Volumenvisualisierung auf dünnen Gittern

Diplomarbeit im Fach Informatik

vorgelegt von

Matthias Hopf

geb. am 27. September 1971 in München

angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Dipl.-Math. Christian Teitzel
Betreuender Hochschullehrer: Prof. Dr. Thomas Ertl

Beginn der Arbeit: 04. Februar 1998

Abgabe der Arbeit: 15. Juli 1998

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Aufgabenstellung . . . 2
1.3 Verwandte Arbeiten . . . 3

2 Dünne Gitter **4**
2.1 Mathematische Grundlagen . . . 4
2.2 Aufbau dünner Gitter . . . 8
2.3 Kombinationsmethode . . . 10
2.4 Beschleunigungsansätze für dünne Gitter . . . 16
2.5 Aufwandsanalyse . . . 17

3 Volumenvisualisierung **21**
3.1 Visualisierungsmethoden . . . 21
3.2 Grundlagen des Raycasting . . . 22
3.3 Integrationsmethoden . . . 22
3.3.1 Röntgen . . . 23
3.3.2 MIP . . . 23
3.3.3 Emissions-Absorptionsmodell . . . 23
3.4 Transferfunktionen und Beleuchtungsberechnung . . . 25

4 Implementierung **26**
4.1 Voraussetzungen . . . 26
4.2 Klassenhierarchie . . . 27
4.3 User-Interface . . . 28
4.4 Programmgerüst . . . 34
4.4.1 Grafische Benutzeroberfläche . . . 34
4.4.2 Repräsentation der Eingabedaten . . . 35
4.4.3 Sichtbestimmung und Berechnung der Raycast-Parameter . . . 35
4.4.4 Hilfesystem . . . 37
4.5 Raycaster . . . 38
4.5.1 Implementierungsvarianten . . . 38
4.5.2 Integrationsmethoden und Beleuchtungsberechnung . . . 40

4.6 Dünne Gitter 41
4.6.1 Standardmethode 41
4.6.2 Kombinationsmethode 42

5 Einblick in die Hardware-Beschleunigung der Kombinationsmethode 44
5.1 Funktionsprinzip 44
5.2 Benötigte OpenGL-Funktionen und -Erweiterungen 46
5.3 Laden der Texturen 47
5.4 Interpolation innerhalb eines dünnen Gitters 48
5.5 Fehlerabschätzung 48

6 Ergebnisse 52
6.1 Voraussetzungen für die Anwendbarkeit der Algorithmen 52
6.2 Vergleich mit Wavelet-basierten Kompressionsverfahren 56
6.3 Vergleich des Speicherbedarfs 58
6.4 Vergleich der benötigten Zeit 58
6.5 Beispiele 60

7 Zusammenfassung und Ausblick 66
7.1 Zusammenfassung der Arbeit 66
7.2 Mögliche Erweiterungen und Verbesserungen 67
7.3 Zukunft der dünnen Gitter in der Visualisierung 68

Literaturverzeichnis 70

Kapitel 1

Einleitung

1.1 Motivation

Viele wissenschaftliche Datensätze, die aus numerischen Simulationen oder Messungen physikalischer Vorgänge resultieren, passen aufgrund ihrer Größe nicht mehr vollständig in den Arbeitsspeicher heutiger Workstations. Daher können diese Daten häufig nicht mehr direkt visualisiert werden und müssen in einem Vorverarbeitungsschritt in der Regel verlustbehaftet komprimiert werden.

Aus skalaren dreidimensionalen Daten kann zum Beispiel für die Darstellung von ISO-Flächen über einen Marching-Cube-Algorithmus eine Geometriebeschreibung extrahiert werden, die während der Darstellung wesentlich weniger Speicher benötigt. Ebenso lassen sich aus den Daten von Strömungssimulationen Teilchenbahnen errechnen, die als Strömungsbänder visualisiert werden können.

Diese Ansätze haben alle gemeinsam, daß für die Extraktion der Visualisierungsdaten potentiell auf den gesamten Datensatz zugegriffen werden muß. Ist der Hauptspeicher einer Workstation für diesen Datensatz zu klein, kann die Extraktion nicht interaktiv erfolgen und hat damit eine dramatische Verschlechterung der Produktivität zur Folge. Speziell die dreidimensionale Struktur der Daten läßt sich erst durch interaktive Volumenvisualisierung gut erkennen, vor allem wenn der Benutzer die Ansicht mit hohen Bildwiederholraten ändern und dadurch um das Volumen „fliegen“ kann. Ein weiterer Nachteil dieser Ansätze ist, daß nur die durch die Extraktion gewonnen Merkmale visualisiert werden und daher nicht der volle Informationsgehalt des Datensatzes ausgenutzt wird.

Um nicht auf die extrahierten Merkmale festgelegt zu sein, müssen Visualisierungstechniken verwendet werden, die auf dem gesamten Datensatz arbeiten, wie zum Beispiel die direkte Volumenvisualisierung. Mit der direkten Darstellung des gesamten Volumens kann der volle Detailreichtum der Daten ausgenutzt werden. Durch das Darstellungsprinzip kann auch ein weites Spektrum an Visualisierungsverfahren genutzt werden, sofern diese durch effiziente Transferfunktionen und Integrationsmethoden beschrieben werden können. Die direkte

Volumenvisualisierung erlaubt es, sowohl ISO-Flächen darzustellen, als auch Bilder semitransparenter Volumen, die zum Beispiel auf dem Emissions-Absorptionsmodell beruhen.

Diese Gruppe von Visualisierungstechniken hat aber den Nachteil, auf den gesamten Datensatz zugreifen zu müssen. Da dieser, wie bereits erwähnt, häufig größer als der zur Verfügung stehende Arbeitsspeicher ist, muß er komprimiert werden. Derzeit viel diskutierte Verfahren, die Datenmengen verlustbehaftet zu reduzieren, sind Darstellungen mittels Wavelets mit anschließender Komprimierung sowie die Dünngitter-Technik.

Während sich bei Wavelets Qualität und Datenmenge der Kompression in weiten Bereichen frei wählen lassen, sind diese Parameter bei der Methode der dünnen Gitter fest. Einige spezielle Eigenschaften der dünnen Gitter machen diese aber gerade für die interaktive Visualisierung interessant, da die Dateninterpolation durch ihren prinzipiellen Aufbau mittels spezieller Algorithmen und die Ausnutzung von 3D-Hardware stark beschleunigt werden kann. Nicht zu unterschätzen ist auch die Tatsache, daß viele numerische Simulationen heutzutage selbst an die Speichergrenzen von Supercomputern stoßen und die Rechenzeit auf diesen Maschinen auch heute noch teuer ist. Immer häufiger werden deshalb schon in der numerischen Berechnung der Simulationsdaten komprimierte Darstellungen verwendet. Gerade die Dünngitter-Technik wird beispielsweise in [2, 5, 6] hierfür eingesetzt.

Um bereits auf dünnen Gittern berechneten Datensätze zu visualisieren, werden die Daten bis jetzt noch regelmäßig in volle, das heißt unkomprimierte Datensätze umgewandelt und anschließend mit Standardmethoden visualisiert. Mit diesem Ansatz bleibt man bei der Visualisierung auf Datensätze beschränkt, deren Größe die Verwendung von Workstations noch zuläßt. Durch die direkte Visualisierung komprimierter Daten würde diese Beschränkung wegfallen. Allerdings muß die ungleich kompliziertere Interpolation auf dünnen Gittern so beschleunigt werden, daß weiterhin interaktives Arbeiten wie mit den Standardmethoden möglich ist.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit ist die erstmalige Implementierung von Volumenvisualisierungstechniken für Daten auf dünnen Gittern. Die Volumenvisualisierung an sich ist dabei keine neue Visualisierungsmethode, allerdings existieren bis jetzt noch keine Ansätze, für die hier nötige Interpolation dünne Gitter einzusetzen.

Die Interpolation auf diesen Gittern ist dabei deutlich aufwendiger als auf vollen Gittern, deshalb sollen verschiedene Ansätze zur Beschleunigung der Interpolation implementiert und getestet werden. Da für eine interaktive Visualisierung kurze Berechnungszeiten erwünscht sind, bildet die Beschleunigung der Interpolation durch den Einsatz von Grafikhardware einen wesentlichen Schwerpunkt der Arbeit. Neben den Interpolationszeiten sollen insbesondere die durch die Beschleunigung bedingten Fehler untersucht werden.

Um den Einfluß der Beleuchtung auf die Visualisierung deutlich zu machen, sind verschiedene Beleuchtungsmodelle zu implementieren und dem Benutzer über eine interaktive Benutzerschnittstelle zugänglich zu machen.

1.3 Verwandte Arbeiten

Neben der Volumenvisualisierung, die in dieser Arbeit behandelt wird, ist die Darstellung sogenannter Strömungsbänder eine weitere Standardmethode der Visualisierung dreidimensionaler Daten der Strömungsmechanik. Parallel zu dieser Diplomarbeit entstand durch Christian Teitzel in [17], basierend auf den gleichen Dünngitter-Klassen, eine Implementierung dieser Visualisierungsmethode.

Auch ISO-Flächen gehören zu den Standard-Visualisierungstechniken. Die erstmalige Implementierung dieser Technik direkt auf dünnen Gittern in [9] verwendet für die Interpolation spezielle Caching-Algorithmen, um die Berechnung zu beschleunigen.

Auszüge dieser Arbeit können in den Technical Reports [18] und [19] eingesehen werden.

Kapitel 2

Dünne Gitter

Die grundsätzlichen Ideen für die dünnen Gitter wurden in den 60er Jahren von Babenko [1] und Smolyak [15] entwickelt. In ihren Abhandlungen zeigen sie, daß eine bestimmte Tensorprodukt-Technik zur Konstruktion von Approximations-Operatoren bis auf logarithmische Faktoren optimale Fehlerraten liefert.

1990 wurde die Dünngitter-Methode durch Zenger [21] erstmals für die Lösung diskreter partieller Differentialgleichungssysteme praktisch eingesetzt. Mit ihrer Hilfe kann die Anzahl der Unbekannten und damit auch die Lösungszeit reduziert werden. Durch die spezielle Interpolationstechnik auf dünnen Gittern erreicht man so eine Auflösung in der Lösung des Problems, die auf herkömmlichen Weg nur über wesentlich feinere Gitter mit entsprechend hohem Lösungsaufwand erreicht werden kann.

2.1 Mathematische Grundlagen

Die hier beschriebenen Grundlagen für dünne Gitter orientieren sich in der Schreibweise und im Inhalt stark an der Standardliteratur. Insbesondere kann in [2] die leicht vereinfachte Rechnungsführung für auf dem Rand verschwindende Funktionen in $\hat{L}_{n,\ldots,n}$ oder in [18] speziell auf den dreidimensionalen Fall angepaßt gefunden werden.

X sei der Raum aller glatten und gleichzeitig in Richtung jeder Koordinatenachse zweimal stetig differenzierbaren Abbildungen auf dem Einheitswürfel und f eine Abbildung aus X:

$$X := \left\{ f : [0,1]^d \to \mathbf{R}^m \;\middle|\; \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \in C^0([0,1]^d),\ i = 1, \ldots, m \right\}, \quad f \in X \tag{2.1}$$

Sei weiterhin $G_{j_1,\ldots,j_d},\ j_i \geq 0$ ein uniformes Gitter auf dem Einheitswürfel mit den Schrittweiten $h_{j_i} = 2^{-j_i},\ i = 1, \ldots, d$ und demnach $\prod_{i=1}^d (2^{j_i} + 1)$ Gitterpunkten. Auf diesen Gittern definieren wir eine Teilordnung: $G_{j_1,\ldots,j_d}$ heißt eine *Verfeinerung* von $G_{k_1,\ldots,k_d}$ oder kurz feiner als $G_{k_1,\ldots,k_d}$ dann und nur dann, wenn $0 \leq k_i \leq j_i,\ i = 1, \ldots, d$ und $\sum_{i=1}^d k_i < \sum_{i=1}^d j_i$. Analog heißt in diesem Fall $G_{k_1,\ldots,k_d}$ gröber als $G_{j_1,\ldots,j_d}$.

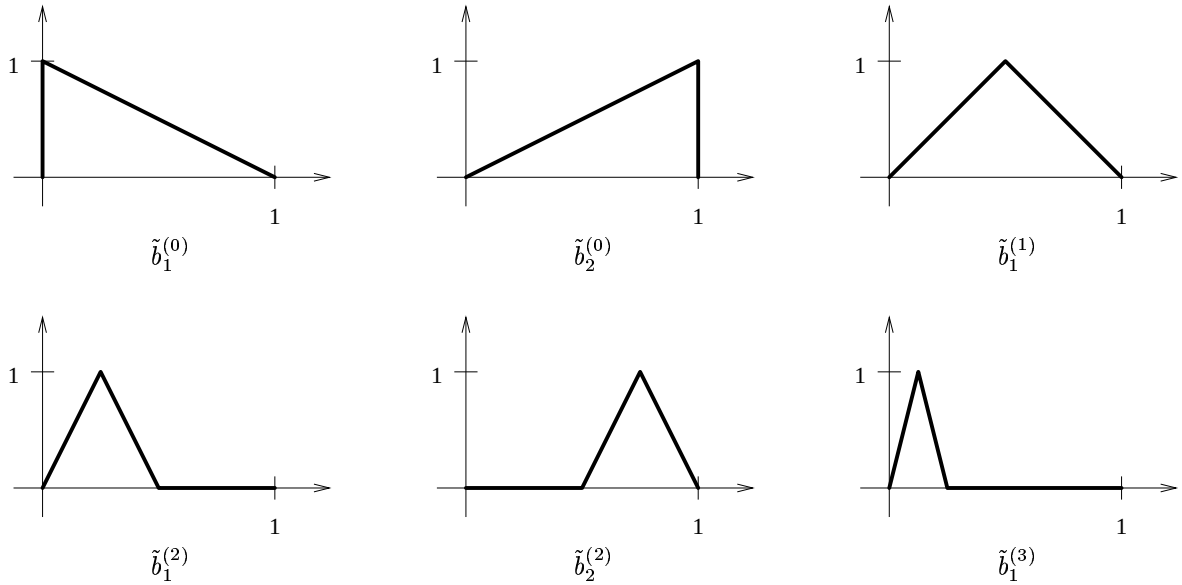


Abbildung 2.1: Beispiele von eindimensionalen Basisfunktionen

Betrachten wir den Abbildungsraum $\hat{L}_{n,\ldots,n}$ aller stetigen, m-wertigen und auf $G_{n,\ldots,n}$ stückweise d-linearen Abbildungen, sowie die Teilräume $S_{j_1,\ldots,j_d} \subseteq \hat{L}_{n,\ldots,n},\ 0 \leq j_i \leq n,\ i = 1,\ldots,d$ aller stetigen, m-wertigen und auf $G_{j_1,\ldots,j_d}$ stückweise d-linearen Abbildungen, die auf allen Gitterpunkten der gröberen Gitter verschwinden. Für $j_i = 0$ entfällt diese Bedingung, so daß insbesondere $S_{0,\ldots,0} \equiv \hat{L}_{0,\ldots,0}$ ist. Die oben definierte Teilordnung auf den Gittern induziert in natürlicher Weise eine Hierarchie auf den Teilräumen und es folgt

$$\hat{L}_{n,\ldots,n} \;=\; \bigoplus_{j_1=0}^{n} \cdots \bigoplus_{j_d=0}^{n} S_{j_1,\ldots,j_d} \quad .$$

In dieser Zerlegung des Abbildungsraumes $\hat{L}_{n,\ldots,n}$ benutzen wir nun stückweise d-lineare Finite Elemente als Basisfunktionen. Die Basis für die Teilräume $S_{j_1,\ldots,j_d}$ mit mindestens einem $j_i = 0$ muß gesondert behandelt werden, um die Abbildungen in $\hat{L}_{n,\ldots,n}$ nicht auszuschließen, die auf dem Rand nicht verschwinden. Abbildung 2.1 verdeutlicht den Aufbau der Basisfunktionen an mehreren eindimensionalen Beispielen, Abbildung 2.2 die Träger und Maxima der Basisfunktionen auf einer Zerlegung des $G_{4,4}$.

$$\tilde{b}_{k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}(x_1,\ldots,x_d) := \prod_{i=1}^{d} w_{j_i}(x_i - m_{k_i}^{(j_i)}) \ , \quad k_i = \begin{cases} 1,2 & \text{für } j_i = 0 \\ 1,\ldots,2^{j_i-1} & \text{für } j_i = 1,\ldots,n \end{cases} , \tag{2.2}$$

$$x_1,\ldots,x_d \in [0,1] \quad ,$$

$$\text{mit} \quad m_k^{(i)} = h_i \cdot \begin{cases} k-1 & \text{für } i = 0 \\ 2k-1 & \text{für } i = 1,\ldots,n \end{cases} \quad , \quad h_i = 2^{-i} \quad ,$$

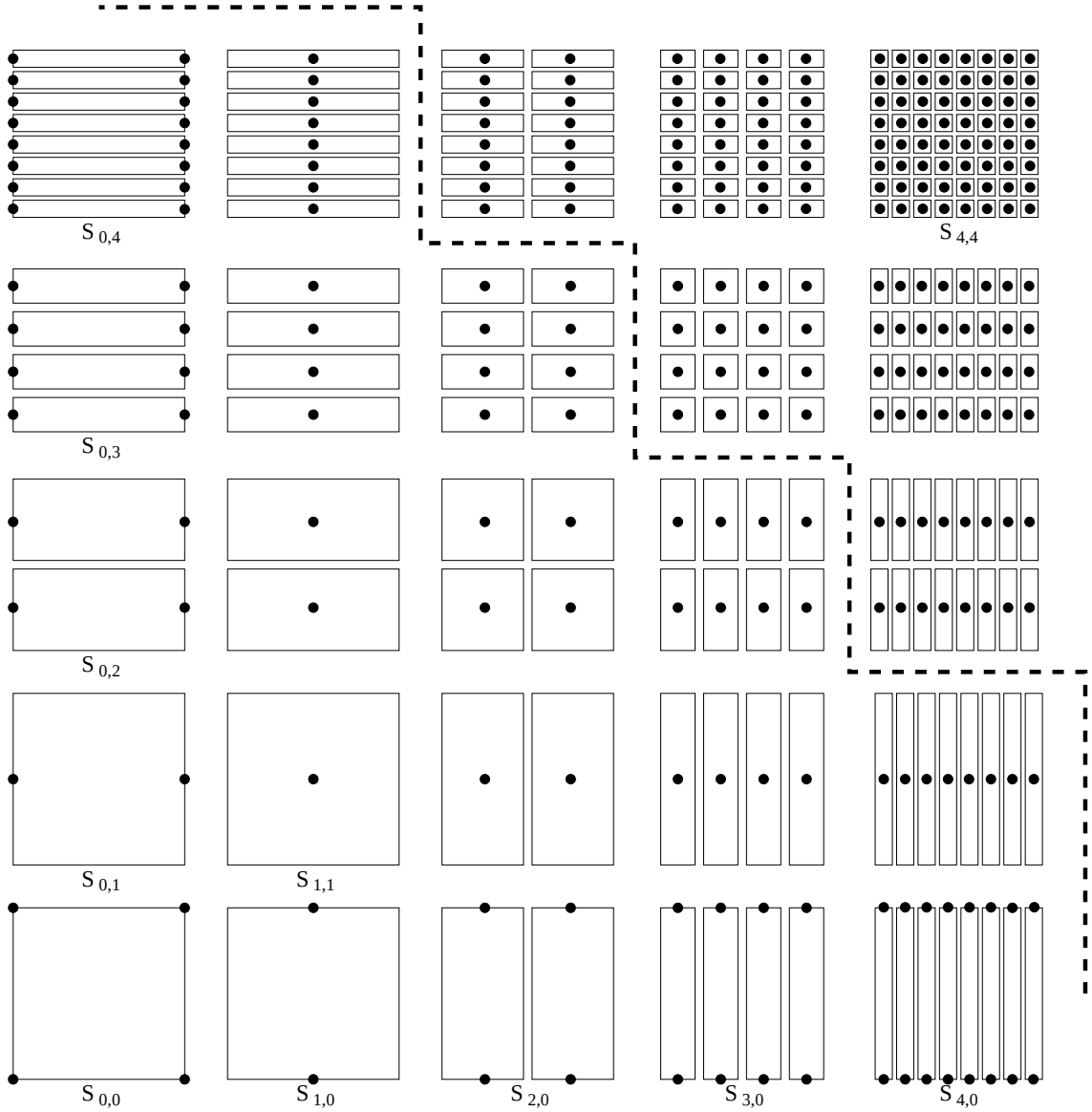


Abbildung 2.2: Die Teilräume S_{j_1,j_2} von $G_{4,4}$

$$\text{und} \quad w_i(x) = \begin{cases} \frac{h_i+x}{h_i} & \text{für} \quad -h_i \leq x \leq 0 \\ \frac{h_i-x}{h_i} & \text{für} \quad 0 \leq x \leq h_i \\ 0 & \text{sonst} \end{cases} . \tag{2.3}$$

Wir betrachten nun die zu der Abbildung f gehörende Interpolationsfunktion

$$\hat{f}_i^n(x_1, \ldots, x_d) = \sum_{j_1=0}^{n} \cdots \sum_{j_d=0}^{n} f_i^{(j_1,\ldots,j_d)}(x_1, \ldots, x_d) \ , \quad i = 1, \ldots, m \ , \tag{2.4}$$

$$f_i^{(j_1,\ldots,j_d)}(x_1, \ldots, x_d) = \sum_{k_1} \cdots \sum_{k_d} c_{i,k_1,\ldots,k_d}^{(j_1,\ldots,j_d)} \cdot \tilde{b}_{k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}(x_1, \ldots, x_d) \quad \text{mit } k_d \text{ aus (2.2).} \tag{2.5}$$

Die Werte $c_{i,k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}$ sind die sogenannten Überschußkoeffizienten zu $\hat{f}^n$ und bilden durch Linearkombinationen mit den Basisfunktionen aus (2.2) die Beiträge (2.5) der entsprechenden Unterräume $S_{j_1,\ldots,j_d}$.

Die Interpolationsfunktion (2.4) ist durch die Grenzwertbildung

$$\hat{f}_i = \lim_{n\to\infty} \hat{f}_i^n := f_i, \quad i = 1, \ldots, m$$

eindeutig bestimmt. Die Grenzwertbildung ist insofern zulässig, als $\hat{L}_{n,\ldots,n} \subset \hat{L}_{n+1,\ldots,n+1}$ gilt und $\lim\limits_{n\to\infty} \hat{L}_{n,\ldots,n}$ in X dicht liegt.

Durch die Grenzwertbildung müssen die Interpolanten $\hat{f}^n$ insbesondere an allen Gitterpunkten des $G_{n,\ldots,n}$ den Wert der Abbildung f annehmen. Dadurch sind in einfacher Weise die Überschußkoeffizienten $c_{i,k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}$ eindeutig bestimmt.

Durch die lineare Interpolation nur endlich vieler Basen ergibt sich zwischen der betrachteten Abbildung f und der dazugehörigen Interpolationsfunktion $\hat{f}^n$ ein Fehler, der nach [2, Seiten 10ff] in L^2 und in L^∞ wie folgt abgeschätzt werden kann:

$$\left\| f_i - \hat{f}_i^n \right\|_2 \ \leq \ \frac{d}{9^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_2 \cdot h_n^2 \ = \ O(h_n^2) \ , \quad i = 1, \cdots, m \ , \tag{2.6}$$

$$\left\| f_i - \hat{f}_i^n \right\|_\infty \ \leq \ \frac{d}{6^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_\infty \cdot h_n^2 \ = \ O(h_n^2) \ , \quad i = 1, \cdots, m \ , \tag{2.7}$$

Des weiteren können die maximalen Beiträge $f^{(j_1,\ldots,j_d)}$ der Unterräume $\mathrm{S}_{j_1,\ldots,j_d}$ abgeschätzt werden, was im folgenden Kapitel ausschlaggebend für die Einführung der dünnen Gitter ist.

2.2 Aufbau dünner Gitter

Nach [2, Seite 11] gilt für die Beiträge (2.5) der Unterräume $S_{j_1,\ldots,j_d}$

$$\left\| f_i^{(j_1,\ldots,j_d)} \right\|_2 \;\leq\; \frac{1}{3^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_2 \cdot h_{j_1}^2 \cdots h_{j_d}^2 \;, \quad i = 1, \cdots, m \;,$$

$$\left\| f_i^{(j_1,\ldots,j_d)} \right\|_\infty \;\leq\; \frac{1}{2^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_\infty \cdot h_{j_1}^2 \cdots h_{j_d}^2 \;, \quad i = 1, \cdots, m \;.$$

Nach dieser Abschätzung haben alle Unterräume $S_{j_1,\ldots,j_d}$ mit $(j_1,\ldots,j_d) \in \mathcal{I}_1(n)$ einen Beitrag der gleichen Größenordnung zu $\hat{f}^n$. Dabei bezeichnet n den sogenannten *Level* des dünnen Gitters, und $\mathcal{I}_1(n)$ ist definiert als

$$\mathcal{I}_1(n) := \left\{ (j_1,\ldots,j_d) \;\middle|\; \sum_{k=1}^{d} j_k = n + d - 1 = \text{const}, \quad j_k \geq 1 \right\} \quad \text{für} \quad n \geq 1 \;.$$

Werden auch auf dem Rand nicht verschwindende Abbildungen betrachtet, ergibt sich die Definition als

$$\mathcal{I}(n) := \left\{ (j_1,\ldots,j_d) \;\middle|\; \sum_{k=1}^{d} \max(1, j_k) = n + d - 1 = \text{const}, \; j_k \geq 0 \right\} \text{ für } \; n \geq 1 \;.$$

Dieser etwas kompliziertere Ansatz ist damit begründet, daß sich so auf natürliche Weise die Gitterpunkte am Rand und die Interpolationswerte in der gleichen Art ergeben, wie sie uns in Kapitel 2.3 bei der Kombinationsmethode begegnen werden.

Neben dem Beitrag der gleichen Größenordnung zu $\hat{f}^n$ haben diese Unterräume zumindest für den randlosen Fall auch die gleiche Dimension, das heißt die gleiche Zahl an Basisfunktionen und somit Überschußkoeffizienten. Die Anzahl der Koeffizienten ist eine natürliche Größe für den benötigten Speicherplatz. Also macht es Sinn, für die Interpolation nur den im folgenden *Dünngitter-Raum* genannten Raum $\tilde{L}_n$ zu betrachten:

$$\tilde{L}_n \;:=\; \bigoplus_{(j_1,\ldots,j_d) \in \mathcal{J}(n)} S_{j_1,\ldots,j_d}$$

mit der Indexmenge

$$\begin{aligned} \mathcal{J}(n) \;&:=\; \bigcup_{i=1,\ldots,n} \mathcal{I}(i) \\ &=\; \left\{ (j_1,\ldots,j_d) \;\middle|\; \sum_{k=1}^{d} \max(1, j_k) \leq n + d - 1 = \text{const}, \; j_k \geq 0 \right\} \quad . \end{aligned} \tag{2.8}$$

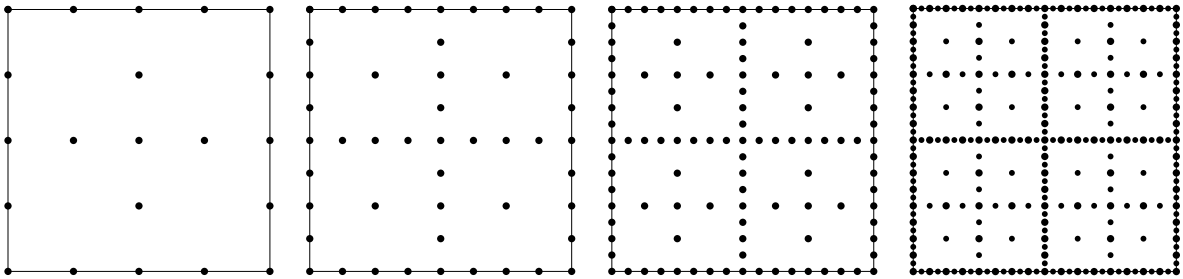


Abbildung 2.3: Dünne Gitter der Dimension 2, Level 2 bis 5

Für die Interpolationsfunktion $\tilde{f}^n \in \tilde{L}_n$ ergibt sich demnach

$$\tilde{f}_i^n := \sum_{(j_1,\ldots,j_d)\in\mathcal{J}(n)} f_i^{(j_1,\ldots,j_d)} \ , \quad i = 1,\ldots,m \tag{2.9}$$

In Abbildung 2.2 sind alle Teilräume, die in $\tilde{L}_4$ enthalten sind, mit einer gestrichelten Linie von den restlichen Teilräumen abgetrennt.

Überträgt man die Definition des Dünngitter-Raumes auf die für die der Gitter $G_{j_1,\ldots,j_d}$ zugrundeliegende Definition, ergeben sich durch das Fehlen der zu den weggelassenen Unterräumen gehörigen Gitterpunkte die sogenannten *dünnen Gitter* $\tilde{G}_n$. Abbildung 2.3 zeigt Gitter mehrerer unterschiedlicher Level für $d = 2$.

Durch das Weglassen mehrerer Unterräume verschlechtert sich der Interpolationsfehler laut [2, Seite 23] auf

$$\left\| f_i - \tilde{f}_i^n \right\|_2 \leq c(n) \cdot \frac{h_n^2}{9^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_2 = O\left(h_n^2 (\log_2 h_n^{-1})^{d-1} \right) \ , \ i = 1,\cdots,m \ , \tag{2.10}$$

$$\left\| f_i - \tilde{f}_i^n \right\|_\infty \leq c(n) \cdot \frac{h_n^2}{6^d} \left\| \frac{\partial^{2d} f_i}{\partial x_1^2 \cdots \partial x_d^2} \right\|_\infty = O\left(h_n^2 (\log_2 h_n^{-1})^{d-1} \right) \ , \ i = 1,\cdots,m \tag{2.11}$$

$$\text{mit } c(n) := 1 + \sum_{k=1}^{d-1} \left(\frac{3}{4}\right)^k \binom{n+k-1}{k} \ .$$

(2.10) und (2.11) im Vergleich mit (2.6) und (2.7) zeigen, daß f durch die Dünngitterabbildung $\tilde{f}^n$ fast so gut approximiert wird wie durch die Vollgitterabbildung $\hat{f}^n$. In Kapitel 6 wird noch gezeigt werden, daß diese Tatsache nur für nach (2.1) glatte Datensätze gilt. Stark fehlerbehaftete oder gar unstetige Datensätze, wie aus einer Kernspinresonanztomographie, sind nicht für diesen Ansatz geeignet.

Die Interpolation auf dünnen Gitter gemäß (2.9) ist, verglichen mit der trilinearen Interpolation auf einem Vollgitter, ziemlich aufwendig und zudem vom Level n abhängig. Um einen Wert an einer gegebenen Stelle zu ermitteln, müssen für auf dem Rand verschwindende Abbildungen im dreidimensionalen Fall

$$m \cdot \sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{m}{6} n(n+1)(n+2) \tag{2.12}$$

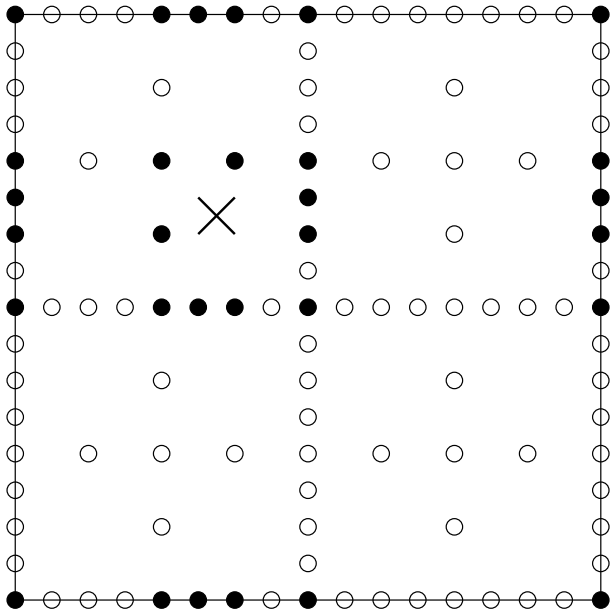


Abbildung 2.4: Für die Interpolation des Wertes X benötigte Basisfunktionen in $\tilde{G}_{4,4}$

Unterräume betrachtet werden und entsprechend viele *wesentliche arithmetische Operationen* (WAO) ausgeführt werden. Jede WAO beinhaltet dabei die Multiplikation für die Linearkombinationen (2.5), die Auswahl und Auswertung der Basisfunktion und die abschließende Addition für den Funktionswert in (2.4). Dabei ist zu beachten, daß die kompakten Träger der Basisfunktionen eines Unterraumes sich nicht überschneiden und daher nur ein Produkt aus Überschußkoeffizient und zugehöriger Basisfunktion je Unterraum gebildet werden muß. Im Vergleich dazu benötigt die trilineare Interpolation auf vollen Gittern nur konstante 8 WAO.

