

# Parallele Volumenvisualisierung auf SIMD Maspar

Studienarbeit im Fach Informatik

vorgelegt von

Matthias Hopf

geb. 27. September 1971 in München

angefertigt am

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

Betreuer: Dr. Roberto Grosso  
Betreuender Hochschullehrer: Prof. Dr. Thomas Ertl

Beginn der Arbeit: 09. August 1995

Abgabe der Arbeit: 02. April 1996



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>5</b>
1.1	Aufgabenstellung . . . . .	5
1.2	Motivation . . . . .	5
<b>2</b>	<b>Theoretische Grundlagen</b>	<b>7</b>
2.1	Direkte Volumenvisualisierung durch Raycasting . . . . .	7
2.2	Transferfunktionen und Beleuchtungsberechnung . . . . .	8
2.2.1	Röntgen . . . . .	8
2.2.2	Mip . . . . .	9
2.2.3	Emission-Absorption . . . . .	9
2.3	Die Shear-Warp-Transformation . . . . .	11
2.3.1	Prinzip und Eigenschaften . . . . .	11
2.3.2	Herleitung der Abbildungsmatrizen . . . . .	12
2.4	Schnelle z-Achsen-Rotation . . . . .	14
2.5	Speicherbedarf . . . . .	16
<b>3</b>	<b>Entwicklungsumgebung</b>	<b>19</b>
3.1	SIMD-Parallelrechner Maspar MP-I . . . . .	19
3.2	Programmiersprache MPL . . . . .	22
3.3	IRIS Explorer Module . . . . .	24
<b>4</b>	<b>Konzepte und Programmdesign</b>	<b>27</b>
4.1	Hauptprogramm raycast . . . . .	27
4.1.1	Unterteilung des Programms in Handlermodule . . . . .	27
4.1.2	Verteilte Datenstrukturen . . . . .	30
4.1.3	Parameterhandling . . . . .	35
4.1.4	Kommunikation mit dem Explorer-Modul . . . . .	38
4.2	Beschreibung der einzelnen Handler . . . . .	41
4.2.1	Typ RAY_H_INIT: Initialisierung . . . . .	41
4.2.2	Typ RAY_H_EXIT: Aufräumen . . . . .	41
4.2.3	Typ RAY_H_INPUT: Volumen einlesen . . . . .	41
4.2.4	Typ RAY_H_OUTPUT: Bild ausgeben . . . . .	42
4.2.5	Typ RAY_H_TRANSFORM: Transformation des Volumens . . . . .	43
4.2.6	Typ RAY_H_START: Initialisierung vor Raycast-Schleife . . . . .	44
4.2.7	Typ RAY_H_STOP: Aufräumen nach Raycast-Schleife . . . . .	44
4.2.8	Typ RAY_H_POS: Ermittlung eines Volumenwertes . . . . .	45
4.2.9	Typ RAY_H_TRANSFER: Transferfunktion . . . . .	45
4.2.10	Typ RAY_H_INTEGRAL: Integration . . . . .	46
4.2.11	Typ RAY_H_STEP: Eine Ebene weiter in der Raycast-Schleife . . . . .	46
4.2.12	Typ RAY_H_RECOLOR: Abschließende Transformation des Bildes . . . . .	47
4.3	Explorer Kommunikationsmodul RaycastMaspar . . . . .	48

4.3.1	Startup und Kommunikation mit dem Hauptprogramm . . . . .	49
4.3.2	Kommunikation mit anderen Explorer-Modulen . . . . .	49
4.4	Explorer Optionsmodul <code>RayOptions</code> . . . . .	50
4.4.1	Prinzip . . . . .	50
4.4.2	Realisierung als Explorer-Modul . . . . .	52
<b>5</b>	<b>Ergebnisse</b>	<b>53</b>
5.1	Verifikation der erhaltenen Bilder . . . . .	53
5.2	Geschwindigkeitsvergleich und Speedup-Messungen . . . . .	55
5.3	Nutzen der Interpolationsfunktionen . . . . .	58
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	<b>60</b>

# 1 Einleitung

## 1.1 Aufgabenstellung

Ziel dieser Studienarbeit ist es, einen Algorithmus der Klasse der Volume-Raycasting-Verfahren zur direkten Volumenvisualisierung für das SIMD-Programmiermodell der Maspar MP-I zu entwickeln und auf dem Parallelrechner zu implementieren. Die Bedienung und Ausgabe des Programms soll über ein Modul in das IRIS Explorer Visualisierungspaket eingebunden werden.

## 1.2 Motivation

Viele wissenschaftliche Untersuchungen — wie zum Beispiel die Computertomographie in der Medizin — ergeben bei ihrer Durchführung große dreidimensionale skalare Datenmengen. Zur Visualisierung dieser Daten werden auch heute noch oft nur mehrere Schnitte durch das Datenvolumen nebeneinander betrachtet. Selten kann mit dieser Methode der Detailreichtum der Daten ausgeschöpft werden und oft bleiben die interessantesten Aspekte unbemerkt. Mit der heute verfügbaren Prozessorleistung bietet sich jedoch die Möglichkeit an, die erhaltenen Daten direkt am Computer zu visualisieren, indem der Rechner dem Betrachter eine zweidimensionale Projektion der Volumendaten präsentiert.

Die heute bekannten Verfahren zur Volumenvisualisierung zerfallen in zwei Klassen:

- Konturflächenbestimmung mit anschließender Darstellung der geometrischen Flächen
- direkte Volumenvisualisierung

Die Visualisierung der Volumendaten durch Isoflächen 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 auch noch den Vorteil, daß durch halbdurchlässige Volumenelemente auch verborgene Details erkannt werden können.

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 in einer Schleife sämtliche Voxel und berechnen ihren Beitrag zu den einzelnen Pixeln. Rückwärts-Projektionsverfahren durchlaufen hingegen sämtliche Pixel und berechnen mittels Raycasting, welche Voxel einen Beitrag hierzu liefern.

Um die direkte Volumenvisualisierung praktisch nutzbar zu machen, sind interaktive Berechnungsraten nötig. Allen hier vorgestellten Verfahren ist jedoch gemeinsam, daß sie äußerst berechnungsintensiv sind. Aus diesem Grund wurden schon frühzeitig Versuche unternommen, parallele Architekturen zu benutzen. Hierzu wird das Problem entweder im Bildraum oder im Objektraum nach dem divide-and-conquer Paradigma aufgeteilt.

Es existieren verschiedene Ansätze für unterschiedliche Parallelarchitekturen. Für Implementierungen auf SIMD Rechnern sind durch die Einschränkungen des Programmiermodells einige Kriterien zu beachten. Ganz entscheidend ist die Gewährleistung der Datenlokalität, da die einzelnen Prozessoren typischerweise nicht über genügend Speicher verfügen, um den gesamten Datensatz aufzunehmen und dieser deshalb auf den Knoten verteilt werden muß. Wichtig ist weiterhin, daß der Algorithmus in seiner Struktur sehr regulär ist und dadurch die Arbeit für die einzelnen Knoten zu jedem Zeitpunkt gleich ausfällt.

Dies kann erreicht werden, indem man das Volumen unmittelbar vor der Projektion durch geeignete Transformationen parallel zur Bildebene ausrichtet, ohne dabei das projizierte Bild zu verändern. Die Shear-Warp Transformation ist hierfür gut geeignet, da für sie nur lokale Kommunikation benötigt wird und der Kommunikationsaufwand bei parallelen Algorithmen meist der begrenzende Faktor ist. Leider ist durch die Struktur des Algorithmus und der Datenaufteilung eine perspektifische Projektion auf SIMD-Rechnern nahezu unmöglich.

Es existieren bereits mehrere Arbeiten zu diesem Thema. Unter anderem wurde in [10], [12] ebenfalls ein Shear-Warp Algorithmus auf einem SIMD-Rechner implementiert. Motiviert wurde die vorliegende Arbeit auch von [1] und [5], [6]. Diese Arbeiten behandeln allerdings die Implementierung des Algorithmus auf MIMD-Maschinen und können deshalb nicht direkt verglichen werden.

Allen diesen Implementierungen ist jedoch gemeinsam, daß keinerlei Angaben über die Strukturierung der Lösungen gemacht wurden. Es muß deshalb davon ausgegangen werden, daß die einzelnen Module der Programme sehr voneinander abhängen und nicht ohne weiteres ausgetauscht werden können. Um das Programm leichter ergänzen zu können und die Beleuchtungsmodelle und Interpolationsroutinen auch zur Laufzeit wechseln zu können, habe ich Konzepte zur Aufgabengliederung entworfen, die sowohl die nötige Effizienz aufweisen, als auch die Implementierung zumindest teilweise verbergen.

## 2 Theoretische Grundlagen

### 2.1 Direkte Volumenvisualisierung durch Raycasting

Beim Raycasting werden die Lichtstrahlen zurückverfolgt, die letztendlich 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 im allgemeinen die Samplingpunkte nicht mit den Zentren der Volumenelemente zusammen, da das Volumen aus den unterschiedlichsten Richtungen betrachtet wird. In der Regel werden deswegen die Samplingwerte durch trilineare Interpolation der angrenzenden Voxel gewonnen; Verfahren höherer oder niedrigerer Ordnung ergeben dementsprechend bessere oder schlechtere Bilder.

Wie die Abbildung 1 zeigt, besteht kein direkt ersichtlicher Zusammenhang zwischen zu betrachtenden Voxeln paralleler Strahlen. Da dies die Parallelisierung auf SIMD-Rechnern erheblich erschwert, wird in dieser Studienarbeit der Ansatz verfolgt, das Volumen durch eine Shear-Warp Transformation senkrecht zu den Augenstrahlen auszurichten. Dann können die Strahlen so gewählt werden, daß die Samplingpunkte in den einzelnen Volumenebenen liegen und außerdem die zu betrachtenden Voxel für alle Strahlen kongruent sind.

Bevor ich näher auf die Shear-Warp Transformation eingehe, möchte ich noch die implementierten Beleuchtungsmodelle vorstellen, da diese von der Transformation nicht betroffen sind.

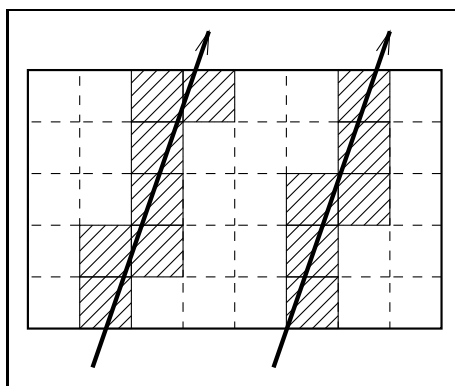


Abbildung 1: Von parallelen Strahlen geschnittene Voxel

## 2.2 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 ein, aber bereits etwas komplexere Modelle wie zum Beispiel Emission-Absorption brauchen für die Modellierung von Transparenz zumindest diese vier Werte. Aufwendigere Modelle können durchaus noch mehr Farbwerte benötigen.

In der derzeitigen Implementierung wird als Transferfunktion ein einfaches Table-Lookup-Verfahren benutzt, um aus dem eindimensionalen Volumenwert einen Farbwert zu gewinnen. In [9] wird die Verwendung einer besseren Transferfunktion vorgeschlagen, die zum Beispiel noch den lokalen Gradienten und gegebenenfalls die zweite Ableitung der Volumendaten verwendet, um sie zu klassifizieren und den Farbwert dementsprechend zu wählen.

Die Integrationsformel kann intuitiv basiert sein wie zum Beispiel bei Mip oder Röntgen. In diesen Fällen erhält man Bilder mit einem hohem Abstraktionslevel, der nicht die 'realen' Verhältnisse wiedergibt. Während diese Bilder durchaus instruktiv sein können, braucht das menschliche Gehirn doch einiges Training, bevor es hieraus die benötigten Informationen erhalten kann. Zumindest für das ungeschulte Auge sind deshalb physikalisch motivierte Beleuchtungsmodelle besser zugänglich.

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 unter anderem in [3].

### 2.2.1 Röntgen

Röntgen wertet ein einfaches Integral über die Volumenwerte aus. Die einzelnen repräsentativen Wellenlängen beeinflussen sich dabei nicht.

$$I^* = \int_{S_{far}}^{S_{near}} q^*(s) ds \quad \text{für } * \text{ aus } \{r, g, b\} \quad (1)$$

Wird diese Gleichung nach Euler diskretisiert, ergibt sich für

$$\Delta s = (S_{near} - S_{far})/n, \quad s_i := S_{far} + i\Delta s \quad (2)$$

nach Vereinfachung die Summe

$$I^* = \Delta s \sum_{i=0}^{n-1} q^*(s_i). \quad (3)$$

Diese Summe wurde in dem *roentgen*-Integrationshandler (siehe 4.2.10) implementiert.

