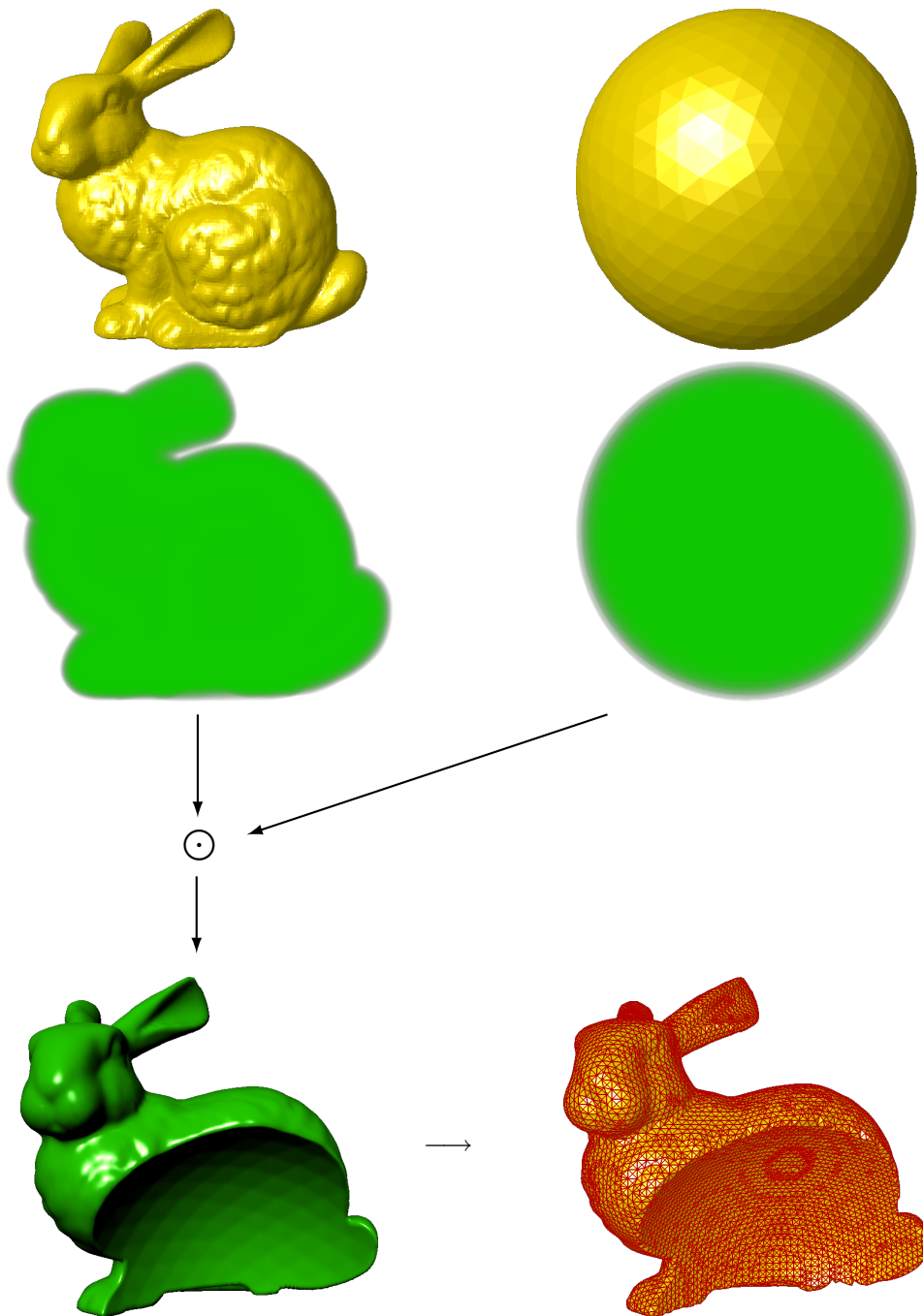


Abbildung 4.6.: Zwei Dreiecksnetze werden in Distanzfelder konvertiert, in dieser Form subtrahiert, und anschließend als Tetraedernetz rekonstruiert, aus dem wiederum leicht ein Dreiecksnetz gewonnen werden kann.