Abbildung 2.4 zeigt an einem Beispiel, welche Basisfunktionen in die Interpolation eines Wertes eingehen. In Kapitel 2.5 werden die Anzahl der wesentlichen arithmetischen Operationen auch für den allgemeinen Fall und mit Beachtung der Ränder untersucht.

2.3 Kombinationsmethode

Durch die Komplexität der Interpolation auf dünnen Gittern hat sich diese in der Numerik nur vereinzelt bei der Berechnung physikalischer Probleme durchgesetzt. In [2] wird zum Beispiel direkt auf dünnen Gittern die Lösung der dreidimensionalen Poisson-Gleichung berechnet. 1992 wurde durch Griebel, Schneider und Zenger in [6] die sogenannte *Kombinationsmethode* eingeführt, die inzwischen aufgrund ihrer Vorteile auf weiteres Interesse gestoßen ist, wie zum Beispiel in [4, 5, 7].

Die Kombinationsmethode baut in numerischen Berechnungen im Prinzip darauf auf, daß eine Lösung auf mehreren kleineren vollen Gittern berechnet wird, um durch geschickte Kombination dieser Lösungen eine Lösung auf einem dünnen Gitter zu erhalten. In [6, Seiten 5ff]

wurde bewiesen, daß der dadurch entstehende Fehler von der gleichen Größenordnung wie der eigentliche Interpolationsfehler auf den dünnen Gittern ist.

Bei der Kombinationsmethode wird für die Darstellung des dünnen Gitters eine Linearkombination mehrerer voller Gitter mit unterschiedlichen Schrittweiten in den verschiedenen Achsen gewählt. Man definiert die Interpolationsfunktion $\bar{f}^n$ im Zweidimensionalen als

$$\bar{f}_i^n = \sum_{\substack{j_1+j_2=n+1 \\ j_1,j_2\geq 1}} \bar{f}_i^{(j_1,j_2)} - \sum_{\substack{j_1+j_2=n \\ j_1,j_2\geq 1}} \bar{f}_i^{(j_1,j_2)}, \quad i=1,\ldots,m \tag{2.13}$$

und im Dreidimensionalen als

$$\bar{f}_i^n = \sum_{\substack{j_1+j_2+j_3=n+2 \\ j_1,j_2,j_3\geq 1}} \bar{f}_i^{(j_1,j_2,j_3)} - 2\cdot \sum_{\substack{j_1+j_2+j_3=n+1 \\ j_1,j_2,j_3\geq 1}} \bar{f}_i^{(j_1,j_2,j_3)} + \sum_{\substack{j_1+j_2+j_3=n \\ j_1,j_2,j_3\geq 1}} \bar{f}_i^{(j_1,j_2,j_3)} . \tag{2.14}$$

Dabei bezeichnet $\bar{f}^{(j_1,\ldots,j_d)}$ die bilineare Interpolationsfunktion im vollen Gitter mit den Schrittweiten $h_{j_i} = 2^{-j_i}$:

$$\bar{f}_i^{(j_1,\ldots,j_d)}(x_1,\ldots,x_d) = \sum_{k_1=0}^{2^{j_1}} \cdots \sum_{k_d=0}^{2^{j_d}} f_i\,(h_{j_1}\cdot k_1,\ldots,h_{j_d}\cdot k_d)\cdot \bar{b}_{k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}(x_1,\ldots,x_d) \quad , \tag{2.15}$$

$$\bar{b}_{k_1,\ldots,k_d}^{(j_1,\ldots,j_d)}(x_1,\ldots,x_d) = \prod_{i=1}^{d} w_{j_i}(x_i - k_i\cdot h_{j_i}) \quad ,$$

mit w_i aus (2.3). Die $\bar{f}^{(j_1,\ldots,j_d)}$ werden auf allen Gittern der Mengen $\mathcal{A}^n$ bis $\mathcal{A}^{n-d+1}$ ausgewertet mit

$$\mathcal{A}^p := \left\{ G_{j_1,\ldots,j_d} \;\middle|\; \sum_{i=1}^{d} j_i = p+d-1, \quad j_i \geq 1 \right\} \quad . \tag{2.16}$$

Analog zur Standardmethode ergibt sich durch das Fehlen der Gitterpunkte des $G_{n,\ldots,n}$, die zu $\bar{f}^n$ nichts beitragen das dünne Gitter der Kombinationsmethode $\bar{G}_n$. Abbildung 2.5 zeigt die Zusammensetzung des $\bar{G}_3$ nach (2.13) aus mehreren kleineren vollen Gittern.

Die Interpolation mit der Kombinationsmethode ist äquivalent zur direkten Interpolation auf dem dünnen Gitter. Um dies zu beweisen, muß erst gezeigt werden, daß die beiden Gitter $\tilde{G}_n$ und $\bar{G}_n$ identisch sind. Zur Vereinfachung wird hier nur der zweidimensionale Fall dargestellt. Die Beweise für den drei- und mehrdimensionalen Fall verlaufen analog.

Wir definieren für die weitere Beweisführung die Mengen T_{j_1,j_2} der Gitterpunkte, auf denen die Maxima der Basisfunktionen der Teilräume S_{j_1,j_2} liegen:

$$T_{j_1,j_2} := \left\{ \left(m_{k_1}^{(j_1)}, m_{k_2}^{(j_2)}\right) \;\middle|\; k_i \text{ aus (2.2)} \right\} \tag{2.17}$$

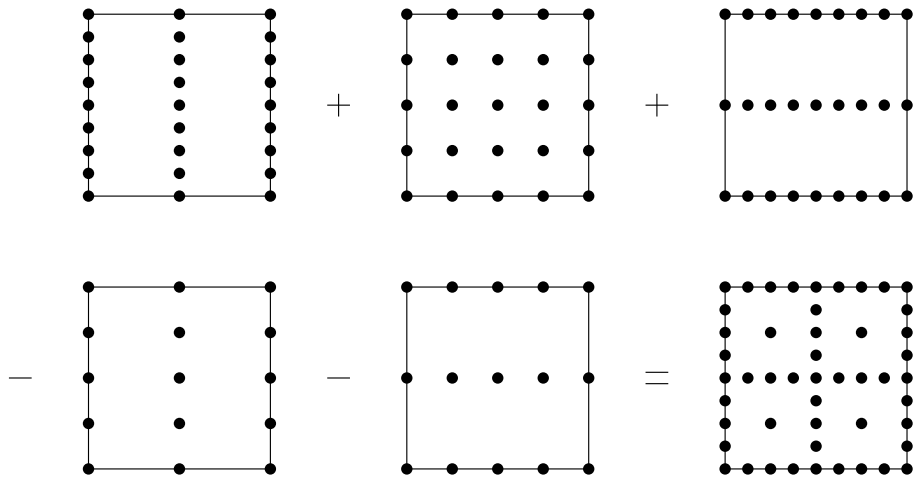


Abbildung 2.5: Zusammensetzung des $\bar{G}_3$ aus mehreren kleineren vollen Gittern in der Kombinationsmethode

Offensichtlich gilt $\{ m_k^{(j)} \mid j = 0, \ldots, i, \quad k \text{ aus } (2.2) \} = G_i$ und damit

$$\bigcup_{\substack{j_1=0,\ldots,i_1 \\ j_2=0,\ldots,i_2}} T_{j_1,j_2} = G_{i_1,i_2} \quad . \tag{2.18}$$

Wegen $G_{j_1,j_2} \subset G_{j_1,j_2+1}$ ist

$$\bar{G}_n = \bigcup_{k=1,\ldots,n} G_{k,n+1-k} \quad \cup \bigcup_{k=1,\ldots,n-1} G_{k,n-k} = \bigcup_{k_1+k_2 \leq n+1} G_{k_1,k_2} \quad . \tag{2.19}$$

Bei einer Analyse der Konstruktionsvorschrift der Interpolationsfunktion $\tilde{f}^n$ in (2.9) mit (2.5) stellt man fest, daß sich $\tilde{G}_n$ offensichtlich durch die Vereinigung aller T_{j_1,j_2} mit $(j_1, j_2) \in \mathcal{J}(n)$ darstellen läßt. Daraus ergibt sich schließlich mit (2.19):

$$\bar{G}_n = \bigcup_{(i_1,i_2)\in\mathcal{J}(n)} G_{i_1,i_2} = \bigcup_{(i_1,i_2)\in\mathcal{J}(n)} \bigcup_{\substack{j_1=0,\ldots,i_1 \\ j_2=0,\ldots,i_2}} T_{j_1,j_2} = \bigcup_{(i_1,i_2)\in\mathcal{J}(n)} T_{i_1,i_2} = \tilde{G}_n \quad .$$

Wie bereits oben bemerkt, nimmt die Interpolationsfunktion $\tilde{f}^n$ an den Stellen des dünnen Gitters den Wert der Ausgangsabbildung f an. Wie im folgenden bewiesen wird, ist durch die Wahl der Vollgitter auch bei der Kombinationsmethode gewährleistet, daß diese Bedingung erfüllt ist.

Es zeigt sich, daß alle Punkte $x = (x_1, \ldots, x_d) \in \bar{G}_n$ in allen Gittern aus (2.16) entweder enthalten sind oder nur linear zwischen zwei Nachbarpunkten interpoliert werden müssen.

Dazu spalten wir die Mengen $\mathcal{A}^p$ abhängig von x in die folgenden Mengen auf:

$$\mathcal{A}^p_{\alpha_1,\ldots,\alpha_d}(x) \;:=\; \left\{ G_{j_1,\ldots,j_d} \;\middle|\; \sum_{i=1}^{d} j_i = p+d-1,\; j_i \geq 1,\; \left\{ \begin{array}{ll} x_i \in G_{j_i} & \text{für } \alpha_i = 1 \\ x_i \notin G_{j_i} & \text{für } \alpha_i = 0 \end{array} \right\} \right\} \quad ,$$

$$\text{mit} \quad \bigcup_{\alpha_1=0}^{1} \cdots \bigcup_{\alpha_d=0}^{1} \mathcal{A}^p_{\alpha_1,\ldots,\alpha_d} \;=\; \mathcal{A}^p$$

$$\text{und} \quad \mathcal{A}^p_{\alpha_1,\ldots,\alpha_d} \cap \mathcal{A}^p_{\beta_1,\ldots,\beta_d} = \emptyset \;\; \forall\, (\alpha_1,\ldots,\alpha_d) \neq (\beta_1,\ldots,\beta_d) \quad .$$

Für die Klassifikation der Punkte x definieren wir noch analog zu (2.17) die Mengen $\bar{T}_{j_1,\ldots,j_d}$ für $j_i \geq 1$:

$$\bar{T}_{j_1,\ldots,j_d} \;:=\; \bigcup_{\alpha_1=0}^{1} \cdots \bigcup_{\alpha_d=0}^{1} \left\{ \begin{array}{ll} T_{\alpha_1 j_1,\ldots,\alpha_d j_d} & \text{für } (1-\alpha_i) j_i \leq 1 \;\forall\, i = 1,\ldots,d \\ \emptyset & \text{sonst} \end{array} \right\} \quad , \qquad (2.20)$$

woraus sich sofort

$$G_{j_1,\ldots,j_d} \;=\; \bigcup_{k_1=1}^{j_1} \cdots \bigcup_{k_d=1}^{j_d} \bar{T}_{k_1,\ldots,k_d}$$

ergibt. Das entspricht im Zweidimensionalen

$$\bar{T}_{j_1,j_2} \;:=\; T_{j_1,j_2} \;\cup\; \left\{ \begin{array}{ll} T_{0,j_2} & \text{für } j_1 = 1 \\ \emptyset & \text{für } j_1 > 1 \end{array} \right\} \cup \left\{ \begin{array}{ll} T_{j_1,0} & \text{für } j_2 = 1 \\ \emptyset & \text{für } j_2 > 1 \end{array} \right\}$$

$$\cup \left\{ \begin{array}{ll} T_{0,0} & \text{für } j_1 = 1 \wedge j_2 = 1 \\ \emptyset & \text{sonst} \end{array} \right\} \quad .$$

x sei im folgenden aus $\bar{T}_{k_1,k_2}$. Da wir momentan nur $x \in \bar{G}_n$ betrachten, ist $k_1 + k_2 \leq n+1$. Wegen (2.18) und (2.20) ist $x_1 \in G_{k_1}$ und $x_2 \in G_{k_2}$.

Es zeigt sich nun, daß $\mathcal{A}^{n-1}_{0,0} = \emptyset$ und damit auch $\mathcal{A}^n_{0,0} = \emptyset$ für alle $x \in \tilde{G}_n$:

$$\mathcal{A}^{n-1}_{0,0} \neq \emptyset \;\Leftrightarrow\; \exists\, j_1, j_2 \text{ mit } j_1 + j_2 = n \;\wedge\; j_1 < k_1 \;\wedge\; j_2 < k_2 \quad .$$

Wegen $j_1 < k_1 \Leftrightarrow j_1 + 1 \leq k_1$ folgt nun aus $k_1 + k_2 \leq n+1$

$$n - j_2 + 1 + k_2 \leq n+1 \quad \Leftrightarrow \quad k_2 \leq j_2 \quad ,$$

was zu einem Widerspruch zur Annahme $j_2 < k_2$ führt.

Ohne Beschränkung der Allgemeinheit betrachten wir nur die Mengen $\mathcal{A}^n_{1,0}$, $\mathcal{A}^{n-1}_{1,0}$, $\mathcal{A}^n_{1,1}$ und $\mathcal{A}^{n-1}_{1,1}$ genauer; $\mathcal{A}^n_{0,1}$ verhält sich aus Symetriegründen äquivalent zu $\mathcal{A}^n_{1,0}$.

$$
\begin{aligned}
\mathcal{A}^n_{1,0} &= \{G_{j_1,j_2} \mid j_1 + j_2 = n+1,\ j_1 \geq k_1,\ 1 \leq j_2 < k_2\} \\
&= \{G_{j_1,j_2} \mid j_2 = n+1-j_1,\ n+1-k_2 < j_1 \leq n\} \\
\text{mit} \quad & \#\mathcal{A}^n_{1,0} \ = \ n - n - 1 + k_2 \ = \ k_2 - 1 \quad , \\
\mathcal{A}^{n-1}_{1,0} &= \{G_{j_1,j_2} \mid j_1 + j_2 = n,\ j_1 \geq k_1,\ 1 \leq j_2 < k_2\} \\
&= \{G_{j_1,j_2} \mid j_2 = n-j_1,\ n-k_2 < j_1 \leq n-1\} \\
\text{mit} \quad & \#\mathcal{A}^{n-1}_{1,0} \ = \ n - 1 - n + k_2 \ = \ k_2 - 1 \ = \ \#\mathcal{A}^n_{1,0} \quad , \\
\mathcal{A}^n_{1,1} &= \{G_{j_1,j_2} \mid j_1 + j_2 = n+1,\ j_1 \geq k_1,\ j_2 \geq k_2\} \\
&= \{G_{j_1,j_2} \mid j_2 = n+1-j_1,\ k_1 \leq j_1 \leq n+1-k_2\} \\
\text{mit} \quad & \#\mathcal{A}^n_{1,1} \ = \ n + 1 - k_2 - k_1 + 1 \ \geq \ 1 \quad , \\
\mathcal{A}^{n-1}_{1,1} &= \{G_{j_1,j_2} \mid j_1 + j_2 = n,\ j_1 \geq k_1,\ j_2 \geq k_2\} \\
&= \{G_{j_1,j_2} \mid j_2 = n-j_1,\ k_1 \leq j_1 \leq n-k_2\} \\
\text{mit} \quad & \#\mathcal{A}^{n-1}_{1,1} \ = \ n - k_2 - k_1 + 1 \ = \ \#\mathcal{A}^n_{1,1} - 1 \quad .
\end{aligned}
$$

Da $x_1 \in G_{k_1}$ ist und für alle Gitter G_{j_1,j_2} gilt, daß

$$
\begin{aligned}
j_1 > k_1 \quad & \text{für} \quad G_{j_1,j_2} \in \mathcal{A}^n_{1,0} \\
j_1 \geq k_1 \quad & \text{für} \quad G_{j_1,j_2} \in \mathcal{A}^{n-1}_{1,0} \quad ,
\end{aligned}
$$

reduziert sich die bilineare Interpolation für diese Gitter auf eine lineare Interpolation zwischen je zwei in x_2 unterschiedlichen Gitterpunkten. Assoziiert man die Gitter $G^1_{j_1,j_2} \in \mathcal{A}^n_{1,0}$ mit den Gittern $G^2_{j_1-1,j_2} \in \mathcal{A}^{n-1}_{1,0}$, stellt man fest, daß diese dieselben Gitterpunkte für die Interpolation benutzen. Demnach ergeben sich bei der Interpolation auf G^1 und G^2 die gleichen Werte. Diese werden in (2.13) voneinander abgezogen und heben sich somit auf. Entsprechendes gilt für die Gitter $G^1_{j_1,j_2} \in \mathcal{A}^n_{0,1}$ und $G^2_{j_1,j_2-1} \in \mathcal{A}^{n-1}_{0,1}$.

Weiterhin gilt für alle $G \in \mathcal{A}^n_{1,1} \cup \mathcal{A}^{n-1}_{1,1}$, daß x in G liegt. Die Interpolationswerte nehmen demnach auf allen diesen Gittern den Funktionswert an. Da die Mächtigkeit von $\mathcal{A}^n_{1,1}$ um eins größer ist als die von $\mathcal{A}^{n-1}_{1,1}$, ergibt sich schließlich

$$
\bar{f}^n(x_1, x_2) \ = \ f(x_1, x_2) \ = \ \tilde{f}^n(x_1, x_2) \quad \forall\, x \in \tilde{G}_n \quad . \tag{2.21}
$$

Damit haben wir bewiesen, daß die Standard- und die Kombinationsmethode für alle Punkte auf dem dünnen Gitter die Werte der Abbildung f annehmen. Für den Beweis, daß die Gleichungen (2.9) und (2.13) wirklich identisch sind, muß noch gezeigt werden, daß sie die gleichen Werte für alle $(x_1, \ldots, x_d) \in G_{n,\ldots,n}$ annehmen. Dies ist hinreichend für die Identität, da zwischen diesen Punkten keine Fallunterscheidungen und damit auch keine Träger der Basisfunktionen überschritten werden, und beide Interpolationsmethoden auf linearen Basisfunktionen beruhen.

Dazu betrachten wir die Gitter $G_{j_1,\ldots,j_d} \in \mathcal{A}^p,\ p = 1,\ldots,n$. Die mit diesen Gittern assoziierten Abbildungsräume $\hat{L}_{j_1,\ldots,j_d}$ können wieder in Teilräume zerlegt werden:

$$\hat{L}_{j_1,\ldots,j_d} \;=\; \bigoplus_{k_1=0}^{j_1} \cdots \bigoplus_{k_d=0}^{j_d} S_{k_1,\ldots,k_d}$$

Mit $\hat{L}_{r_1,\ldots,r_d} \subseteq \hat{L}_{s_1,\ldots,s_d}$ für alle $r_i \leq s_i\ \forall\, i = 1,\ldots,d$ ergibt sich sofort

$$\bar{L}_n \;\subseteq\; \bigoplus_{p=1}^{n} \bigoplus_{\substack{(j_1,\ldots,j_d)\text{mit}\\ G_{j_1,\ldots,j_d}\in\mathcal{A}^p}} S_{j_1,\ldots,j_d} \;=\; \bigoplus_{\substack{(j_1,\ldots,j_d)\text{mit}\\ G_{j_1,\ldots,j_d}\in\mathcal{A}^n}} S_{j_1,\ldots,j_d} \;\subseteq\; \bar{L}_n \quad .$$

Damit läßt sich nun zeigen, daß der von den $S_{k_1,\ldots,k_d}$ für alle $G_{j_1,\ldots,j_d} \in \mathcal{A}^n$ aufgespannte Raum mit $\tilde{L}_n$ identisch ist:

$$\bar{L}_n \;=\; \bigoplus_{\substack{j_1+\cdots+j_d=n+d-1\\ j_i\geq 1}} \bigoplus_{k_1=0}^{j_1} \cdots \bigoplus_{k_d=0}^{j_d} S_{k_1,\ldots,k_d} \;=\; \bigoplus_{(k_1,\ldots,k_d)\in\mathcal{I}(n)} S_{k_1,\ldots,k_d} \;=\; \tilde{L}_n \quad .$$

Wie wir bereits gezeigt haben, ist $\bar{f}^n(x) = \tilde{f}^n(x) = f(x)$ für alle Dünngitterpunkte. Diese Punkte sind linear unabhängig, da an ihnen die Werte der Ausgangsabbildung angenommen werden. Durch die Konstruktion der dünnen Gitter entspricht die Anzahl der Basisfunktionen eines dünnen Gitters der Anzahl der Dünngitterpunkte. Da die beiden Räume identisch sind und der Dimension des Raumes entsprechend viele linear unabhängige Funktionen auf die gleichen Werte abgebildet werden, gilt schließlich ohne Einschränkungen:

$$\bar{f}^n(x) \;=\; \tilde{f}^n(x)$$

Die Kombinationsmethode hat gegenüber der Standard-Dünngitter-Methode eine Reihe von Vorteilen: Sie muß nicht mit komplizierten Datenstrukturen kämpfen, Ränder müssen nicht durch ausgefeilte Sonderfälle integriert werden, sondern sind durch ihr Prinzip bereits enthalten, und es müssen keine neuen mathematischen Methoden zum Beispiel für Differentialgleichungen auf dünnen Gittern entworfen werden, es reichen die bereits bekannten Methoden für volle Gitter.

Der wichtigste Vorteil der Kombinationsmethode ist aber die theoretisch wesentlich schnellere Interpolation. Für die Auswertung eines Beitrags (2.15) sind 2^d Basisfunktionen und damit ebensoviele WAO nötig, da die Träger der Basisfunktionen eines Unterraumes disjunkt sind. Im Dreidimensionalen werden demnach für eine Auswertung von (2.14)

$$8m \cdot \sum_{i=n-2}^{n} \frac{i(i+1)}{2} \;=\; 12mn(n-1) + 8 \quad \text{WAO}$$

benötigt, was eine ganze Größenordnung weniger ist als (2.12).

Ein Nachteil der Kombinationsmethode ist, daß etliche Funktionswerte mehrfach gespeichert werden müssen, da sie in mehreren der vollen Gitter vorkommen. Dadurch erhöht sich, wie in Kapitel 2.5 noch gezeigt wird, der Speicherbedarf gegenüber der Standardmethode deutlich. Die Ersparnis gegenüber einem vollen Gitter überwiegt aber noch stark. Man könnte durch die hierarchische Speicherung der vollen Gitter den Speicherbedarf dem der Standardmethode angleichen, allerdings würde auch die Interpolation entsprechend komplex und damit langsam werden. Dadurch würde der Vorteil der Kombinationsmethode wegfallen.

2.4 Beschleunigungsansätze für dünne Gitter

Eine der wichtigsten Aufgaben dieser Arbeit ist es, die Interpolation auf dünnen Gittern so zu beschleunigen, daß interaktives Arbeiten möglich wird. Dabei besteht ein Ansatz darin, die Teilräume des dünnen Gitters bei der Interpolation wegzulassen, die nur einen kleinen Beitrag zum interpolierten Gesamtwert haben. Den maximalen Beitrag eines Unterraumes kann man in einem Vorverarbeitungsschritt ermitteln und speichern. Danach kann für die Interpolation eine Schranke ε vorgegeben werden, für die die Unterräume nicht beachtet werden, deren maximaler Beitrag ε nicht überschreitet.

Dieser Ansatz hat sich bei den dünnen Gittern als nicht sehr erfolgreich erwiesen. Durch die zusätzlichen Daten erhöht sich der Speicheraufwand etwas, und die notwendigen Vergleiche machen einen großen Teil der Zeitersparnis wieder zunichte. Noch viel schwerwiegender ist aber, daß ε für eine meßbare Zeitersparnis so groß gewählt werden muß, daß die durch die Interpolationsfehler der dünnen Gitter ohnehin schon schlechteren Bilder unzumutbar stark in Mitleidenschaft gezogen werden. Wenn man bei der Berechnung den Level reduziert, ergeben sich ähnliche Interpolationsfehler bei vergleichbarer Zeitersparnis.

Wenn der algorithmische Ansatz scheitert, führt ein typischer Weg für die Beschleunigung unmittelbar zu *brute force*, also reiner Rechenkraft. Neben dem trivialen Ansatz, einen schnelleren Computer einzusetzen, spezielle Compileroptionen zu verwenden und kritische Codesegmente zu optimieren, werden oft auch Multiprozessormaschinen ausgenutzt. Hier existiert eine Vielzahl unterschiedlicher Architekturen und demnach auch Ansätze zur Beschleunigung. Neben reinen Multiprozessormaschinen gibt es auch Maschinen mit Spezialprozessoren, wie zum Beispiel DSPs oder auch Grafikprozessoren.

Da die Kombinationsmethode im dreidimensionalen Fall sehr viele trilineare Interpolationen durchführen muß, liegt es nahe, diese Aufgabe der Grafik-Hardware schnellerer Silicon Graphics Workstations zu überlassen. Die Grafiksysteme *Maximum Impact* und *Reality Engine 2*, die am Institut zur Verfügung stehen, besitzen für die Ausgabe von 3D-Texturen Spezial-Hardware, die ohne Prozessorunterstützung Texturen trilinear interpolieren kann.

Die mathematischen Grundlagen der Hardware-unterstützten Kombinationsmethode entsprechen denen der normalen Kombinationsmethode. Allerdings kann die Hardware nur dann optimal eingesetzt werden, wenn nicht nur ein Wert an einer bestimmten Stelle ermittelt werden soll, sondern gleich eine ganze Fläche auf einmal interpoliert wird. Zum Volume-Raycasting

existiert ein Standard-Algorithmus, der die Hardware für die schnelle Darstellung voller Gitter ausnutzt. Dieser Algorithmus kann für die Kombinationsmethode abgewandelt werden. Aufgrund der beschränkten Größe des Framebuffers der verwendeten Grafikkarten ergibt sich allerdings noch ein weiterer Quantisierungsfehler bei der Interpolation, auf den in Kapitel 5.5 noch genauer eingegangen wird.

2.5 Aufwandsanalyse

Der Hauptvorteil der dünnen Gitter ist die gewaltige Speicherersparnis. Während man für ein volles Gitter $G_{n,\ldots,n}$ $m \cdot (2^n + 1)^d$ Speicherstellen benötigt, ergibt sich die Anzahl der Gitterpunkte eines dünnen Gitters $\tilde{G}_n$ und damit (multipliziert mit m) auch die Anzahl der Überschußkoeffizienten nach [2, Seiten 19ff] als

$$\#\tilde{G}_n^{(1)} = 2^n + 1 \ , \quad \#\tilde{G}_n^{(d)} = 3 \cdot \#\tilde{G}_n^{(d-1)} \ + \ \sum_{i=1}^{n-1} 2^i \cdot \#\tilde{G}_{n-i}^{(d-1)} \quad \text{für } d > 1 \ .$$

Da diese Definition rekursiv über die Dimension läuft, wurde in der Bezeichnung ausnahmsweise die Dimension mit aufgeführt.

In der Realität wird die Speicherersparnis nicht ganz so groß ausfallen, da durch den komplizierten Aufbau der dünnen Gitter einige Datenstrukturen für die Verwaltung angelegt werden müssen.

Die Kombinationsmethode benötigt mit

$$m \cdot \sum_{i=n-d+1}^{n} \ \sum_{j_1+\cdots+j_d=i} \ \prod_{k=1}^{d} 2^{j_k} + 1$$

Speicherplätzen zwar deutlich mehr als die Standard-Dünngitter-Methode; in der Tabelle 2.1 kann man sich aber überzeugen, daß die Speicherersparnis gegenüber den vollen Gittern noch deutlich überwiegt.

In dieser Tabelle ist auch der Texturspeicherbedarf der Hardware-unterstützten Kombinationsmethode aufgeführt. Theoretisch müßte jene die gleiche Anzahl an Überschußkoeffizienten wie die normale Kombinationsmethode aufweisen. Unter OpenGL sind allerdings Texturen Hardware-bedingt in jeder Achse in der Größe auf Zweierpotenzen beschränkt. In Kapitel 5.3 wird dieser Zusammenhang noch deutlicher aufgeführt. Weiterhin gilt es zu beachten, daß es aufgrund der beschränkten Framebuffer-Tiefe vollkommen ausreichend ist, Texturen mit nur 16 Bit zu allozieren, während die Überschußkoeffizienten in der Standard- und Kombinationsmethode als Gleitkommazahlen mit 32 Bit gespeichert werden.

Der Hauptnachteil der dünnen Gitter ist die ungleich kompliziertere Interpolation auf den Daten. Man beachte, daß die trilineare Interpolation auf vollen Gittern unabhängig von der Levelzahl nur 2^d wesentliche arithmetische Operationen (WAO) benötigt.

Zwei Dimensionen:

Level	Kantenlänge	Volles Gitter	Standard	Kombi	Hardware
1	3	9	9	9	16
2	5	25	21	39	80
3	9	81	49	109	256
4	17	289	113	271	704
5	33	1 089	257	641	1792
6	65	4 225	577	1 475	4 352
7	129	16 641	1 281	3 333	10 240
8	257	66 049	2 817	7 431	23 552
9	513	263 169	6 145	16 393	53 248
10	1 025	1 050 625	13 313	35 851	118 784
15	32 769	$1{,}07 \cdot 10^9$	589 825	1 638 421	5 767 168
20	1 048 577	$1{,}10 \cdot 10^{12}$	$2{,}41 \cdot 10^7$	$6{,}82 \cdot 10^7$	$2{,}47 \cdot 10^8$
30	$1{,}07 \cdot 10^9$	$1{,}15 \cdot 10^{18}$	$3{,}54 \cdot 10^{10}$	$1{,}02 \cdot 10^{11}$	$3{,}82 \cdot 10^{11}$
50	$1{,}13 \cdot 10^{15}$	$1{,}27 \cdot 10^{30}$	$5{,}97 \cdot 10^{16}$	$1{,}75 \cdot 10^{17}$	$6{,}71 \cdot 10^{17}$

Drei Dimensionen:

Level	Kantenlänge	Volles Gitter	Standard	Kombi	Hardware
1	3	27	27	27	64
2	5	125	81	162	448
3	9	729	225	630	1 984
4	17	35 937	593	1 997	7 040
5	33	35 937	1 505	5 687	22 016
6	65	274 625	3 713	15 188	63 488
7	129	2 146 689	8 961	38 868	173 056
8	257	16 974 593	21 249	96 471	452 608
9	513	$1{,}35 \cdot 10^8$	49 665	233 949	1 146 880
10	1 025	$1{,}08 \cdot 10^9$	114 689	557 030	2 834 432
15	32 769	$3{,}52 \cdot 10^{13}$	6 619 137	35 520 576	204 734 464
20	1 048 577	$1{,}15 \cdot 10^{18}$	$3{,}32 \cdot 10^8$	$1{,}89 \cdot 10^9$	$1{,}17 \cdot 10^{10}$
30	$1{,}07 \cdot 10^9$	$1{,}24 \cdot 10^{27}$	$4{,}29 \cdot 10^{11}$	$4{,}03 \cdot 10^{12}$	$2{,}69 \cdot 10^{13}$
50	$1{,}13 \cdot 10^{15}$	$1{,}43 \cdot 10^{45}$	$4{,}48 \cdot 10^{17}$	$1{,}10 \cdot 10^{19}$	$7{,}86 \cdot 10^{19}$

Alle Angaben in Speicherplätzen und nicht in Bytes.