### 2.2.2 Mip

Der *mip*-Integrationshandler sucht für jede Wellenlänge unabhängig das Maximum auf dem Augenstrahl. Im allgemeinen wird aber Mip nur auf Graustufenvolumen angewandt.

$$I^* = \max_{s \in [S_{far}, S_{near}]} q^*(s) \quad \text{für } * \text{ aus } \{r, g, b\} \quad (4)$$

Im Diskreten ergibt dies:

$$I^* = \max_{i \in \mathbf{Z}_n} q^*(s_i) \quad s_i \text{ siehe (2), } \mathbf{Z}_n = \{0, \dots, n-1\} \quad (5)$$

### 2.2.3 Emission-Absorption

Das Emission-Absorption-Modell ist ein einfaches physikalisch basiertes Beleuchtungsmodell ohne Scattering-Effekte und ohne lokale Beleuchtungsberechnung. Durch dieses Modell lassen sich Volumen sehr gut darstellen, während die teilweise darin enthaltenen Flächen schlecht zur Geltung kommen.

Da keine lokale Beleuchtung in das Modell mit eingeht, müssen die Gradienten der Voxel nicht berechnet werden, was die Berechnungsdauer deutlich erniedrigt.

Nach [3, Seite 13-14] ergibt sich bei der Vereinfachung der Transfergleichung das Integral

$$I^* = I_{far}^* e^{-\tau(S_{far}, S_{near})} + \int_{S_{far}}^{S_{near}} q^*(s) e^{-\tau(s, S_{near})} ds \quad \text{für } * \text{ aus } \{r, g, b\} \quad (6)$$

mit der sogenannten optischen Tiefe

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(s) ds . \quad (7)$$

Mit  $q^*$  wird dabei der Emissions-Koeffizient, mit  $\kappa$  der Absorptions-Koeffizient bezeichnet.  $I_{far}^*$  wird über die Randbedingungen (Hintergrundfarbe) eindeutig spezifiziert.

Mit den aus (2) und (7) gewonnenen Gleichungen

$$\tau(s_{k_1}, s_{k_2}) = \sum_{j=k_1+1}^{k_2} \tau(s_{j-1}, s_j) , \quad s_0 = S_{far} \quad \text{und} \quad s_n = S_{near} \quad (8)$$

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

$$\theta_k = e^{-\tau(s_{k-1}, s_k)} \quad \text{und} \quad b_k^* = \int_{s_{k-1}}^{s_k} q^*(s) e^{-\tau(s, s_k)} ds \quad (9)$$

$I^*$  wie folgt darstellen läßt:

$$\begin{aligned}
I^* &= I_{far}^* e^{-\left(\sum_{k=1}^n \tau(s_{k-1}, s_k)\right)} + \sum_{k=1}^n \int_{s_{k-1}}^{s_k} q^*(s) e^{-\left(\tau(s, s_k) + \sum_{j=k+1}^n \tau(s_{j-1}, s_j)\right)} ds \\
&= I_{far}^* \prod_{k=1}^n \theta_k + \sum_{k=1}^n b_k^* \prod_{j=k+1}^n \theta_j = \sum_{k=0}^n b_k^* \prod_{j=k+1}^n \theta_j \quad \text{mit} \quad b_0^* := I_{far}^* \quad (10)
\end{aligned}$$

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  $\theta_k$  durch

$$\theta_k = e^{-\kappa_k \Delta s} \quad \text{und} \quad (11)$$

$$\begin{aligned}
b_k^* &= \int_0^{\Delta s} q_k^* e^{-\kappa_k s} ds = q_k^* \left( \frac{1}{-\kappa_k} e^{-\kappa_k \Delta s} + \frac{1}{\kappa_k} \right) \\
&= \frac{q_k^*}{\kappa_k} (1 - \theta_k) \quad (12)
\end{aligned}$$

im Diskreten exakt dargestellt werden. Auch wenn die Exponentialfunktion in (12) nur einmal für jeden Farbwert pro Bild berechnet werden muß, indem man die Einträge der Farbtabelle transformiert, wird  $b_k^*$  oft nur über das 1. Taylorpolynom zum Entwicklungspunkt  $(\kappa_k \Delta s)_0 = 0$  approximiert:

$$\begin{aligned}
b_k^* &= \frac{q_k^*}{\kappa_k} (1 - e^{-\kappa_k \Delta s}) = \frac{q_k^*}{\kappa_k} (1 - (1 - \kappa_k \Delta s + o(\kappa_k \Delta s))) \\
&\approx q_k^* \Delta s \quad \text{für} \quad \kappa_k \Delta s \ll 1 \quad (13)
\end{aligned}$$

In der Praxis hat sich gezeigt, daß Voxel mit einer niedrigen Opazität in der Regel als nur schwach emittierend dargestellt werden sollen, da sonst bereits das Integral über wenige Voxel den erlaubten Intensitätsbereich verläßt. Im Colortable-Modul des IRIS Explorer ist es allerdings schwer, sehr niedrige Farbwerte exakt zu bearbeiten. Es wäre daher sinnvoll, sämtliche Farbwerte vor ihrer Benutzung mit ihrer Opazität zu multiplizieren. Da sich diese Multiplikation nur auf die Farbtabellewerte außerhalb der Beleuchtungsberechnung direkt auswirkt, wird durch diese Modifikation die physikalisch korrekte Modellierung des Beleuchtungsmodells nicht negativ beeinflusst. In dem implementierten Handler (siehe 4.2.9) ist für die Aktivierung dieser Funktion bereits ein Schalter vorgesehen, die eigentliche Farbmodifikation ist aber noch nicht implementiert.

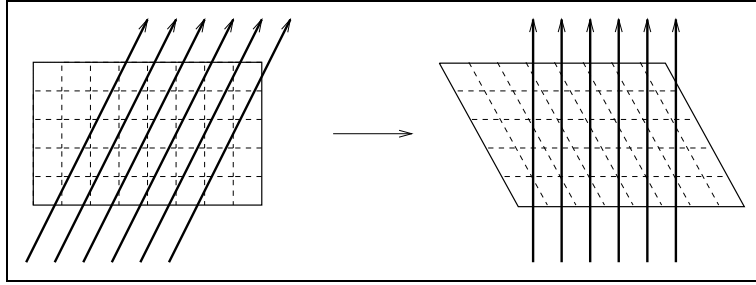


Abbildung 2: Ausrichtung des Volumens senkrecht zu den Augenstrahlen

## 2.3 Die Shear-Warp-Transformation

Wird ein Volumen mit Hilfe einer Abbildung ins Zweidimensionale projiziert, so kann man die in der Abbildung  $\mathcal{A}$  enthaltene dreidimensionale Rotation  $\mathcal{R}$  in eine Scherung des Volumens entlang einer Volumenachse und in eine zweidimensionale Skalierung und Rotation senkrecht zur Projektionsachse zerlegen. Sowohl die Scherung des Volumens als auch die Skalierung des berechneten Bildes sind sehr effizient durchführbar und lassen sich gut parallelisieren. Des weiteren können etliche Alias-Effekte schon im Ansatz reduziert werden.

### 2.3.1 Prinzip und Eigenschaften

Mit der Shear-Warp-Transformation wird das zu betrachtende Volumen senkrecht zu den Augenstrahlen ausgerichtet. Dazu wird das Koordinatensystem, in dem die Projektion stattfindet, entlang einer Volumenachse so geschert, daß die Augenstrahlen senkrecht auf den einzelnen Ebenen des Volumens eintreffen. Um die Transformation gemäß Abbildung 2 durchzuführen, ist es notwendig, eine Achse des Koordinatensystems so zu wählen, daß die Winkel zwischen den Augenstrahlen und dieser Achse bezüglich den beiden anderen Koordinatenachsen stets kleiner als  $\frac{\pi}{2}$  sind. Wie ich in 2.5 noch erläutern werde, wird das berechnete Zwischenbild und damit der Speicherverbrauch beliebig groß, wenn sich einer der Winkel  $\frac{\pi}{2}$  nähert. Des weiteren hängt die Darstellung des Volumens im Speicher von der Koordinatenachse ab, weshalb man im allgemeinen als Koordinatenachse die  $z$ -Achse fest vorgibt und das Volumen einer Translation unterwirft, wenn einer der beiden Winkel  $\frac{\pi}{4}$  überschreitet.

Wird der Abstand  $\Delta x$  der einzelnen Augenstrahlen voneinander so gewählt, daß die Strahlen in der ersten Volumenebene durch die Voxelmittelpunkte gehen, bleibt der Abstand der von parallelen Augenstrahlen geschnittenen Voxel konstant. Auf diese Weise werden Alias-Effekte reduziert und die Implementierung gerade auf SIMD-Maschinen erleichtert. Allerdings wird dies durch eine abschließende Skalierung erkauft, die bei bewegten Bildern nötig ist, um bei allen Bildern die gleiche Pixelgröße und Aspect-Ratio zu gewährleisten.

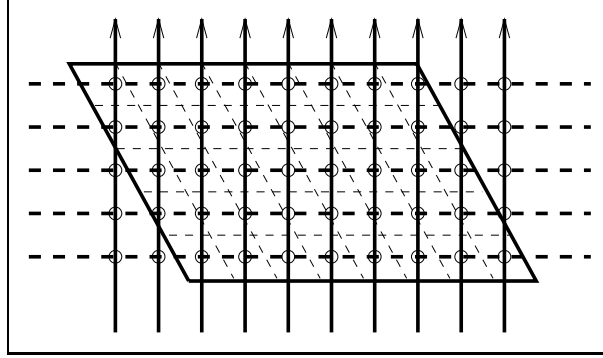


Abbildung 3: Auswahl der Samplingpunkte

Es ist klar, daß mit dieser Methode keine Bilder beliebiger Größe berechnet werden können, da der Abstand der Augenstrahlen fest vorgegeben ist. Allerdings kann als Strahlenabstand auch  $\frac{1}{n}$ tel des ursprünglich berechneten  $\Delta x$  gewählt werden. Die resultierenden Bilder werden dabei um den Faktor  $n$  vergrößert. Es sind zwar nicht mehr Details vorhanden, da die betrachtete Datenmenge sich nicht erhöht, aber die subjektive Bildqualität kann so entscheidend verbessert werden.

Da alle Augenstrahlen konstruktionsbedingt durch die Mittelpunkte der Voxel der ersten Volumenebene gehen, liegt es nahe, diese Punkte als erste Samplingpunkte zu benutzen. Wird jetzt noch der Abstand der einzelnen Volumenebenen als Schrittweite gewählt — wie in Abbildung 3 ersichtlich — kann die trilineare Interpolation zur Gewinnung der Samplingwerte ohne Verfälschung der Ergebnisse durch eine bilineare Interpolation ersetzt werden. Diese Vereinfachung existiert natürlich nicht mehr, wenn man zur Verbesserung der Abbildung mittels Oversampling die Anzahl der Samplingpunkte vervielfacht.

Bis jetzt wurden die Strahlen nicht betrachtet, die das Volumen verlassen oder nicht durch die Stirnseite eintreten. Untersucht man die Strahlenwege genauer, stellt man fest, daß die austretenden Strahlen wie in Abbildung 4 genau den neu eintretenden Strahlen entsprechen, wenn man das Volumen als Torus ansieht. Sobald ein Strahl also das Volumen verläßt, speichert man seinen Farbwert, assoziiert ihn mit einen neuen Strahl und paßt seine Koordinaten so an, daß er auf der anderen Seite des Volumens zu liegen kommt. Auf Parallelrechnern wie der Maspar MP-I kann die Anpassung der Koordinaten entfallen, wenn die Torus-Eigenschaft der lokalen Kommunikationsnetze ausgenutzt werden kann.