## 5 Zusammenfassung

In dieser Arbeit wurden drei fundamentale Volumenrepräsentationen gegenübergestellt. Datensätze auf Basis des strukturell sehr einfachen *regulären Voxelgitters* sind das Ausgangsmaterial vieler medizinischer Anwendungen wie Segmentierung und Visualisierung. Demonstriert wurde ein texturbasierter Visualisierungsalgorithmus, der eine Auswahl der betrachteten Regionen des Volumens über die Transferfunktion sowie das Ausblenden beliebig geformter Bereiche (Selektion) ermöglicht. Ein schnelles, GPU-basiertes Verfahren wurde implementiert, um geschlossene, orientierte Dreiecksnetze in die Form eines binären Voxelgitters zu bringen.

Wie sich zeigt, ist ein binäres Volumen für die Darstellung von Schnitten zweier Modelle suboptimal, da die diskretisierte Funktion diskontinuierlich ist. Die zweite grundlegende Volumendarstellung als implizite Funktion eröffnet neue Möglichkeiten. Ein Algorithmus zur Generierung eines *diskretisierten Abstandsfeldes* wurde implementiert, der gleichfalls die Rechenkapazitäten der GPU durch Parallelisierung der Abstandsberechnungen ausnutzt. Dieses Feld besitzt die selbe einfache Gitterstruktur wie das binäre Voxelgitter, hat jedoch einen kontinuierlichen Verlauf. Dies erlaubt eine bessere Darstellung durch die exaktere Rekonstruktion von Gradienten und Oberflächennormalen, sowie die Visualisierung verschiedener Offsetflächen der impliziten Funktion über die Wahl der Transferfunktion. Es wurde demonstriert, dass Schnitte zwischen solchen diskreten Distanzfeldern eine wesentlich getreue Wiedergabe der Schnittflächen erlauben.

Besonders für Simulationen mit FEM-Methoden relevant ist die Volumendarstellung als *Tetraedernetz*. Ein aktuelles, marching-cubes-ähnliches Verfahren wurde implementiert, dass aus einer als implizite Funktion gegebenen Geometrie eine Tetraedisierung erzeugt. Ein geschlossenes Tetraedernetz beinhaltet zugleich eine Triangulierung der Oberfläche, so dass hier eine Rückkonvertierung zu einer Oberflächenrepräsentation stattfinden kann.

Die gezeigte Konvertierbarkeit zwischen den Flächen- und Volumenrepräsentationen erlaubt daher die einfache Durchführung von volumetrischen Modellierungsoperationen oder komplexen Simulationen auf Objekten, die initial und im Ergebnis als Dreiecksnetze vorliegen sollen.

## 5.1 Ausblick

Vielfältige Verbesserungen der gezeigten grundlegenden Verfahrensweisen sind möglich und wünschenswert. Die zentralen Konflikte bei Volumenrepräsentationen treten in der Praxis zwischen den Dimensionen Speicherbedarf, Geschwindigkeit und Präzision auf. Die Diskretisierung in Form eines Voxelgitters ist extrem speicheraufwendig. Die maximale Dimension eines Volumenmodells wird im Wesentlichen durch den verfügbaren Hauptspeicher eines Rechners limitiert. Für möglichst hohe Detailtreue sind hohe Auflösungen hingegen wünschenswert. Die Ausführungsgeschwindigkeit von Operationen im Volumen ist wiederum in der Regel von der Anzahl der Volumenelemente abhängig, nimmt also mit zunehmender Auflösung drastisch ab.