Tabelle 2.1: Größe und somit Speicherbedarf der vollen und dünnen Gitter, Speicherbedarf der Kombinationsmethode und benötigter Texturspeicher für die Hardware-beschleunigte Kombinationsmethode

Allgemein betrachtet benötigt eine Interpolation auf dünnen Gittern im d-Dimensionalen ohne Betrachtung der Ränder

$$m \cdot \sum_{i=1}^{n} \binom{d+i-2}{i-1} = \frac{m}{(d-1)!} \sum_{i=1}^{n} \prod_{j=i}^{i+d-2} j \quad \text{WAO}$$

und mit Betrachtung der Ränder

$$m \cdot \sum_{k=1}^{d} \sum_{i=1}^{n} 2^{d-k} \binom{d}{k} \binom{k+i-2}{i-1} + m \cdot \ 2^d \cdot 1 \cdot 1 \quad \text{WAO} \ .$$

Diese Formeln ergeben direkt aus der Kombinatorik. Im Dreidimensionalen benötigt man demnach

$$m \cdot \sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{m}{6} n(n+1)(n+2) \quad \text{WAO}$$

ohne Betrachtung der Ränder und

$$m \left(\frac{1}{6} n(n+1)(n+2) + 12n + 3n(n+1) + 8 \right) = \frac{m}{6} \left(n^3 + 21n^2 + 92n + 48 \right) \quad \text{WAO} \ , \quad (2.22)$$

wenn man die Ränder mit einbezieht.

Die Interpolation mit der Kombinationsmethode ist gegenüber der Standardmethode asymptotisch wesentlich schneller. Für die Auswertung eines Beitrags (2.15) sind 2^d Basisfunktionen und damit ebensoviele WAO nötig. Im Zweidimensionalen werden demnach für eine Auswertung von (2.13)

$$4m \cdot \sum_{i=n-1}^{n} i = 4m(2n-1) \quad WAO$$

benötigt. Für drei Dimensionen ergeben sich

$$8m \cdot \sum_{i=n-2}^{n} \frac{i(i+1)}{2} = 12mn(n-1) + 8 \quad \text{WAO} \ ,$$

was eine ganze Größenordnung weniger ist als (2.22).

Bei genauerer Betrachtung der Tabelle 2.2 erkennt man allerdings, daß im Dreidimensionalen dieser Vorteil erst bei einem Level zum Tragen kommt, der durch die extreme Größe der Datenmengen momentan sowieso noch keine Rolle spielt. Nicht zu vergessen ist allerdings, daß nur die Kombinationsmethode durch ihre trilineare Interpolation einen Zugang für die Hardware-Beschleunigung bietet. Außerdem hat sich gezeigt, daß die Kombinationsmethode auch in den theoretisch ungünstigen Fällen $n \ll 50$ bis um den Faktor 10 schneller ist als die Standardmethode, da die einfachere Rechenvorschrift rein iterativ in einer sehr kurzen Schleife durchgeführt werden kann, die ohne Probleme vollständig in den First-Level-Cache

Zwei Dimensionen:

Level	Ohne Rand	Mit Rand	Kombi
1	1	9	4
2	3	15	12
3	6	22	20
4	10	30	28
5	15	39	36
8	36	72	60
10	55	99	76
15	120	184	116
20	210	294	156
30	465	589	236
40	820	984	316
50	1 275	1 479	396
100	5 050	5 454	796
1 000	500 500	504 504	7 996

Drei Dimensionen:

Level	Ohne Rand	Mit Rand	Kombi
1	1	27	8
2	4	54	32
3	10	90	80
4	20	136	152
5	35	193	248
8	120	440	680
10	220	678	1 088
15	680	1 588	2 528
20	1 540	3 048	4 568
30	4 960	8 118	10 448
40	11 480	16 888	18 728
50	19 600	30 358	29 408
100	171 700	203 208	118 808
1 000	$1,67 \cdot 10^8$	$1,70 \cdot 10^8$	$1,20 \cdot 10^7$

Tabelle 2.2: Anzahl wesentlicher arithmetischer Operationen für Interpolation auf dünnen Gittern, für die Standardmethode ohne und mit Rand, sowie für die Kombinationsmethode

paßt. Es entstehen außerdem keine zusätzlichen Verzögerungen durch bedingte Sprünge oder rekursiven Abstieg.

Die Auswahl, welche Basisfunktion für die betrachtete Stelle herangezogen werden muß, kann im Aufwand mit der Wahl der Zelle für die trilineare Interpolation verglichen werden. Da die Anzahl der zu betrachtenden Basisfunktionen direkt mit der Gesamtanzahl der wesentlichen arithmetischen Operationen korreliert, kann diese Auswahl als implizit in der Anzahl der WAO enthalten betrachtet werden. Auch die Auswertung der Basisfunktion ist von der gleichen Komplexität wie die Auswertung der trilinearen Basisfunktionen.

Kapitel 3

Volumenvisualisierung

3.1 Visualisierungsmethoden

Die beiden am häufigsten benutzten Verfahren zur Volumenvisualisierung sind nach [3, Seite 1035] Konturflächenbestimmung mit anschließender Darstellung der geometrischen Fläche und direkte Volumenvisualisierung.

Die Visualisierung der Volumendaten durch ISO-Flächen hat den Nachteil, daß durch ihr Prinzip nicht das Volumen, sondern nur eine Grenzschicht dargestellt wird, und Effekte wie allmähliche Übergänge nicht dargestellt werden können. Die direkte Volumenvisualisierung hat dagegen den Vorteil, daß durch halbdurchlässige Volumenelemente auch verborgene Details erkannt werden können. Zudem kann die direkte Volumenvisualisierung durch die Auswahl entsprechender Beleuchtungsfunktionen auch zur Darstellung ausgewählter ISO-Flächen benutzt werden. Die ISO-Flächendarstellung durch direkte Volumenvisualisierung kann sogar genauer sein, da kein Zwischenschritt über eine Polygondarstellung mit entsprechenden Rundungsverlusten und Aliasing-Effekten durchgeführt wird. Allerdings berechnet der normale Marching-Cubes-Algorithmus, im Gegensatz zur direkten Volumendarstellung, exakte Schnittpunkte innerhalb der Interpolationsfunktionen und kann daher mit aktiviertem Anti-Aliasing ähnlich gute Resultate liefern.

Die Bestimmung von ISO-Flächen direkt auf dünnen Gittern sowie Beschleunigungsverfahren für diese Algorithmen werden in [9] behandelt.

Die direkte Volumenvisualisierung kann wiederum in zwei Klassen aufgeteilt werden:

- Vorwärts-Projektion oder Objektraumverfahren
- Rückwärts-Projektion oder Bildraumverfahren

Vorwärts-Projektionsverfahren wie das sogenannte Splatting durchlaufen sämtliche Voxel in einer Schleife und berechnen deren Beitrag zu den einzelnen Pixeln. Für viele Datenstrukturen können diese Verfahren durch Spezialhardware beschleunigt werden. Da die Beiträge der Voxel auf die Pixel aber im allgemeinen über einen kantenabschwächenden Kernel, beispielsweise

einem Gauß-Kernel, verteilt werden, sind die resultierenden Bilder häufig nicht so scharf wie mit Rückwärts-Projektionsverfahren berechnete. Diese durchlaufen sämtliche Pixel und berechnen mittels Raycasting, welche Voxel einen Beitrag hierzu liefern.

Da die Rückwärts-Projektionsverfahren im allgemeinen bessere Bilder liefern, und sich die untersuchten Beschleunigungsansätze gut für diese Verfahren eignen, wurde für diese Arbeit das Bildraumverfahren Raycasting gewählt.

3.2 Grundlagen des Raycasting

Beim Raycasting werden die Lichtstrahlen zurückverfolgt, die das Bild auf der Netzhaut des Betrachters erzeugen. Auf dem Weg dieser sogenannten Augenstrahlen durch das Volumen werden die Beiträge der Volumenelemente für mehrere repräsentative Wellenlängen aufintegriert. Wird das Integral diskretisiert, fallen die Samplingpunkte im allgemeinen nicht mit den Zentren der Volumenelemente zusammen, da das Volumen aus den unterschiedlichsten Richtungen betrachtet wird. In der Regel werden daher die Samplingwerte durch trilineare Interpolation der angrenzenden Voxel gewonnen; Interpolationsverfahren höherer oder niedrigerer Ordnung ergeben dementsprechend bessere oder schlechtere Bilder.

Da die Volumendaten meist keine Farbinformation, sondern nur skalare Werte enthalten, muß vor der Integration aus diesen ein Farbwert gewonnen werden. Diesen Schritt kann man wieder in zwei Teilschritte unterteilen, die Anwendung einer Transferfunktion und die Beleuchtungsberechnung. Die Transferfunktion gewinnt aus einem skalaren Wert einen Farbwert. Durch die Beleuchtungsberechnung wird dieser Farbwert zum Beispiel mit dem Gradienten an der aktuellen Interpolationsstelle so verknüpft, daß ein stärkerer dreidimensionaler Eindruck entsteht. Besonders ISO-Flächen sind erst auf diese Weise im Raum lokalisierbar. In Kapitel 3.4 wird genauer auf die Beleuchtungsberechnung eingegangen.

3.3 Integrationsmethoden

Die Integrationsformeln physikalisch motivierter Modelle, wie zum Beispiel des Emissions-Absorptions-Modells, sind in der Regel aufwendiger und in der Berechnung langsamer. Ihr Vorteil ist, die „realen“ Verhältnisse in einem Volumen mit semitransparenter Materie wiederzugeben und dadurch auch dem ungeschulten Auge zugänglich zu sein.

In diesen Modellen wird ausgehend von der Transporttheorie des Lichts eine Näherung der Transfergleichung ermittelt, die einerseits die gewünschte Realitätsnähe erreicht, andererseits aber durch ihren Berechnungsaufwand die Bildwiederholrate nicht zu sehr senkt. Eine ausführliche Herleitung verschiedener Modelle findet man in [8].

Auf der anderen Seite haben nicht direkt von der Transporttheorie abgeleitete Integrationsmodelle den Vorteil, die Bilder nicht zu überladen und den Betrachter nicht zu überfordern. Mit Hilfe des MIP-Modells kann man zum Beispiel die Gefäße in einer Computertomographie gut darstellen.

3.3.1 Röntgen

Die Integrationsmethode Röntgen wertet ein einfaches Integral über die Volumenwerte aus. Die einzelnen repräsentativen Wellenlängen beeinflussen sich dabei nicht. Das diskretisierte Integral wurde in der Klasse `LightingXRay` implementiert.

$$I^* = \int_{S_{\text{far}}}^{S_{\text{near}}} q^*(s)\, ds \quad \text{für } * \text{ aus } \{r, g, b\} \tag{3.1}$$

Wird diese Gleichung nach Euler diskretisiert, ergibt sich für

$$\Delta s = (S_{\text{near}} - S_{\text{far}})/n\ , \quad s_i := S_{\text{far}} + i\Delta s \tag{3.2}$$

nach Vereinfachung die Summe

$$I^* = \Delta s \sum_{i=0}^{n-1} q^*(s_i). \tag{3.3}$$

3.3.2 MIP

MIP steht für *Maximum Intensy Projection*. Bei dieser Integrationsmethode wird für jede Wellenlänge unabhängig das Maximum auf dem Augenstrahl gesucht. Im allgemeinen wird MIP nur auf skalare Größen und daher mit einer Graustufen- oder Falschfarbentransferfunktion angewandt. Die Implementierung dieser Methode mit einer sehr einfachen Falschfarbentransferfunktion ist in der Klasse `LightingMip` zu finden.

$$I^* = \max_{s \in [S_{\text{far}}, S_{\text{near}}]} q^*(s) \quad \text{für } * \text{ aus } \{r, g, b\} \tag{3.4}$$

Diskretisiert ergibt dies:

$$I^* = \max_{i \in \mathbb{Z}_n} q^*(s_i) \quad s_i \text{ siehe (3.2)}, \quad \mathbb{Z}_n = \{0, \ldots, n-1\} \tag{3.5}$$

3.3.3 Emissions-Absorptionsmodell

Das Emissions-Absorptionsmodell ist ein einfaches, physikalisch basiertes Beleuchtungsmodell ohne Scattering-Effekte. Wird dieses Modell ohne lokale Beleuchtungsberechnung eingesetzt, lassen sich Volumen sehr gut darstellen, während die darin enthaltenen Flächen teilweise schlecht zur Geltung kommen. Die Klasse `LightingEmAbISO` implementiert dieses Modell, wobei sie speziell auf die Visualisierung von lokal beleuchteten ISO-Flächen zugeschnitten wurde.

Nach [8, Seite 13–14] ergibt sich bei der Vereinfachung der Transfergleichung das Integral

$$I^* = I^*_{\text{far}} e^{-\tau(S_{\text{far}}, S_{\text{near}})} + \int_{S_{\text{far}}}^{S_{\text{near}}} q^*(s) e^{-\tau(s, S_{\text{near}})}\, ds \quad \text{für } * \text{ aus } \{r, g, b\} \tag{3.6}$$

mit der sogenannten optischen Tiefe

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(s)\, ds\ . \tag{3.7}$$

Mit q^* wird dabei der Emissions-Koeffizient, mit κ der Absorptions-Koeffizient bezeichnet. I^*_{far} wird über die Randbedingungen (Hintergrundfarbe) eindeutig spezifiziert.
Mit den aus (3.2) und (3.7) gewonnenen Gleichungen

$$\tau(s_{k_1}, s_{k_2}) = \sum_{j=k_1+1}^{k_2} \tau(s_{j-1}, s_j)\ , \quad s_0 = S_{\text{far}} \quad \text{und} \quad s_n = S_{\text{near}} \tag{3.8}$$

kann man das Integral so in n Teilintegrale zerlegen, daß sich mit den Abkürzungen

$$\theta_k = e^{-\tau(s_{k-1}, s_k)} \quad \text{und} \quad b^*_k = \int_{s_{k-1}}^{s_k} q^*(s) e^{-\tau(s, s_k)}\, ds \tag{3.9}$$

I^* wie folgt darstellen läßt:

$$\begin{aligned} I^* &= I^*_{\text{far}} e^{-(\sum_{k=1}^{n} \tau(s_{k-1}, s_k))} + \sum_{k=1}^{n} \int_{s_{k-1}}^{s_k} q^*(s) e^{-(\tau(s, s_k) + \sum_{j=k+1}^{n} \tau(s_{j-1}, s_j))}\, ds \\ &= I^*_{\text{far}} \prod_{k=1}^{n} \theta_k + \sum_{k=1}^{n} b^*_k \prod_{j=k+1}^{n} \theta_j = \sum_{k=0}^{n} b^*_k \prod_{j=k+1}^{n} \theta_j \quad \text{mit} \quad b^*_0 := I^*_{\text{far}} \end{aligned} \tag{3.10}$$

Werden jetzt $\forall s_{k-1} \leq s < s_k :\ q^*(s) := q^*_k \wedge \kappa(s) := \kappa_k$ als stückweise konstant angenommen, können alle b^*_k und θ_k durch

$$\theta_k = e^{-\kappa_k \Delta s} \quad \text{und} \tag{3.11}$$

$$\begin{aligned} b^*_k &= \int_0^{\Delta s} q^*_k e^{-\kappa_k s}\, ds = q^*_k \left(\tfrac{1}{-\kappa_k} e^{-\kappa_k \Delta s} + \tfrac{1}{\kappa_k}\right) \\ &= \frac{q^*_k}{\kappa_k} (1 - \theta_k) \end{aligned} \tag{3.12}$$

im Diskreten exakt dargestellt werden. Auch wenn die Exponentialfunktion in (3.12) nur einmal für jeden Farbwert pro Bild berechnet werden muß, indem man die Einträge der Farbtabelle transformiert, werden b^*_k und θ_k oft nur über das 1. Taylorpolynom zum Entwicklungspunkt $(\kappa_k \Delta s)_0 = 0$ approximiert:

$$\begin{aligned} b^*_k &= \frac{q^*_k}{\kappa_k} (1 - e^{-\kappa_k \Delta s}) = \frac{q^*_k}{\kappa_k} (1 - (1 - \kappa_k \Delta s + o\,(\kappa_k \Delta s))) \\ &\approx q^*_k\, \Delta s \quad \text{für} \quad \kappa_k \Delta s \ll 1 \end{aligned} \tag{3.13}$$

$$\theta_k = e^{-\kappa_k \Delta s} = 1 - \kappa_k \Delta s + o\,(\kappa_k \Delta s) \approx 1 - \kappa_k \Delta s \tag{3.14}$$

3.4 Transferfunktionen und Beleuchtungsberechnung

Vor dem eigentlichen Integrationsprozeß müssen die in der Regel eindimensionalen Volumenwerte auf einen Farbwert und seine Opazität (RGBA) abgebildet werden. Die Opazität geht nicht in die primitiven Beleuchtungsmodelle (MIP, Röntgen) ein, etwas komplexere Modelle wie Emission-Absorption brauchen für die Modellierung von Transparenz mindestens diese vier Werte. Aufwendigere Modelle können durchaus noch mehr Farbwerte für zusätzliche repräsentative Wellenlängen benötigen, zum Beispiel um Frequenzverschiebungen an optischen Gittern zu simulieren.

Die Transferfunktion bildet dabei die vorhandenen Daten auf einen Farbwert ab, der der lokalen Klassifikation der Daten dient. Einfache Modelle bilden skalare Volumenwerte auf Graustufen ab oder interpolieren innerhalb einer interaktiv erstellten Farbtabelle. Komplexere Modelle können auch den Gradienten, höhere Ableitungen oder externe Segmentierungs- und Klassifikationsdaten ausnutzen, um beispielsweise in den Daten einer tomographischen Aufnahme den Organen verschiedene Farben zuzuordnen. ISO-Flächen werden in der Klasse `LightingEmAbISO` dadurch gewonnen, daß die Volumendaten in der Regel auf die transparente Farbe und beim Auftreten einer Fläche auf eine konstante opaque Farbe abgebildet werden.

Der so erhaltene Farbwert geht nun in die Beleuchtungsberechnung ein, die in der Regel die Farbe direkt übernimmt oder als Grundfarbe für ein einfaches Lambert- oder Phong-Modell benutzt. Diese Modelle sind zum Beispiel in [3, Seiten 722–731] ausführlich beschrieben.

Die Transfer- und die Beleuchtungsfunktion beeinflussen sich gegenseitig und sind zusätzlich noch von der verwendeten Integrationsmethode abhängig. Zudem sind sie sehr zeitkritisch, da sie in der innersten Schleife des Raycasters ausgeführt werden müssen. Aus diesen Gründen hat es sich als sinnvoll erwiesen, alle drei Schritte zusammenzufassen und gezielte Kombinationen dem Benutzer zur Auswahl zu stellen. Die derzeit implementierten Kombinationen werden in Kapitel 4.5.2 vorgestellt.

Kapitel 4

Implementierung

4.1 Voraussetzungen

Im Rahmen dieser Arbeit wurde das Programm *sgrid* geschrieben, das volle und dünne Gitter laden und über verschiedene Algorithmen als Volumen darstellen kann. Dabei können interaktiv die aktuelle Sicht, die vordere und hintere Clip-Ebene, Auflösung, Schrittweite, der Level des zu visualisierenden dünnen Gitters und die verwendete Integrationsmethode gewählt werden.

Für die Anbindung an die Grafik-Hardware existiert mit *OpenGL* ein allgemein anerkannter Standard, der von Silicon Graphics entwickelt wurde und inzwischen vom *OpenGL Architecture Review Board* (ARB) gepflegt und weiterentwickelt wird. Um die Raycasting-Parameter leicht interaktiv wählen zu können, wurde eine Viewer-Klasse von SGIs *Inventor*-Paket verwendet. Inventor ist ein Scene-Graph-API mit dazugehörigen Bibliotheken, die ebenfalls auf OpenGL aufbauen.

Das Programm wurde für Silicon Graphics Workstations in der Programmiersprache C++ für das SGI-eigene UNIX-Derivat IRIX geschrieben. Für das User-Interface wurde das im Developer-Paket enthaltene *RapidApp* [14] benutzt. Die Sprache C++ wurde gewählt, da sie durch ihre Nähe zu C sehr schnell ist und eine gute Anbindung an die Systembibliotheken und insbesondere an OpenGL bietet. Außerdem vereinfacht sie den Programmentwurf durch objektorientierte Ansätze und wird von RapidApp gut unterstützt.

Die Hardware-beschleunigte Kombinationsmethode funktioniert aufgrund der Transparenz von OpenGL auf allen SGI Workstations. Allerdings ist die Berechnung auf Rechnern mit einer Grafikkarte ohne Volumentextur-Unterstützung deutlich langsamer als die normale Kombinationsmethode, da alle Grafikoperationen in Software ausgeführt werden müssen. Zur Zeit der Entstehung dieser Arbeit gibt es am Institut genau zwei Rechner mit geeigneten Grafikkarten mit 3D-Textur-Unterstützung, einer *Maximum Impact* und einer *Reality Engine 2*.

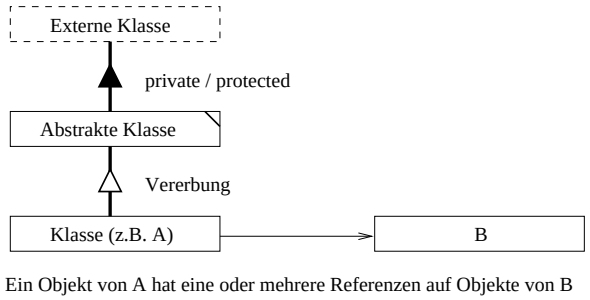


Abbildung 4.1: Legende für die Klassendiagramme

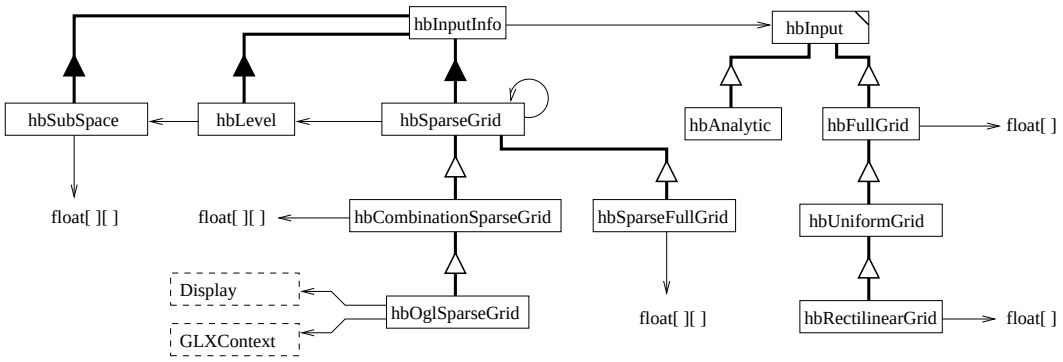


Abbildung 4.2: Hierarchie der Interpolationsklassen von `sgrid`

4.2 Klassenhierarchie

Wie aus dem Klassendiagramm der Interpolationsklassen in Abbildung 4.2 hervorgeht, existieren zwei voneinander getrennte Hierarchien. Eine ist von `hbInput` abgeleitet, die andere protected abgeleitet von `hbInputInfo`. Die von `hbInput` abgeleiteten Klassen stehen dabei für Eingabefunktionen, zum Beispiel volle Gitter wie `hbUniformGrid` oder analytische Funktionen wie `hbAnalytic`. Die von `hbInputInfo` abgeleiteten Klassen bilden die Interpolationsfunktionen. Wie bereits in den Namen zu erkennen ist, ist in der Klasse `hbSparseGrid` die Standardmethode der Dünngitter-Interpolation, in `hbCombinationSparseGrid` die Kombinationsmethode und in `hbOglSparseGrid` die Hardware-unterstützte Kombinationsmethode implementiert.

Eine Besonderheit bietet die Klasse `hbSparseFullGrid`. Sie interpoliert ihre Daten auf einem vollen Gitter, bietet aber die Schnittstelle der Dünngitter-Klassen. Mit ihrer Hilfe können nach dem Laden eines vollen Gitters die mit den entsprechenden dünnen Gittern erstellten Bilder direkt mit dem ursprünglichen Gitter verglichen werden.

Die Klassen `hbInputInfo`, `hbSparseGrid`, `hbLevel`, `hbSubSpace`, `hbInput`, `hbAnalytic` und `hbFullGrid` wurden von Christian Teitzel schon vor meiner Arbeit erstellt. Sie wurden teilweise in der Semantik erweitert und an die Bedürfnisse von `sgrid` angepaßt. Die Legende in Abbildung 4.1 beschreibt die verwendeten Symbole.

Das Klassendiagramm in Abbildung 4.3 zeigt die Abhängigkeiten der Klassen von `sgrid`, die für die Volumendarstellung und das User-Interface zuständig sind. Dabei haben sowohl das `MainPanel`-Objekt als auch der gerade aktive `Raycaster` eine Referenz auf ein Objekt der im Moment benutzten Dünngitter-Interpolationsklasse `hbSparseGrid` oder einer davon abgeleiteten Klasse der Abbildung 4.2.

Alle Klassen, die von `VkComponent`, `VkGenericDialog`, `VkFileSelectionDialog`, `VkTabbedDeck`, `VkWindow` oder `VkSimpleWindow` abgeleitet sind, wurden zunächst mit RapidApp automatisch generiert und später mit Semantik ausgebaut. Eine genauere Beschreibung der Implementierung des User-Interfaces ist in Kapitel 4.4.1 zu finden.

Die Klasse `SoXtExaminerViewer` wird von RapidApp zur Verfügung gestellt und enthält einen ExaminerViewer des *Inventor*-Pakets. Mit diesem Viewer kann der Benutzer die aktuelle Sicht auf das Volumen sowie die vordere und hintere Clip-Ebene festlegen.

Alle verwendeten Raycaster-Varianten sind als Subklassen der abstrakten Klasse `Raycaster` implementiert. Diesen ist Kapitel 4.5.1 gewidmet. Die von der abstrakten Klasse `Lighting` abgeleiteten Klassen enthalten die Integrationsmethoden sowie die Beleuchtungsberechnung. In Kapitel 4.5.2 sind sie genauer beschrieben.

Die Klasse `apTime` wird nur zur Zeitmessung benutzt und bedarf im folgenden keiner weiteren Erklärung. Sie stammt ursprünglich von Roberto Grosso.

Alle Komponenten des User-Interfaces wurden mit einer Online-Hilfe versehen, um die Bedienung des Programms möglichst einfach zu gestalten. Die Implementierung dieser Online-Hilfe in der Klasse `helpWindow` benutzt HTML als Datenformat und wird in Kapitel 4.4.4 genauer beschrieben.

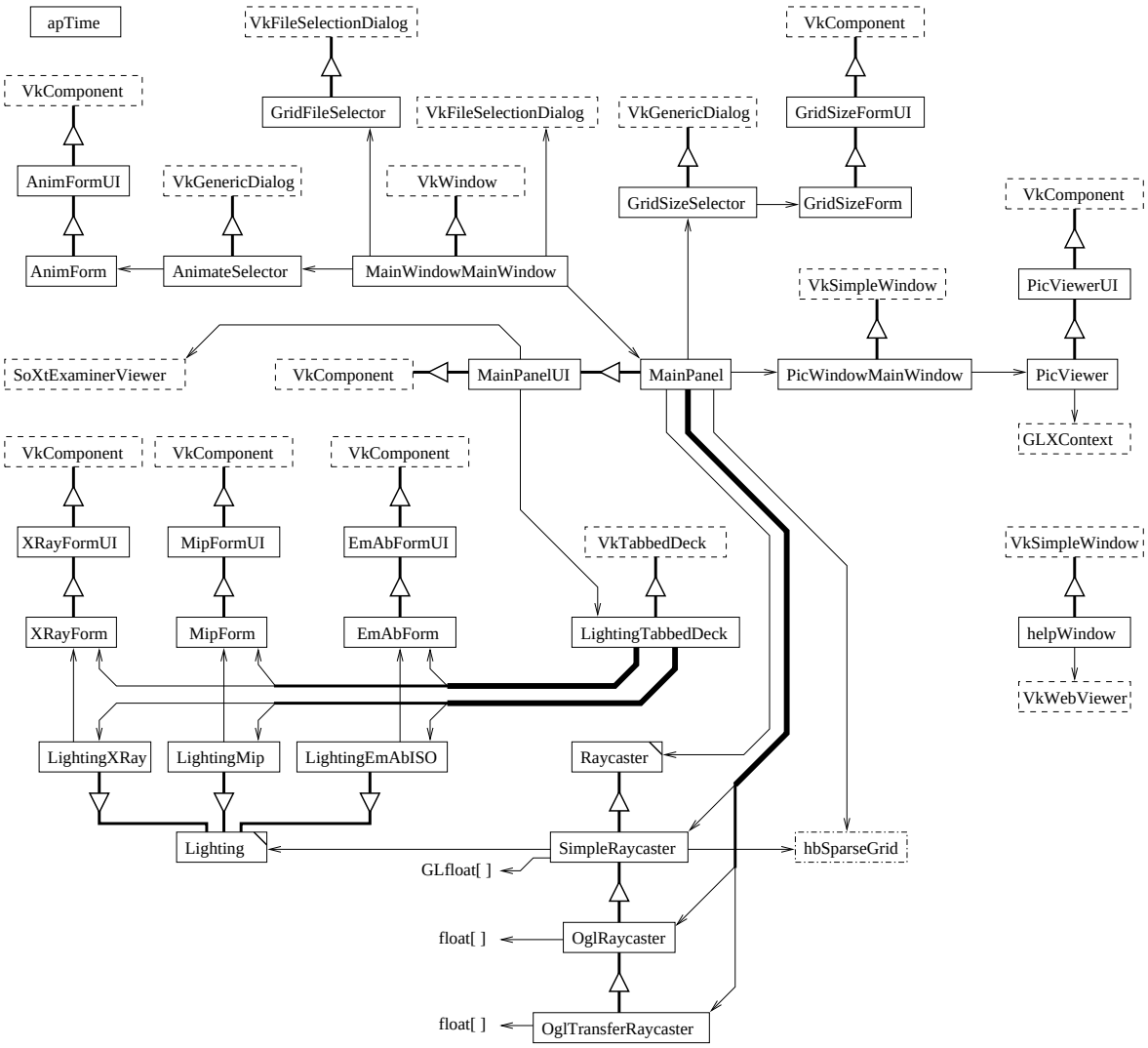
4.3 User-Interface

Abbildung 4.4 zeigt das User-Interface des Programms mit allen wesentlichen Einstellmöglichkeiten. Dieses Fenster ist, wie auch das in Abbildung 4.5 dargestellte Bildanzeigefenster, immer zu sehen.

Der rechte Teil des Hauptfensters wird durch einen Inventor Examiner-Viewer ausgefüllt, mit dem der Benutzer die aktuelle Sicht auf das Volumen wählen kann. Das Volumen wird dabei durch einen Drahtgitterwürfel angedeutet. Mehr Informationen über die Möglichkeiten des Inventor Viewers enthält [20].

Im linken Teil des Hauptfensters kann bei der Standardmethode eingestellt werden, bis zu welchem Level die dünnen Gitter bei der Interpolation benutzt werden sollen. Wie noch in Kapitel 4.6.1 beschrieben wird, muß der maximal verwendete Level bereits beim Laden des dünnen Gitters feststehen. Da bei der Kombinationsmethode die Größe der Vollgitter vom Level abhängt, ist der Regler außer bei der Standardmethode wirkungslos.