### 2.3.2 Herleitung der Abbildungsmatrizen

Die gewünschte Abbildung  $\mathcal{A} := \mathcal{P} \cdot \mathcal{R} \cdot \mathcal{E}$  ist in der Regel durch die Rotationsmatrix  $\mathcal{R}$  und die Voxelgröße  $\vec{e}$  gegeben. Man kann die Abbildung  $\mathcal{A}$  nun in ein Produkt einer Transposition  $\mathcal{T}$ , einer Scherung  $\mathcal{S}$ , der Projektion  $\mathcal{P}$  und einer anschließenden Rotation mit

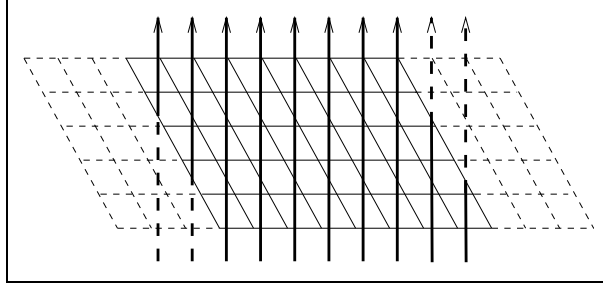


Abbildung 4: Wrap-Around aus dem Volumen austretender Strahlen

Skalierung im Zweidimensionalen  $\mathcal{Z}$  zerlegen:

$$w \cdot \mathcal{A} = w \cdot \mathcal{P} \cdot \mathcal{R} \cdot \mathcal{E} =: \mathcal{Z} \cdot \mathcal{P} \cdot \mathcal{S} \cdot \mathcal{T} \quad (14)$$

mit

$$\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad (15)$$

$$\mathcal{R} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, \quad \mathcal{R} \text{ orthonormal, } \det \mathcal{R} = +1 \quad (16)$$

$$\mathcal{E} = \begin{pmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & e_z \end{pmatrix}, \quad \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} =: \vec{e}, \quad (17)$$

$$\mathcal{Z} = \begin{pmatrix} k & l & 0 \\ m & n & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (18)$$

$$\mathcal{S} = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}, \quad -1 \leq x, y \leq 1, \quad (19)$$

$$\mathcal{T} = \text{Transpositionsmatrix, } \det \mathcal{T} = +1$$

$$w > 0$$

Mit  $w$  wird dabei der noch festzulegende abschließende Gesamtskalierungsfaktor bezeichnet (siehe 2.5).

Eingesetzt in (14) ergibt dies:

$$\mathcal{A} = \begin{pmatrix} wae_x & wbe_y & wce_z \\ wde_x & wee_y & wfe_z \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} k & l & xk + yl \\ m & n & xm + yn \\ 0 & 0 & 0 \end{pmatrix} \quad (20)$$

Hieraus folgen unmittelbar die Koeffizienten der Matrix  $\mathcal{Z}$ :

$$\Rightarrow k = wae_x, \quad l = wbe_y, \quad m = wde_x, \quad n = wee_y \quad (21)$$

Die Koeffizienten  $x$  und  $y$  der Matrix  $\mathcal{S}$  ergeben sich aus (20) wie folgt:

$$wce_z = xk + yl \quad \wedge \quad wfe_z = xm + yn \quad (22)$$

$$\Rightarrow \begin{pmatrix} wae_x & wbe_y \\ wde_x & wee_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} wce_z \\ wfe_z \end{pmatrix} \quad (23)$$

$$\Rightarrow x = \frac{e_z}{e_x} \cdot \frac{ce - bf}{ae - bd}, \quad y = \frac{e_z}{e_y} \cdot \frac{af - cd}{ae - bd} \quad (24)$$

Die Matrix  $\mathcal{T}$  läßt sich nicht so einfach durch eine Formel ausdrücken. Eine Möglichkeit ihrer Berechnung besteht darin, die soeben ermittelten Werte  $x$  und  $y$  auf ihre Grenzen  $-1$  und  $+1$  zu testen und für den Fall des Überschreitens die Transpositionsmatrix so anzupassen, daß die neu ermittelten Werte wieder in ihren Grenzen liegen. In der Praxis bedeutet das, daß nach der Berechnung von  $x$  und  $y$  das Volumen gegebenenfalls um  $\frac{\pi}{2}$  gedreht und die Werte neu berechnet werden.

Die Matrix  $\mathcal{Z}$  mit den Elementen  $k, l, m$  und  $n$  beschreibt eine zweidimensionale Skalierung und Rotation, die noch weiter zerlegt werden muß, um effizient auf einem Multiprozessor-system durchgeführt werden zu können.

## 2.4 Schnelle $z$ -Achsen-Rotation

Man kann  $\mathcal{Z}$  in ein Produkt einer Skalierung  $\mathcal{K}$  und einer einfachen Rotation  $\mathcal{Q}$  zerlegen. Da diese Transformationen nur im Zweidimensionalen durchgeführt werden, werde ich im Folgenden nur mit den direkt abgeleiteten  $2 \times 2$  Matrizen arbeiten:

$$\mathcal{Z}' = \mathcal{K}' \cdot \mathcal{Q}' \quad (25)$$

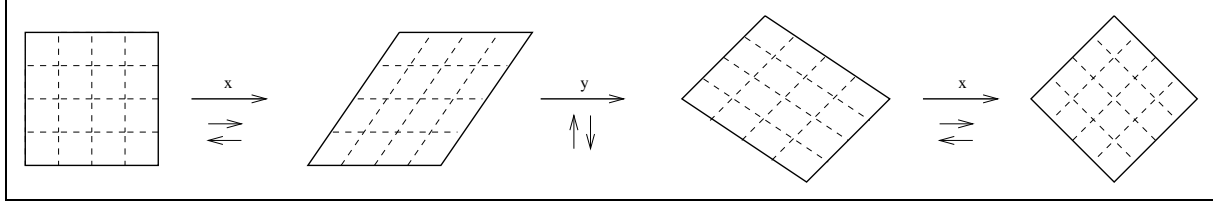


Abbildung 5: Zweidimensionale Rotation durch mehrere Scherungen

mit

$$\mathcal{Z}' = \begin{pmatrix} k & l \\ m & n \end{pmatrix} \quad \text{mit } k, l, m, n \text{ aus (21)} \quad (26)$$

$$\mathcal{K}' = \begin{pmatrix} p & 0 \\ 0 & q \end{pmatrix} \quad (27)$$

$$\mathcal{Q}' = \begin{pmatrix} t & -s \\ s & t \end{pmatrix} \quad \text{mit } s^2 + t^2 = 1 \quad (28)$$

Wird nach  $p$ ,  $q$ ,  $s$  und  $t$  aufgelöst, ergibt sich:

$$p = \sqrt{k^2 + l^2}, \quad t = \frac{k}{p} \quad (29)$$

$$q = \sqrt{m^2 + n^2}, \quad s = \frac{m}{q}$$

Die Rotation  $\mathcal{Q}$  kann nun weiter in ein Produkt aus drei zweidimensionalen Scherungen zerlegt werden, die wie in Abbildung 5 ersichtlich hintereinander ausgeführt werden:

$$\mathcal{Q}' = \mathcal{S}'_x \cdot \mathcal{S}'_y \cdot \mathcal{S}'_x \quad (30)$$

mit

$$\mathcal{S}'_x = \begin{pmatrix} 1 & s_x \\ 0 & 1 \end{pmatrix}, \quad \mathcal{S}'_y = \begin{pmatrix} 1 & 0 \\ s_y & 1 \end{pmatrix} \quad (31)$$

Setzt man nun  $s$  und  $t$  ein, ergibt sich nach [10]

$$\mathcal{S}'_x = \begin{pmatrix} 1 & -\sqrt{\frac{1-t}{1+t}} \\ 0 & 1 \end{pmatrix} \quad \text{und} \quad \mathcal{S}'_y = \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix}. \quad (32)$$

Diese reinen Scherungen können auf Parallelrechnern mit lokaler XNet Kommunikation wie der Maspar MP-1 effizient durchgeführt werden, da die Kommunikationsstruktur sehr regelmäßig ist.

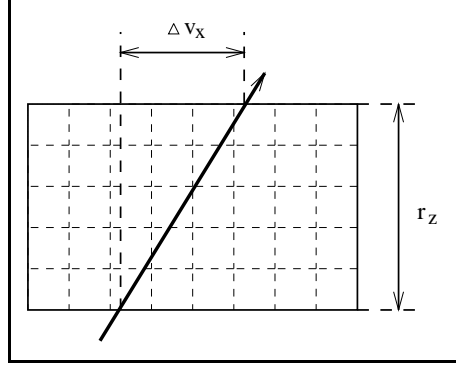


Abbildung 6: Zurückgelegte Entfernung eines Strahls beim Weg durch das Volumen

## 2.5 Speicherbedarf

Bereits relativ kleine Volumen der Größe  $256^3$  benötigen auf kleineren Parallelrechnern mit ihren 16 MByte den gesamten verfügbaren Hauptspeicher. Um so viel Speicher wie möglich für das Volumen frei zu halten, ist es nötig, den Speicherverbrauch des berechneten Bildes möglichst klein zu halten, ohne Einbußen in der Qualität hinnehmen zu müssen. Ein Bild der Größe  $512^2$  benötigt zum Beispiel in der verwendeten Float-RGBA-Darstellung immerhin 4 MByte.

Die Bildgröße ist mit der Anzahl der betrachteten Augenstrahlen identisch. Neben den Augenstrahlen, die in das Volumen durch die Stirnseite eintreten, müssen noch die Strahlen betrachtet werden, die das Volumen verlassen und von der anderen Seite erneut in das Volumen geschickt werden.

Gemäß Abbildung 6 verläßt ein Strahl in Richtung  $\vec{v}$  mit  $|\vec{v}| > 0$ , der das Volumen der Größe  $\vec{r}$  an der Stirnseite betritt, die Ebene durch die Rückseite des Volumens in einer Entfernung  $\Delta\vec{v}$  vom Eintrittspunkt mit

$$\Delta\vec{v} = \begin{pmatrix} r_z \frac{v_x}{v_z} \\ r_z \frac{v_y}{v_z} \\ r_z \end{pmatrix} \quad (33)$$

Die Größe des benötigten Zwischenbildes  $\vec{o}$  für jedes Bild erhält man, indem man den Weg eines an der äußersten Ecke eintretenden Strahles verfolgt:

$$\vec{o} = \begin{pmatrix} r_x + |\Delta v_x| \\ r_y + |\Delta v_y| \end{pmatrix} \quad (34)$$

Für  $|\frac{v_{x,y}}{v_z}| \rightarrow \infty$  geht auch  $|o_{x,y}|$  gegen unendlich. Der Grenzfall  $v_z = 0$  tritt ein, wenn der Winkel zwischen der  $z$ -Achse und den Augenstrahlen  $\frac{\pi}{2}$  beträgt.

Das Volumen kann offensichtlich immer so transponiert werden, daß der Winkel zwischen Haupt-Projektionsachse und Augenstrahlen  $\frac{\pi}{4}$  nie überschreitet. In diesem Fall erhält man als maximale Bildgröße

$$\underline{d} = \begin{pmatrix} r_x + r_z \\ r_y + r_z \end{pmatrix} \quad (35)$$

In meinem Programm gehe ich davon aus, daß die Größe des Volumen in allen drei Dimensionen identisch ist. In diesem Fall reduziert sich (35) auf

$$\underline{d}' = \begin{pmatrix} 2r_x \\ 2r_x \end{pmatrix} \quad \text{mit } r_x = r_y = r_z \quad (36)$$

Volumen mit kleinerer  $z$ -Ausdehnung können auch ohne weiteres für die Berechnung verwendet werden; allerdings kann durch eine Translation eine andere Achse die Haupt-Projektionsachse werden, wodurch Schwierigkeiten bei der Datenaufteilung auf die einzelnen Prozessoren auftreten können oder diese gar unmöglich machen.

Des weiteren beruhen alle bisherigen Berechnungen auf Voxel der Größe  $e_{x,y,z} = 1$ . Haben die Voxel keine Würfelform, können jedoch alle Berechnungen so transformiert werden, daß die Beziehung (35) auch dann noch gilt; allerdings sind die größten möglichen Winkel dann von der Haupt-Projektionsachse abhängig. Aus Gründen der Anschauung habe ich schließlich darauf verzichtet. Eine mögliche Formulierung der Bedingung, wann eine Transposition des Volumens nötig wird, habe ich bereits in 2.3.2 gezeigt.

Das berechnete Zwischenbild muß noch rotiert und skaliert werden. Betrachtet man in Abbildung 5 die Grenzen eines Bildes bei der Rotation, so stellt man fest, daß diese während der gesamten Transformation einen  $\sqrt{2}o'_x \times \sqrt{2}o'_y$  großen Rahmen nicht verlassen, solange der Rotationswinkel  $\frac{\pi}{4}$  nicht überschreitet. Setzt man die Größen aus (36) ein, stellt man fest, daß ein Arbeitsspeicher der Größe

$$\underline{u}' = \begin{pmatrix} 3r_x \\ 3r_y \end{pmatrix} \quad (37)$$

die Bedingungen ausreichend erfüllt.

Um die erhaltene Bildinformation nicht durch den abschließenden Skalierungsschritt zu reduzieren, habe ich mich dazu entschlossen, die erhaltenen Bilder nur gegebenenfalls zu vergrößern, nicht aber zu verkleinern. Der Faktor  $w$  in der Gleichung (14) muß dabei so angepaßt werden, daß  $p$  und  $q$  möglichst nie kleiner als 1 werden, aber 1 auch angenommen wird, um den Speicherverbrauch nicht unnötig zu erhöhen. Für Voxelgrößen, die keiner Würfelform entsprechen, ist keine optimale Lösung für alle drei Haupt-Projektionsachsen möglich.