Starre, reguläre Datenstrukturen bieten nicht genug Flexibilität, um das gewünschte Gleichgewicht zwischen diesen Parametern herzustellen und je nach Anwendung, dynamisch zu verändern. Einen vielversprechenden Ausweg verspricht daher die Verwendung adaptiver Strukturen, wie sie etwa in der Form von *adaptively sampled distance fields* [PF01] in jüngerer Zeit entwickelt wurden. Zukünftige und bereits heute beispielsweise im Visible Human Project [The] erzeugte, besonders große medizinische Datensätze, sind auf absehbare Zeit nur mit solchen adaptiven Methoden beherrschbar.

## Abbildungsverzeichnis

2.1.	2D–Schema des Ablaufs des Voxelisierungsalgorithmus. . . . .	7
2.2.	Die in dieser Arbeit verwendeten Polygonmodelle, sowie in der Auflösung $256^3$ erzeugte Voxelisierungen. . . . .	9
2.3.	Transferfunktion mittels 1D–Textur–Lookup. . . . .	14
2.4.	3D–Textur im (r,s,t)–Koordinatensystem. . . . .	15
2.5.	Drei texturierte Polygonstapel. . . . .	16
2.6.	Schnitt des Texturquaders mit einer Ebene senkrecht zur Sicht- richtung. . . . .	17
2.7.	Visualisierung eines Voxelmodells. . . . .	19
2.8.	Darstellung eines CT–Datensatzes mit unterschiedlicher Färbung in verschiedenen Wertebereichen der Transferfunktion. . . . .	19
2.9.	Interaktive Selektion von Ausschnitten. . . . .	22
2.10.	Aus einer diskretisierten Kugel wurde das ”armadillo”–Modell herausgeschnitten. . . . .	23
2.11.	Selektion mit einem voxelisierten Kugelmodell. . . . .	23
2.12.	Das voxelisierte ”schädel”–Modell in zwei weiteren Auflösungen. . . . .	24
3.1.	Konstruktion der Prismen für benachbarte Dreiecke. . . . .	28
3.2.	Lokales Koordinatensystem eines Dreiecks. . . . .	29
3.3.	Pseudocode für die Distanzberechnung. . . . .	30
3.4.	Fehlerhaftes Vorzeichen bei der Distanzberechnung. . . . .	31
3.5.	Orientierte Boundingbox für ein Dreieck. . . . .	32
3.6.	Darstellung der Isofläche $f = 0$ für Distanzfelder einiger Modelle. . . . .	34
3.7.	Darstellung verschiedener Isoflächen. . . . .	35
3.8.	Selektion auf Basis von Distanzfeldern. . . . .	37
4.1.	Zwei Tetraeder im BCC–Gitter. . . . .	40
4.2.	Abbildung der geometrischen Anordnung auf die Indexbeziehung zwischen äußeren und inneren Gitterpunkten. . . . .	43
4.3.	Zuordnung der Halbkanten in der BccGridPoint–Struktur. . . . .	44
4.4.	Zuordnung eines Parameterwertes auf einer diagonalen Halbkante. . . . .	45
4.5.	Tetraedisierung des aus dem ”schädel”–Modell erzeugten Distanz- feldes. . . . .	47
4.6.	Zwei Dreiecksnetze werden in Distanzfelder konvertiert, in dieser Form subtrahiert, und anschließend als Tetraedernetz rekonstruiert. . . . .	49
A.1.	Vereinfachte Darstellung der GPU–Pipeline. . . . .	63



## Tabellenverzeichnis

2.1. Zeiten für die Voxelisierung von Polygonmodellen. . . . .	8
3.1. Zeiten für die Erzeugung eines diskretisierten Distanzfeldes. . . .	36