Weitere Einstellungsmöglichkeiten bieten einige Regler für die Anzahl der Samples pro Strahl, die Größe des Ausgabebildes, den Vergrößerungsfaktor bei der Ausgabe und dem Epsilon-Wert



Alle Referenzen auf Standard-Widget-Objekte sind der Übersicht halber nicht dargestellt.

Abbildung 4.3: Hierarchie der Raycasterklassen und des User-Interfaces von `sgrid`

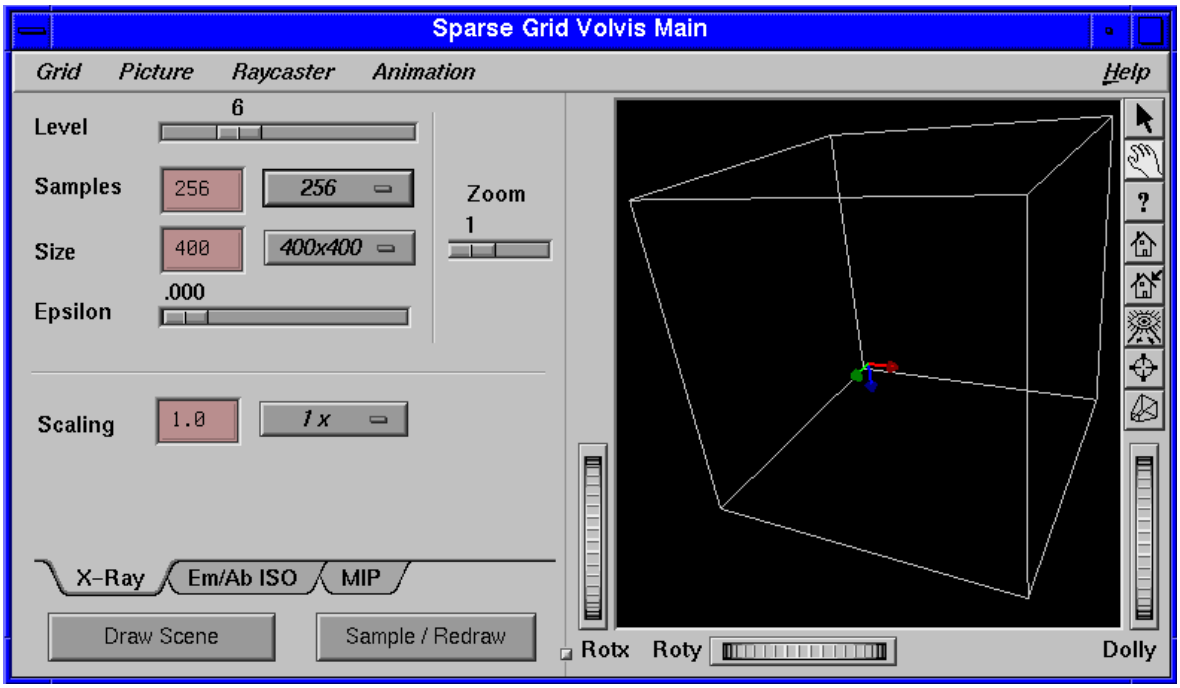


Abbildung 4.4: Das User-Interface von `sgrid`

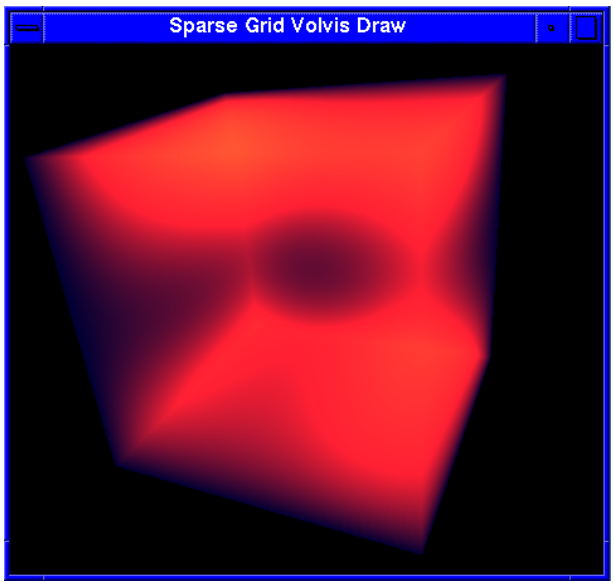


Abbildung 4.5: Das Fenster zur Bildanzeige

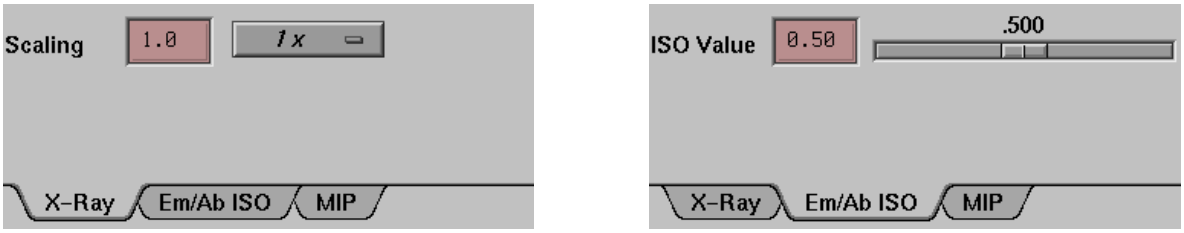


Abbildung 4.6: Einstellmöglichkeiten für verschiedene Beleuchtungsmodelle

für die Beschleunigung durch Weglassen einzelner Teilräume bei der Interpolation (siehe auch Kapitel 2.4).

Unterhalb der allgemeinen Einstellungselemente können das gewünschte Beleuchtungsmodell und die entsprechende Integrationsmethode gewählt werden. Die veränderbaren Parameter können für jedes Modell unterschiedlich sein. Abbildung 4.6 zeigt die Widgets der Beleuchtungsmodelle Röntgen (*X-Ray*) und Emission-Absorption für ISO-Flächen (*Em/Ab ISO*). Bei dem Röntgen-Modell werden die Volumendaten noch mit dem eingestellten Wert skaliert, bevor sie integriert werden. Für die Darstellung von ISO-Flächen muß der zu visualisierende ISO-Wert gewählt werden. Das MIP-Modell hat keine benutzerdefinierbaren Parameter und ist daher nicht aufgeführt.

Über das *Grid*-Menü können sowohl verschiedene Arten von Gittern geladen als auch das Programm beendet werden. Im Dateifenster, das in Abbildung 4.7 gezeigt wird, bietet `sgrid` dem Benutzer eine Reihe verschiedener Datenformate für das Laden voller und dünner Gitter an. Des weiteren können hier die Interpolationsmethode und der Level des zu ladenden Gitters eingestellt werden. Wie in Kapitel 4.6.1 beschrieben, kann für die Standardmethode der Level bei der Interpolation unabhängig vom Level beim Laden des Gitters gewählt werden. Natürlich macht es keinen Sinn, die Daten mit einem höheren Level zu visualisieren, als sie geladen wurden. Wird beim Laden nicht ein dünnes, sondern ein volles Gitter ausgewählt, wandelt das Programm dieses automatisch in ein dünnes Gitter des angegebenen Levels um.

Manche Laderoutinen benötigen für die Durchführung noch zusätzliche Informationen. Beispielsweise ist in „rohen“ Vollgitterdaten keine Informationen über die Größe des Volumens gespeichert. Diese müssen über einen weiteren Dialog dem Programm mitgeteilt werden. `sgrid` versucht dabei, über eine Heuristik sinnvolle Werte voreinzustellen, so daß der Benutzer oft nur die vorgegebenen Werte bestätigen muß. Der Dialog für rohe Vollgitterdaten ist in Abbildung 4.8 dargestellt. Gitter, die nicht der einzigen unterstützten Auflösungsvariante $2^i + 1 \times 2^i + 1 \times 2^i + 1$ entsprechen oder aus nicht-kubischen Voxeln bestehen, werden automatisch in das nächst größere Gitter umgewandelt.

Nach dem Laden des Gitters können in beliebiger Zahl Bilder mit den eingestellten Parametern in der Sicht des Inventor-Widgets berechnet werden, indem der Knopf *Draw Scene* mit der Maus betätigt wird. Wird das Anzeigefenster von einem anderen Fenster überlagert, kann mit *Sample/Redraw* ein bereits berechnetes Bild neu dargestellt werden. Eine automatische Refresh-Funktion ist derzeit nicht implementiert (siehe auch Kapitel 7.2).

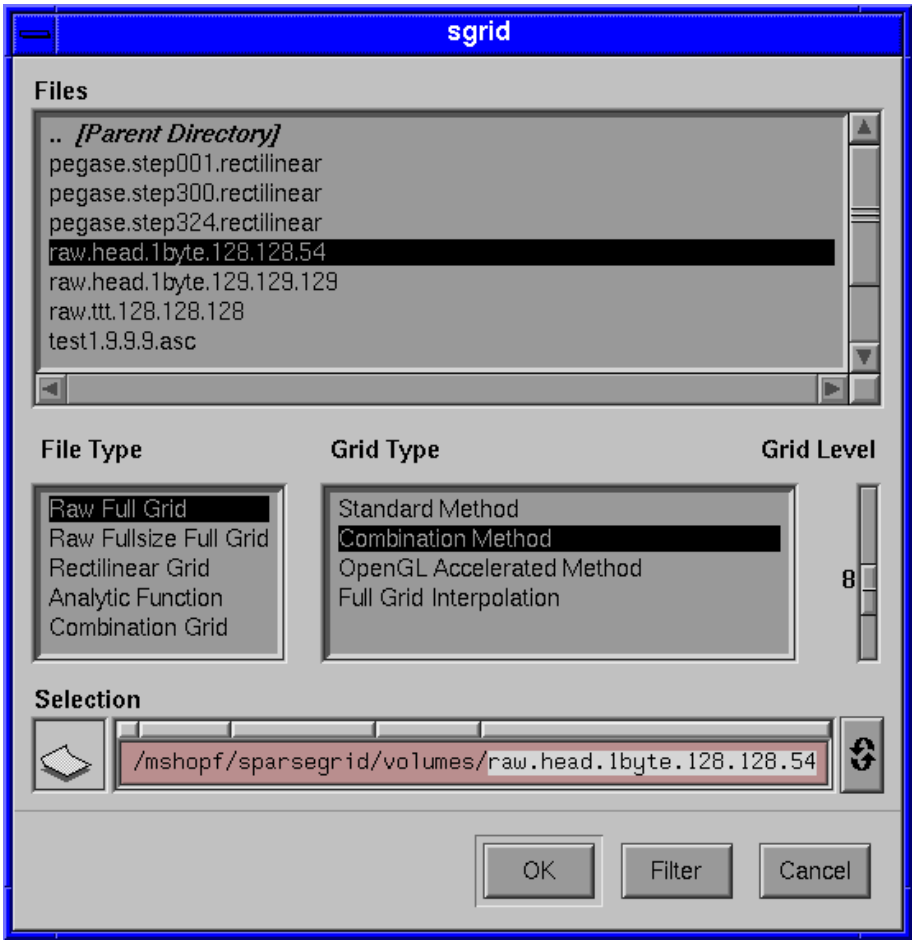


Abbildung 4.7: File-Dialog zur Auswahl des zu ladenden Gitters

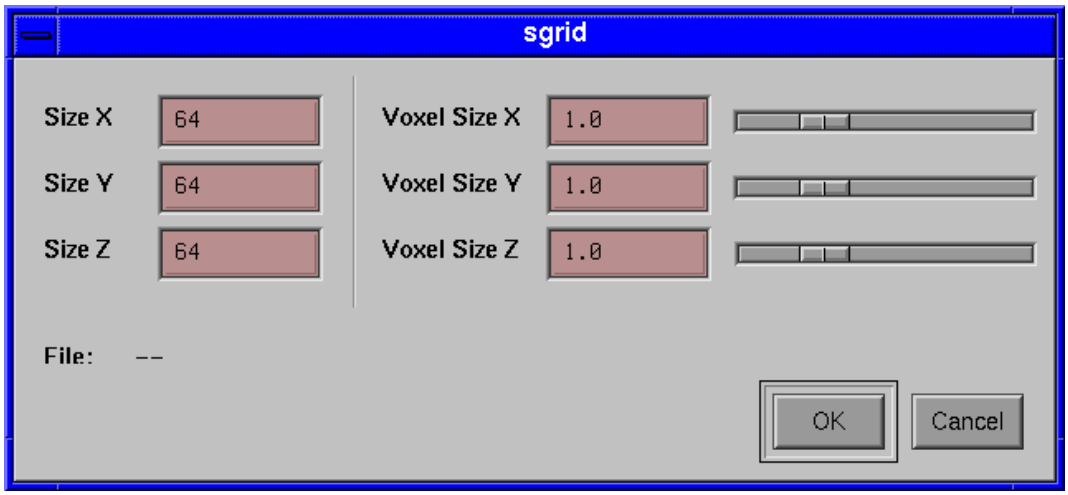


Abbildung 4.8: Dialog zum Einstellen der Größe des zu ladenden Gitters

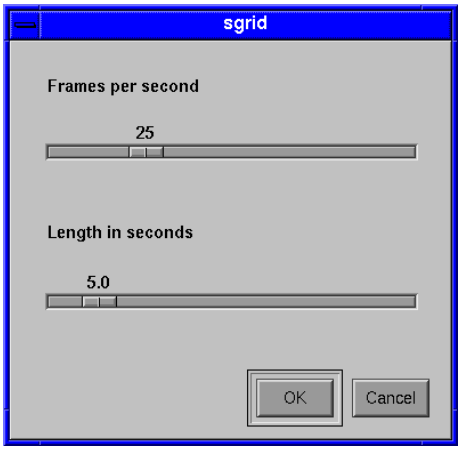


Abbildung 4.9: Einstellungen für die Animationsfunktion

Über das Menü des Hauptfensters können noch einige weitere Funktionen und Parameter gewählt werden. Das Menü *Picture* enthält im wesentlichen Funktionen, um das berechnete Bild zu speichern. `sgrid` speichert zusätzlich automatisch jedes berechnete Bild im aktuellen Verzeichnis im *Portable PixMap*-Format unter dem Namen `sgrid_pic.ppm`.

Im Menü *Raycaster* kann der benutzte Raycaster-Typ ausgewählt werden. *Ray by Ray* ist nur noch aus historischen Gründen enthalten, und beinhaltet den Standard-Raycaster, der für jeden Pixel einzeln einen Strahl durch das Volumen schickt und das Volumen entsprechend abtastet. *Plane by Plane* ist der Default-Raycaster für alle Visualisierungsarten, die keine Normalen für die Beleuchtungsberechnung benötigen. Er ist daher erheblich schneller als *Plane by Plane + Normals*.

Im *Animation*-Menü kann eine komplette Animation oder nur eine Sicht auf das Volumen erstellt, geladen und gespeichert, sowie die Berechnung der Animation gestartet werden. Bei der Berechnung werden alle Einzelbilder unter den Namen `sgrid_anim_`***`wxyz`***`.ppm` gespeichert, wobei ***`wxyz`*** eine fortlaufende vierstellige Nummer darstellt. Vor der Erstellung der Animation muß über einen weiteren Dialog (siehe Abbildung 4.9) noch die Anzahl der Abtastungen pro Sekunde sowie die Dauer der Sequenz in Sekunden eingestellt werden. Sobald nun der Knopf *Sample/Redraw* gedrückt wird, wird vom Programm die aktuelle Sicht des Inventor-Fensters in den eingestellten Abständen gespeichert. Jetzt kann man die Sicht auf das Volumen verändern und die Animations-Routine speichert automatisch die Parameter für die jeweils entsprechende Sicht auf das Volumen. Außerdem kann auch ausgenutzt werden, daß die Sicht im Inventor-Viewer einfach in eine gleichförmige Rotationen um eine beliebige Achse im Raum versetzt werden kann.

Zu bemerken ist noch, daß *Draw Scene* immer die erste Szene einer Sequenz berechnet, wenn eine Animation geladen oder erstellt worden ist. Ist gerade keine Animation aktiv, gilt die aktuelle Ansicht im Inventor-Widget als aktuelle Sequenz und kann daher auch über das Animation-Menü gespeichert werden. Auf diese Weise läßt sich leicht eine einmal erreichte Sicht im Inventor-Widget als Animation speichern und später nach dem Laden die gleiche Sicht erneut berechnen.

4.4 Programmgerüst

4.4.1 Grafische Benutzeroberfläche

Die Klassen für die grafische Benutzeroberfläche wurden im wesentlichen von *RapidApp* erstellt, einem Programm von SGI für die interaktive GUI-Programmierung. Das Programm trennt dabei die Implementierung des User-Interfaces deutlich von der Semantik, die hinter diesem steckt. Zur Anbindung an X11 benutzt RapidApp dabei die auf das Standard-X11-Toolkit *Motif* aufsetzende *ViewKit*-Klassenbibliothek. ViewKit enthält unter anderem auch Klassen, die die Verwendung von OpenGL oder Inventor deutlich erleichtern. Wie in Abbildung 4.3 ersichtlich ist, sind dabei alle Methoden, die mit dem X11-Toolkit interagieren, in den Klassen mit den Namen `xyzUI` untergebracht. Davon abgeleitet werden dann die Klassen mit den Namen `xyz`, in denen die Semantik zu den User-Interface-Klassen implementiert werden soll. Die Trennung erfolgt dabei so sauber, daß in den allermeisten Fällen die UI-Klassen nicht mehr verändert und nur noch die Semantik-Klassen ausgebaut werden müssen. Die einzige Ausnahme scheinen hier Dialoge zu sein, die von der Klasse `VkFileSelectionDialog` abgeleitet sind.

Der C++-Code, der von RapidApp erstellt wird, erlaubt es, eigene Erweiterungen zu dem generierten Code hinzuzufügen. Dazu werden in die Klassendefinition und in jede generierte Methode Kommentarzeilen der Form

```
//---- Start editable code block: MainPanel constructor
//---- End editable code block: MainPanel constructor
```

hinzugefügt, wobei RapidApp den Code innerhalb dieser Blöcke unverändert übernimmt, wenn das User-Interface verändert wird und deshalb der automatisch generierte C++-Code neu erzeugt werden muß. Dieses Prinzip hat sich manchmal als lästig, aber meistens als sehr gut funktionierend erwiesen.

Nicht sehr hoch ist der Abstraktionslevel allerdings bei der Parameterverwaltung. RapidApp stellt dem Programmierer hier keinerlei Hilfsroutinen zur Verfügung, und alle Parameter, die nicht Event-Charakter besitzen, müssen direkt über die entsprechenden fehleranfälligen, weil nicht streng typisierten, Motif-Funktionen abgefragt werden. Das Event-Modell ist nicht besonders flexibel, aber sehr einfach zu benutzen. Beim Eintreten eines definierten Events, zum Beispiel wenn der Benutzer einen Knopf drückt, einen Menüpunkt auswählt oder einen Wert ändert, wird eine vorher eingestellte Methode an der entsprechenden Semantik-Klasse aufgerufen. Hier könnten der Methode auch leicht als Aufrufparameter die dazugehörigen Daten der veränderten Widgets übergeben werden, was die Programmierung vereinfachen und die Abstraktion fördern würde.

Etwas unglücklich ist auch die Aufteilung der Hauptsemantik. Während in der Klasse `MainPanel` nahezu alle Fäden des Programms zusammenlaufen und daher fast alle Hauptaktionen von hier gestartet werden, enthält die Klasse `MainWindowMainWindow` die Semantik zu den Menüpunkten, da sich diese von der GUI-Struktur her gesehen außerhalb des `MainPanel` befinden. Die meisten Menüfunktionen springen daher direkt in eine Methode des `MainPanel`-

Objektes, das von `MainWindowMainWindowUI` erzeugt worden ist, da sie auf Datenstrukturen zugreifen müssen, die nur innerhalb dieser Klasse sichtbar sind.

4.4.2 Repräsentation der Eingabedaten

Für die Repräsentation der Eingabedaten wurden mehrere Klassen geschrieben, aus denen durch Interpolation an den Punkten dünner Gitter ein Objekt der ausgewählten Dünngitter-Klasse erzeugt wird. Alle bis jetzt implementierten Eingabeklassen arbeiten dabei auf vollen Gittern, da derzeit noch kein Dateiformat für dünne Gitter existiert, das implementiert werden könnte. Die Eingabeklassen repräsentieren jeweils eine bestimmte Vollgitterart:

- `hbFullGrid` wird für uniforme Vollgitterdaten eingesetzt, auf denen nicht interpoliert werden muß. Dies ist genau dann der Fall, wenn die Eingabedaten die Größe $2^p + 1 \times 2^p + 1 \times 2^p + 1$ haben. Alle Laderoutinen erwarten Rohdaten, die als Bytes vorliegen.
- `hbUniformGrid` wird für reguläre uniforme Vollgitterdaten eingesetzt. Die für die dünnen Gitter benötigten Werte werden durch trilineare Interpolation auf den Vollgitterdaten generiert. Alle Laderoutinen erwarten Rohdaten, die als Bytes vorliegen.
- `hbRectilinearGrid` wird für rectilineare Gitter verwendet. Bei dem zu dieser Klasse gehörenden Fileformat ist die Größe des Gitters bereits im Format mit enthalten und muß nicht vom Benutzer angegeben werden. Die Beschreibung des Formats ist in der Datei `hbRectilinearGrid.h` enthalten.
- `hbAnalytic` enthält eine analytische Abbildung. Sie ist in der Datei `hbAnalytic.h` genau beschrieben. In Kapitel 6.1 wird eine der Teilfunktionen genauer dargestellt.

Um aus den Eingabedaten ein dünnes Gitter zu generieren, wird ein Objekt der Eingabeklasse dem dünnen Gitter bei seiner Konstruktion übergeben. Das Objekt erzeugt sich dann bei einem Aufruf der Methode `setCoeff()` selbst, indem es von der Eingabeklasse über die Methode `getData()` die benötigten Gitterwerte erhält.

4.4.3 Sichtbestimmung und Berechnung der Raycast-Parameter

Im Prinzip ist es möglich, im Inventor-Widget an beliebiger Stelle im Szenen-Graphen die Kontrolle zu übernehmen, um hier durch direkte Aufrufe von OpenGL-Funktionen in das Inventor-Widget zu zeichnen. Dadurch könnte man auch die Volumenvisualisierung direkt im Inventor-Fenster vornehmen. Diese Lösung scheint aber nur angebracht, wenn die Framerate der eigenen Routine so hoch liegt, daß der Benutzer hier in der Bedienung nur unmerklich gebremst wird. Dafür ist jedoch die Interpolation auf dünnen Gittern zu aufwendig. Außerdem hätte diese Lösung den Nachteil, nur beschränkte Kontrolle über die gezeichneten Dreiecke zu haben, da die aktuelle Perspektive und die Viewing-Position über die ModelView-Matrix von Inventor vorgegeben sind. Noch schwerwiegender ist die Tatsache, daß man durch diese Lösung nur mit großer Mühe reine Softwarelösungen wie `hbSparseGrid`

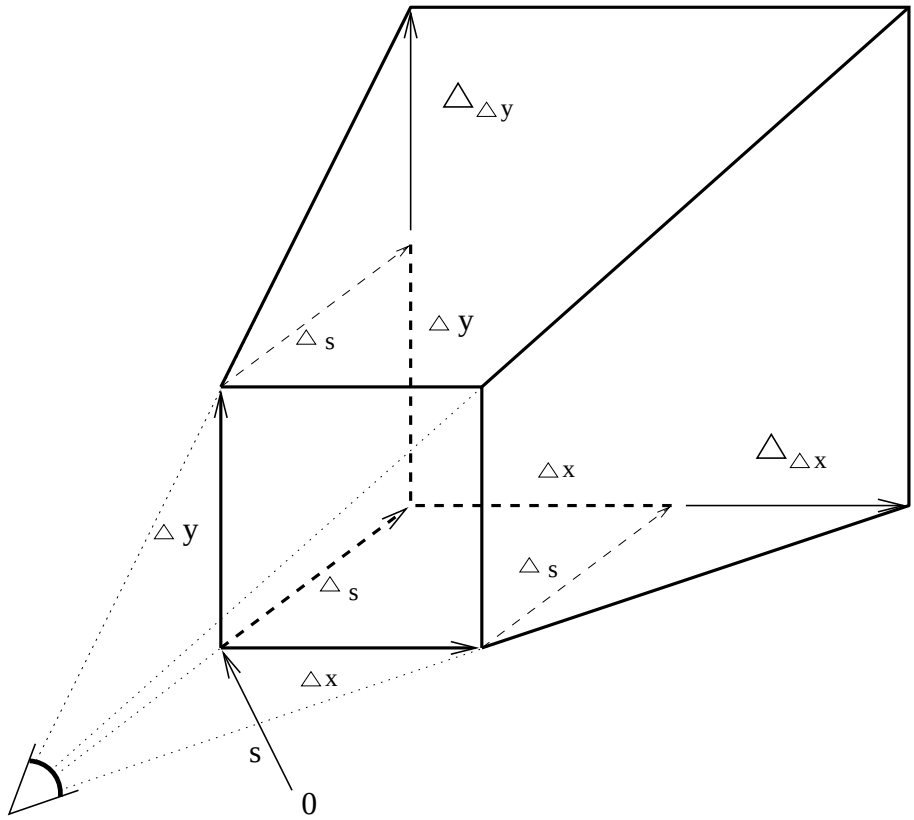


Abbildung 4.10: Benötigte Vektoren für die Strahlenrückverfolgung

oder `hbCombinationSparseGrid` integrieren kann. Auch die Hardware-beschleunigte Interpolationsklasse `hbOglSparseGrid` verwendet für die Beleuchtungsberechnung den Prozessor, so daß die Integration in das Inventor-Fenster nicht praktikabel erscheint.

Aus diesen Gründen ist es besser, von Inventor das aktuelle Viewing-Volume der Kamera abzufragen und aus den hier erhaltenen Werten die Parameter für das Raycasting selbst zu berechnen. Weil das berechnete Bild in einem getrennten Fenster angezeigt wird, stören auch keine Hilfslinien oder Koordinatensysteme des Inventor-Widgets die Sicht.

Als Parameter für die Raycasting-Funktion werden der Vektor $\vec{s}$ auf die linke untere Ecke der vorderen Clip-Ebene, die Differenzvektoren $\Delta\vec{x}$ und $\Delta\vec{y}$ zu der rechten unteren und linken oberen Ecke, sowie der Differenzvektor $\Delta_{\vec{s}}$ zur hinteren Clip-Ebene benötigt. Für die Perspektivenkorrektur werden außerdem die Unterschiede $\Delta_{\Delta\vec{x}}$ und $\Delta_{\Delta\vec{y}}$ der Differenzvektoren $\Delta\vec{x}$ und $\Delta\vec{y}$ zwischen der vorderen und der hinteren Clip-Ebene benötigt. Abbildung 4.10 verdeutlicht den Zusammenhang dieser Vektoren.

Die Berechnung dieser Vektoren gestaltet sich relativ einfach, da Inventor Ebenen- und Linien-Klassen zur Verfügung stellt, die Methoden zur Schnittpunktsberechnung enthalten. Über die Kamera erhält man die für die Berechnung benötigten Clip-Ebenen sowie die Linien durch die vier Eckpunkte der aktuellen Anzeige. Die Schnittpunkte der beiden Ebenen mit diesen vier Linien ergeben genau die Eckpunkte des *Viewing Frustrums*, also des Teils des Volumens,

global function **getViewPos** *Berechne Viewing Frustrum für das gegebene Inventor-Widget*
in: **SoXtExaminerViewer** i: *Inventor-Widget*
out: $\vec{s}$: *Vektor zur linken unteren Ecke der vorderen Clip-Ebene*
$\Delta\vec{x}$, $\Delta\vec{y}$: *Differenzvektoren zur rechten unteren und linken oberen Ecke*
$\Delta_{\vec{s}}$, $\Delta_{\Delta\vec{x}}$, $\Delta_{\Delta\vec{y}}$: *Differenzvektoren von der vorderen zur hinteren Clip-Ebene*

{

SoCamera $c = i \rightarrow$ **getCamera**
SbViewVolume $v = c \rightarrow$ **getViewVolume**

Berechne Entfernung der vorderen und hinteren Clip-Ebene

$$d_{\text{near}} = \max\left(\min\left(v \rightarrow \textbf{getPlane}(c \rightarrow \textbf{nearDistance}) \rightarrow \textbf{getDistanceFromOrigin}, \sqrt{3}\right), -\sqrt{3}\right)$$
$$d_{\text{far}} = \max\left(\min\left(v \rightarrow \textbf{getPlane}(c \rightarrow \textbf{farDistance}) \rightarrow \textbf{getDistanceFromOrigin}, \sqrt{3}\right), -\sqrt{3}\right)$$

SbPlane p_{near} $(v \rightarrow \textbf{zVector},\ d_{\text{near}})$, p_{far} $(v \rightarrow \textbf{zVector},\ d_{\text{far}})$ *Erstelle Clip-Ebenen*

Berechne die Geraden durch die Eckpunkte des Viewing Frustrums

$$v \rightarrow \textbf{projectPointToLine}\left(\binom{0}{0},\ l_{\text{s}}\right)$$
$$v \rightarrow \textbf{projectPointToLine}\left(\binom{1}{0},\ l_{\text{x}}\right), \quad v \rightarrow \textbf{projectPointToLine}\left(\binom{0}{1},\ l_{\text{y}}\right)$$

Schneide Geraden mit den Clip-Ebenen

$$p_{\text{near}} \rightarrow \textbf{intersect}\ (l_{\text{s}}, \vec{s}_{\text{near}})$$
$$p_{\text{near}} \rightarrow \textbf{intersect}\ (l_{\text{x}}, \vec{x}_{\text{near}}),\ p_{\text{near}} \rightarrow \textbf{intersect}\ (l_{\text{y}}, \vec{y}_{\text{near}})$$
$$p_{\text{far}} \rightarrow \textbf{intersect}\ (l_{\text{s}}, \vec{s}_{\text{far}})$$
$$p_{\text{far}} \rightarrow \textbf{intersect}\ (l_{\text{x}}, \vec{x}_{\text{far}}), \quad p_{\text{far}} \rightarrow \textbf{intersect}\ (l_{\text{y}}, \vec{y}_{\text{far}})$$

Erstelle Differenzvektoren

$$\vec{s} = \vec{s}_{\text{near}}$$
$$\Delta\vec{x} = \vec{x}_{\text{near}} - \vec{s}_{\text{near}}, \qquad \Delta\vec{y} = \vec{y}_{\text{near}} - \vec{s}_{\text{near}}$$
$$\Delta_{\vec{s}} = \vec{s}_{\text{far}} - \vec{s}_{\text{near}}$$
$$\Delta_{\Delta\vec{x}} = \vec{x}_{\text{far}} - \vec{s}_{\text{far}} - \Delta\vec{x}, \qquad \Delta_{\Delta\vec{y}} = \vec{y}_{\text{far}} - \vec{s}_{\text{far}} - \Delta\vec{y}$$

}

Abbildung 4.11: Pseudo-Code für die Berechnung des Viewing Frustrums

der das angezeigte Inventor-Widget gerade ausfüllt.

Mit Hilfe dieser Parameter kann dann der in Kapitel 4.5.1 beschriebene Raycaster aufgerufen werden. Abbildung 4.11 zeigt die Routine zur Berechnung der Vektoren in Pseudo-Code. Zur Vereinfachung werden in den meisten Beispielen in Pseudo-Code nur die Klassenabhängigkeiten beachtet, die für das Verständnis unmittelbar notwendig sind.

4.4.4 Hilfesystem

Das Hilfesystem benutzt die Fähigkeit von ViewKit, jedem Event einen Hilfetext zuzuordnen. Die Standard-Hilferoutinen sind allerdings nicht dazu geeignet, einen längeren Text auszugeben oder gar Verweise von einer Hilfeseite auf eine andere anzubieten. Das API des Hilfesystems ist jedoch gut dokumentiert. Es besteht aus drei globalen C-Funktionen, die vom Hilfesystem zur Initialisierung und für die Darstellung eines Hilfefensters aufgerufen werden.