Die Größe des berechneten Zwischenbildes nimmt ihr Maximum  $\underline{d}'$  dann an, wenn  $x = 1$  und  $y = 1$  sind, die Projektionsachse mit den Augenstrahlen also den Winkel  $\frac{\pi}{4}$  einschließt.

Die erwünschte Größe des Bildes  $g_x = \sqrt{2}r_x$ ,  $g_y = \sqrt{2}r_y$  ist dann aber kleiner als  $\bar{\varrho}'$ . Der gewünschte Effekt kann offensichtlich am ehesten bewirkt werden kann, wenn man  $w = \sqrt{2}$  wählt. In diesem Grenzfall werden die Skalierungsfaktoren  $p, q = 1$ .

Das endgültige skalierte Bild kann höchstens die maximale Ausdehnung wie das rotierte Zwischenbild besitzen, da die größte darzustellende Entfernung — die Raumdiagonale durch das Volumen — skaliert mit  $w$  den Wert  $\sqrt{6} \approx 2,4495 < 3$  nicht überschreitet. Die Faktoren  $p$  und  $q$  beeinflussen die Skalierung also so, daß keine relevanten Bildteile verloren gehen. Es muß aber trotzdem der Speicher für den Skalierungsschritt in der Größe

$$\vec{z}' = \begin{pmatrix} 3\sqrt{2}r_x \\ 3\sqrt{2}r_y \end{pmatrix} \quad (38)$$

alloziert werden, da in dem parallelen Skalierungsschritt die Daten nicht umverteilt werden können. Der zusätzliche Speicher wird benötigt, um das nach der Skalierung größere Bild aufnehmen zu können.

## 3 Entwicklungsumgebung

### 3.1 SIMD-Parallelrechner Maspar MP-I

Die Entwicklung des Programms sollte auf einer Maspar MP-I durchgeführt werden. Dieser Computer gehört zu der Klasse der Feldrechner. Darunter versteht man Parallelrechner mit SIMD-Architektur mit Prozessor-Maskierung.

Die MP-I besteht aus einem UNIX-Frontend, der sogenannten Array Control Unit (ACU), dem Prozessor Array bestehend aus 1024 – 16384 Prozessor Elementen (PEs) und Kommunikationshardware, sowie zusätzlichen parallelen Ein-/Ausgabe-Systemen. Abbildung 7<sup>1</sup> zeigt eine Übersicht über ihre Struktur. Die MP-I des Lehrstuhls IV enthält 1024 Prozessoren mit jeweils 16 KByte Speicher. Für die Performancetests hat uns die Universität Stuttgart freundlicherweise ihre Maspar MP-I mit 16384 Prozessoren zur Verfügung gestellt.

#### Das UNIX-Subsystem

Das Frontend besteht aus einer VAXstation 3520 mit ULTRIX, die über TCP/IP an das Internet angeschlossen ist. Auf dem Frontend werden die Programme für das Prozessor Array cross-compiled und gestartet. Die Warteschlangen für die Programmausführung werden ebenfalls auf dem Frontend verwaltet. Neuere Versionen der MP-I sowie des Nachfolgermodells MP-II werden auch mit einer DEC-Alpha Workstation anstelle der VAXstation ausgeliefert.

Mit der ACU kann das Frontend über Shared Memory und DMA kommunizieren. Es werden Bibliotheksfunktionen sowohl zum Speichertransfer als auch zur Koordination bereitgestellt.

#### Die Array Control Unit

Die ACU übernimmt sämtliche an das Prozessor Array gerichtete Befehle, die nicht auf mehrfachen Datensätzen arbeiten. Sie besteht aus einem RISC-Prozessor mit Harvard-Architektur, 1 MByte physikalischen Befehlsspeicher (4 GByte virtuell) und 128 KByte Datenspeicher. Der Befehlssatz des Prozessors enthält spezielle mikrocodierte Befehle, um effizient mit dem Prozessor Array kommunizieren zu können.

Obwohl die ACU deutlich schneller als ein PE ist, werden bestimmte Befehle wie zum Beispiel Floating-Point Berechnungen zum UNIX-Subsystem geschickt, weil die Berechnung dort trotz Kommunikations- und Koordinationsaufwand schneller vonstatten geht.

---

<sup>1</sup>Entnommen aus [7]

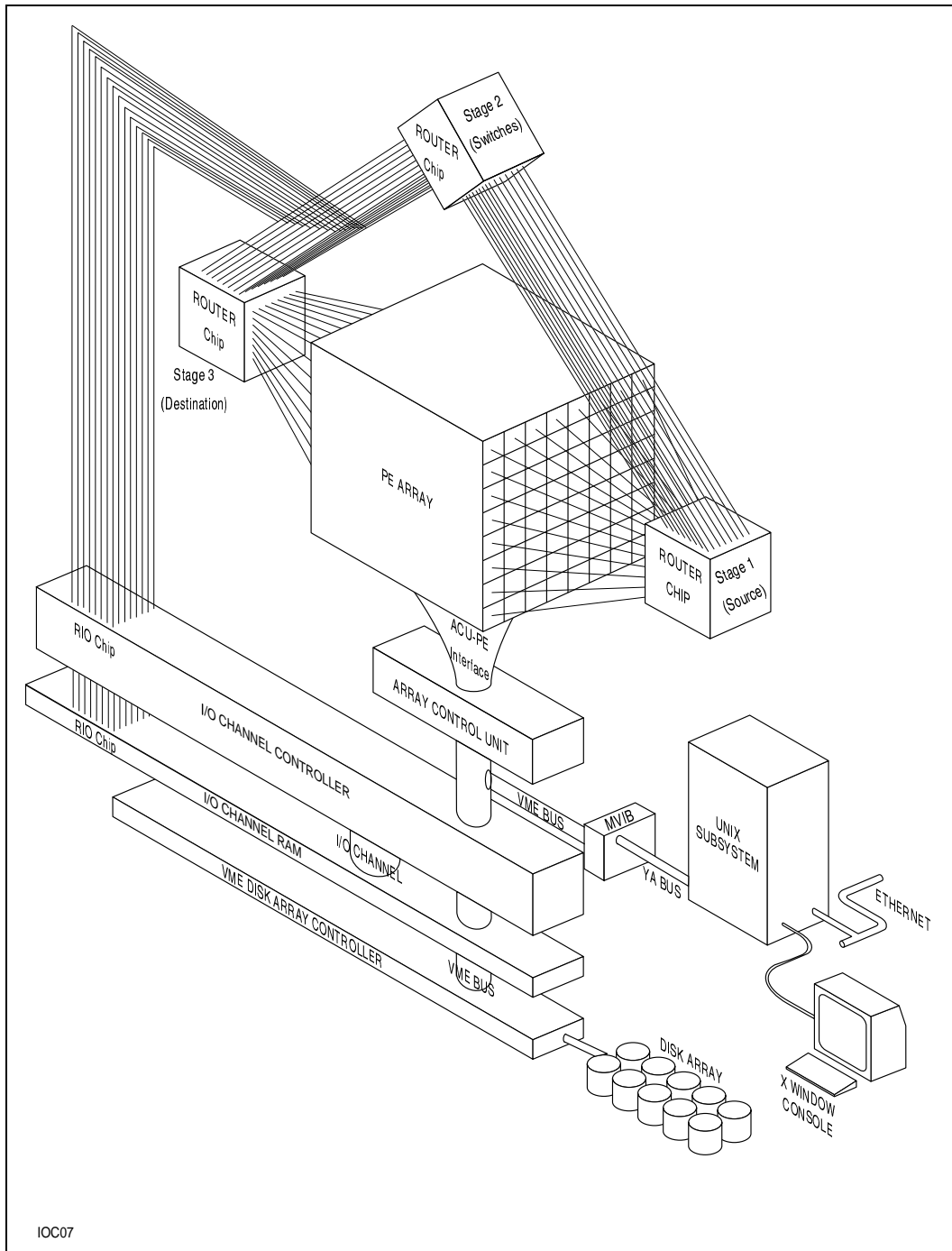


Abbildung 7: Überblick über die Maspar MP-1

## **Das Prozessor Array**

1 bis 16 Prozessorboards mit je 1024 PEs können in der MP-I installiert werden. Jeweils  $4 \times 4 = 16$  Prozessoren werden dabei zu einem Cluster zusammengefaßt, wobei ein Prozessorchip auf dem Board zwei Cluster enthält. Die Aufteilung in Cluster spielt bei der Kommunikation über den Globalen Router eine große Rolle, wie ich unten noch erläutern werde. Jeder Prozessor kann auf mindestens 16 KByte lokalen Speicher zugreifen, prinzipiell ist der Speicher sogar nahezu beliebig ausbaufähig, da die PEs über 32 Bit breite Adreßregister verfügen. Praktisch existieren jedoch nur Konfigurationen mit 16 und 64 KByte je PE.

## **Kommunikation mit der ACU**

Die ACU kann über einen Broadcast Daten simultan an alle PEs senden. In der Gegenrichtung erfolgt die Kommunikation über einen ODER-Reduktionsbaum, über den die aktiven PEs ihre Daten an die ACU zurückschicken können. Es existieren Bibliotheksroutinen zur Summenbildung, Maximumsuche und anderem, die diesen Reduktionsbaum nutzen.

## **Lokale Kommunikation der PEs über XNet**

Die PEs sind in einem zweidimensionalen Array angeordnet und können direkt auf den Speicher ihrer Nachbarn in den vier Himmelsrichtungen sowie den vier diagonalen Richtungen zugreifen. Das Netz ist in Form eines Torus angeordnet, so daß an den Rändern keine Spezialfälle auftreten. Dieses sogenannte X-Netzwerk ist zwar nur ein Bit breit ausgelegt, aber alle PEs können es gleichzeitig nutzen. Man kann auch auf Daten weiter entfernter PEs zugreifen, nur erhöht sich dabei der Zeitaufwand. Für Spezialfälle existieren spezielle Mechanismen, um dazwischenliegende inaktive PEs auszunutzen und zur Geschwindigkeitserhöhung zu einer Pipeline zusammenzuschließen.

## **Globale Kommunikation**

Über den Globalen Router können beliebige PEs miteinander kommunizieren. Der Router besteht aus einem hierarchischen dreistufigen Kreuzschienenverteiler, der auf beiden Seiten mit allen Clustern verbunden ist. Er arbeitet verbindungsorientiert, wobei theoretisch alle Cluster gleichzeitig eine Verbindung aufbauen können. Treten bei einem Verbindungsaufbau Konflikte auf, weil mehrere PEs eines Clusters gleichzeitig eine Verbindung anfordern oder auf mehrere PEs eines Clusters gleichzeitig zugegriffen werden soll, werden die Konflikte automatisch aufgelöst, indem die Anforderungen nacheinander ausgeführt werden. Trotz seiner Leistungsfähigkeit ist der Globale Router deshalb deutlich langsamer als das XNet, das jedoch nur gleichgerichtete Kommunikation erlaubt.

## Prozessor Elemente

Die einzelnen Prozessor Elemente sind einfach aufgebaute 4 Bit Prozessoren mit 48 32-Bit Registern (8 reserviert), einer Fließkommaeinheit für 32 Bit und 64 Bit IEEE- und VAX-Format, einem 4 Bit Bus zum lokalen Speicher sowie einigen zusätzlichen Einheiten für Logik und ACU Kommunikation.

### 3.2 Programmiersprache MPL

Die Maspar MP-I kann in Fortran mit Spracherweiterungen für parallele Datenstrukturen und in MPL programmiert werden. MPL ist auf K&R C basiert, neuere Versionen sind auch im wesentlichen ANSI-kompatibel. MPL besitzt etliche Spracherweiterungen für parallele Datenstrukturen und Programmausführung. Compiliert wird ein MPL-Programm auf dem Frontend mit `mpl_cc`. Man muß keinerlei Einstiegsroutinen bereitstellen, das Programm wird bei seinem Aufruf vom System automatisch zur ACU übertragen und gestartet. Für die meisten Standard-Funktionen der `libc` und der `libm` enthält die `libmpl` plurale Versionen wie zum Beispiel `p_sin()`.