## Literaturverzeichnis

- [AB03] H. Aanæs and J. A. Bærentzen. Pseudo-Normals for Signed Distance Computation. In *Proc. Conf. Vision Modeling and Visualization*, pages 407–413, 2003.
- [Bak89] T. J. Baker. Automatic mesh generation for complex three-dimensional regions using a constrained delaunay triangulation. *Engineering with Computers*, 5:161–175, 1989.
- [Bat82] K. J. Bathe. *Finite Element Procedures in Engineering Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [BBB<sup>+</sup>97] J. Bloomental, C. Bajaj, J. Blinn, M. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to implicit surfaces*. Morgan Kaufmann Publishers, 1997.
- [BFH<sup>+</sup>04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.
- [Blo88] J. Bloomental. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.
- [BMW01] D. E. Breen, S. Mauch, and R. T. Whitaker. 3D Metamorphosis Between Different Types of Geometric Models. *Computer Graphics Forum*, 20(3):36–48, 2001.
- [BNC99] M. Bro-Nielsen and S. Cotin. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. In *Proc. Eurographics '99*, pages 57–66, 1999.
- [Bor86] G. Borgefors. Distance Transformations in Digital Images. *Computer Vision, Graphics, and Image Processing*, 34:344–371, 1986.
- [BS89] R. E. Bank and L. R. Scott. On the Conditioning of Finite Element Equations with Highly Refined Meshes. *SIAM Journal on Numerical Analysis*, 26(6):1383–1394, 1989.

- [CBC<sup>+</sup>01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. Richard Fright, B. C. McCallum, and T. R. Evans. Reconstruction and Representation of 3D Objects With Radial Basis Functions. *Computer Graphics Proceedings, Annual Conference Series*, pages 67–76. ACM Press / ACM SIGGRAPH, August 2001.
- [CDM<sup>+</sup>02] B. Cutler, J. Dorsey, L. McMillan, M. Müller, and R. Jagnow. A Procedural Approach to Authoring Solid Models. *ACM Transactions on Graphics*, 21(3):302–311, July 2002.
- [CKY00] M. Chen, A. Kaufman, and R. Yagel, editors. *Volume graphics*. Springer Verlag, 2000.
- [COLS98] D. Cohen-Or, D. Levin, and A. Solomovici. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics*, 17(2):116–141, April 1998.
- [Cor] NVIDIA Corporation. offizielle CUDA Website. <http://developer.nvidia.com/object/cuda.html>.
- [DEL<sup>+</sup>99] J. Dorsey, A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. Modeling and rendering of weathered stone. In *Proc. ACM SIGGRAPH 1999*, Computer Graphics Proceedings, Annual Conference Series, pages 225–234. ACM Press / ACM SIGGRAPH, 1999.
- [ED08] Kenny Erleben and Henrik Dohmann. *Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra*, volume 3 of *GPU Gems*, pages 741–763. Addison–Wesley, 2008.
- [EHK<sup>+</sup>06] Klaus Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-time volume graphics*. A K Peters Ltd., 2006.
- [FC00] S. Fang and H. Chen. *Hardware accelerated voxelisation*, pages 301–315. Volume Graphics. Springer, 2000.
- [Fuc98] A. Fuchs. Automatic Grid Generation with Almost Regular Delaunay Tetrahedra. *Seventh International meshing Roundtable*, pages 133–148, 1998.
- [FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice, second edition*. Addison–Wesley, 1990.
- [GF05] Chris Guy and Dominic Ffytche. *An Introduction to The Principles of Medical Imaging*. Imperial College Press, revised edition, 2005.