Es existiert noch ein weiteres hierauf basierendes System, das von der SGI-eigenen Hilfe *insight* benutzt wird und sowohl Grafiken als auch Beispiel-Code enthalten kann. Für ein Projekt der Größenordnung dieser Arbeit ist dieses System aber vollkommen überdimensioniert.

Für die Darstellung von Hilfetexten bietet sich HTML als allgemein anerkannter Standard an. Insbesondere die Fähigkeiten, Querverweise und Bilder einzufügen, prädestiniert HTML für diese Anwendung. Die gleichen Seiten können so ebenfalls im WWW präsentiert werden.

Als Browser wird die ViewKit-Klasse `VkWebViewer` eingesetzt, die einen minimalistischen Web-Browser mit HTML-2.0-Unterstützung bietet. Leider hat sich die Aufrufsemantik von Version 1.0 zu 2.0 derartig geändert, daß das Programm nur auf Rechnern mit neuerem IRIX-System compiliert und auch verwendet werden werden kann. Als Alternative wurde deshalb ein Verfahren implementiert, das einen bereits laufenden *Netscape Navigator* als Hilfe-Browser verwendet.

4.5 Raycaster

4.5.1 Implementierungsvarianten

Bei der Implementierung eines Raycasters gibt es zwei prinzipiell verschiedene Varianten, wie die Abbildung 4.12 verdeutlicht. Entweder man betrachtet jeden Strahl für sich und integriert über die Samplingwerte entlang jedes einzelnen Strahls, oder man betrachtet alle Strahlen parallel und integriert sie erst, wenn man alle Samplingwerte innerhalb einer Ebene durch das Volumen erhalten hat.

In einer reinen Softwarelösung sind beide Algorithmen in Bezug auf die Geschwindigkeit gleichwertig, mit dem Unterschied, daß die erste Methode weniger Speicher benötigt. In der Implementierung äußert sich der Unterschied der beiden Varianten im wesentlichen in der Vertauschung der inneren und äußeren Schleife.

Soll jedoch die Hardware die Berechnung der Samplingwerte beschleunigen, darf nicht nur ein Samplingwert pro Aufruf der entsprechenden Routine berechnet werden, da auf diese Weise der Overhead für das Ansprechen der Hardware jeden Geschwindigkeitsgewinn wieder zunichte machen würde. In der zweiten Implementierungsvariante kann je Aufruf der Samplingfunktion eine ganze Ebene durch das Volumen abgetastet werden, so daß hier die Hardware mit ihrer Fähigkeit, texturierte Polygone zu zeichnen, ausgenutzt werden kann. Diese zweite Variante ist in Abbildung 4.13 als Pseudo-Code aufgeführt. Diese Technik wurde für die Visualisierung von vollen Gittern mit der Grafik-Hardware von SGI bereits in [11] vorgestellt.

Da alle Methoden außer der Hardware-beschleunigten Kombinationsmethode von der gleichzeitigen Berechnung mehrerer Interpolationswerte nicht profitieren, enthält die Klasse `hbSparseGrid` eine Routine, die die Interpolation einer ganzen Ebene auf die Interpolation einzelner Werte abbildet. Diese Methode ist ebenfalls in Abbildung 4.13 zu finden. Sie wird an die Subklassen weitervererbt, aber in `hbOglSparseGrid` überschrieben. Die Implementierung in dieser Klasse wird in Kapitel 5.4 genauer betrachtet.

In der tatsächlich verwendeten Routine wird die Funktion für die Beleuchtungsberechnung nur für Pixel aufgerufen, die innerhalb des betrachteten Volumens liegen. Um den Pseudo-Code nicht unnötig kompliziert zu machen, wurde auf die Darstellung dieser Tatsache verzichtet.

Viele der implementierten Beleuchtungsfunktionen benötigen für ihre Berechnungen keine Normalen. Daher hat es sich als sinnvoll erwiesen, die zweite Raycaster-Variante zweimal zu

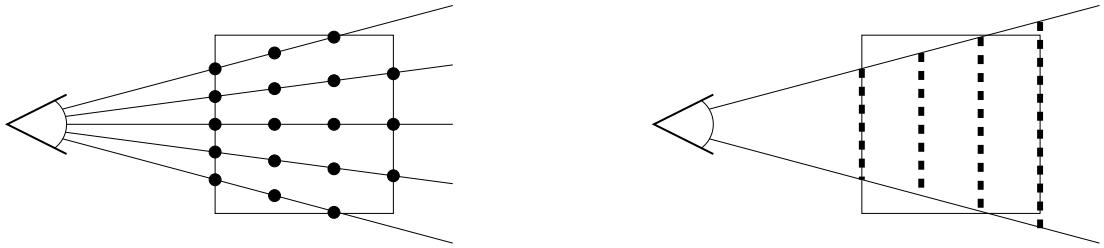


Abbildung 4.12: Prinzipielle Implementierungsvarianten eines Raycasting-Algorithmus

```
global function raycast                 Aufruf des Raycasters
in:             h_x, h_y:               Bitmap-Größe
                h_z:                    Anzahl der zu zeichnenden Ebenen
                s⃗:                      Vektor zur linken unteren Ecke der vorderen Clip-Ebene
                Δx⃗, Δy⃗:                 Differenzvektoren zur rechten unteren / linken oberen Ecke
                Δ_s⃗, Δ_Δx⃗, Δ_Δy⃗:       Differenzvektoren von der vorderen zur hinteren Clip-Ebene
                shade:                  Funktion zur Beleuchtungsberechnung
out:            b[·][·]:                Bitmap mit dem Ergebnis
{
  glXMakeCurrent (...)                  Aktuellen Grafikkontext für Beschleunigung wiederherstellen
  b[·][·] = 0                           Bitmap löschen
  s⃗ := s⃗ + Δ_s⃗ ,   Δx⃗ := Δx⃗ + Δ_Δx⃗ ,   Δy⃗ := Δy⃗ + Δ_Δy⃗   Bei der hinteren Ebene beginnen
  for z = [0 ... h_z]                   Ebene für Ebene berechnen, von hinten nach vorne
      t := calcValueRect (s⃗, Δx⃗, Δy⃗, h_x, h_y)   Interpoliere Daten in Zwischen-Bitmap t
      s⃗ := s⃗ − (1/h_z) Δ_s⃗ ,   Δx⃗ := Δx⃗ − (1/h_z) Δ_Δx⃗ ,   Δy⃗ := Δy⃗ − (1/h_z) Δ_Δy⃗   Zur nächsten Ebene
      for (i, j) = [0 ... h_y − 1] × [0 ... h_x − 1]   Beleuchtungsberechnung und Kopieren in die Bitmap
          shade (t[i][j], (0 0 0), j, i, z, b[i][j])
}

global function calcValueRect           Interpolation eines ganzen Rechtecks in einem dünnen Gitter
                                        (Default-Routine)
in:             s⃗:                      Linke untere Ecke des Rechtecks
                Δx⃗, Δy⃗:                 Differenzvektor zur rechten unteren / linken oberen Ecke
                h_x, h_y:               Größe des Rechtecks in Pixeln
out:            t[·][·]:                Array mit interpolierten Werten
{
  for (i, j) = [0 ... h_y − 1] × [0 ... h_x − 1]   Interpoliere Werte für alle Pixel des Rechtecks
      t[i][j] := calcValue (s⃗ + j (Δx⃗/h_x) + i (Δy⃗/h_y))
}
```

Abbildung 4.13: Pseudo-Code für den Raycaster und Ebeneninterpolation für die Standardmethode

```
global            s_xray:          Skalierungsfaktor

global function   shade            Beleuchtungsberechnung für X-Ray
in:               v:               Interpolierter Datenwert
                  n⃗:               Normale
                  x, y, z:         Aktuelle Position
in/out:           b:               Aktuelle Bitmap-Position
{
  b_r := b_r + 8 · s_xray · v                          Integrationsmethode Röntgen,
  b_g := b_g + 1 · s_xray · v                      einfache Falschfarben-Transferfunktion
  b_b := 0,2
}
```

Abbildung 4.14: Pseudo-Code für die Integrationsmethode Röntgen

implementieren. Eine Implementierung berechnet nur die Funktionswerte, während die andere zusätzlich auf den bereits interpolierten Werten Normalen für die Beleuchtungsberechnung erstellt. Eine weitere mögliche Implementierungsvariante würde die Normalenberechnung im Objektraum durchführen und die erhaltenen Normalen genauso wie die Funktionswerte interpolieren. Diese Variante dürfte deutlich genauere Ergebnisse erzielen, wurde aber noch nicht implementiert. Eine genauere Diskussion der Ergebnisse und noch zu implementierender Varianten findet sich in Kapitel 6 beziehungsweise Kapitel 7.2.

4.5.2 Integrationsmethoden und Beleuchtungsberechnung

Wie bereits in Kapitel 3.4 beschrieben, beeinflussen sich die Wahl der Transfer- und Beleuchtungsfunktion sowie der Integrationsmethode gegenseitig. Zudem sind sie sehr zeitkritisch, da sie in der innersten Schleife des Raycasters ausgeführt werden müssen. Aus diesen Gründen hat es sich als sinnvoll erwiesen, alle drei Schritte zusammenzufassen und sinnvolle Kombinationen dem Benutzer zur Auswahl zu stellen. Derzeit ist noch keine Transferfunktion außer einer einfachen Falschfarbenfunktion für MIP und Röntgen implementiert, so daß für diese beiden Integrationsmethoden keine weitere Auswahl besteht. Für das Emissions-Absorptions-Modell mit ISO-Flächen ist als Beleuchtungsfunktion ein einfaches Lambert-Modell vorgesehen, wie es zum Beispiel in [3, Seiten 723–725] beschrieben ist.

Um die Auswahl der Integrationsmethode möglichst einfach zu gestalten und dennoch für die Programmierung ein gutes Abstraktionsmodell zu erhalten, wurden alle Integrations / Beleuchtungs-Kombinationen in verschiedenen Subklassen der abstrakten Klasse `Lighting` implementiert. Die derzeit aktive Raycaster-Klasse ruft dann in ihrer innersten Schleife eine virtuelle Methode der aktiven Integrationsklasse auf. Die Methode erhält beim Aufruf als Parameter die aktuelle Position, den interpolierten Funktionswert und die Normale an dieser Position, sowie einen Zeiger auf die aktuelle Position im Bildspeicher. Abbildung 4.14 zeigt diese Methode exemplarisch für die Integrationsmethode Röntgen mit einer einfachen Falschfarben-Transferfunktion.

4.6 Dünne Gitter

Alle Implementierungsvarianten der Interpolation auf dünnen Gittern sind in der Klasse `hbSparseGrid` und ihren Subklassen enthalten. Bevor sie für die Interpolation benutzt werden können, müssen sie initialisiert werden. Neben der Gitter- und Datendimension und dem Level für die Interpolation benötigen sie dabei die Eingabedaten.

Nach der Initialisierung müssen die internen Datenstrukturen aufgebaut werden. Dies geschieht über einen Aufruf der virtuellen Methode `setCoeff`(). Die Implementierung für diese Methode ist stark von der verwendeten Interpolationsmethode abhängig. Die Eingabedaten werden nach dem Aufbau der Datenstrukturen nicht mehr benötigt und können freigegeben werden.

Nun können die Objekte für die Interpolation der vorher übermittelten Eingabedaten benutzt werden. Hierzu existiert die mehrfach überladene virtuelle Methode `calcValue`(). Da die zusätzlich implementierten Interpolationsmethoden teilweise nur in der Volumenvisualisierung sinnvoll eingesetzt werden können, sind nicht immer alle überladenen Varianten der Methode implementiert.

4.6.1 Standardmethode

Die Interpolationsklassen für die Standardmethode `hbSparseGrid` wurden bereits vor dieser Arbeit von Christian Teitzel im Rahmen der Visualisierung von Strömungsbändern durch Particle Tracing [17] erstellt. Die Standardmethode ist bedingt durch den Aufbau der Interpolation die einzige der implementierten Methoden, die bei der Interpolation nicht den gleichen Level verwenden muß, mit dem sie erstellt wurde. Bei allen anderen abgeleiteten Klassen muß der gewünschte Level bereits beim Aufbau der internen Datenstrukturen bekannt sein. Zum Beispiel sind die Größen der kleinen vollen Gitter der Kombinationsmethode vom Level abhängig. Da diese nur beim Erstellen des dünnen Gitters mit Funktionswerten gefüllt werden, kann der zu visualisierende Level nach der Initialisierung nicht mehr verändert werden.

Des weiteren ist diese Methode prinzipbedingt die einzige, bei der die bereits in Kapitel 2.4 beschriebene Beschleunigung über die Toleranzgrenze ε implementiert ist. Außerdem verfügt bedingt durch die Entstehungsgeschichte nur sie über die überladenen Interpolationsmethoden, die zusätzlich zum Datenwert auch noch die Jakobimatrix an der betrachteten Stelle generieren.

Für die Interpolation enthält die Klasse `hbSparseGrid` eine Tabelle mehrerer Objekte der Klasse `hbLevel`. Jedes Objekt dieser Klasse enthält mit einer Tabelle von Objekten der Klasse `hbSubSpace` alle Unterräume, die genau einem Level entsprechen. Bei der Interpolation werden alle Beiträge der einzelnen Level aufaddiert, welche sich durch die Summe der Beiträge der Unterräume ergeben. In den Unterräumen muß noch die Basisfunktion ausgewählt und ausgewertet werden, innerhalb deren Träger der zu interpolierende Punkt liegt.

Aufwendig gestaltet sich bei der Standardmethode die Interpolation auf dem Rand nicht verschwindender Abbildungen. Hierfür wird bei der Initialisierung für jeden Rand ein zweidimensionales dünnes Gitter erzeugt, das die Randwerte enthält und zusätzlich ausgewertet werden muß. Dabei sind eine Reihe von Sonderfällen zu beachten, die die Programmierung relativ aufwendig machen.

4.6.2 Kombinationsmethode

Bei der Kombinationsmethode in der Klasse `hbCombinationSparseGrid` werden für die internen Datenstrukturen mehrere dreidimensionale Arrays angelegt, die mit den bereits nach (2.14) gewichteten Eingabedaten gefüllt werden. Der wesentliche Aufwand bei der Interpolation besteht dann nur noch in einer Schleife über alle Arrays, in der die zum aktuellen Punkt gehörende Zelle ermittelt und die Beiträge der kleinen vollen Gitter nach der trilinearen Interpolation aufaddiert werden müssen.

Da die Interpolation so einfach ist, kann sie gut optimiert werden. Es hat sich gezeigt, daß die Optimierung eine sehr große Rolle spielt. Obwohl die Kombinationsmethode im Aufwand, wie in Kapitel 2.5 gezeigt, für Level $\ll 50$ gegenüber der Standardmethode in der Anzahl wesentlicher arithmetischer Operationen deutlich unterlegen ist, ist sie in der Praxis auch in diesen ungünstigen Fällen um circa den Faktor 10 schneller als die Standardmethode. Der Pseudo-Code für die Kombinationsmethode ist in Abbildung 4.15 dargestellt.

Da die Beschleunigung der Kombinationsmethode durch die Hardware einen Schwerpunkt dieser Arbeit darstellt, ist diesem Ansatz ein eigenes Kapitel 5 gewidmet.

```
global            n_0:                 Anzahl der vollen Gitter
                  g_x[·], g_y[·], g_z[·]: Level der vollen Gitter
                  a[·][·][·][·]:       Array der dreidimensionalen vollen Gitter

global function   setCoeff             Erstellen der vollen Gitter für die Kombinationsmethode
in:               d:                   Eingabedaten [0,1]^3 → [0,1]
                  l:                   Level
{
  n_0 := 0
  for (i,j) = [1...l] × [1...l]                                   Alle Texturen für i+j+k ≤ l+2 erzeugen
      if (i+j+1 ≤ l+2)
         dataToArray (n_0, d, i, j, l+2-i-j, 1) ,      n_0 := n_0 + 1
  for (i,j) = [1...l] × [1...l]                                   Alle Texturen für i+j+k ≤ l+1 erzeugen
      if (i+j+1 ≤ l+1)
         dataToArray (n_0, d, i, j, l+1-i-j, -2) ,     n_0 := n_0 + 1
  for (i,j) = [1...l] × [1...l]                                   Alle Texturen für i+j+k ≤ l erzeugen
      if (i+j+1 ≤ l)
         dataToArray (n_0, d, i, j, l-i-j, 1) ,        n_0 := n_0 + 1
}                                                   n_0 enthält nun die Gesamtzahl an vollen Gittern

function dataToArray      Erstelle kleineres volles Gitter aus der Datenfunktion
in:       n:              Gitter-Nummer
          d:              Eingabedaten [0,1]^3 → [0,1]
          l_x, l_y, l_z:  Level in Richtung der drei Achsen
          f:              Skalierungsfaktor
{
  g_x[n] := l_x ,     g_y[n] := l_y ,     g_z[n] := l_z          Größe für die Benutzung in calcValue() sichern
  a[n][·][·][·] := 0
  for (i,j,k) = [0...2^{l_z}] × [0...2^{l_y}] × [0...2^{l_x}]     Eingabedaten abtasten und volle Gitter erstellen
      a[n][i][j][k] := f · d(k/2^{l_x}, j/2^{l_y}, i/2^{l_z})
}

global function calcValue     Interpolation mit der Kombinationsmethode
in:             s:            Zu interpolierende Position s ∈ [0,1]^3
out:            v:            Interpolierter Wert
{
  v := 0
  for i = [0...n_0 - 1]                                           Durchlaufe alle vollen Gitter
      p_x := ⌊s_x · 2^{g_x[i]}⌋ ,   p_y := ⌊s_y · 2^{g_y[i]}⌋ ,   p_z := ⌊s_z · 2^{g_z[i]}⌋      Ermittle Zellennummer
      δ_x := s_x · 2^{g_x[i]} - p_x ,  δ_y := s_y · 2^{g_y[i]} - p_y ,  δ_z := s_z · 2^{g_z[i]} - p_z   Berechne Position in Zelle
                                                                  Interpoliere im vollen Gitter trilinear
      v := v + a[i][p_x][p_y][p_z]          · (1-δ_x) · (1-δ_y) · (1-δ_z)
             + a[i][p_x+1][p_y][p_z]        · δ_x     · (1-δ_y) · (1-δ_z)
             + a[i][p_x][p_y+1][p_z]        · (1-δ_x) · δ_y     · (1-δ_z)
             + a[i][p_x+1][p_y+1][p_z]      · δ_x     · δ_y     · (1-δ_z)
             + a[i][p_x][p_y][p_z+1]        · (1-δ_x) · (1-δ_y) · δ_z
             + a[i][p_x+1][p_y][p_z+1]      · δ_x     · (1-δ_y) · δ_z
             + a[i][p_x][p_y+1][p_z+1]      · (1-δ_x) · δ_y     · δ_z
             + a[i][p_x+1][p_y+1][p_z+1]    · δ_x     · δ_y     · δ_z
}
```

Abbildung 4.15: Pseudo-Code für die Interpolation mit der Kombinationsmethode

Kapitel 5

Einblick in die Hardware-Beschleunigung der Kombinationsmethode

5.1 Funktionsprinzip

Die Hardware-beschleunigte Interpolation auf dünnen Gittern macht sich die Tatsache zu Nutze, daß als Basisfunktionen in der Kombinationsmethode die sehr einfachen Funktionen der trilinearen Interpolation zum Einsatz kommen. Betrachtet man in Abbildung 5.1 die Pipeline des Grafikstandards OpenGL, so erkennt man, daß sich eine Reihe von Zwischenschritten für die Interpolation auf dünnen Gittern mit der Kombinationsmethode nutzen lassen.

OpenGL unterstützt die Texturierung von geometrischen Primitiven durch mehrere Schnittstellen. Erste Hardware-Beschleunigungsansätze arbeiteten nur auf Flächentexturen und benutzten nur Nearest-Neighbor-Interpolation, bestachen aber durch ihre Geschwindigkeit gegenüber den reinen Software-Lösungen. Mit fortschrittlicherer Hardware konnte auf Texturen auch bilinear interpoliert werden und schließlich wurden auch *MIP-Maps* unterstützt. Da die beiden Interpolationswerte verschiedener MIP-Levels auch noch linear interpoliert werden, muß moderne Grafik-Hardware für deren Unterstützung dreifach linear interpolieren. Daher können von diesen modernen Grafikkarten Volumen-Texturen mit trilinearer Interpolation ohne großen Aufwand unterstützt werden. Genauere Informationen über MIP-Mapping sowie dessen Implementierung in OpenGL findet man unter anderem in [13].

Da in dem Programm `sgrid` die Sicht auf das Volumen über ein Inventor-Widget gewählt wird, könnte man die Transformation und das Clipping auch OpenGL überlassen. Allerdings würde in diesem Fall die reine Software-Lösung ausscheiden, und man hätte sich voll und ganz auf die Hardware festgelegt. Zudem müßten immer noch die zu zeichnenden Ebenen bestimmt werden, was vom Aufwand dem Ermitteln des Viewing Frustrums in etwa entspricht. Aus diesen Gründen wurde nur der Teil der Pipeline genutzt, der direkt mit dem Raster-Subsystem gekoppelt ist.

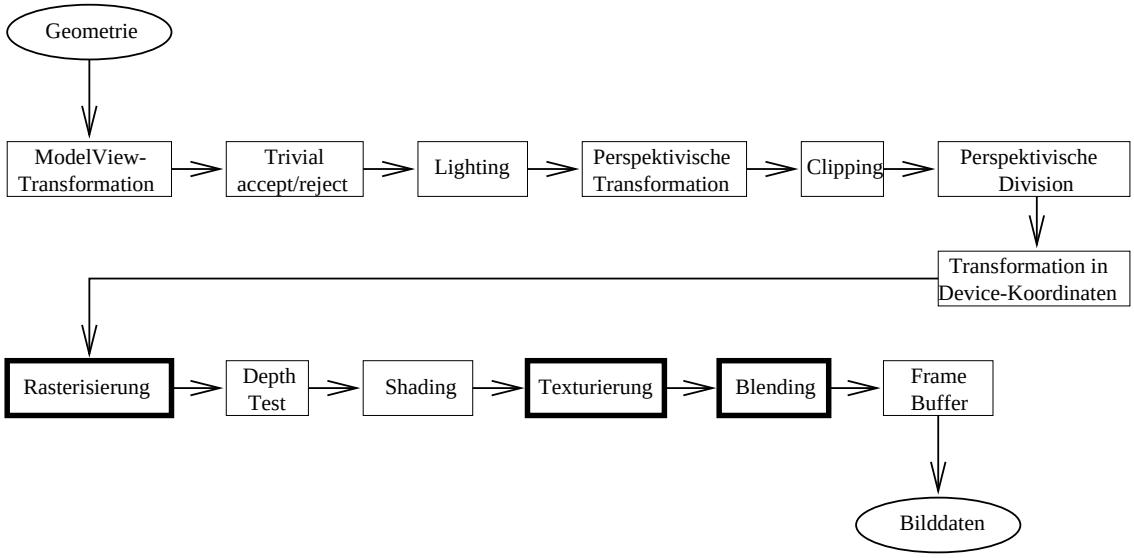


Abbildung 5.1: Der für die Hardware-Beschleunigung relevante Teil der OpenGL-Grafikpipeline

Um durch die Ausnutzung der Hardware auch wirklich eine Beschleunigung der Interpolation zu erzielen, ist es unumgänglich, nicht nur einen Wert zu interpolieren, sondern gleich ein ganzes Rechteck in beliebiger Lage durch das Volumen zu berechnen. Dies hängt damit zusammen, daß die Texturierung einzelner Punkte in OpenGL durch ihre geringe Signifikanz in der Computergrafik nicht optimiert ist und die gleiche Anzahl an Initialisierungs- und Zeichenroutinen aufgerufen werden muß, wie bei der Texturierung eines ganzen Vielecks. Auf leistungsfähiger Grafik-Hardware können viele Pixel zur gleichen Zeit texturiert werden, und somit steigt der Gesamtaufwand gegenüber einem Pixel nicht wesentlich.

Die Interpolation eines Funktionswertes an einer gegebenen Stelle gemäß (2.14) benötigt trilinear interpolierte Funktionswerte mehrerer voller Gitter. Werden beim Laden des dünnen Gitters bereits mehrere Volumentexturen erstellt, die als Texturwerte die Funktionswerte der Eingangsdaten an den Gitterstellen erhalten, kann nun jede dieser Texturen dazu verwendet werden, schnell innerhalb eines dünnen Gitters gemäß (2.15) trilinear zu interpolieren.

Die so erhaltenen Werte müssen zum Teil addiert und zum Teil subtrahiert werden. Da im Framebuffer und im Texturspeicher alle Werte durch die Grafik-Hardware nach der OpenGL-Spezifikation auf den Wertebereich $[0, 1]$ eingeschränkt sind, kann man nicht einfach die Funktionswerte vor dem Transport in den Texturspeicher negativ skalieren. Erst während des *Blending*-Schritts werden daher die Pixel der texturierten Rechtecke entsprechend dem Vorzeichen der Gleichung (2.14) zu den bisherigen Werten im Framebuffer addiert beziehungsweise von den vorherigen Werten subtrahiert.

Eine weitere Möglichkeit, die Hardware für die Additionen und Subtraktionen zu benutzen, wäre die Verwendung des *Accumulation Buffers* von OpenGL. Auf diese Methode wird noch nochmal in Kapitel 7.2 verwiesen.

Sind Rechtecke mit allen Texturen gezeichnet worden, können die Pixelwerte des Framebuffers ausgelesen und mit der eingestellten Methode integriert werden. Auch hier kann noch, wie in Kapitel 7.2 zu lesen, die Hardware ausgenutzt werden, um die Beleuchtungsberechnung und Integration zu beschleunigen.

5.2 Benötigte OpenGL-Funktionen und -Erweiterungen

Da der Benutzer nicht unbedingt während einer Berechnung die Vielzahl gezeichneter texturierter Polygone zu sehen bekommen soll, ist es sinnvoll, einen nicht sichtbaren Teil des Framebuffers für die Zeichenoperationen zu verwenden. OpenGL bietet für diese Zwecke den *Auxiliary Buffer* an. Allerdings ist bei diesem Speicherbereich genauso wie bei einem „gewöhnlichen“ Ausschnitts des Framebuffers nicht gewährleistet, daß der Inhalt durch über den Bildschirm bewegte Fenster nicht zerstört wird.

SGI hat die Problematik erkannt und bietet auf ihren Workstations als eine experimentelle Erweiterung (SGIX) zu OpenGL 1.0 die sogenannten *P-Buffer* an. Die Erstellung eines P-Buffers und des dazugehörigen Kontextes kann nur über die ebenfalls experimentellen Erweiterungsfunktionen `glXChooseFBConfigSGIX`(), `glXCreateContextWithConfigSGIX`() und `glXCreateGLXPbufferSGIX`() erfolgen und unterscheidet sich daher deutlich von der bisher vorhandenen Standardmethode in OpenGL.

Volumen-Texturen sind erst seit der OpenGL-Version 1.1 fester Bestandteil des Standards. Bereits vorher gab es eine Reihe von Herstellern wie zum Beispiel SGI, die eine gemeinsam entwickelte Erweiterung (EXT) zur Verwendung von 3D-Texturen implementierten.

Um die Texturen in einem genau spezifizierten Format speichern zu können, wurden einige Erweiterungen der bekannten Pixelformate eingeführt. Auf diese Weise kann der Hardware über die Formatbezeichnung `GL_LUMINANCE16_EXT` mitgeteilt werden, daß die Texturdaten wenn möglich als 16 Bit-Werte gespeichert werden sollen. Allerdings ist der Spezifikation nach die Hardware berechtigt, eine andere Speicherart vorzuziehen, wenn sie das gewünschte Format nicht unterstützt. 16 Bit reichen für die Texturdaten aufgrund der geringen Framebuffer-Tiefe voll aus, wie noch in Kapitel 5.5 verdeutlicht wird.

Um mehrere Texturen ansprechen zu können, mußte in OpenGL 1.0 bei jedem Umschalten der Textur die gewünschte Textur neu geladen werden. Bei schnellen Grafikkarten geschieht dies zwar mit hoher Geschwindigkeit, aber für eine Vielzahl von Texturen, die jeweils nur für genau ein Polygon gebraucht werden, ist dieses Verfahren indiskutabel. Deshalb wurde mit den *Texture Names* eine Erweiterung eingeführt, die dieses Problem adressiert und inzwischen in OpenGL 1.1 aufgenommen wurde. Hierbei kann jeder geladenen Textur eine Nummer zugeordnet werden. Nur wenn eine neue Textur mit einer bereits vergebenen Nummer geladen wird, wird die alte Textur gelöscht. OpenGL kümmert sich dabei automatisch um die Verwaltung des Texturspeichers und behält so viele Texturen wie möglich im Speicher der Grafikkarte.

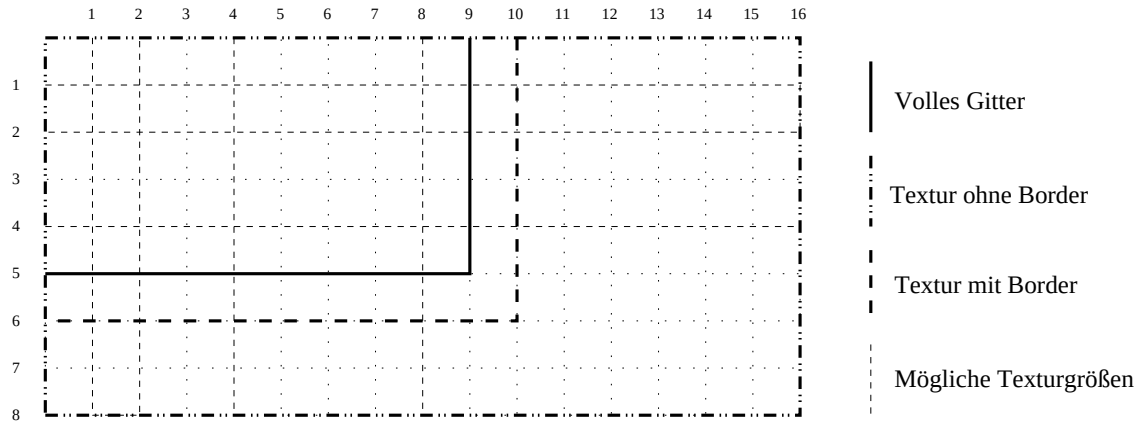


Abbildung 5.2: Beispiel eines vollen Gitter und der hierfür benötigten Texturgrößen

Die texturierten Polygone müssen im Framebuffer addiert beziehungsweise subtrahiert werden. Diese Blending-Operationen sind keine Standardoperationen in der Computergrafik. Allerdings existiert eine allgemein anerkannte Erweiterung des OpenGL-Standards, der die beiden Operatoren `GL_FUNC_ADD_EXT` und `GL_FUNC_REVERSE_SUBTRACT_EXT` zur Verfügung stellt.

5.3 Laden der Texturen

Um die vollen Gitter der Kombinationsmethode in den Texturspeicher zu laden, müssen im wesentlichen die Funktionswerte der Eingabedaten an den Gitterstellen abgetastet und für den Aufruf der Funktion `glTexImage3DEXT()` umformatiert werden.

Besondere Beachtung verdient dabei die Berechnung der Größe der zu ladenden Textur. Wie in Abbildung 5.2 gezeigt, verlangt der OpenGL-Standard, daß alle Texturen in jeder Dimension eine Größe haben, die als Zweierpotenz geschrieben werden kann. Die vollen Gitter, die für die Kombinationsmethode benötigt werden, haben jedoch alle eine Größe der Form $2^i + 1 \times 2^j + 1 \times 2^k + 1$. Um diese Gitter in Texturen umwandeln zu können, müssen folglich Texturen der Größe $2^{i+1} \times 2^{j+1} \times 2^{k+1}$ alloziert werden. Die Texturen werden dann mit Nullen aufgefüllt, was auch *padding* genannt wird. Dadurch wird sehr viel Texturspeicher verschwendet. Auch auf Grafiksystemen wie der Reality Engine ist dieser eine knappe Ressource, und die maximal betrachtbare Volumengröße wird durch die schlechte Ausnutzung des Speichers um ein bis zwei Level reduziert. Theoretisch erlaubt OpenGL Texturen mit einem, in jeder Dimension in jeweils beiden Richtungen einen Pixel breiten Rand zu versehen, allerdings wird dieser sogenannte *Border* bis heute noch von keiner Hardware unterstützt.