Die zentrale Spracherweiterung besteht in sogenannten pluralen Typen. Variablen, die in ihrer Deklaration `plural` enthalten, werden im lokalen Speicher der PEs abgelegt und sind somit mehrfach vorhanden. Man kann sowohl singuläre (im normalen ACU-Datenspeicher abgelegte) Zeiger auf plurale Daten als auch plurale Zeiger auf singuläre oder plurale Daten definieren. Auf diese Variablen wird genauso zugegriffen, wie man es in einem gewöhnlichen C-Programm gewohnt ist. Wird allerdings ein pluraler Zeiger auf singuläre Daten zum Schreiben dereferenziert, können die Resultate bei Kollisionen nicht vorhergesehen werden. Im Beispiel der Abbildung 8<sup>2</sup> ist `*ohMy = tigers;` eine derartige gefährliche Zuweisung. Des weiteren sind Casts von pluralen Datentypen auf singuläre Typen verboten. Casts von singulären Typen auf plurale sind erlaubt und werden auch automatisch vorgenommen.

Zum Reduzieren pluraler Daten existieren außer diversen Bibliotheksroutinen auch die Primitive `globalor`. Der Wert aller aktiven PEs wird mit dieser Funktion oder-verknüpft. Mit der `proc` Anweisung kann man auch den Wert einer pluralen Variablen eines bestimmten PEs adressieren.

Die PEs können mit pluralen Kontrollanweisungen deaktiviert werden. Am Ende eines solchen Blocks wird die Deaktivierung wieder aufgehoben. Zu den pluralen Kontrollanweisungen gehören `if`, `while`, `for`, `switch` und andere. Mit `all` können kurzfristig alle PEs aktiviert werden. Wichtig ist hierbei noch, daß in einem pluralen `if` mit einem zugehörigen `else`-Block beide Blöcke ausgeführt werden, wenn die plurale Bedingung auf manchen Prozessoren erfüllt ist und auf anderen nicht. Es ist garantiert, daß kein Prozessor beide Blöcke abarbeitet, aber die Ausführungszeit erhöht sich entsprechend. Genauso ist eine Schleife erst dann beendet, wenn alle PEs die Bedingung zum Verlassen der Schleife erfüllen.

---

<sup>2</sup>Entnommen aus [8]

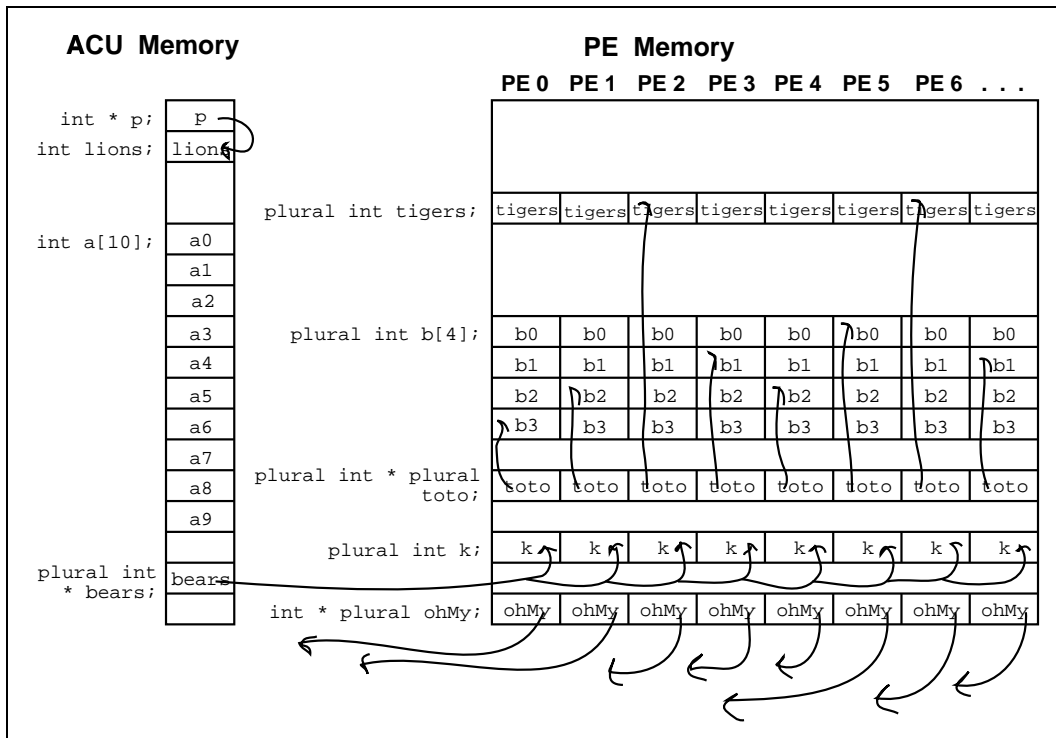


Abbildung 8: Speicherzugriffsmöglichkeiten

Die Variablen `nproc`, `nxproc` und `nyproc` enthalten die Größe des Prozessor Arrays, wobei mit `iproc`, `ixproc` und `iyproc` auf den Index jedes PEs im Prozessor Array zugegriffen werden kann. `nproc` gibt dabei die Gesamtanzahl an Prozessoren an, `nxproc` und `nyproc` die Ausdehnung des Arrays in den zugehörigen Richtungen.

Zur Kommunikation über das XNet existieren die `xnet` Konstrukte, mit denen man spezifizieren kann, auf welchem PE in welcher Entfernung relativ zum aktuellen PE der angegebene Ausdruck ausgewertet wird. Mit `xnetN[1].j` kann man zum Beispiel auf die Variable `j` des nördlichen Nachbarprozessors zugreifen. `xnet` Konstrukte können sowohl auf der linken als auch auf der rechten Seite einer Zuweisung auftreten. Mit `xnetc` und `xnetp` stehen weiterhin Spezialfunktionen bereit, mit denen man Zeit sparen kann, wenn Teile des Prozessor Arrays nicht aktiv sind. Für nähere Erläuterungen verweise ich auf [8].

Mit dem `router` Konstrukt kann man über den Globalen Router einen Ausdruck auf einem entfernten PE auswerten lassen. Zum Beispiel holt `router[x][y].j` für jeden PE den Inhalt der (pluralen) Variable `j` des PEs, dessen Koordinaten in den (pluralen) Variablen `x` und `y` stehen.

### 3.3 IRIS Explorer Module

Mit dem Visualisierungspaket IRIS Explorer von Silicon Graphics können nahezu beliebige Daten visualisiert werden, indem man die Datenströme mehrerer Eingabe-, Berechnungs- und Ausgabemodule miteinander verbindet. Dies geschieht im sogenannten *Map Editor*, der in Abbildung 9 gezeigt wird; Module können hier der Map hinzugefügt, Eingabe- und Ausgabeströme miteinander verbunden und die Modulparameter verändert werden. Hierzu werden alle benutzten Module auf der Oberfläche des Map Editors als kleine Fenster dargestellt, zwischen denen die Datenverbindungen aufgebaut werden können. Bei der Programmierung der Module können einige variable Parameter spezifiziert werden. Diese können dann in den Modulfenstern durch Standardwidgets verändert werden. Leider ist gerade dieser Teil im Explorer noch mit einigen Fehlern behaftet; zum Beispiel war es unmöglich, ScrollLists einzusetzen. Die Fenster können auch auf ihre volle Größe gebracht werden, um Parameter leichter einzustellen und Ausgaben in höherer Auflösung zu erhalten.

Sowohl das Funktionstemplate als auch der Aufbau der Modulschnittstelle und die Position und Größe der Widgets im Modulfenster werden im *Module Builder* festgelegt. Von diesem werden dann eine Reihe von C-Modulen mit Makefile sowie andere Datenfiles erzeugt. Das Hauptmodul enthält nur das Funktionstemplate und muß für die gewünschte Funktion erweitert werden. Zur Kommunikation mit dem Explorer-Hauptprogramm stehen etliche Bibliotheksfunktionen zur Verfügung.

Die Hauptfunktion des selbstgeschriebenen Moduls wird immer dann aufgerufen, wenn sich irgendein Eingabeparameter verändert hat. Dies ist sowohl dann der Fall, wenn der Benutzer ein Widget bedient hat, als auch bei Veränderungen der Eingabeströme durch vorgeschaltete Module. Module können nur dann zu einem Ring geschlossen werden, wenn

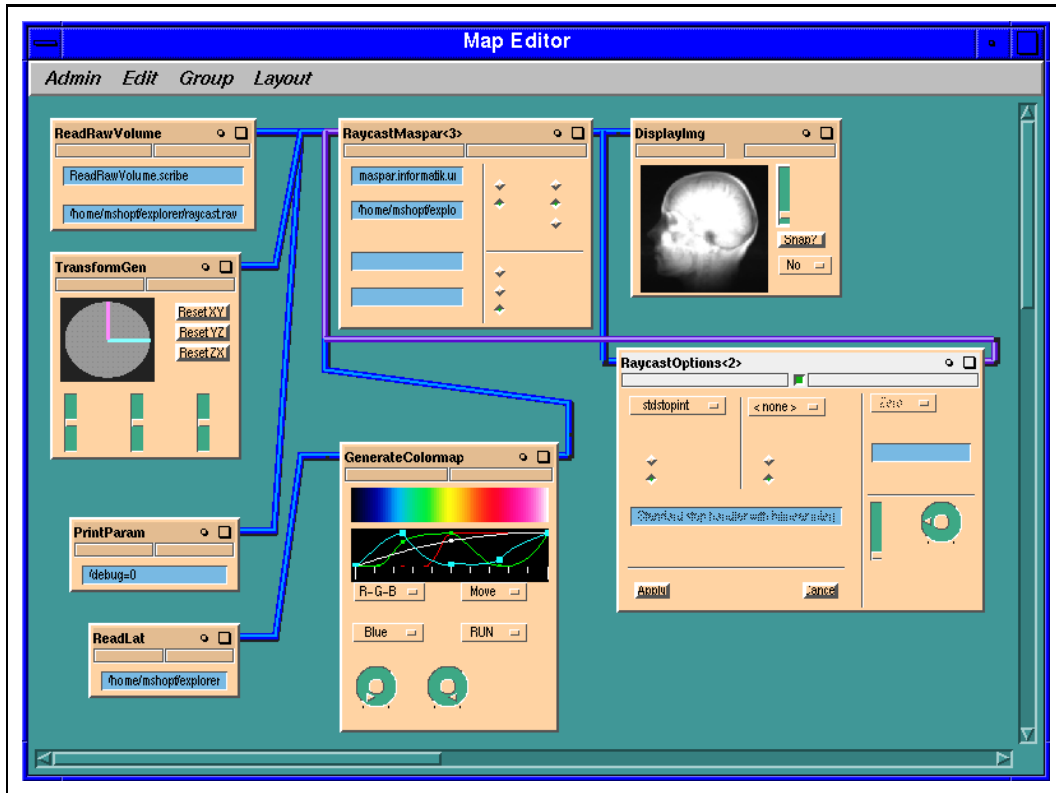


Abbildung 9: Der *Map Editor* des Explorers

eines der beteiligten Module ein *Loop Controller* ist. Auf eine Änderung der Eingabegrößen darf dieses Modul nicht immer mit einer Änderung seiner Ausgabegrößen reagieren, da sonst eine Endlosschleife provoziert wird.

Die Kommunikation der Module untereinander durch das Explorer-Hauptprogramm erfolgt über bestimmte vordefinierte Datentypen. In den in 4.3 und in 4.4 beschriebenen Modulen werden neben Standard-Parameter wie *int* und *char \** die Explorer-Typen *cxLattice* und *cxParameter* verwendet. Der *cxLattice* Datentyp beschreibt ein multidimensionales Datenfeld mit Koordinaten und mehreren Datenwerten beliebigen Formats. Über den *cxParameter* Datentyp erhält das Modul Informationen über Widget-Inhalte und sonstige Eingaben. Näheres über diese Typen findet man unter anderem in [11].

## 4 Konzepte und Programmdesign

Ein vorrangiges Ziel dieser Arbeit ist es, eine effiziente Implementierung des Shear-Warp-Algorithmuses zu erhalten. Da die verwendete Maspar MP-I zudem nur mit 16 MByte PE-Speicher ausgerüstet ist, war es auch notwendig, mit Speicher sparsam umzugehen. Gleichzeitig soll in der Arbeit gezeigt werden, daß gute Strukturierung der Aufgabe bei geschickter Implementierung ohne große Performance-Verluste erreicht werden kann. MPL besitzt zwar keinerlei objektorientierte Ansätze, allerdings hätte man aufgrund des hohen Speicherbedarfs virtueller Klassen das Programm auch nicht durchgehend objektorientiert gestalten können.