- [GH97] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Conference Proceedings*, pages 209–216, 1997.
- [HCK<sup>+</sup>99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *SIGGRAPH 99 Conference Proceedings*, pages 277–285, August 1999.
- [JBS06] M. W. Jones, J. A. Bærentzen, and M. Sramek. 3D Distance Fields: A Survey of Techniques and Applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.
- [JS00] Mark W. Jones and R. Satherly. Voxelisation: Modelling for volume graphics. In *Proc. Vision Modeling and Visualization (VMV 2000)*, pages 319–326, 2000.
- [Kau87] A. Kaufman. Efficient Algorithms for 3D Scan–Conversion of Parametric Curves, Surfaces, and Volumes. In *Computer Graphics (Proc. ACM SIGGRAPH 87)*, pages 171–179, 1987.
- [KBSS01] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature Sensitive Surface Extraction from Volume Data. *Computer Graphics Proceedings, Annual Conference Series*, pages 57–66. ACM Press / ACM SIGGRAPH, August 2001.
- [LC87] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH 87 Conference Proceedings)*, volume 21, pages 163–170, 1987.
- [LH91] David Laur and Pat Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Computer Graphics (Proc. ACM SIGGRAPH 91)*, volume 25, pages 285–288, 1991.
- [Loh88] R. Lohner. Generation of three–dimensional unstructured grids by the advancing front method. *International Journal for Numerical Methods in Fluids*, 8:1135–1149, 1988.
- [LS07] François Labelle and Jonathan R. Shewchuck. Isosurface Stuffing: Fast Tetrahedral Meshes with Good Dihedral Angles. *ACM Transactions on Graphics*, 26(3):Article 57, July 2007.
- [Mau] S. Mauch. A Fast Algorithm for Computing the Closest Point and Distance Transform. <http://www.acm.caltech.edu/~seanm/software/cpt/cpt.pdf>.

- [MB05] Tom McReynolds and David Blythe. *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann Publishers, 2005.
- [MBTF03] N. Molino, R. Bridson, J. Teran, and R. Fedkiw. A crystalline, red green strategy for meshing highly deformable objects with tetrahedra. *Twelfth International Meshing Roundtable*, pages 103–114, 2003.
- [Mul03] J. C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*, 54(6), November 2003.
- [Nay99] D. J. Naylor. Filling Space with Tetrahedra. *International Journal for Numerical Methods in Engineering*, 44(10):1383–1395, 1999.
- [OH99] J. F. O’Brien and J. K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proc. ACM SIGGRAPH 1999*, Computer Graphics Proceedings, Annual Conference Series, pages 137–146. ACM Press / ACM SIGGRAPH, 1999.
- [OLG<sup>+</sup>07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
- [PF01] Ronald N. Perry and Sarah F. Frisken. Kizamu: A System for Sculpting Digital Characters. In *Proc. ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 47–56. ACM Press / ACM SIGGRAPH, 2001.
- [PSOP01] Bernhard Preim, Wolf Spindler, Karl J. Oldhafer, and Heinz-Otto Peitgen. 3D-Interaction Techniques for Planning of Oncologic Soft Tissue Operations. In *Proc. Graphics Interface 2001*, pages 183–190, 2001.
- [RP66] A. Rosenfeld and J. L. Pfaltz. Sequential Operations in Digital Picture Processing. *Journal of the ACM*, 13(4):471–494, 1966.
- [Set99] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, United Kingdom, 2nd edition, 1999.
- [Shi05] Peter Shirley. *Fundamentals of computer graphics*. A K Peters, Ltd., 2005.

- [SJ00] R. Satherly and M. W. Jones. Vector city vector distance transform. Technical report, University of Wales Swansea, February 2000.
- [SK99] M. Sramek and A. Kaufman. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):251–267, 1999.
- [SPG03] C. Sigg, R. Peikert, and M. Gross. Signed Distance Transform Using Graphics Hardware. In *Proceedings of Visualization 2003*, pages 83–90. IEEE Press, 2003.
- [The] The National Library of Medicine, Bethesda. The Visible Human Project. [http://www.nlm.nih.gov/research/visible/visible\\\_human.html](http://www.nlm.nih.gov/research/visible/visible\_human.html).
- [UK88] C. Upson and M. Keeler. V-buffer: Visible volume rendering. In *SIGGRAPH 88 Conference Proceedings*, pages 59–64, 1988.
- [WG91] Jane Wilhelms and Allen Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. In *Computer Graphics (Proc. ACM SIGGRAPH 91)*, volume 25, pages 275–284, 1991.
- [WK93] S. W. Wang and A. E. Kaufman. Volume sampled voxelization of geometric primitives. In *Proc. Visualization 93*, pages 78–84. IEEE CS Press, 1993.
- [WK94] S. Wang and A. Kaufman. Volume-Sampled 3D Modeling. *IEEE Computer Graphics and Applications*, 14(5):26–32, 1994.