Bei der Größenberechnung ist zu beachten, daß entsprechend [12] die Grafik-Hardware der Maximum Impact Texturen mit einer Größe ≤ 16 Pixeln in der Richtung der x-Achse eventuell falsch interpoliert. Folglich müssen die Texturen gegebenenfalls künstlich vergrößert werden.

Da im Framebuffer alle Werte auf den Bereich $[0, 1]$ durch das sogenannte *clamping* eingeschränkt werden, müssen alle Texturwerte mit dem Faktor $1/n_+$ skaliert werden, wobei n_+

die Anzahl der Texturen angibt, die einen positiven Beitrag zu (2.14) haben. Diese Tatsache wird in Kapitel 5.5 nochmal aufgegriffen.

Der Pseudo-Code in Abbildung 5.3 verdeutlicht den Ablauf bei der Erstellung der Texturen.

5.4 Interpolation innerhalb eines dünnen Gitters

Für die im Pseudo-Code in Abbildung 5.5 dargestellte eigentliche Interpolation eines Rechtecks innerhalb des dünnen Gitters wird zu Beginn der Framebuffer gelöscht, für das Blending die Addition der Werte im Framebuffer aktiviert und der sichtbare Bereich in der Größe des zu interpolierenden Rechtecks festgelegt.

In der Hauptschleife wird mit jeder Textur ein Rechteck in den Framebuffer gezeichnet. Dafür müssen die Gitterkoordinaten auf Texturkoordinaten umgerechnet werden. Abbildung 5.4 verdeutlicht den Zusammenhang der beiden Koordinatensysteme.

Da die Werte im Framebuffer nie negativ werden dürfen, müssen zunächst alle Texturen aktiviert werden, die positiv zum Interpolationswert beitragen. Dann kann das Compositing auf Subtraktion geschaltet und schließlich alle negativ beitragenden Texturen gezeichnet werden. Diese Reihenfolge hat gleichzeitig noch den Vorteil, daß nicht mehrfach zwischen Addition und Subtraktion umgeschaltet werden muß. Allerdings wird dieser Vorteil durch die benötigte Skalierung wieder mehr als aufgewogen. In Kapitel 5.5 wird hierauf näher eingegangen.

Momentan nimmt die Interpolationsroutine noch keinerlei Rücksicht auf die Grenzen des Volumens. Da der im Pseudo-Code in Abbildung 4.13 beschriebene Raycaster in der wirklichen Implementierung die Beleuchtungsberechnung und Integration nur auf Pixel durchführt, die tatsächlich innerhalb des Volumens liegen, spielt dies in der Praxis keine Rolle. Im Pseudo-Code des Raycasters ist diese Abfrage allerdings zur Vereinfachung nicht enthalten.

Nachdem am Ende der Schleife auf die Grafik-Hardware gewartet wurde, wird der Framebuffer ausgelesen und die Ergebnisse um den Faktor n_+ wieder auf den kompletten Wertebereich zurückskaliert.

Die so erhaltenen Werte können nun dem Raycaster übergeben werden. In Kapitel 4.5.1 wurde beschrieben, wie die Ebenen mit Samplingwerten vom Raycaster verarbeitet werden.

5.5 Fehlerabschätzung

Wie in Kapitel 5.3 bereits beschrieben, müssen alle Texturdaten um den Faktor $1/n_+$ skaliert werden, damit keine Verfälschung der Werte durch das Clamping im Framebuffer auf den Wertebereich $[0, 1]$ auftritt. Durch diese Skalierung werden der Genauigkeit der Hardware-beschleunigten Kombinationsmethode strenge Grenzen gesetzt. In den heutigen OpenGL-Implementierungen ist auf SGI-Maschinen jeder verfügbare Framebuffer auf 12 Bit pro Farbkanal beschränkt. Es existieren bis heute keine Ansätze, weniger Farbkanäle zu allozieren und

```
global            n_0:                 Anzahl der vollen Gitter
                  n_+:                 Anzahl der Gitter mit positivem Beitrag zu (2.14)
                  g_x[·], g_y[·], g_z[·]: Level der vollen Gitter

global function   setCoeff             Erstellen der vollen Gitter für die Hardware-beschleunigte Kombinationsmethode
in:               d:                   Eingabedaten [0,1]^3 → [0,1]
                  l:                   Level
{
  glXMakeCurrent (...)                                                    Aktuellen Grafikkontext setzen
  n_0 := 0,   n_+ := 0
  for (i,j) = [1...l] × [1...l]        Die Anzahl der Gitter zählen, die einen positiven Beitrag zum Funktionswert haben
      if (i+j+1 ≤ l+2)                 Diese Zahl wird in dataToTexture() benötigt und daher als erstes berechnet
         n_+ := n_+ + 1
  for (i,j) = [1...l] × [1...l]
      if (i+j+1 ≤ l)
         n_+ := n_+ + 1

  for (i,j) = [1...l] × [1...l]                Nun die Texturen mit einem positiven Beitrag generieren
      if (i+j+1 ≤ l+2)
         dataToTexture (n_0, d, i, j, l+2-i-j, 1) ,   n_0 := n_0 + 1
  for (i,j) = [1...l] × [1...l]
      if (i+j+1 ≤ l)
         dataToTexture (n_0, d, i, j, l-i-j, 1) ,     n_0 := n_0 + 1
  for (i,j) = [1...l] × [1...l]        Nun die mit 2 skalierten Texturen mit einem negativen Beitrag generieren
      if (i+j+1 ≤ l+1)
         dataToTexture (n_0, d, i, j, l+1-i-j, 2) ,   n_0 := n_0 + 1
}                                                      n_0 enthält nun die Gesamtzahl an vollen Gittern

function dataToTexture    Erstelle kleineres volles Gitter aus der Datenfunktion und transportiere es
                          in den Texturspeicher
in:       n:              Gitter-Nummer
          d:              Eingabedaten [0,1]^3 → [0,1]
          l_x, l_y, l_z:  Level in Richtung der drei Achsen
          f:              Skalierungsfaktor
{
  glBindTextureEXT (GL_TEXTURE_3D_EXT, n)                                Zu speichernde Textur auswählen
  glTexParameteri (...)                                                  Weitere Texturparameter setzen
  glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE)           Modulation-Mode setzen

  g_x[n] := l_x,     g_y[n] := l_y,     g_z[n] := l_z                   Größe für die Benutzung in calcValueRect() sichern
  s_x := 2^(l_x+1),  s_y := 2^(l_y+1),  s_z := 2^(l_z+1)   Texturen müssen in jeder Richtung die Größe 2^i besitzen (siehe Text)
  t[·] := 0
  for (i,j,k) = [0...2^l_z] × [0...2^l_y] × [0...2^l_x]   Eingabedaten samplen und für TexImage3DEXT() formatieren
      t[k + s_x(j + s_y i)] := f/n_+ · d(k/2^l_x, j/2^l_y, i/2^l_z)   Die Datenwerte müssen mit 1/n_+ skaliert werden (siehe Text)
                                                                       Und nun die Daten in den Texturspeicher transportieren
  glTexImage3DEXT (GL_TEXTURE_3D_EXT, 0, GL_LUMINANCE16_EXT, s_x, s_y, s_z, 0, GL_LUMINANCE, GL_FLOAT, t)
}
```

Abbildung 5.3: Pseudo-Code für die Erstellung der Texturen

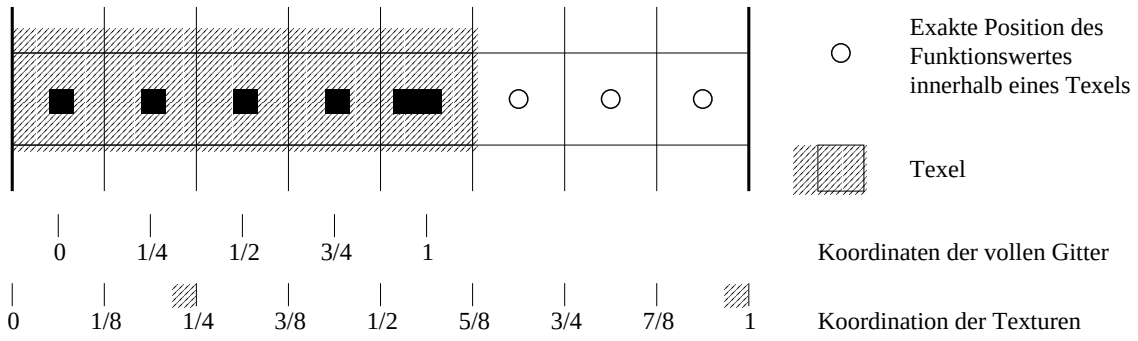


Abbildung 5.4: Umrechnung der Gitter- in Texturkoordinaten für eine Textur der Größe 5 (Level 2)

---

global function **calcValueRect** — *Interpolation eines ganzen Rechtecks auf einem dünnen Gitter mit Hardware-Beschleunigung*

in: $\vec{s}$: *Linke untere Ecke des Rechtecks*

$\Delta\vec{x}$, $\Delta\vec{y}$: *Differenzvektor zur rechten unteren / linken oberen Ecke*

h_x, h_y: *Größe des Rechtecks in Pixeln*

out: $t[\cdot][\cdot]$: *Array mit interpolierten Werten*

{

glViewport $(0,\ 0,\ h_x,\ h_y)$ — *Zeichenbereich festlegen*

glClear (GL_COLOR_BUFFER_BIT) — *Framebuffer löschen*

glBlendEquationEXT (GL_FUNC_ADD_EXT) — *Beim Blending die gezeichneten Texturen im Framebuffer aufaddieren*

for $i\ =\ [0 \ldots n_0 - 1]$

$b_x := \frac{1}{4 \cdot 2^{g_x[i]}}$, $b_y := \frac{1}{4 \cdot 2^{g_y[i]}}$, $b_z := \frac{1}{4 \cdot 2^{g_z[i]}}$ — *Texturkoordinaten-Offset berechnen*

glBindTextureEXT (GL_TEXTURE_3D_EXT, i) — *Die richtige vorher geladene Textur auswählen*

if $(i = n_+)$ — *Wenn alle Texturen mit einem positiven Beitrag zu (2.14) gezeichnet worden sind,*

glBlendEquationEXT (GL_FUNC_REVERSE_SUBTRACT_EXT) — *alle weiteren vom Framebuffer subtrahieren*

glBegin (GL_TRIANGLE_STRIP) — *Ein texturiertes Rechteck zeichnen, das den sichtbaren Bereich ausfüllt*

glTexCoord3f $\left(\frac{s_x}{2} + b_x, \frac{s_y}{2} + b_y, \frac{s_z}{2} + b_z\right)$, **glVertex2f** $(0,\ 0)$

glTexCoord3f $\left(\frac{s_x + \Delta\vec{x}_x}{2} + b_x, \frac{s_y + \Delta\vec{x}_y}{2} + b_y, \frac{s_z + \Delta\vec{x}_z}{2} + b_z\right)$, **glVertex2f** $(1,\ 0)$

glTexCoord3f $\left(\frac{s_x + \Delta\vec{y}_x}{2} + b_x, \frac{s_y + \Delta\vec{y}_y}{2} + b_y, \frac{s_z + \Delta\vec{y}_z}{2} + b_z\right)$, **glVertex2f** $(0,\ 1)$

glTexCoord3f $\left(\frac{s_x + \Delta\vec{x}_x + \Delta\vec{y}_x}{2} + b_x, \frac{s_y + \Delta\vec{x}_y + \Delta\vec{y}_y}{2} + b_y, \frac{s_z + \Delta\vec{x}_z + \Delta\vec{y}_z}{2} + b_z\right)$, **glVertex2f** $(1,\ 1)$

glEnd ()

glFlush () — *Auf die Grafikpipeline warten*

glReadPixels $(0,\ 0,\ h_x,\ h_y,$ GL_RED, GL_FLOAT, t) — *Ergebnis von der Grafik-Hardware holen*

for $(i, j)\ =\ [0 \ldots h_y - 1]\ \times\ [0 \ldots h_x - 1]$

$t[i][j] := n_+ \cdot t[i][j]$ — *Endgültiges Ergebnis auf den richtigen Wertebereich skalieren*

}

Abbildung 5.5: Pseudo-Code für die Interpolation mit der Hardware-beschleunigten Kombinationsmethode

Level	n_+	8 Bit Farbtiefe	12 Bit Farbtiefe	16 Bit Farbtiefe	32 Bit Farbtiefe
1	1	8,0 Bit $\hat{=}$ $10^{-2,4}$	12,0 Bit $\hat{=}$ $10^{-3,6}$	16,0 Bit $\hat{=}$ $10^{-4,8}$	32,0 Bit $\hat{=}$ $10^{-9,6}$
2	3	6,4 Bit $\hat{=}$ $10^{-1,9}$	10,4 Bit $\hat{=}$ $10^{-3,1}$	14,4 Bit $\hat{=}$ $10^{-4,3}$	30,4 Bit $\hat{=}$ $10^{-9,2}$
3	7	5,2 Bit $\hat{=}$ $10^{-1,6}$	9,2 Bit $\hat{=}$ $10^{-2,8}$	13,2 Bit $\hat{=}$ $10^{-4,0}$	29,2 Bit $\hat{=}$ $10^{-8,8}$
4	13	4,3 Bit $\hat{=}$ $10^{-1,3}$	8,3 Bit $\hat{=}$ $10^{-2,5}$	12,3 Bit $\hat{=}$ $10^{-3,7}$	28,3 Bit $\hat{=}$ $10^{-8,5}$
5	21	3,6 Bit $\hat{=}$ $10^{-1,1}$	7,6 Bit $\hat{=}$ $10^{-2,3}$	11,6 Bit $\hat{=}$ $10^{-3,5}$	27,6 Bit $\hat{=}$ $10^{-8,3}$
6	31	3,0 Bit $\hat{=}$ $10^{-0,9}$	7,0 Bit $\hat{=}$ $10^{-2,1}$	11,0 Bit $\hat{=}$ $10^{-3,3}$	27,0 Bit $\hat{=}$ $10^{-8,1}$
7	43	2,6 Bit $\hat{=}$ $10^{-0,8}$	6,6 Bit $\hat{=}$ $10^{-2,0}$	10,6 Bit $\hat{=}$ $10^{-3,2}$	26,6 Bit $\hat{=}$ $10^{-8,0}$
8	57	2,2 Bit $\hat{=}$ $10^{-0,6}$	6,2 Bit $\hat{=}$ $10^{-1,9}$	10,2 Bit $\hat{=}$ $10^{-3,1}$	26,2 Bit $\hat{=}$ $10^{-7,9}$
9	73	1,8 Bit $\hat{=}$ $10^{-0,5}$	5,8 Bit $\hat{=}$ $10^{-1,7}$	9,8 Bit $\hat{=}$ $10^{-3,0}$	25,8 Bit $\hat{=}$ $10^{-7,8}$
10	91	1,5 Bit $\hat{=}$ $10^{-0,5}$	5,5 Bit $\hat{=}$ $10^{-1,7}$	9,5 Bit $\hat{=}$ $10^{-2,9}$	25,5 Bit $\hat{=}$ $10^{-7,7}$
11	111	1,2 Bit $\hat{=}$ $10^{-0,4}$	5,2 Bit $\hat{=}$ $10^{-1,6}$	9,2 Bit $\hat{=}$ $10^{-2,8}$	25,2 Bit $\hat{=}$ $10^{-7,6}$
12	133	0,9 Bit $\hat{=}$ $10^{-0,3}$	4,9 Bit $\hat{=}$ $10^{-1,5}$	8,9 Bit $\hat{=}$ $10^{-2,7}$	24,9 Bit $\hat{=}$ $10^{-7,5}$
15	211	0,3 Bit $\hat{=}$ $10^{-0,1}$	4,3 Bit $\hat{=}$ $10^{-1,3}$	8,3 Bit $\hat{=}$ $10^{-2,5}$	24,3 Bit $\hat{=}$ $10^{-7,3}$
20	381	—	3,4 Bit $\hat{=}$ $10^{-1,0}$	7,4 Bit $\hat{=}$ $10^{-2,2}$	23,4 Bit $\hat{=}$ $10^{-7,0}$

Tabelle 5.1: Genauigkeit der Interpolation beziehungsweise Anzahl gültiger Bits im Framebuffer für unterschiedliche Level und Farbtiefen

diese dafür mit einer größeren Bittiefe zu verwenden. Ein Grund mag in der verwendeten Hardware liegen, die auf Multiplikationen mit maximal 12 Bit pro Multiplikand optimiert ist.

Die Skalierung reduziert die Anzahl der für die Berechnung zur Verfügung stehenden Bits auf $12 - \log_2 n_+$. Für Level 10 bedeutet dies, daß 91 von 136 Funktionen einen positiven Beitrag zu (2.14) haben, und durch die Skalierung beinahe 7 Bit an Genauigkeit verloren gehen. Somit kann auf heute verfügbarer Hardware nur eine Genauigkeit von gut 5 Bit erreicht werden. In Tabelle 5.1 sind für verschiedene Level die effektive Genauigkeit für verschiedene Framebuffer-Tiefen aufgeführt. Man beachte, daß die Werte für 16 und 32 Bit Framebuffer-Tiefe sowie für die Level 12, 15 und 20 derzeit rein theoretischer Natur sind, da noch keine Architekturen mit diesen Farbtiefen beziehungsweise Texturgrößen zur Verfügung stehen.

Wenn jedoch alle positiv und negativ beitragenden Texturen nicht nacheinander, sondern im Wechsel gezeichnet werden, könnte dies die notwendige Skalierung reduzieren. Diese interessante Fragestellung wurde jedoch aus Zeitgründen nicht mehr vollständig innerhalb dieser Arbeit analysiert. Schließlich gilt es zu beachten, daß die Werte im Framebuffer nie negativ werden können. Man erinnere sich daran, daß dies auch der Hauptgrund dafür ist, daß zuerst alle texturierten Polygone mit einem positivem Beitrag zum Interpolationswert gezeichnet werden und erst dannach jene mit einem negativem Beitrag.

Kapitel 6

Ergebnisse

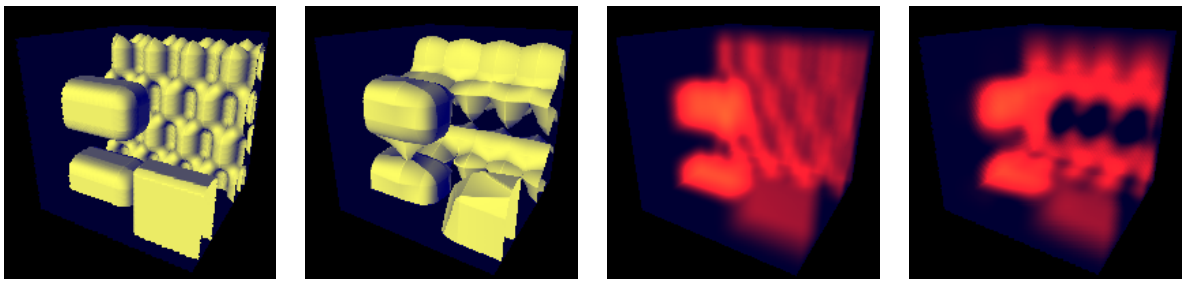
6.1 Voraussetzungen für die Anwendbarkeit der Algorithmen

Die Interpolation auf dünnen Gittern ist im wesentlichen für zwei Anwendungsfälle geeignet. Im ersten Fall sind bereits die Eingabedaten nur auf dünnen Gittern bekannt. Für die Anwendbarkeit der in dieser Arbeit beschriebenen Algorithmen muß für diese Gruppe von Daten nichts mehr nachgewiesen werden, da alle Voraussetzungen bereits beim Generieren der Gitter erfüllt sein müssen. Anders sieht dies aus, wenn die dünnen Gitter erst aus einem vollen Gitter erzeugt werden müssen, um sie für die interaktive Darstellung großvolumiger Datensätze zu verwenden.

Nach den in Kapitel 2.1 beschriebenen mathematischen Grundlagen der dünnen Gitter muß die zu interpolierende Funktion in jede Richtung zweimal stetig differenzierbar sein. Theoretisch ist das jedoch keine Einschränkung, da die vorgegebene Funktion im allgemeinen nur an diskreten Gitterpunkten bekannt ist. Daher kann eine beliebige Anzahl von Funktionen zum Beispiel im C^6 erstellt werden, die als Funktionswerte an den Gitterpunkten genau die Werte des Eingabegitters annehmen.

Andererseits enthalten die Abschätzungen des Interpolationsfehlers (2.10) und (2.11) im Dreidimensionalen die sechste Ableitung der zu interpolierenden Funktion, die außer bei analytischen Funktionen gewöhnlich nicht bekannt ist. Die Abschätzung ist somit in der Praxis für die Beurteilung der Qualität der Interpolation relativ wertlos und zeigt nur die prinzipielle Konvergenz auf.

Die folgenden Beispiele sollen einen Eindruck davon vermitteln, welche Datensätze sich für die Kompression mit dünnen Gittern eignen. Um die Unterschiede zu den vollen Gittern deutlich herauszustellen, sind die meisten Bilder mit der Kombinationsmethode als ISO-Flächen berechnet. Diese sind für Kompressionsartefakte wesentlich anfälliger als die Röntgen-Integrationsmethode, wie in Abbildung 6.1 deutlich zu sehen ist. Da zur Berechnung von ISO-Flächen für die Beleuchtungsfunktion die Normalen auf den Flächen gebildet werden, sind auch lokale Schwankungen der Interpolationsfunktion gut zu erkennen.



test3.9.9.9, Volles Gitter, ISO-Wert 0,2, 9^3 | test3.9.9.9, Kombi, ISO-Wert 0,2, Level 4 | test3.9.9.9, Volles Gitter, Röntgen, 9^3 | test3.9.9.9, Kombi, Röntgen, Level 4

Abbildung 6.1: Vergleich der Artefaktanfälligkeit von ISO-Flächen zur Röntgen-Integrationsmethode

Für die Untersuchung der Interpolationsfehler hat sich ein analytischer Datensatz mit der Funktion

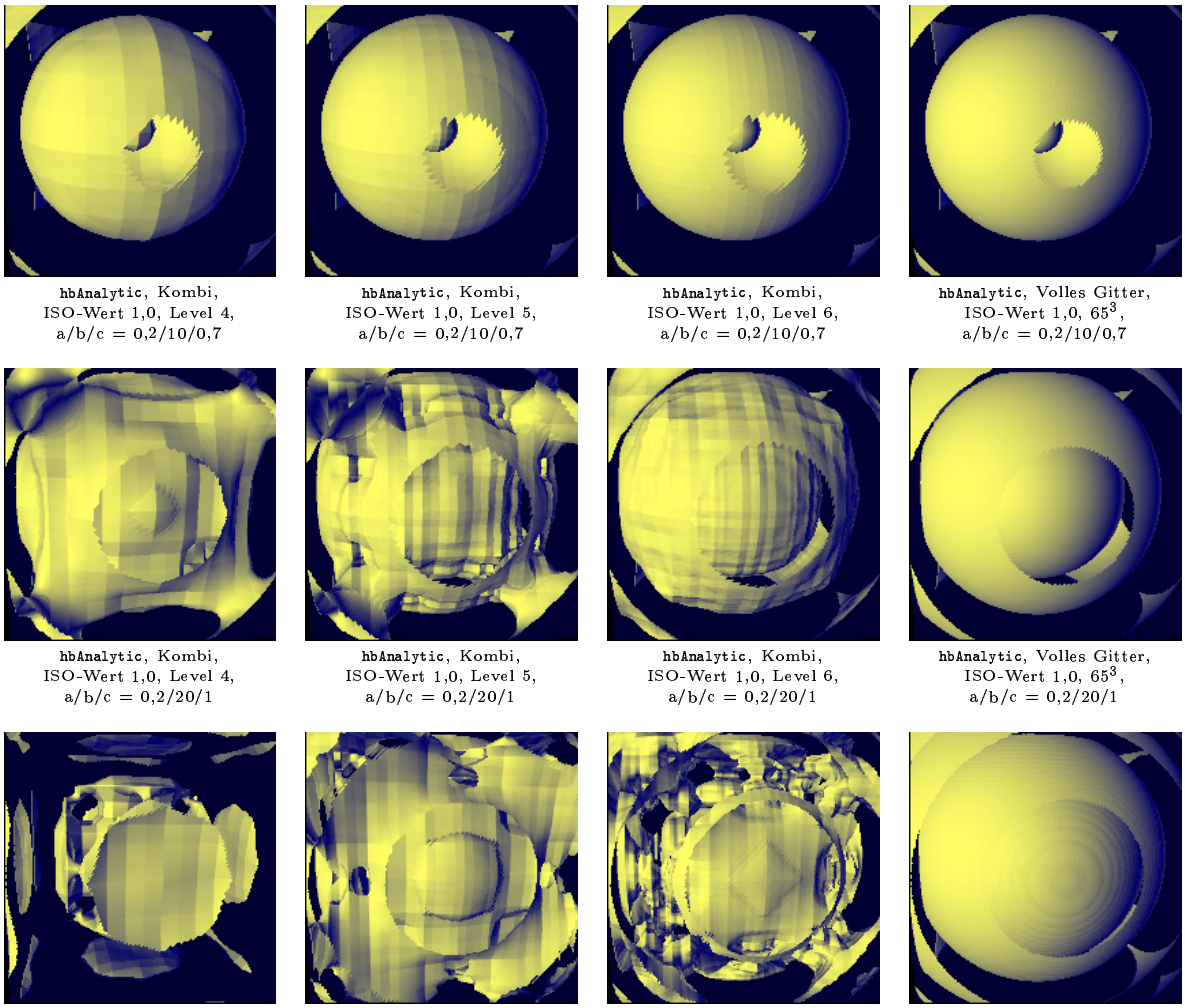
$$\begin{aligned} f(r^2) &= c \cdot e^{a \cdot r^2} \cdot (1 + \cos(b \cdot \sqrt{r^2}))\ , \\ r^2 &= (x - 0{,}5)^2 + (y - 0{,}5)^2 + (z - 0{,}5)^2\ , \quad x, y, z \in [0, 1] \end{aligned} \tag{6.1}$$

als recht aussagekräftig erwiesen. In Abbildung 6.2 ist gut zu erkennen, daß die Fähigkeit der dünnen Gitter, die Funktion zu approximieren, sehr stark von der Frequenz b der Funktion abhängt. Die Grenze zu nicht mehr tolerierbaren Verzerrungen und Funktionswertverfälschungen ist ziemlich scharf — während für $b = 10$ bereits Level 4 eine sehr gute Approximation der Funktion liefert, kann für $b = 20$ auch Level 6 nicht mehr voll überzeugen. Für $b = 30$ ist schließlich selbst bei Level 6 kaum noch Ähnlichkeit mit dem Datensatz vorhanden, obwohl die Funktion des Datensatzes aus C^∞ ist und somit die Vorraussetzungen an dünne Gitter voll erfüllt. Dünne Gitter reagieren also insbesondere auf hohe Frequenzen sehr empfindlich. Anders ausgedrückt steigt auch in der Praxis der Interpolationsfehler für Funktionen mit wachsenden höheren Ableitungen gemäß (2.10) und (2.11).

Eine weitere Möglichkeit, das Kompressionsverhalten der dünnen Gitter zu untersuchen, besteht darin, einen für dünne Gitter schlecht geeigneten Datensatz zu filtern. Anschließend kann man die Bilder vergleichen, die durch die Visualisierung mit dünnen und vollen Gittern gewonnen werden. Durch die Tiefpaß-Filterung werden Kanten geglättet und hohe Frequenzen aus dem Datensatz entfernt. Wie man in Abbildung 6.3 deutlich erkennen kann, nähern sich die Visualisierungsergebnisse von Voll- und Dünngitterdaten mit steigenden Filtergrößen schnell an. Als Filter wurde ein ungewichteter Box-Filter (*Blur*) verwendet. Ein zum Vergleich herangezogener Gauß-Filter ergab ähnliche Ergebnisse.

Abbildung 6.4 zeigt, daß die Stetigkeit des Datensatzes für die Praxis zwar keine Voraussetzung, aber dennoch von Vorteil ist. Der Testdatensatz der Größe 9^3 wurde bewußt unstetig konstruiert. Durch die trilineare Interpolation ist dies allerdings auch beim Betrachten des vollen Gitters nicht mehr erkennbar.