Um das Programm in logische Abschnitte unterteilen zu können, habe ich ein Handler-Konzept entwickelt. Ein einzelner logischer Schritt wird jeweils in einem gekapselten Modul durchgeführt. Da die Handler-Module jeweils mit den gleichen Parametern aufgerufen werden müssen, können allerdings die aktuellen Zustandsgrößen nicht von Modul zu Modul weitergereicht werden. Sie müssen als globale Variablen allgemein bekannt sein. Diesem konzeptionellen Nachteil steht der große Vorteil gegenüber, einzelne Handler aktivieren und deaktivieren sowie zur Laufzeit austauschen zu können. Des weiteren können neue Handlermodule einfach in das Programm integriert werden, ohne die bisherigen Handler entfernen zu müssen.

Das Handler-Konzept wurde an einigen Stellen allerdings noch nicht konsequent verwirklicht. Zum Beispiel ist die Interaktion mit dem Explorer Kommunikationsmodul teilweise noch im Hauptprogramm zu finden, da das Ziel einer im Explorer integrierten Visualisierungsoberfläche erst spät in die Zielsetzung mit einging.

### 4.1 Hauptprogramm raycast

Das MPL-Programm `raycast` kann auf der Maspar manuell oder automatisch durch das Explorer Kommunikationsmodul gestartet werden. Es existiert neben der Möglichkeit, mit dem Explorer-Modul zu kommunizieren, auch noch die einer Kommandozeilen-orientierten Interaktion, auf die ich aber hier nicht näher eingehen möchte. Die Parameter können hier über `stdin` in der gleichen Syntax eingetippt werden, wie das Explorer-Modul sie sendet. Die Parameter-Syntax ist in Abschnitt 4.1.3 beschrieben.

#### 4.1.1 Unterteilung des Programms in Handlermodule

Für die Unterteilung des Problems in mehrere unabhängige Module werden diese Module — im folgenden Handler genannt — nach ihrem Aufgabenbereich klassifiziert. Die einzelnen Aufgabenbereiche und dazugehörige Handlertypen werde ich zusammen mit den jeweils implementierten Handlern in 4.2 beschreiben. Es können von einem Typ durchaus mehrere Handler existieren; bei einigen Typen können mehrere Handler gleichzeitig aktiv sein, bei anderen nicht.

Ein Handler besteht aus einer Status-Struktur, einer Informations-Struktur und einer Reihe von Methoden. Die Methoden sind als Funktionspointer in der Status-Struktur realisiert:

```

162  /* Type of handlerinfo_t; this one is defined inside a handler by itself          File raycast.h
163  * and provides additional data */
164  typedef struct
165  {
166  const char *Name;           /* Name of the handler (the handlertype is already known) */
167  const char *Description;   /* Informative Description */
168  const char *Version;       /* (RCS) Version String */
169  parameter_t *Parameter;    /* Pointer to parameter description array (may be NULL) */
170 } handlerinfo_t ;
171
172  /* Type of handler */
173  typedef struct handler
174  {
175  struct handler_list *List;           /* Has to be 1. entry! */
176  bool Active;
177  bool Initialized;
178  long Time;           /* in microseconds; used for time measuring */
179  int (*Init) (void);
180  int (*Start) (void); /* only needed for handlers of dir H_COMP */
181  int (*Do) (void);
182  int (*Exit) (void);
183  handlerinfo_t *Info;
184 } handler_t;

```

Jeder Handler wird durch seinen eindeutigen Namen in der Informationsstruktur identifiziert. Die Beschreibung sowie der Versionsstring werden vom Explorer-Modul nur für den Benutzer ausgegeben und haben keine weitere Bedeutung.

Die einzelnen Handler sind jeweils mit NULL abgeschlossen in Arrays zusammengefaßt. Die Arrays werden über je eine `handler_list`-Struktur je Handlertyp verwaltet, die wiederum in einem Array zusammengefaßt sind. Der Handlertyp ist aus Sicherheitsgründen sowohl durch die Position der Struktur im Array festgelegt als auch in der Struktur selber aufgeführt. Er wird in der Initialisierungsfunktion auf Übereinstimmung getestet:

```

189  typedef struct handler_list          File raycast.h
190  {
191  handler_t *Handler;           /* Has to be 1. entry! */
192  func_t FType;
193  handler_t *MutualActive;      /* points to active handler when MutualExclude==1 and */
194  /* the handler has a Do method */
195  bool MutualExclude;         /* Only one handler is allowed to be active at a time */
196  bool NeedsActive;           /* There has always to be an active handler */
197 } handler_list_t;

```

Jeder Handler wird von der Hauptfunktion *ray\_cast* () über seine Methoden angesprochen. Derzeit existieren vier verschiedene Methoden:

- Init** Diese Methode wird garantiert aufgerufen, bevor irgendeine andere Methode benutzt wird. In dieser Methode sollte jeder Handler seinen benötigten Speicher allozieren und Variablen initialisieren.
- Exit** Diese Methode wird aufgerufen, wenn der Handler deaktiviert wurde. Der gesamte selbst allozierte Speicher muß hier zurückgegeben werden, da er gegebenenfalls von anderen Handlern benötigt wird.
- Do** In der Hauptfunktion werden der Reihe nach die *Do*-Methoden aller aktiver Handler aufgerufen. Die Handlertypen RAY\_F\_POS, RAY\_F\_INTEGRAL, RAY\_F\_TRANSFER und RAY\_F\_STEP liegen innerhalb der Schleife durch alle Volumenebenen und werden entsprechend für jede Ebene einmal aufgerufen.
- Start** Vor der Schleife durch alle Volumenebenen müssen diverse Handler ihre Variablen und Datenarrays initialisieren. Im *stdstart*-Handler werden deshalb alle *Start*-Methoden aufgerufen, wenn sie vorhanden sind. Es wäre auch denkbar, den Handlertyp RAY\_F\_START zu entfernen und den Aufruf der *Start*-Methoden in das Hauptprogramm zu verlegen.

Das Handler-Konzept wird von einer Reihe von Support-Funktionen unterstützt:

- ray\_cast** Dies ist die eigentliche Hauptfunktion. Sie initialisiert alle Handler und berechnet ein Bild mit den aktuellen Daten, indem sie die einzelnen Handlertypen der Reihe nach aufruft. Die Schleife durch alle Volumenebenen ist ebenfalls in dieser Funktion enthalten.
- ray\_init** Diese Funktion ruft die *Init*-Methoden aller noch nicht initialisierten aktiven Handlermodule auf.
- ray\_exit** Diese Funktion ruft die *Exit*-Methoden aller initialisierten Handler auf.
- ray\_call** Mit dieser Funktion werden die *Do*-Methoden aller aktiven Handler eines bestimmten Handlertyps aufgerufen.
- ray\_gethandler** Die einzelnen Handler sind eindeutig durch ihre Namen bestimmt. Mit dieser Funktion kann die Status-Struktur eines Handlers gefunden werden, dessen Name bekannt ist.
- ray\_activate** Einzelne Handler können mit dieser Funktion aktiviert und deaktiviert werden. Die Funktion kümmert sich darum, daß der Handler initialisiert wird. Außerdem werden alle anderen Handler des übergebenen Typs deaktiviert, wenn der Typ nur einen aktiven Handler erlaubt.

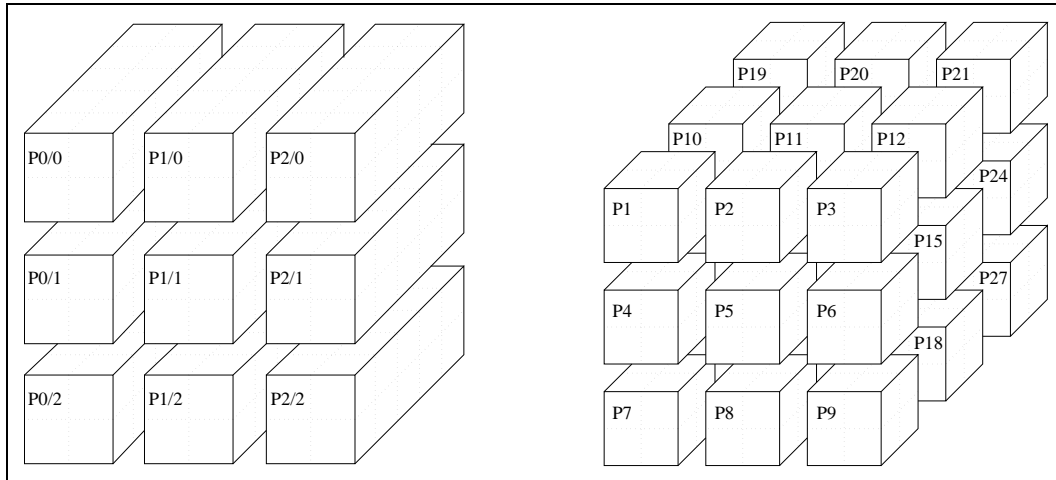


Abbildung 10: Repräsentation des Volumens durch Säulen oder Teilvolumina

#### 4.1.2 Verteilte Datenstrukturen

Auf SIMD-Maschinen ist die Bereitstellung der Daten ein wichtiger Parallelisierungsaspekt. Alle Datenstrukturen müssen so ausgelegt sein, daß jederzeit alle Prozessoren ausgelastet sind, um so einen optimalen Datendurchsatz zu gewährleisten. Aus diesem Grund werde ich erst die verwendeten Datenstrukturen erläutern, bevor ich auf die Struktur des Programms zu sprechen komme.

##### **Volumendaten: vol\_t \*V**

Die Volumendaten können auf die einzelnen PEs auf zwei grundverschiedene Arten verteilt werden. Abbildung 10 zeigt, daß entweder das Volumen in allen drei Dimensionen gleichmäßig unterteilt wird oder jedes PE eine Säule durch das ganze Volumen erhält. Auf Parallelrechnern ohne XNet-Kommunikation ist wahrscheinlich die erste Möglichkeit die bessere Wahl, da die sonst nötige Transformation bei Betrachtungswinkeln  $> \frac{\pi}{4}$  wegfällt. Andererseits kann auf der Maspar die Torus-Eigenschaft des XNet nicht ausgenutzt werden, da die für eine Ebene zuständigen Prozessoren nicht das gesamte Prozessor-Array ausfüllen. Des weiteren wird die Koordinierung in diesem Fall schwierig, da in einem Berechnungsschritt mehrere Ebenen betrachtet werden müssen. Außerdem ist für die Unterteilung der Integration notwendig, daß das Integral über den Strahl in mehrere Teilintegrale unterteilt werden kann. Mir ist zwar kein Beleuchtungsmodell bekannt, das diese Möglichkeit nicht bietet, aber die Komposition der Teilergebnisse muß extra implementiert werden. Aus diesen Gründen habe ich mich für die Volumendarstellung in Säulenform entschieden.

Da die Größe des Volumen-Arrays und damit die Größe der einzelnen Säulen zur Compilezeit nicht feststeht, kann das Volumen-Array nur über einen Pointer auf neu allozierten

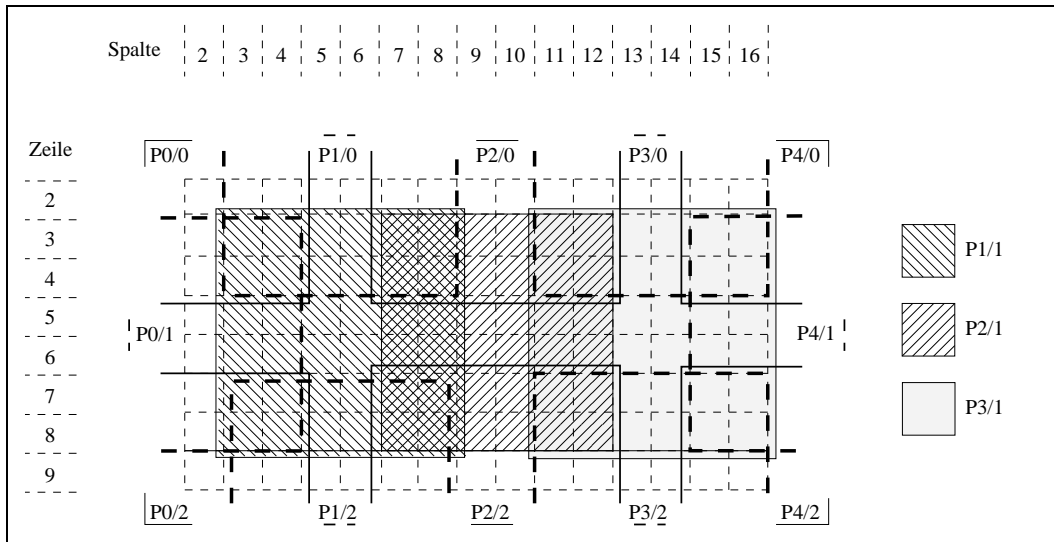


Abbildung 11: Überlappende Speicherung der Säulen auf den einzelnen Prozessoren

Speicher repräsentiert werden. Die Indizierung muß von Hand durchgeführt werden. Um Speicher zu sparen, werden die Daten nur in 8 Bit Genauigkeit gespeichert:

```

72 typedef plural uchar vol_t; File rayvolume.h
105 /* The volume is a 3D-array. The x-index runs fastest,
106  * followed by the y-index. The z-index (plane number)
107  * has got the biggest step size. */
108 extern vol_t *V;
109 /* Every processor has an array of this size,
110  * including all duplicated edges! */
111 extern ushort Size_X, Size_Y, Size_Z;
112 /* The total volume size */
113 extern ushort Total_X, Total_Y, Total_Z;

```

Da sowohl bei der Bilinearen Interpolation als auch bei einer gegebenenfalls nötigen Gradientenbildung noch die Werte angrenzender Prozessoren in zumindest einer Richtung benötigt werden, werden die Teilvolumen überlappend und ihre Randwerte damit doppelt gespeichert, um den Kommunikationsaufwand zu erniedrigen. Als Beispiel, wie eine Ebene des Volumens letztendlich auf den Prozessoren verteilt ist, habe ich in Abbildung 11 den Fall  $\text{Size}_X = \text{Size}_Y = 6$ ;  $\text{OVERLAP\_LEFT} = \text{OVERLAP\_RIGHT} = 1$  dargestellt. Diese beiden Defines definieren die Größe des überlappenden Bereichs.