## A GPU-Programmierung: Ein Überblick

Einige der im Text vorgestellten Algorithmen nutzen die für Grafikkarten verfügbaren Fähigkeiten der parallelen Datenverarbeitung mittels spezialisierter, *Shader* genannter Programme. Dieser Anhang soll die verwendeten Begriffe klären, um das Verständnis der entsprechenden Abschnitte zu erleichtern. Für eine detailliertere Einführung in diese Techniken mit Bezug auf Volumengrafik sei auf [EHK<sup>+</sup>06] verwiesen.

Die Visualisierung dreidimensionaler Grafiken basiert heute überwiegend auf der Projektion und Rasterisierung planarer Polygone, in aller Regel Dreiecke. Komplexe Szenarien umfassen oft mehrere Hunderttausend Polygone, die mit möglichst hohem Durchsatz transformiert, beleuchtet, und in den Bildspeicher geschrieben werden sollen.

Ursprünglich komplett in Software implementiert, ist die Unterstützung dieses Prozesses durch spezialisierte Grafikkarten-Hardware mittlerweile immens fortgeschritten. Deren Prozessor, die GPU (*Graphics Processing Unit*), sorgt zunächst nur für die Berechnung von Beleuchtung und Transformation der Polygoneckpunkte, sowie die Rasterisierung (Scankonvertierung) der auf die Bildebene projizierten Polygone. In der Visualisierungs-Pipeline übernimmt die Hardware die Bearbeitung an zwei Stellen: Der Bearbeitung und Transformation von Polygonpunkten (*vertex processing*), sowie der Darstellung interpolierter Attribute wie Farben und Texturen (*fragment processing*). *Fragment* ist ein OpenGL-spezifischer Ausdruck; ein Fragment ist praktisch mit einem Pixel in einem rasterisierten Polygon identisch.

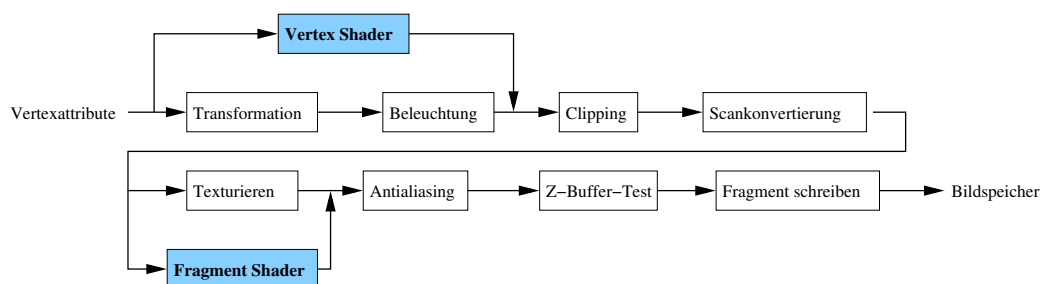


Abbildung A.1.: Vereinfachte Darstellung der GPU-Pipeline: Die obere Zeile beschreibt die Verarbeitung der Polygoneckpunkte (*vertex processing*). Die untere Zeile zeigt die Operationen auf Pixelebene (*fragment processing*). Die unterlegten alternativen Pfade sind bei aktiviertem Vertex Shader bzw. Fragment Shader aktiv.