Die Abbildung zeigt eindeutig, daß für eine gute Darstellung der Unstetigkeitsstellen eine hohe Levelzahl notwendig ist. Insbesondere der der Größe des Volumens entsprechende Level 3 ist



`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 4, a/b/c = 0,2/10/0,7

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 5, a/b/c = 0,2/10/0,7

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 6, a/b/c = 0,2/10/0,7

`hbAnalytic`, Volles Gitter, ISO-Wert 1,0, 65^3, a/b/c = 0,2/10/0,7

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 4, a/b/c = 0,2/20/1

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 5, a/b/c = 0,2/20/1

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 6, a/b/c = 0,2/20/1

`hbAnalytic`, Volles Gitter, ISO-Wert 1,0, 65^3, a/b/c = 0,2/20/1

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 4, a/b/c = 0,2/30/1,5

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 5, a/b/c = 0,2/30/1,5

`hbAnalytic`, Kombi, ISO-Wert 1,0, Level 6, a/b/c = 0,2/30/1,5

`hbAnalytic`, Volles Gitter, ISO-Wert 1,0, 65^3, a/b/c = 0,2/30/1,5

Abbildung 6.2: Kompressionsverhalten des analytischen Datensatzes

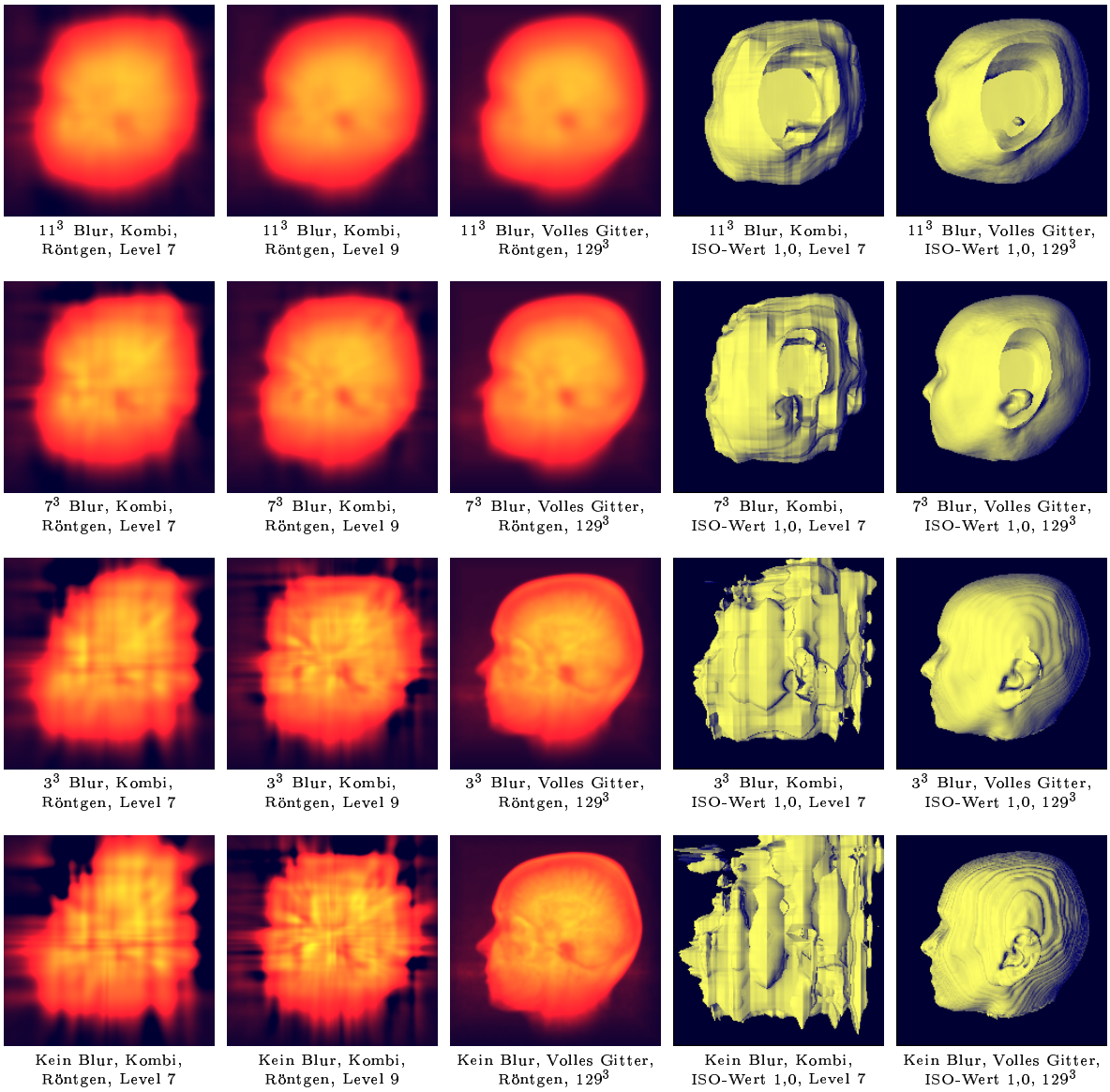


Abbildung 6.3: Kompressionsverhalten des gefilterten `head` Datensatzes

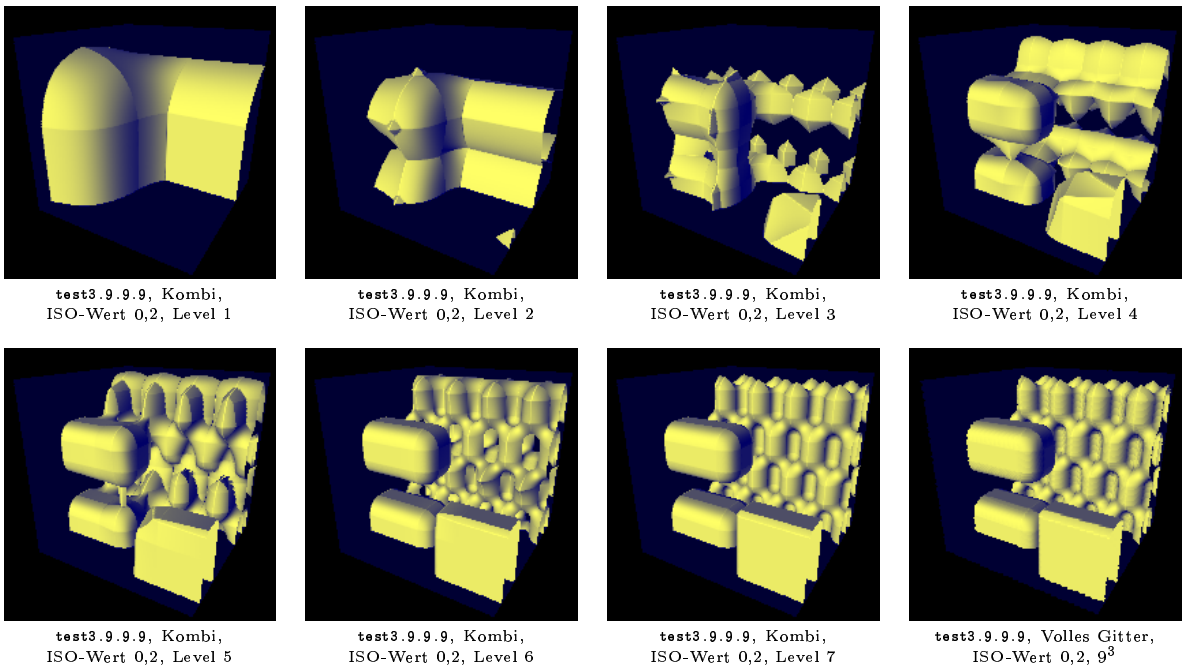


Abbildung 6.4: Wiedergabequalität eines unstetigen Testdatensatzes in Abhängigkeit des Levels

in keiner Weise für die Visualisierung des Datensatzes geeignet. Auch hier ist sehr deutlich zu sehen, daß vor allem die hohen Frequenzen im Schachbrett-Muster an der Rückseite des Volumens Schwierigkeiten bei der korrekten Darstellung machen.

Durch die geringe Volumengröße machen sich fehlende Koeffizienten stärker bemerkbar. Andererseits sind relativ zum vollen Gitter mehr Koeffizienten vorhanden als bei größeren Volumen.

Als weiteren Aspekt kann man Abbildung 6.4 entnehmen, daß für Level 7 exakt das gleiche Bild entsteht wie für das volle Gitter. Nach der Theorie der dünnen Gitter muß dies so sein, da gemäß der Index-Menge in (2.8) wegen $3+3+3 \leq 7+3-1$ alle Gitterpunkte des $G_{3,3,3}$ im $\tilde{G}_7$ enthalten sind. Zudem werden alle Gitterpunkte, die nicht im vollen Gitter vorkommen, durch trilineare Interpolation gewonnen und verändern dadurch ebenfalls nicht die Interpolation. Dieses Bild ist also als überzeugender Test für die Richtigkeit der Implementierung anzusehen.

Zusammenfassend kann man sagen, daß dünne Gitter auch mit unstetigen Datensätzen brauchbare Ergebnisse liefern können, wenn man zur Visualisierung einen genügend hohen Level einsetzt. Empfindlich reagieren sie hauptsächlich auf hohe Frequenzen im Datensatz. In diesen Fall muß häufig der Level so hoch gewählt werden, daß sich der Einsatz der dünnen Gitter nicht mehr lohnt.

6.2 Vergleich mit Wavelet-basierten Kompressionsverfahren

Im Gegensatz zur Dünngitter-Methode, die durch ihren prinzipiellen Aufbau die Anzahl der Koeffizienten für eine gegebene Gitterauflösung vorbestimmt, ist diese Anzahl und damit auch

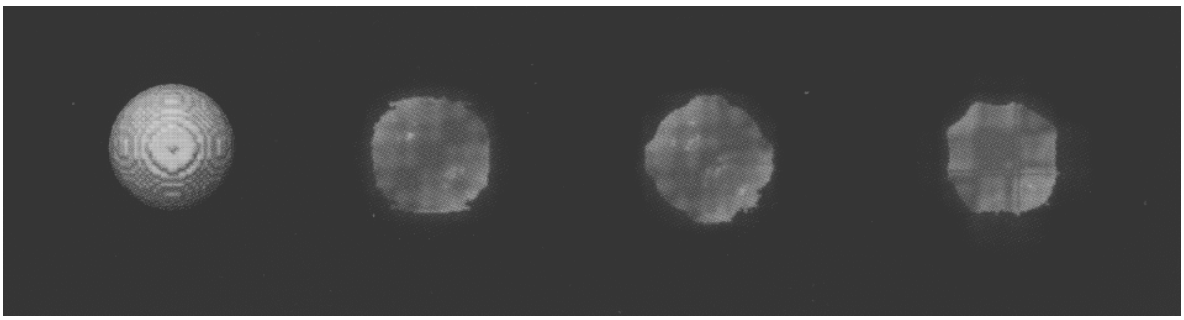
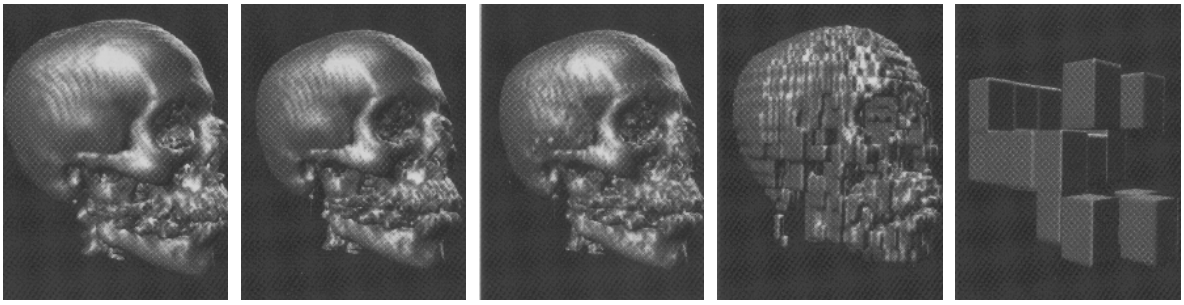


Abbildung 6.5: Artefakte bei verschiedenen Wavelets (aus [10, Seiten 149ff])



Haar-Wavelets, 100% aller Koeffizienten | Haar-Wavelets, 15,5% aller Koeffizienten | Haar-Wavelets, 6,8% aller Koeffizienten | Haar-Wavelets, 0,3% aller Koeffizienten | Haar-Wavelets, 0,02% aller Koeffizienten

Abbildung 6.6: ISO-Flächen-Bestimmung mit Haar-Wavelets zum Vergleich (aus [10, Seiten 149ff])

der Kompressionsgrad und die visuelle Qualität bei der Komprimierung mittels Wavelets frei wählbar. Auch kann die Wahl der Basisfunktionen an den zu visualisierenden Datensatz angepaßt werden.

Durch diese Eigenschaften sind sie im Gegensatz zu dünnen Gittern in der Lage, nahezu beliebige Datensätze in akzeptabler Weise zu komprimieren. Nähert sich jedoch die Kompressionsrate der der dünnen Gitter, haben die Wavelets in der Darstellungsqualität keinen erheblichen Vorteil mehr.

Die Artefakte, die bei der Kompression mit Wavelets auftreten, sind dabei erstaunlicherweise von ähnlicher Natur wie bei den dünnen Gittern, wovon man sich in Abbildung 6.5 im Vergleich zu Abbildung 6.2 überzeugen kann.

In Abbildung 6.6 sind noch einige Bilder eines Kopf-Datensatzes mit Haar-Wavelets visualisiert und mit unterschiedlichen Kompressionsraten dargestellt. Die Anzahl der Koeffizienten eines in etwa vergleichbaren dünnen Gitters mit Level 8 gegenüber der eines entsprechenden vollen Gitters beträgt ungefähr 0,126% mit der Standardmethode und 0,575% mit der Kombinationsmethode. Die Kompressionsrate ist mit der der beiden rechten Bilder in Abbildung 6.6 vergleichbar.

Dabei sind zwei ganz wesentliche Punkte zu beachten. Erstens sind die Datenstrukturen durch den Aufbau der Kompressionsmethode und der Wavelet-Dekomposition deutlich komplizierter

Level	5	6	7	8	9	10	11
Größe	33^3	65^3	129^3	257^3	513^3	1025^3	2049^3
Volles Gitter	128kB	1MB	8MB	64MB	512MB	4GB	32GB
Standardmethode	6kB	15kB	35kB	83kB	200kB	450kB	1MB
Kombinationsmethode	22kB	59kB	152kB	377kB	914kB	2,1MB	5MB
Hardware-Methode	43kB	124kB	338kB	884kB	2,2MB	5,4MB	13,1MB
Anzahl kleiner Gitter	31	46	64	85	109	136	166

Tabelle 6.1: Speicherbedarf der verschiedenen Interpolationsroutinen

als bei den dünnen Gitter, vor allem im Vergleich zur Kombinationsmethode. Der Speicherbedarf wächst so noch einmal erheblich, während bei der Kombinationsmethode außer für die Koeffizienten kein weiterer nennenswerter Speicherbedarf besteht. Zweitens ist, bedingt durch die meist wesentlich komplizierteren Basisfunktionen, die Interpolation langsamer und es existieren noch keine Beschleunigungsansätze mit Standard-Hardware.

6.3 Vergleich des Speicherbedarfs

In Kapitel 2.5 wurde bereits genau auf die Anzahl des benötigten Speichers der einzelnen Interpolationsmethoden eingegangen. Um einen genauen Überblick über den Speicherbedarf der unterschiedlichen Routinen zu erlangen, ist in Tabelle 6.1 der benötigte Speicher für verschiedene Level aufgeführt. Alle Methoden benutzen als Datentyp `float` benutzen, mit Ausnahme der Hardware-beschleunigten Kombinationsmethode. Bei dieser werden die Daten als `short`-Werte in den Texturspeicher geschrieben. Aufgrund der bereits erwähnten beschränkten Genauigkeit des Framebuffers ist dies vollkommen ausreichend. Werden die Daten statt mit 16 Bit nur mit 8 Bit Genauigkeit in den Texturspeicher geladen, reicht dieser noch für Level 12 aus. Momentan ist jedoch die Genauigkeit des Framebuffers momentan bereits bei Level 11 für eine effektive Nutzung zu gering. Deshalb ist diese Möglichkeit bis jetzt noch nicht in dem Programm integriert.

Die Tabelle zeigt deutlich die extreme Speicherersparnis, die dünne Gitter gegenüber vollen Gittern bringen. Bereits auf nicht besonders gut ausgestatteten Workstations können dünne Gitter mit Level 11 visualisiert werden, während selbst Großrechner nicht genügend Hauptspeicher für volle Gittern der entsprechenden Größe besitzen. In der Tabelle ist auch die Anzahl der kleineren vollen Gitter angegeben, die für die Interpolation mit der Kombinationsmethode benötigt werden. Sie entspricht der Anzahl der für die Hardware-Methode benötigten Texturen.

6.4 Vergleich der benötigten Zeit

Die Zeitmessungen in Tabelle 6.2 wurden auf einer SGI Onyx mit einer Reality Engine 2 und 4 R10000 Prozessoren mit 196 MHz am Lehrstuhl für grafische Datenverarbeitung durch-

Benötigte Rechenzeit:

Level	5	6	7	8	9	10	11
Volles Gitter	1,37 s	2,73 s	5,3 s	11,4 s	†	†	†
Standardmethode	181 s	510 s	1 425 s	3 870 s	11 000 s*	31 300 s*	86 500 s*
Kombinationsmethode	20,3 s	60 s	172 s	475 s	1 775 s	5 250 s	14 800 s*
Hardware	0,95 s	2,26 s	5,6 s	13,7 s	33,7 s	82 s	199 s

Samples pro Sekunde und Beschleunigungsfaktoren:

Level	5	6	7	8	9	10	11
Volles Gitter	934k	938k	952k	898k	†	†	†
Standardmethode	7,1k	5k	3,59k	2,65k	1,86k*	1,3k*	947*
	8,9×	8,5×	8,3×	8,1×	6,2×	6,0×	5,8×
Kombinationsmethode	63k	42,7k	29,8k	21,6k	11,5k	7,8k	5,5k
	21×	27×	31×	35×	53×	64×	74×
Hardware	1,34M	1,13M	914k	747k	608k	500k	412k

* hochgerechnet † wegen zu hohem Speicherbedarf nicht möglich

Standardmethode und Kombinationsmethode: CPU-Zeit

Hardware-beschleunigte Kombinationsmethode: Echtzeit

Alle Angaben für Bilder der Größe 200^2, gerechnet auf SGI Onyx mit Reality Engine 2, 16 MB Texturspeicher, 196 MHz R10000

Tabelle 6.2: Rechenzeit, Samples/Sekunde und Beschleunigungsfaktoren in Abhängigkeit von der Interpolationsmethode und vom Level

geführt. Die Zeiten der Standard- und der Kombinationsmethode sind in CPU-Sekunden angegeben, während die Hardware-beschleunigte Kombinationsmethode in Echtzeit gemessen wurde, da die CPU hier die meiste Zeit mit Warten auf die Grafik-Hardware verbringt. Die Interpolation auf vollen Gittern ist komplett in Software geschrieben und daher sehr langsam.

Die Sampling-Schrittweite wird von `sgrid` automatisch an den eingestellten Level angepaßt. Bei einer Verdopplung der Gittergröße, also beim Erhöhen des Levels um eins, ergibt sich für die Vollgitter-Interpolation demnach theoretisch eine Verdopplung der Laufzeit bei gleicher Bildgröße. Ab Level 8 bricht die Performance der Vollgitter-Interpolation leicht ein, da der Cache aufgrund der Größe des Datensatzes nicht mehr voll wirksam ist.

Als Integrationsmethode wurde Röntgen gewählt, da die Berechnungsdauer hier unabhängig vom Datensatz ist und keine Normalen benötigt werden. Für ISO-Flächen braucht das Programm etwa doppelt so lange, da je gezeichneter Fläche eine zweite Fläche zur Normalenbestimmung in Beobachterrichtung benötigt wird. Hierfür könnte zwar die zuletzt gezeichnete Fläche benutzt werden, allerdings kann diese perspektivisch verzerrt und dadurch unbrauchbar sein.

In Tabelle 6.3 sind die Samplingraten in Abhängigkeit von der Bildgröße angegeben. Hier sieht man sehr deutlich, daß die Hardware-Methode die Berechnung sehr kleiner Bilder kaum beschleunigt, da ein großer Teil des Vorteils der wesentlich schnelleren Interpolation von dem Zeitbedarf für die Initialisierung der Hardware und dem höheren Aufwand je vollem Gitter zunichte gemacht wird.

Level 6:

Größe	10^2	50^2	100^2	200^2	400^2
Volles Gitter	320k	941k	955k	969k	819k
Standardmethode	5,03k	5,05k	5,05k	5,05k*	5k*
	8,5×	8,3×	8,3×	8,3×	8,1×
Kombinationsmethode	43k	42k	42k	42k	40,6k
	1,9×	15×	24×	27×	26×
Hardware	80k	640k	1M	1,13M	1,06M

Level 10:

Kombinationsmethode	7,7k	7,9k	7,9k	7,8k*	7,9k*
	6×	43×	61×	64×	61×
Hardware	46k	340k	480k	500k	480k

* hochgerechnet

Tabelle 6.3: Samples/Sekunde in Abhängigkeit von der Bildgröße

Bei einem Vergleich der Zeiten für die Interpolation auf vollen Gittern, der Hardware-beschleunigten Interpolation auf dünnen Gittern und der bekannten Verfahren zur Hardware-unterstützten Volumenvisualisierung voller Gitter fällt auf, daß die maximale Pixelfüllrate der Hardware mit der derzeitigen Implementierung noch nicht voll ausgelastet ist. Dies ist ein deutliches Indiz dafür, daß die ohnehin schon erhebliche Beschleunigung noch stark vergrößert werden kann, indem wie in Kapitel 7.2 beschrieben die Hardware auch für die Beleuchtungsberechnung hergezogen wird. Die Implementierung hätte aber den Rahmen dieser Arbeit gesprengt.

Ebenfalls interessant ist Tabelle 6.4 mit einem Vergleich der beiden verfügbaren Grafikkarten mit Volumentextur-Unterstützung, der Reality Engine 2 und der Maximum Impact. Da der Texturspeicher der Maximum Impact nur für die Gitter bis Level 7 ausreicht, bleibt der Test auf diese Volumengrößen beschränkt.

6.5 Beispiele

Die Abbildungen 6.7 bis 6.15 zeigen noch einige Bilder, die mit `sgrid` berechnet worden sind. Dabei werden insbesondere alle weiteren Datensätze gezeigt, mit denen die Funktionalität und die Visualisierungsqualität der Algorithmen getestet wurden.

Ein Vergleich der verschiedenen Interpolationsroutinen ist in Abbildung 6.7 dargestellt. Auffällig sind die leichten Artefakte in der Abbildung der Hardware-beschleunigten Kombinationsmethode, die aufgrund der geringen effektiven Framebuffer-Tiefe von nur etwa 7 Bit entstehen. Dieses Problem wurde bereits in Kapitel 5.5 behandelt. Abbildung 6.8 verdeutlicht die Abhängigkeit dieses Phänomens vom Level des visualisierten Volumens.

Viel deutlicher tritt dieser Effekt bei der Visualisierung von ISO-Flächen in Abbildung 6.9 zu Tage. Da die Gradienten im Bildraum berechnet werden, können sich bereits kleine Ungenauigkeiten der Interpolationswerte stark auf die so erstellten Normalen auswirken. In höheren

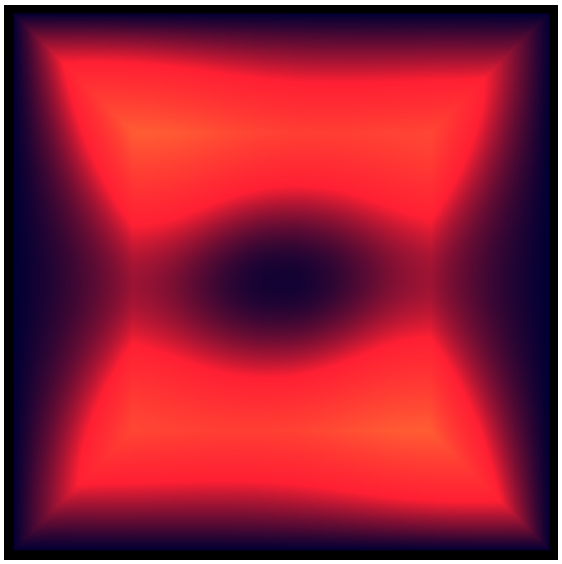
Bildgröße 400^2

Level	5	6	7
Maximum Impact	674k	670k	615k
	1,9×	1,6×	1,4×
Reality Engine 2	1,28M	1,06M	864k

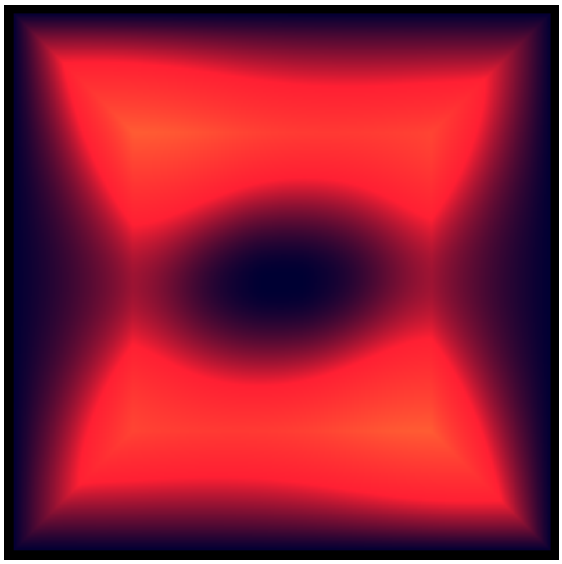
Bildgröße 100^2

Level	5	6	7
Maximum Impact	660k	640k	595k
	1,9×	1,6×	1,4×
Reality Engine 2	1,23M	1,01M	853M

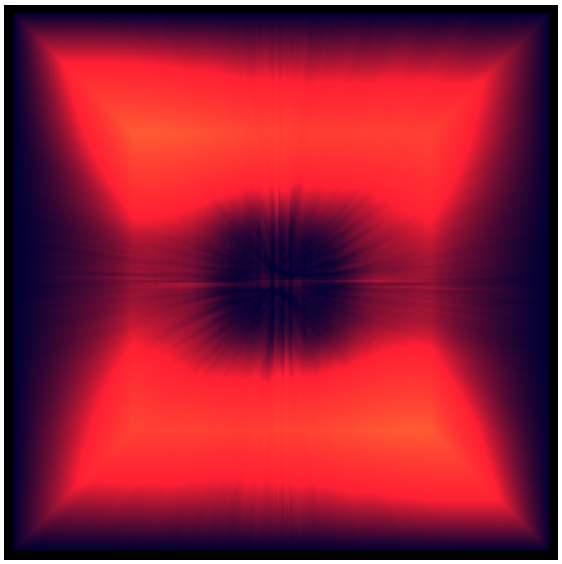
Tabelle 6.4: Samples/Sekunde der Hardware-beschleunigten Kombinationsmethode in Abhängigkeit von der Grafikkarte



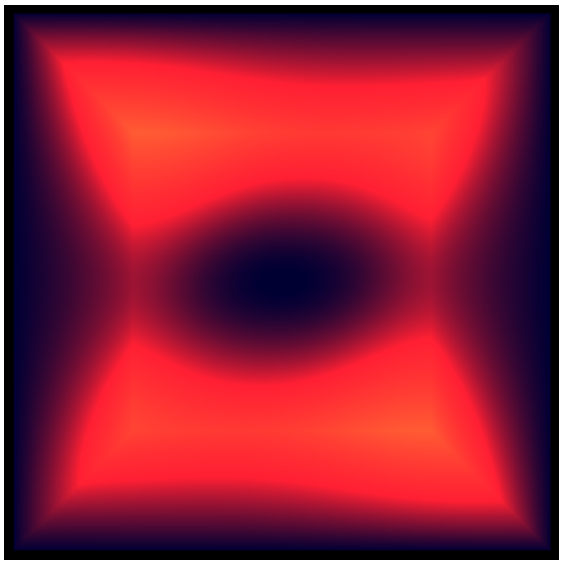
`cavity4`, Standardmethode, Röntgen, Level 6



`cavity4`, Kombinationsmethode, Röntgen, Level 6

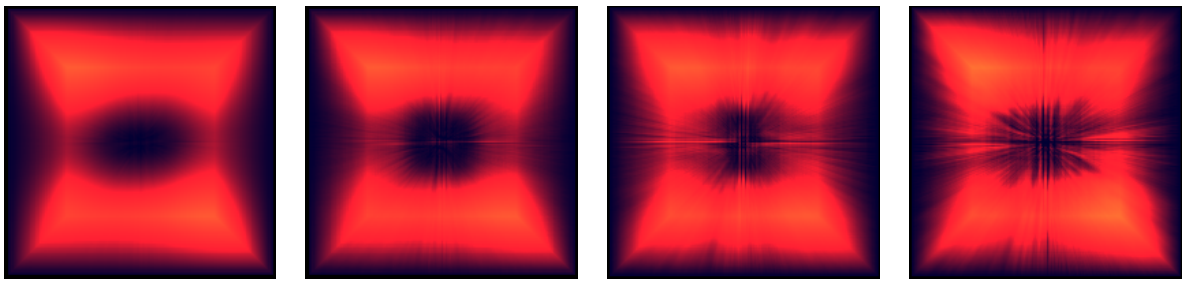


`cavity4`, OpenGL-Methode, Röntgen, Level 6



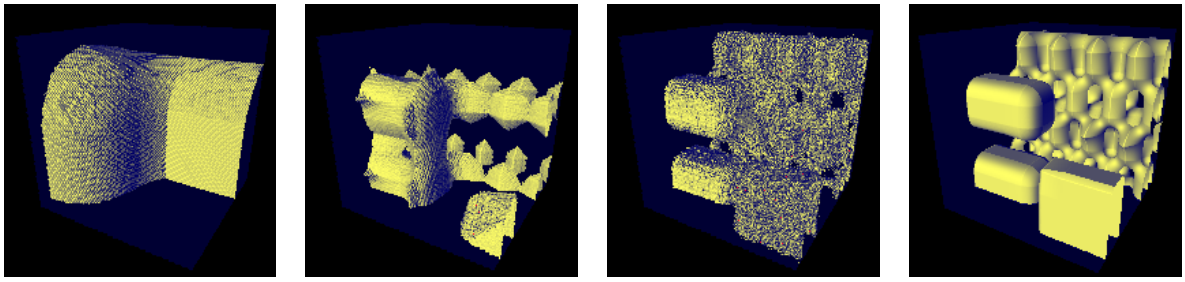
`cavity4`, Volles Gitter, Röntgen, 65^3

Abbildung 6.7: Vergleich der Standardmethode, der Kombinationsmethode, der Hardware-beschleunigte Kombinationsmethode und des vollen Gitters



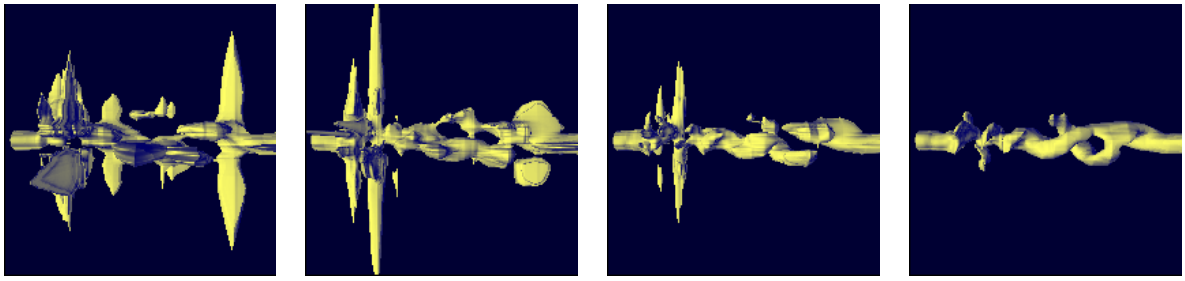
`cavity4`, OpenGL, Röntgen, Level 5 | `cavity4`, OpenGL, Röntgen, Level 6 | `cavity4`, OpenGL, Röntgen, Level 7 | `cavity4`, OpenGL, Röntgen, Level 8

Abbildung 6.8: Abhängigkeit der Genauigkeit im Framebuffer vom Level bei der Hardware-beschleunigten Kombinationsmethode



`test3.9.9.9`, OpenGL, ISO-Wert 0,2, Level 1 | `test3.9.9.9`, OpenGL, ISO-Wert 0,2, Level 3 | `test3.9.9.9`, OpenGL, ISO-Wert 0,2, Level 6 | `test3.9.9.9`, Kombi, ISO-Wert 0,2, Level 6

Abbildung 6.9: Unbrauchbare ISO-Flächen mit der Hardware-beschleunigten Kombinationsmethode aufgrund geringer Framebuffer-Tiefe

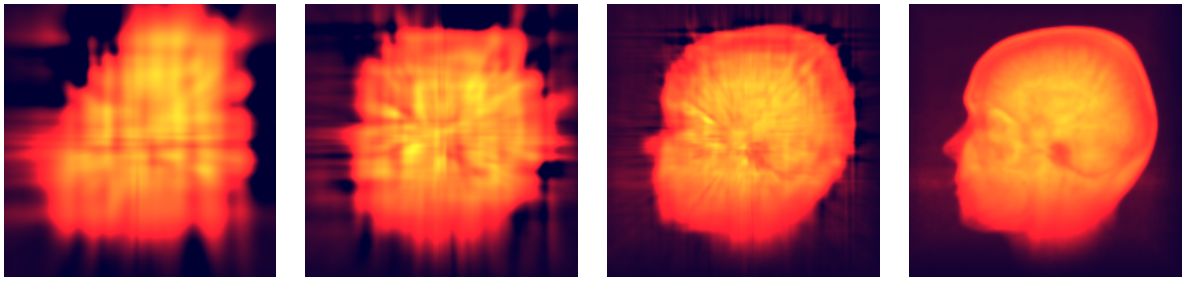


`Pegas324`, Kombi, ISO-Wert 1,2, Level 8 | `Pegas324`, Kombi, ISO-Wert 1,2, Level 10 | `Pegas324`, Kombi, ISO-Wert 1,2, Level 12 | `Pegas324`, Volles Gitter, ISO-Wert 1,2, 257^3

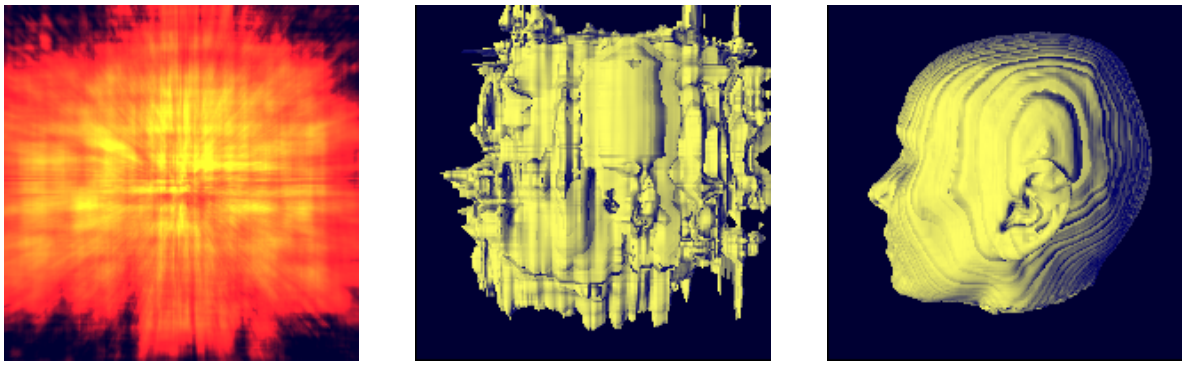
Abbildung 6.10: Beispiel eines Strömungsdatensatzes, der nicht gut für die Komprimierung mittels dünner Gitter geeignet ist (siehe Text)

Leveln hat dies zur Folge, daß die Normalen eher weißem Rauschen ähneln als den gewünschten Gradienten.