In der Variablen VolumeState wird festgehalten, ob das Volumen von einem Transformations-Handler irreversibel verändert wurde. In diesem Fall muß der gerade aktive Input-Handler das Volumen neu einlesen beziehungsweise anfordern.

### Berechnungsdaten: comp\_t \*C

Um die Unterteilung des Algorithmus in kleinere, leichter überschaubare Abschnitte zu ermöglichen, müssen die Berechnung des RGBA-Wertes eines Voxels und die Integration voneinander getrennt werden. Da der Overhead deutlich zu groß wäre, für jeden einem Prozessor zugehörigen Voxel der Reihe nach mehrere Handler aufzurufen, habe ich mich dazu entschlossen, daß jeder Handler alle Strahlen behandeln muß, die einem Prozessor zugeteilt sind. Die berechneten Werte werden in einem Zwischenspeicher geschrieben, der sogenannten *Computation Area*. Dieser Zwischenspeicher wird von den Lookup-, den Transfer- und dem Integrations-Handlern benötigt.

```
131  /* Explanation of all values:                                     File raypixel.h
132  * l: lookup/interpolation value (pos handlers)
133  * r: red component of computation area
134  * g: dto. green component
135  * b: dto. blue component
136  * a: dto. alpha value */
137 typedef plural struct { float l, r, g, b, a; } comp_t ;
145  /* The Computation Area is a 2D-array. The x-index runs fastest,
146  * followed by the y-index. Size is PSize_X * PSize_Y */
147 extern comp_t *C;
```

Derzeit wird in den Lookup-Handlern nur ein Wert im Volumen ermittelt. Für bessere Beleuchtungsmodelle ist es hier sinnvoll, zusätzlich noch den Gradienten zu ermitteln. Hierfür muß die Struktur noch erweitert werden.

### Zwischenbild: pix\_t \*P

Nach (36) in 2.5 benötigt man für das Zwischenbild die doppelte Größe des Volumens sowohl in  $x$ - als auch in  $y$ -Richtung. Dabei wird für jede Volumen-Ebene immer nur auf einem Gebiet in Volumengröße gearbeitet, da die Anzahl gerade betrachteter Strahlen immer der Anzahl an Volumenelementen in einer Ebene entspricht. Werden zur Auflösungserhöhung mehrere Strahlen je Volumenelement benutzt, wie in 2.3.1 angedeutet, erhöht sich die benötigte Größe entsprechend.

Um das aktuelle Gebiet des Zwischenbildes immer mit den Volumendaten korreliert auf allen PEs verteilt verfügbar zu haben, muß das Zwischenbild hierarchisch abgespeichert werden. Das Bild wird gemäß Abbildung 12 in vier Teilbilder zerlegt, die wiederum auf alle Prozessoren verteilt werden. Auf den einzelnen PEs sind die für den jeweiligen Prozessor vorgesehenen partiellen Teilbilder in einem zweidimensionalen Array gespeichert:

```
95  /* t: transparency of pixel array (1 - alpha) */                 File raypixel.h
96  typedef plural struct { float r, g, b, t; } pix_t ;
148  /* Another 2D-array. x-index runs fastest. */
149 extern pix_t *P;
```

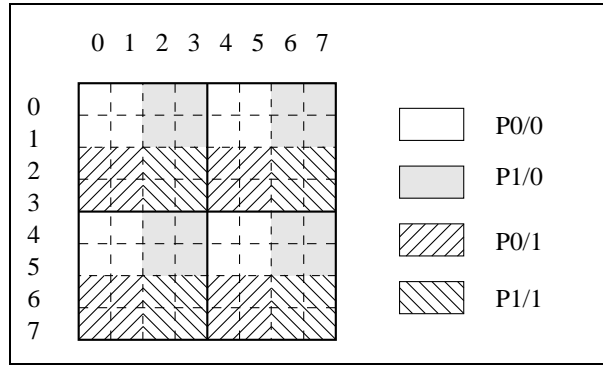


Abbildung 12: Aufteilung des *Pixel Arrays* auf die einzelnen Prozessoren

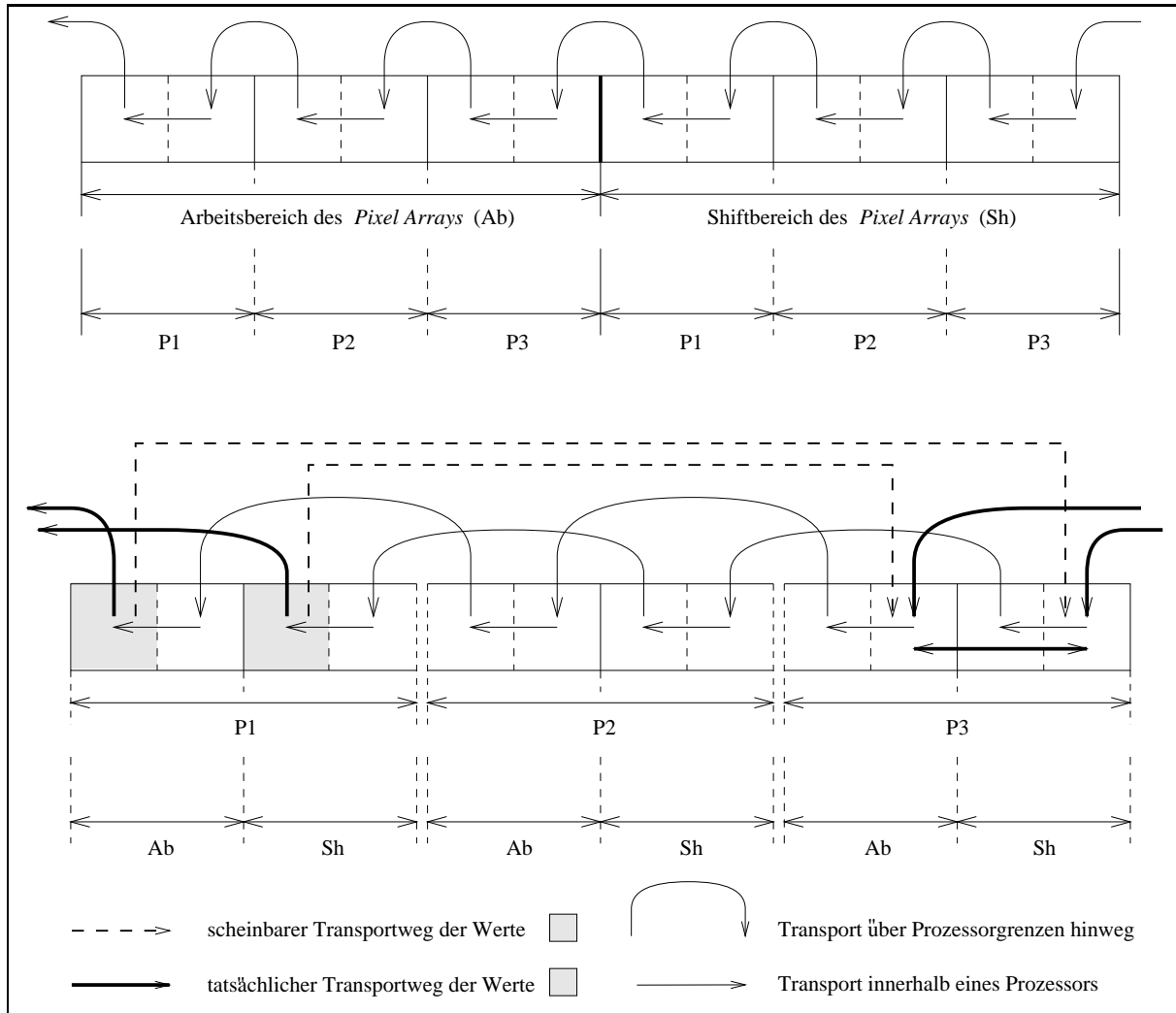
```

150 /* Every processor has an (working) pixel array of this size
151 * (well, it has actually four times this value and a bit more..) */
152 extern ushort PSize_X, PSize_Y;

```

Aufgrund des Shear-Warp-Ansatzes (14) müßte das Volumen bei jedem Schritt von Ebene zu Ebene entsprechend der Scherungsmatrix  $\mathcal{S}$  auf dem Prozessor Array verschoben werden, damit das Zwischenbild und die benötigten Volumendaten auf das jeweils richtige PE zu liegen kommen. Der Gesamtaufwand ist hier bei einem Winkel von  $\frac{\pi}{4}$  und der Kantenlänge  $n$  gleich  $O(n^4)$ , da insgesamt  $n^3$  Volumenelemente bei geschickter Implementierung im Durchschnitt um  $\frac{n}{4}$  Elemente verschoben werden müssen. Verschiebt man in jedem Schritt hingegen nicht das Volumen, sondern das Koordinatensystem, in dem die Augenstrahlen aufintegriert werden, muß nur das Zwischenbild entsprechend verschoben werden. Der Gesamtaufwand reduziert sich so auf  $O(n^3)$ ; allerdings muß das Zwischenbild nach der Integration wieder zu seinem Ursprungsort geschoben werden. Aber auch dieser letzte Schritt hat nur einen Aufwand von  $O(n^3)$ . Derzeit wird für das Bild nur der Offset des richtigen Prozessors ermittelt, die Verschiebung um einzelne Pixel ist noch nicht implementiert. Das Bild wird nicht wirklich auf den Prozessoren umhergeschoben, wodurch sich der Gesamtaufwand nochmal erniedrigt. Da die Ausgabe-Handler in der Regel sowieso die Daten sequentiell ausgeben müssen, wird ihnen auch diese Aufgabe übergeben.

Nutzt man beim Verschieben des Zwischenbildes die Torus-Eigenschaft des XNet-Netzwerks aus und wählt eine geschickte Indizierung der Daten im Zwischenbild, können mit minimalem Aufwand die Strahlen aussortiert, gespeichert und durch neue ersetzt werden, die das Volumen verlassen. Abbildung 13 zeigt das Vorgehen bei einem West-Shift der Daten innerhalb einer Bildzeile. Dabei ist im *Pixel Array* etwas zusätzlicher Platz nötig, um die Daten erst nur innerhalb jedes einzelnen Prozessors verschieben und die Kommunikation erst im Anschluß durchführen zu können.



Die Abbildung zeigt den gewünschten Datentransportweg mit getrenntem *Pixel Array* und Shiftbereich sowie den Transportweg der tatsächlichen Implementierung mit einem kombinierten *Pixel Array*. Um den Datentransport vollständig regelmäßig gestalten zu können, werden die gekennzeichneten Werte nach dem Transport ausgetauscht und nicht schon im Vorfeld gesondert behandelt.

Abbildung 13: Kommunikationsstruktur für eine Bildzeile bei einem West-Shift

## Ausgabedaten: out\_t \*0

Der abschließende Skalierungsschritt vergrößert die Bilddatenmenge und bewirkt durch nichtganzzahlige Vergrößerungsraten, daß das Ergebnisbild nicht mehr uniform auf allen Prozessoren verteilt ist. Die Größe des Teilbildes, das jedes PE berechnet, kann um je ein Pixel in  $x$ - und in  $y$ -Richtung schwanken. Dies wirkt sich auch für die Ausgabe an das Explorer-Modul negativ aus, wie ich noch erläutern werde.