Diese starren Funktionsblöcke wurden im Laufe der Zeit durch bis heute zunehmend leistungsfähigere, programmierbare Komponenten abgelöst: den sogenannten *Vertex Shader*– beziehungsweise *Fragment Shader*–Einheiten (der vor allem aus der Direct3D-API bekannte Begriff *Pixel Shader* ist synonym). Abbildung A.1 zeigt die Einordnung der Shader in die Pipeline. Nach anfänglich sehr begrenzten Möglichkeiten der Programmierung einfacher Matrix- und Vektoroperationen in einem Assemblerdialekt, ist heute die Entwicklung von sehr komplexen Programmen in C-ähnlichen Hochsprachen mit den üblichen Datenstrukturen, Kontrollflusskonstrukten, Unterfunktionen, etc. möglich. Ein einfacher Vertex Shader in der mit OpenGL spezifizierten Shadersprache GLSL (*GL Shading Language*) sieht folgendermaßen aus:

```
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor = gl_Color;
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Die Eingabe an einen Vertex Shader ist ein Eckpunkt (*Vertex*) eines Polygons, mit einer konfigurierbaren Zahl von Attributen versehen. Im Beispiel wird der Punkt in die Bildebene projiziert, und zusammen mit dem zugeordneten Farbwert und Texturkoordinaten an die nächste Pipelinestufe übergeben.

Ein Fragment Shader wird für jeden zu füllenden Bildpunkt des rasterisierten Polygons mit den zwischen den Eckpunkten linear interpolierten Vertex-Attributen aufgerufen. Ein einfaches Beispiel, das einen Texturzugriff an den interpolierten Vertex-Texturkoordinaten ausführt, und mit dem gleichfalls interpolierten Farbwert kombiniert, zeigt das folgende Codefragment:

```
uniform sampler2D texture;

void main(void)
{
    vec4 texel = texture2D(texture, gl_TexCoords[0]);
    gl_FragColor = texel * gl_Color;
}
```

Diese Programme werden auf der CPU übersetzt, und über OpenGL (oder eine andere API wie Direct3D) an die GPU übertragen und aktiviert.



Da Shaderprogramme prinzipiell beliebige Berechnungen ausführen können und auf massiv datenparallele Ausführung spezialisiert sind (sogenannte SIMD-Architektur, von *Single Instruction Multiple Data*; der Begriff *stream processing* ist gleichfalls geläufig), sind sie auch für die Berechnung ganz allgemeiner Problemstellungen hochinteressant. Entscheidend bei der Formulierung eines Algorithmus für die GPU ist die Datenübergabe an die Shaderprogramme, da diese ja keinen Zugriff auf den Hauptspeicher der CPU haben. Die grundlegenden diesbezüglichen Techniken sind:

- Konstanten (auch Arrays oder Records), können direkt in GPU-Register übertragen werden.
- Dynamisch variierende Werte werden als Vertexattribute definiert, die mit einer Polygondefinition von dem auf der CPU ausgeführten Programmen kommen, oder auch erst im Vertex Shader berechnet werden.
- Im Fragment Shader werden aus diesen Attributen, die von der Hardware linear interpoliert werden können, Ergebnisse berechnet. Diese werden im Bildspeicher abgelegt, und können dort wieder von der CPU ausgelesen werden.
- Zum tabellenartigen Zugriff auf größere externe Datenmengen dienen ein- bis dreidimensionale Texturen, auf die im Fragment Shader zugegriffen werden kann. Hier wird oft die in der Texturhardware eingebaute lineare Interpolation genutzt.
- Indirektion ist möglich, indem aus einer Textur Werte ausgelesen werden, die wiederum als Texturkoordinaten für den Zugriff auf eine weitere Textur verwendet werden.

Je mehr sich die Algorithmen von der Berechnung rein grafischer Ausgaben entfernen, desto mehr erscheinen diese Techniken als Umweg. Wünschenswert wäre eine allgemein gehaltene Programmierschnittstelle, die die Formulierung datenparalleler Algorithmen unterstützt. Es hat sich hier der Begriff GPGPU (*General Purpose GPU*) etabliert [OLG<sup>+</sup>07]. Mit CUDA [Cor] und Brook [BFH<sup>+</sup>04] befinden sich derartige Frameworks in der Entwicklung.