Datensätze mit starken lokalen Schwankungen sind für die Visualisierung mit dünnen Gittern ungeeignet. Während sich Strömungsdatensätze, wie an den `cavity`-Beispielen gesehen, prinzipiell gut darstellen lassen, gibt es doch Beispiele turbulenter Strömungen, die sich nicht gut mit dünnen Gittern verarbeiten lassen, weil sie auf vollen Gittern berechnet wurden. In Abbildung 6.10 ist ein solcher Datensatz zu sehen, der aus einer numerischen Strömungssimulation stammt.



`raw.head.128.128.54`, Kombi, Röntgen (Skalierung 10×), Level 7 | `raw.head.128.128.54`, Kombi, Röntgen (Skalierung 10×), Level 9 | `raw.head.128.128.54`, Kombi, Röntgen (Skalierung 10×), Level 12 | `raw.head.128.128.54`, Volles Gitter, Röntgen (Skalierung 10×), 129^3

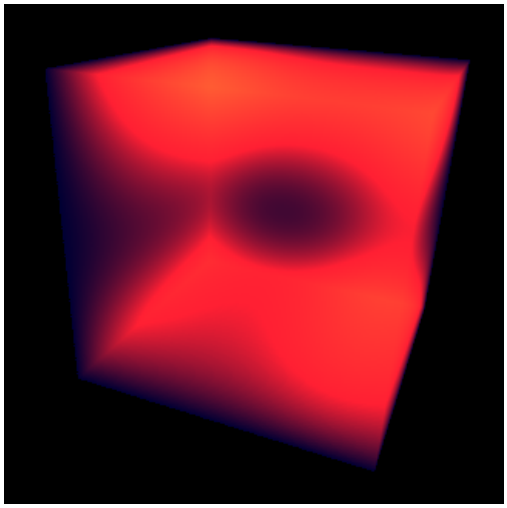


`raw.head.128.128.54`, OpenGL, Röntgen (Skalierung 10×), Level 11 | `raw.head.128.128.54`, Kombi, ISO-Wert 0,08, Level 9 | `raw.head.128.128.54`, Volles Gitter, ISO-Wert 0,08, 129^3

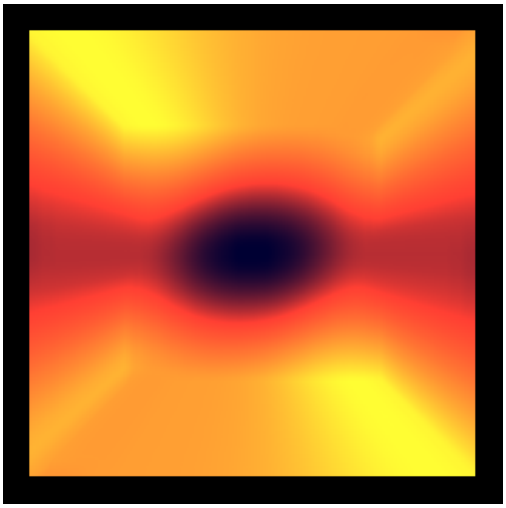
Abbildung 6.11: Die Kernspinresonanz-Tomographieaufnahme eines Kopfes als Beispiel eines unstetigen, hochfrequenten und daher für dünne Gitter unbrauchbaren Datensatzes

Ebenfalls schlecht an dünne Gitter angepaßt ist der Datensatz in Abbildung 6.11. Allein die Tatsache, daß an den Übergangsbereichen der verschiedenen Materiedichten wie beim Übergang von der Luft in die Haut Unstetigkeitsstellen existieren, kann erklären, warum sich der Datensatz schlecht für die Visualisierung mit dünnen Gittern eignet. Es ist jedoch erstaunlich, welch hohe Level benutzt werden müssen, um auch nur die grobe Form des Kopfes erahnen zu lassen. Offensichtlich wirkt sich die Kompression bei dünnen Gittern nicht nur lokal, sondern gleich auf das ganze Volumen aus.

Abbildungen 6.12 und 6.13 zeigen Beispiele für die verschiedenen Integrationsmethoden anhand der bereits in den Abbildungen 6.7 und 6.8 dargestellten Druckverteilung des `cavity` Datensatzes. In Abbildung 6.14 ist die Temperaturverteilung des Datensatzes mit verschiedenen Integrationsmethoden visualisiert. Zum Abschluß ist in Abbildung 6.15 die analytische Lösung der Legendre-Funktion des Wasserstoff-Atoms dargestellt. Auf diesen Bildern kann man auch deutlich die Artefakte erkennen, die durch die dünnen Gitter entstehen.

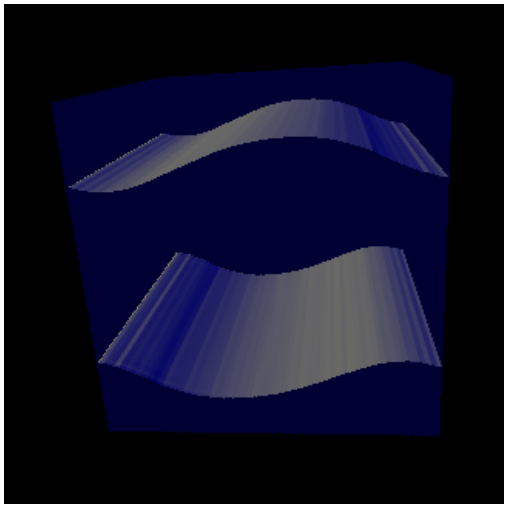


cavity4, Kombinationsmethode, Röntgen, Level 6

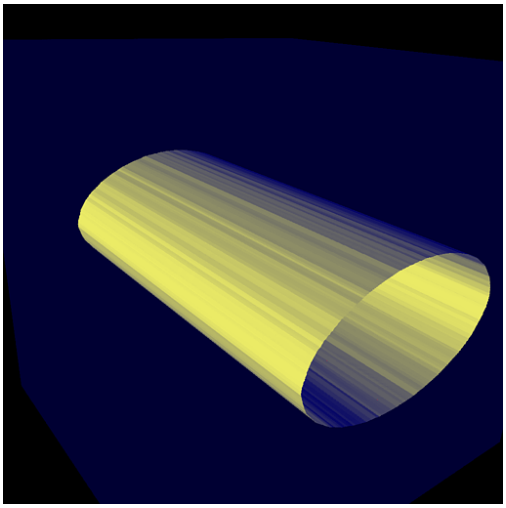


cavity4, Kombinationsmethode, MIP, Level 6

Abbildung 6.12: Röntgen- und MIP-Beleuchtungsmodell am Beispiel des Cavity4-Datensatzes, berechnet mit der Kombinationsmethode

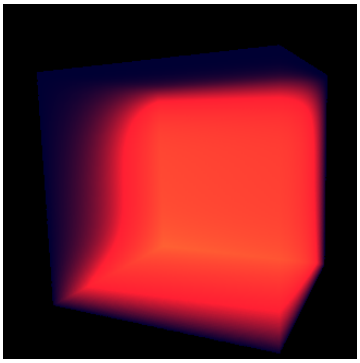


cavity4, Kombinationsmethode, ISO-Wert 0,25, Level 6

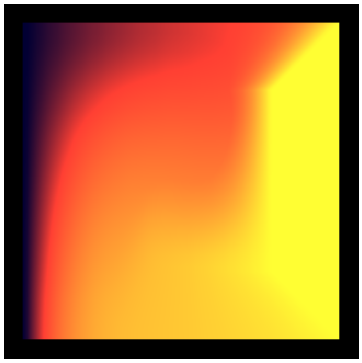


cavity4, Kombinationsmethode, ISO-Wert 0,5, Level 7

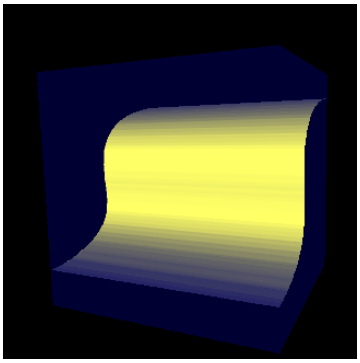
Abbildung 6.13: Zwei ISO-Flächen des Cavity4-Datensatz, Kombinationsmethode



cavity5, Kombi, Röntgen, Level 6

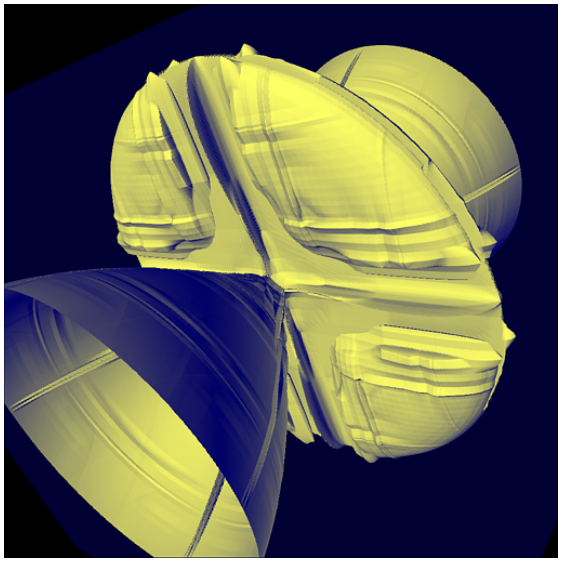


cavity5, Kombi, MIP, Level 6

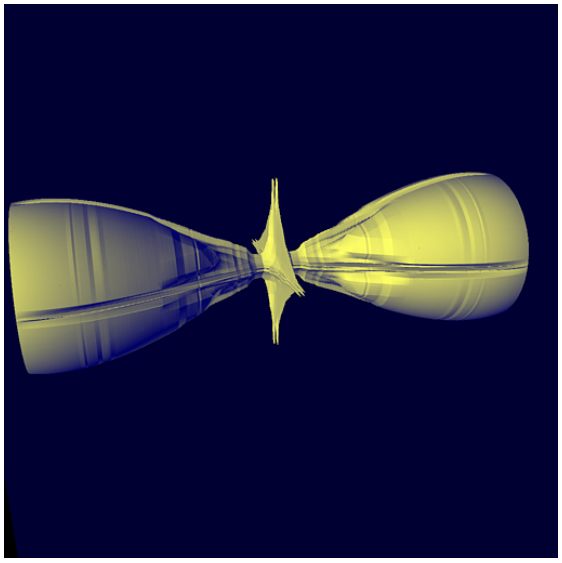


cavity5, Kombi, ISO-Wert 0,5, Level 6

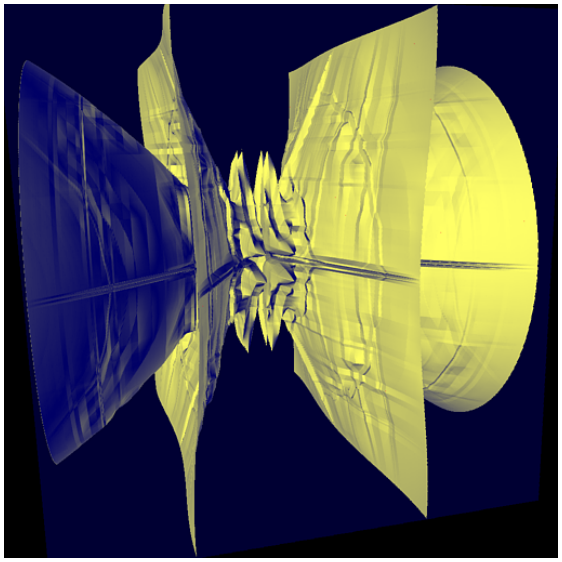
Abbildung 6.14: Beispiel-Ansichten des Cavity5-Datensatzes



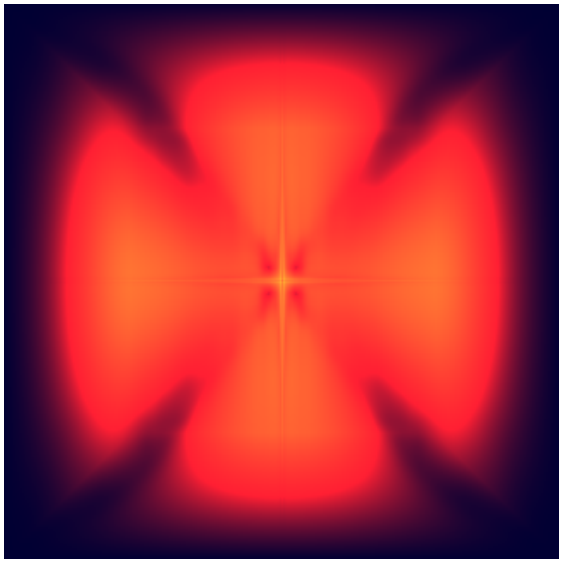
`hydrogen`, Kombi, ISO, Level 9



`hydrogen`, Kombi, ISO, Level 9



`hydrogen`, Kombi, ISO, Level 9



`hydrogen`, Kombi, Röntgen, Level 9

Abbildung 6.15: Die Orbitalfunktionen des Wasserstoff-Atoms

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung der Arbeit

Im Rahmen dieser Diplomarbeit wurde ein Programm für die bisher noch nicht verfügbare Volumenvisualisierung dünner Gitter erstellt und mit verschiedenen Datensätzen getestet. Es wurden mit MIP, Röntgen und ISO-Flächen mehrere verschiedene Integrations- und Beleuchtungsfunktionen implementiert und in eine grafische Benutzeroberfläche integriert.

Da die Interpolation auf dünnen Gittern deutlich aufwendiger als auf vollen Gittern ist, wurden mit der Kombinationsmethode und einer Epsilon-Schranke bei der Standardmethode verschiedene Ansätze zur Beschleunigung untersucht und implementiert. Dabei hat sich die Hardware-beschleunigte Kombinationsmethode als effizienteste Variante bewährt. Diese baut auf der Fähigkeit moderner Grafikkarten auf, Dreiecke mit dreidimensionalen Texturen zu versehen und auf diesen dann mit hoher Geschwindigkeit trilinear zu interpolieren. Die Implementierung benötigt viele Erweiterungen von OpenGL, die zum Teil noch experimentell und nur in der Implementierung von Silicon Graphics vorhanden sind.

Auch wenn sich die Geschwindigkeit der Interpolation mit der Hardware-beschleunigten Kombinationsmethode gegenüber der Standardmethode teilweise um den Faktor 400 und mehr erhöht, ist die Interpolation auf vollen Gittern noch deutlich schneller. Benutzt letztere ebenfalls die Grafik-Hardware, steht die Dünngitter-Methode im Bezug auf die Rechenzeit auf verlorenem Posten.

Andererseits können mittels dünner Gitter auf der Reality Engine 2 noch Volumen der Größe 2049^3 visualisiert werden. Bei Verwendung von 8 Bit-Werten für die Texturen lassen sich sogar Volumen der Größe 4097^3 visualisieren, während mit vollen Gittern spätestens jenseits der 256^3 bei Ausnutzung der Hardware nur mit einer Unterteilung des Volumens in mehrere Texturblöcke gearbeitet werden kann, und Volumen mit mehr als 512^3 Voxeln selbst im Hauptspeicher gut ausgebauter Workstations nicht mehr bearbeitet werden können.

Ein großes Problem stellt momentan die Tiefe des Framebuffers dar. Keine der verwendeten Grafikkarten stellt Framebuffer mit mehr als 12 Bit pro Farbkanal zur Verfügung. Die Genauigkeit der Interpolation fällt demnach ab Level 8 sehr stark ab.

7.2 Mögliche Erweiterungen und Verbesserungen

Wie bereits in Kapitel 4 angedeutet, muß das User-Interface noch in seiner Funktionalität verbessert werden. Unter anderem fehlt eine automatische Refresh-Funktion, die das Bild im Anzeige-Fenster nach dem Verdecken durch ein anderes Fenster neu zeichnet. Für die Klasse `hbAnalytic` wurde noch kein Fenster zum Einstellen der Parameter erstellt. Daher werden sie von der Konsole abgefragt. Momentan wird auch zur Visualisierung bei Abbildungen mit mehrdimensionalem Wertebereich immer nur der oberste Vektoreintrag benutzt. Hier sollte dem Benutzer eine Interaktionsmöglichkeit geboten werden.

Auch die Animationsfunktion hat noch eine prinzipbedingte Schwäche. Wenn die Samplingroutine durch die Alarmfunktion zufällig genau dann aufgerufen wird, wenn das Inventor-Fenster eine neue Ansicht zeichnet, kann es passieren, daß das ermittelte Viewing Frustrum mit der aktuellen Ansicht nichts zu tun hat, weil die Vektoren gerade in dem Momennt ausgelesen werden, in dem sie von Inventor überschrieben werden.

Momentan kann das Programm nur bestehende volle Gitter und analytische Datensätze darstellen, da noch kein Fileformat für dünne Gitter oder Gitter der Kombinationsmethode existiert. Eine Laderoutine für ein entsprechendes Datenformat könnte aber ohne Probleme integriert werden.

Um die begrenzte Genauigkeit der Hardware-beschleunigten Kombinationsmethode zu verbessern, kann die Summation der interpolierten kleinen vollen Gitter auch im *Accumulation Buffer* von OpenGL durchgeführt werden. Es muß noch untersucht werden, inwiefern diese Methode die Interpolation verlangsamt, und wie stark sich die erhöhte Genauigkeit auf die visuelle Qualität der berechneten Bilder auswirkt.

Wenn alle positiv und negativ beitragenden Texturen nicht nacheinander, sondern im Wechsel in den Framebuffer gezeichnet werden können, würde dies die notwendige Skalierung reduzieren und somit ebenfalls die Genauigkeit erhöhen. Dazu muß jedoch sichergestellt werden, daß die Werte im Framebuffer nie negativ werden, was durch die Art der Interpolation nicht auf triviale Weise sichergestellt ist. Ob hierfür eine genau definierte Zeichenreihenfolge ausreicht, oder ob komplexere Abfragen notwendig sind, muß noch geklärt werden.

Die Hardware-Beschleunigung bei der Interpolation ist nahezu ausgereizt. Allerdings wartet die CPU noch auf die Grafik-Hardware, bevor sie mit der Beleuchtungsberechnung beginnt. Eine verbesserte Variante könnte die Beleuchtungsberechnung einer schon interpolierten Ebene durchführen, während die Hardware bereits eine neue Ebene zeichnet.

An sich könnte die Beleuchtungsberechnung und die Integration noch sehr stark beschleunigt werden. In der derzeitigen Implementierung ist dieser Prozeß noch rein CPU-basiert, einfache Beleuchtungsfunktionen wie zum Beispiel Röntgen können aber genauso gut mittels Grafik-Hardware durchgeführt werden. Man muß allerdings genau abwägen, ob die Hardware trotz höherer Belastung die Berechnung beschleunigen kann. Schließlich ist die Beleuchtungsberechnung die letzte Hauptaufgabe, die der CPU momentan noch bleibt. Noch weitere Verlagerungen der Aufgaben in die Hardware könnte sogar die Berechnung verlangsamen.

Das größte Problem bei der Beleuchtungsberechnung in Hardware dürfte darin bestehen, daß die Daten bereits im Framebuffer liegen. Man muß also einen weiteren Framebuffer allozieren, und die Daten von Framebuffer zu Framebuffer mit Compositing kopieren. Notfalls können auch die Daten des einen Framebuffers in den Texturspeicher kopiert und anschließend eine Fläche mit dieser Textur in den anderen Framebuffer gezeichnet werden. Wieder gilt dabei zu beachten, daß auch hier die Framebuffer-Tiefe der Genauigkeit enge Grenzen setzt.

Schwieriger wird es, komplexere Beleuchtungsfunktionen mit der Hardware zu realisieren, wie solche, die die Normalen an allen betrachteten Punkten benötigen. Da die derzeit implementierte Normalenbestimmung für die Interpolation mittels Hardware nicht geeignet ist, muß hier vorher noch ein anderes Verfahren eingesetzt werden, das die Normalen in einem Vorverarbeitungsschritt berechnet und diese dann ebenso wie den Funktionswert im Objektraum interpoliert. Dabei wird zusätzlicher Texturspeicher benötigt, ein Aspekt, der die Anwendbarkeit dieser Methode wieder stark einschränkt.

Die Beleuchtungsberechnung an sich ist zumindest für ISO-Flächen auch in Hardware möglich, wie am Lehrstuhl für grafische Datenverarbeitung bereits im Programm ***TIVoR*** [16] gezeigt wurde.

7.3 Zukunft der dünnen Gitter in der Visualisierung

Für die Kompression und anschließende Darstellung beliebiger voller Gitter haben sich dünne Gitter in vielen Fällen als relativ unbrauchbar erwiesen, da sie durch ihren Aufbau nur eine sehr geringe Anzahl an Koeffizienten besitzen und demnach fein aufgelöste Volumen nur ungenügend reproduzieren. Volumen, die nur niedrige Frequenzen und geringes Rauschen enthalten, machen dabei deutlich weniger Probleme als hochfrequente oder gar unstetige Datensätze.

Es zeigt sich, daß gerade für hochfrequente Daten Ansätze mit an den Datensatz anpaßbarem Kompressionsverhalten besser geeignet sind. Als Beispiel hierfür seien Wavelets genannt. Diese Verfahren haben jedoch das Problem, daß die Visualisierung momentan noch wesentlich aufwendiger ist als die der dünnen Gitter und daher nicht interaktiv geschehen kann, da sich die Hardware durch die komplizierteren Basisfunktionen nicht zur Beschleunigung eignet.

In der Numerik hingegen haben sich in einigen Bereichen zur Lösung partieller Differentialgleichungssysteme bereits dünne Gitter etabliert. Wenn für ein Problem gezeigt werden kann, daß die durch die Interpolation induzierten Fehler akzeptabel sind, können Lösungen des Gleichungssystems direkt auf dünnen Gittern in kürzerer Zeit in erheblich höherer Auflösung und damit auch höherer Genauigkeit als auf vollen Gittern berechnet werden.

Große dünne Gitter können ohne direkte Visualisierung gar nicht mehr dargestellt werden, da für das Umschichten der Lösung auf ein entsprechendes volles Gitter der Hauptspeicher heutiger Workstations bei weitem nicht ausreicht. Hier wird die direkte Volumenvisualisierung auf dünnen Gittern auch in Zukunft noch benötigt werden.

Dünne Gitter werden wegen ihrer Vorzüge in der Nummerik auch weiterhin benutzt werden. Für die hieraus resultierenden Datensätze ist die direkte Visualisierung dünner Gitter von großem Vorteil. Allerdings kann die Dünngitter-Technik nur mit Einschränkungen für die Kompression bereits bestehender Vollgitterdaten angewandt werden.

Literaturverzeichnis

[1] K. I. Babenko. Approximation by trigonometric polynomials in a certain class of periodic functions of several variables. *Soviet Mathematics*, 1:672–675, 1960. Translation of Doklady Akademii Nauk SSSR.

[2] H.-J. Bungartz. *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung.* PhD thesis, Technische Universität München, 1992.

[3] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice.* Addison-Wesley, second edition, 1993.

[4] M. Griebel and W. Huber. Turbulence simulation on sparse grids using the combination method. SFB-Bericht 342/19/94 A, Institut für Informatik, TU München, December 1994.

[5] M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. The combination technique for parallel sparse-grid-preconditioning or -solution of pde's on multiprocessor machines and workstation networks. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Second Joint International Conference on Vector and Parallel Processing*, pages 217–228, Berlin, 1992. CONPAR/VAPP, Springer-Verlag.

[6] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In P. de Groen and R. Beauwens, editors, *International Symposium on Iterative Methods in Linear Algebra*, pages 263–281, Amsterdam, 1992. IMACS, Elsevier.

[7] M. Griebel and V. Thurner. The efficient solution of fluid dynamics problems by the combination technique. SFB-Bericht 342/1/93 A, Institut für Informatik, TU München, 1993.

[8] H.-C. Hege, T. Höllerer, and Stalling D. Volume Rendering. Technical Report 93-7, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1993.

[9] N. Heußer and M. Rumpf. Efficient visualization of data on sparse grids. In H.-C. Hege and K. Polthier, editors, *Visualization and Mathematics*, Berlin. Springer-Verlag. In preparation.

[10] G. M. Nielson, H. Hagen, and H. Müller. *Scientific Visualization: Overviews, Methodologies, and Techniques.* IEEE Computer Society Press, Los Alamitos, CA, 1997.

[11] SGI. Volume rendering using re2 technology. Technical report, SGI, 1994.

[12] Silicon Graphics Inc., Mountain View, California. *Man Page glteximage3dext (3)*, 1996.

[13] Silicon Graphics Inc., Mountain View, California. *OpenGL on Silicon Graphics Systems*, 1996.

[14] Silicon Graphics Inc., Mountain View, California. *RapidApp User's Guide*, 1996.

[15] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics*, 4:240–243, 1963. Translation of Doklady Akademii Nauk SSSR.

[16] O. Sommer, A. Dietz, R. Westermann, and T. Ertl. An Interactive Visualization and Navigation Tool for Medical Volume Data. In N. M. Thalmann and V. Skala, editors, *WSCG '98, The Sixth International Conference in Central Europe on Computer Graphics and Visualization '98*, volume II, pages 362–371, Plzen, Czech Republic, February 1998. University of West Bohemia Press.

[17] C. Teitzel, R. Grosso, and T. Ertl. Particle Tracing on Sparse Grids. In D. Bartz, editor, *Visualization in Scientific Computing '98*, pages 81–90, Wien, April 1998. Springer-Verlag. Proceedings of the Eurographics Workshop in Blaubeuren, Germany.

[18] C. Teitzel, M. Hopf, R. Grosso, and T. Ertl. Volume Ray Casting on Sparse Grids. Technical Report 5/1998, Universität Erlangen-Nürnberg, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX), Erlangen, March 1998.

[19] C. Teitzel, M. Hopf, R. Grosso, and T. Ertl. Volume Visualization on Sparse Grids. Technical Report 8/1998, Universität Erlangen-Nürnberg, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX), Erlangen, July 1998.

[20] J. Werneke. *The Inventor Mentor, Programming Object-Oriented 3D Graphics with OpenInventor.* Addison-Wesley, release 2 edition, 1994.

[21] C. Zenger. Sparse grids. In *Parallel Algorithms for Partial Differential Equations: Proceedings of the Sixth GAMM-Seminar*, Kiel, 1990.

Abbildungsverzeichnis

2.1 Beispiele von eindimensionalen Basisfunktionen 5

2.2 Die Teilräume S_{j_1,j_2} von $G_{4,4}$ 6

2.3 Dünne Gitter der Dimension 2, Level 2 bis 5 9

2.4 Für die Interpolation des Wertes X benötigte Basisfunktionen in $\tilde{G}_{4,4}$ 10

2.5 Zusammensetzung des $\bar{G}_3$ aus mehreren kleineren vollen Gittern in der Kombinationsmethode 12

4.1 Legende für die Klassendiagramme 27

4.2 Hierarchie der Interpolationsklassen von `sgrid` 27

4.3 Hierarchie der Raycasterklassen und des User-Interfaces von `sgrid` 29

4.4 Das User-Interface von `sgrid` 30

4.5 Das Fenster zur Bildanzeige 30

4.6 Einstellmöglichkeiten für verschiedene Beleuchtungsmodelle 31

4.7 File-Dialog zur Auswahl des zu ladenden Gitters 32

4.8 Dialog zum Einstellen der Größe des zu ladenden Gitters 32

4.9 Einstellungen für die Animationsfunktion 33

4.10 Benötigte Vektoren für die Strahlenrückverfolgung 36

4.11 Pseudo-Code für die Berechnung des Viewing Frustrums 37

4.12 Prinzipielle Implementierungsvarianten eines Raycasting-Algorithmus 39

4.13 Pseudo-Code für den Raycaster und Ebeneninterpolation für die Standardmethode 39

4.14 Pseudo-Code für die Integrationsmethode Röntgen 40

4.15 Pseudo-Code für die Interpolation mit der Kombinationsmethode 43

5.1 Der für die Hardware-Beschleunigung relevante Teil der OpenGL-Grafikpipeline 45

5.2 Beispiel eines vollen Gitter und der hierfür benötigten Texturgrößen 47

5.3 Pseudo-Code für die Erstellung der Texturen 49

5.4 Umrechnung der Gitter- in Texturkoordinaten für eine Textur der Größe 5 (Level 2) 50

5.5 Pseudo-Code für die Interpolation mit der Hardware-beschleunigten Kombinationsmethode 50

6.1 Vergleich der Artefaktanfälligkeit von ISO-Flächen zur Röntgen-Integrationsmethode 53

6.2 Kompressionsverhalten des analytischen Datensatzes 54

6.3 Kompressionsverhalten des gefilterten `head` Datensatzes 55

6.4 Wiedergabequalität eines unstetigen Testdatensatzes in Abhängigkeit des Levels 56

6.5 Artefakte bei verschiedenen Wavelets (aus [10, Seiten 149ff]) 57

6.6 ISO-Flächen-Bestimmung mit Haar-Wavelets zum Vergleich (aus [10, Seiten 149ff]) 57

6.7 Vergleich der Standardmethode, der Kombinationsmethode, der Hardware-beschleunigte Kombinationsmethode und des vollen Gitters 61

6.8 Abhängigkeit der Genauigkeit im Framebuffer vom Level bei der Hardware-beschleunigten Kombinationsmethode 62

6.9 Unbrauchbare ISO-Flächen mit der Hardware-beschleunigten Kombinationsmethode aufgrund geringer Framebuffer-Tiefe 62

6.10 Beispiel eines Strömungsdatensatzes, der nicht gut für die Komprimierung mittels dünner Gitter geeignet ist (siehe Text) 62

6.11 Die Kernspinresonanz-Tomographieaufnahme eines Kopfes als Beispiel eines unstetigen, hochfrequenten und daher für dünne Gitter unbrauchbaren Datensatzes 63

6.12 Röntgen- und MIP-Beleuchtungsmodell am Beispiel des Cavity4-Datensatzes, berechnet mit der Kombinationsmethode 64

6.13 Zwei ISO-Flächen des Cavity4-Datensatz, Kombinationsmethode 64

6.14 Beispiel-Ansichten des Cavity5-Datensatzes 64

6.15 Die Orbitalfunktionen des Wasserstoff-Atoms 65

Tabellenverzeichnis

2.1 Größe und somit Speicherbedarf der vollen und dünnen Gitter, Speicherbedarf der Kombinationsmethode und benötigter Texturspeicher für die Hardware-beschleunigte Kombinationsmethode 18

2.2 Anzahl wesentlicher arithmetischer Operationen für Interpolation auf dünnen Gittern, für die Standardmethode ohne und mit Rand, sowie für die Kombinationsmethode 20

5.1 Genauigkeit der Interpolation beziehungsweise Anzahl gültiger Bits im Framebuffer für unterschiedliche Level und Farbtiefen 51

6.1 Speicherbedarf der verschiedenen Interpolationsroutinen 58

6.2 Rechenzeit, Samples/Sekunde und Beschleunigungsfaktoren in Abhängigkeit von der Interpolationsmethode und vom Level 59

6.3 Samples/Sekunde in Abhängigkeit von der Bildgröße 60

6.4 Samples/Sekunde der Hardware-beschleunigten Kombinationsmethode in Abhängigkeit von der Grafikkarte 61

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, daß die Arbeit veröffentlicht wird und daß in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Erlangen, den 15. Juli 1998

(Matthias Hopf)