Das sogenannte *Output Array* enthält dieses Ergebnisbild; allerdings sind bereits alle Daten auf einen anderen Wertebereich skaliert, um Speicher zu sparen:

```
118 typedef plural struct { unsigned char r, g, b, t; } out_t ;           File raypixel.h
142     /* Another 2D-array. x-index runs fastest. */
143 extern out_t *O;
154     /* For every processor: valid pic area */
155 extern plural ushort PixSize_X, PixSize_Y;
```

Der Transparenzwert hat in der Struktur keinerlei Bedeutung, er wird zum Beispiel nicht zum Explorer geschickt. Durch seine Anwesenheit wird die Struktur allerdings auf eine Größe von vier Bytes gebracht und damit gegebenenfalls der Zugriff auf die Struktur durch Verhinderung von Missalign-Effekten beschleunigt. Zudem kann der Transparenz-Wert im Recolor-Handler verwendet werden, um zusätzliche Informationen in das Bild einzuarbeiten.

### 4.1.3 Parameterhandling

Um die einzelnen Handlermodule zu beeinflussen, müssen in der Regel mehrere Parameter verändert werden. Da sowohl die Art und der Wertebereich als auch die Anzahl an Parametern sich selbst für Handler des gleichen Typs unterscheiden können, ist ein flexibles Konzept sowohl zur Parameterverwaltung als auch zur Einstellung nötig.

Im Hauptprogramm werden die zu einem Handler gehörenden Parameter in einem Strukturarray beschrieben. Jeder Parameter enthält dabei einen eigenen Eintrag; das Ende des Arrays wird mit NULL markiert.

```
143     /* Type of parameter_t; an array of this type defines the parameters for all       File raycast.h
144     * handlers as well as the type of the parameter. */
145 typedef struct
146 {
147     const char *Name;           /* Name of the parameter (namespace local for */
148                               /* each handler) (NULL after last parameter) */
149     const char *Description;    /* Description */
150     const char *Type;          /* Type of the parameter: 'a' (toggle active, only first character), */
151                               /* one of 's' (select), 't' (text), 'i' (int=slider), 'd' (dial=double) */
152     bool      Active;          /* Parameter is active/inactive */
153     char      Text [MAXLEN_TEXT_PARAMETER]; /* Text parameter and Select field */
```

```

154  int      Slider;                                     /* 'i' parameter */
155  double   Dial;                                       /* 'd' parameter */
156  double   MinVal, MaxVal; /* Minimum / maximum value for 'i' and 'd' parameters */
157 } parameter_t;

```

Der Parametertyp besteht aus einem oder zwei Buchstaben, die die Charaktereigenschaften des Parameters spezifizieren:

- 'a'        *active*: Der Parameter kann aktiviert und deaktiviert werden.
- 's'        *select*: Der Benutzer kann aus mehreren Möglichkeiten auswählen.
- 't'        *text*: Der Parameter besteht aus einem String, zum Beispiel einem Filenamem.
- 'i'        *integer*: Ein ganzzahliger Parameter wird benötigt.
- 'd'        *double*: Ein reeler Parameter wird benötigt.
- 'ax' mit  $x \in \{s, t, i, d\}$  Der entsprechende Parameter kann zusätzlich aktiviert und deaktiviert werden.

Für das Parameter-Handling existieren eine Reihe von Support-Funktionen:

**ray\_getparameter** Jeder Parameter hat im lokalen Namensraum eines Handlers einen eindeutig bestimmten Namen. Die Struktur eines Parameters, dessen Name bekannt ist, kann mit dieser Funktion gefunden werden.

**ray\_template** Mit dieser Funktion wird das unten beschriebene Handlerstatus-Template generiert.

**ray\_parseopt** Mit dieser Funktion kann eine vom Benutzer eingegebene oder vom Explorer gesendete Option analysiert und der entsprechende Parameter angepaßt werden.

Um die Parameter und ihren Status sowie den Handlerstatus zum Explorer Optionsmodul senden zu können, wird ein Template mit der folgenden EBNF generiert:

```

Template ::= { Handler }* 'done\n'
Handler  ::= { '+' | '-' } Name '#' Description '@' { Parameter }* '\n'
Parameter ::= '␣' { '+' | '-' } Name '#' Description '@' { Type }*
Name     ::= { alphanumeric }+
Description ::= { alphanumeric | '␣' | punctuation }*
Type     ::= { 'a' } | { 's=' (nicht spezifiziert) } | { 't=' TextValue } |
             { 'i=' NumValue } | { 'd=' NumValue }
TextValue ::= ''' { alphanumeric | '␣' | punctuation }* '''
NumValue  ::= Number '/' Number '/' Number '/'
Number   ::= { numeric | '.' }+

```

Die numerischen Werte bestehen jeweils aus dem aktuellen Wert sowie ihrem Minimum und Maximum. Keine Beschreibung darf die Zeichen '#' oder '@', den Zeilenvorschub '\n' oder das Textendezeichen '\0' enthalten. Mit '+' und '-' wird spezifiziert, ob der entsprechende Handler oder Parameter aktiv oder inaktiv ist.

Zum Verändern der Parameter habe ich das in 4.4 beschriebene Explorer-Modul entwickelt, mit dem für jeden Handler die hier beschriebenen Parametertypen eingestellt werden können. Das Template für den Datentyp 's' ist noch nicht spezifiziert, da dieser Typ im Moment noch nicht unterstützt wird.

Das Explorer Optionsmodul schickt alle Parameter, die geändert wurden, über die Standardeingabe an das Hauptprogramm. Das Format für die Parameteroptionen kann auch ohne die Explorer-Module verwendet werden, wenn das Hauptprogramm von Hand gestartet wird. Die Parameter können sowohl beim Start als Argumente übergeben werden als auch nach dem Start über *stdin*. Die EBNF für die Parameterübergabe lautet wie folgt:

```
Options ::= { [ Option ] { '\_ ' | '\n' } }*
Option  ::= Special | Handler | Parameter
Special ::= '/' { 'go' | 'forceinput' | { 'debug=' Number } |
              { 'x=' Number } | { 'y=' Number } |
              { 'matrix=' MatValue } | { 'cmap=' CMapValue } }
Handler ::= { '+' | '-' | '@' } Name
Parameter ::= '! { '+' | '-' | '@' } Name [ '=' { NumValue | TextValue } ]
Name      ::= { alphanumeric }+
TextValue ::= ''' { alphanumeric | '\_ ' | punctuation }* '''
NumValue  ::= Number '/' Number '/' Number '/'
MatValue  ::= { Number ':' } (16x)
CMapValue ::= Number ':' Number '/' Number '/' Number '/' Number
Number    ::= { numeric | '.' }+
```

Die einzelnen Optionen können demnach in drei grundsätzlich unterschiedliche Typen unterteilt werden:

- **Special**

Spezielle Optionen können nicht verallgemeinert werden. Deshalb werde ich alle bis jetzt implementierten Optionen erläutern:

**/go** Der Berechnungsvorgang wird gestartet. Das Volumen wird wenn nötig geladen, ein Bild berechnet und gespeichert. Danach können wieder neue Optionen übergeben werden. **/go** muß mit '\n' terminiert werden.

**/forceinput** Das Volumen wird mit dieser Option für ungültig erklärt. Das Explorer-Modul schickt diese Option, wenn sich der Eingabedatenstrom des Moduls geändert hat.

**/debug** In der Debug-Version des Hauptprogramms kann mit dieser Option der aktuelle Debuglevel angegeben werden.

- /matrix** Die nachfolgenden 16 Werte spezifizieren die aktuelle Abbildungsmatrix.
- /cmap** Mit dieser Option wird ein Farbtabelleintrag gespeichert. Derzeit unterstützt das Hauptprogramm nur eine einzige Farbtabelle mit 256 Einträgen. Die fünf Parameter spezifizieren den Index sowie die RGBA-Werte. Alle Werte müssen im Bereich [0, 255] liegen.
- /x und /y** Wird `raycast` nicht im Explorer-Modus gestartet, ist es etwas mühsam, eine Rotationsmatrix von Hand auszurechnen und einzutippen. Deshalb kann mit diesen beiden Befehlen das Volumen um die gegebene Achse um den angegebenen Winkel gedreht werden. Die erstellte Rotationsmatrix wird mit der aktuellen Matrix multipliziert, es können also Animationen hergestellt werden, wenn man zum Beispiel mehrfach `"/x=10 /go"` eintippt.

- **Handler**

Handler können mit dieser Option aktiviert ('+'), deaktiviert ('-') oder nur als aktuell erklärt ('@') werden. Es können jeweils nur die Parameter des aktuellen Handlers geändert werden. Wird ein Handler aktiviert oder deaktiviert, wird er ebenfalls zum aktuellen Handler.

- **Parameter**

Parameter können aktiviert ('+') und deaktiviert ('-') werden, wenn das dazugehörige Template dies erlaubt. Mit '@' wird der Status des Parameters nicht geändert, sondern nur sein Wert. Soll der Wert des Parameters geändert werden, muß auf seinen Namen der gewünschte Wert in dem Format folgen, den der Parameter erwartet.

#### 4.1.4 Kommunikation mit dem Explorer-Modul

Wird beim Start des Hauptprogramms als Argument `'explorer a.b.c.d x'` mit einer numerischen Internetadresse a.b.c.d und einer Portnummer x übergeben, startet das Programm im Explorer-Modus und versucht, eine Verbindung mit dem Explorer auf dieser Adresse aufzubauen. Über diese Datenverbindung bekommt das Hauptprogramm sowohl die Parameteroptionen als auch die Volumendaten. Das Optionstemplate und die Bilddaten werden ebenfalls über die Verbindung in Richtung Explorer-Modul übermittelt. Auf `stdout` gibt das Hauptprogramm nur informative Nachrichten sowie Debugging-Meldungen aus. Das Explorer-Modul empfängt diese Nachrichten und schreibt sie in das Log-Fenster des Explorers.

Nachdem die Datenverbindung aufgebaut wurde, schickt `raycast` seine Copyright-Meldung. Es benutzt die Datenverbindung, um dem Explorer Kommunikationsmodul sowohl Informationen zu schicken, als auch um Anforderung zu senden. Zum Einleiten eines Informationsaustausches wird jeweils ein Schlüsselwort terminiert mit '\n' gesendet. Die folgenden Schlüsselworte sind bekannt:

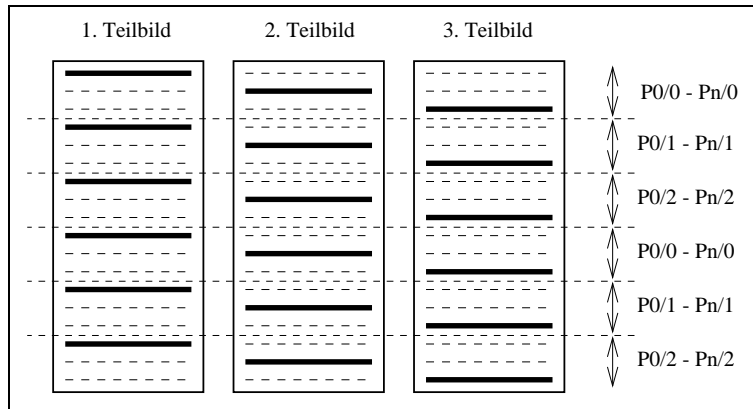


Abbildung 14: Teilbilder bei der Übertragung zum Explorer

- ok** raycast ist bereit, Parameteroptionen zu erhalten.
- error** Ein Fehler ist aufgetreten. Der Fehlergrund wird in der nächsten Zeile beschrieben. In aller Regel kann nach einem Fehler mit der Arbeit fortgefahren werden.
- options** Es folgt das Parameter-Template mit dem in 4.1.3 beschriebenen Aufbau.
- in** Hiermit zeigt raycast, daß es die Volumendaten benötigt. Das Explorer Kommunikationsmodul RaycastMaspar schickt daraufhin die Größe des Volumens mit drei ASCII-Zahlen, terminiert mit '\n', und im Anschluß die Volumendaten. Abgeschlossen wird die Übertragung mit 'done\n'.
- out** Die Übertragung der Bilddaten wird angekündigt. Es folgen die absolute Größe des Bildes in x- und in y-Richtung, die maximale Anzahl Zeilen je Teilbild und die Anzahl Iterationen. Das Bild wird gemäß Abbildung 14 in Teilbildern übertragen, um der Repräsentation der Daten auf den PEs möglichst gut zu entsprechen. Für eine komplette Umwandlung des Bildes im Datenspeicher der ACU ist im allgemeinen nichtgenügend Speicher vorhanden, deshalb mußte diese Zwischenlösung benutzt werden.
- Etliche Teilschritte der Bildübertragung können noch deutlich beschleunigt werden; allerdings ist es auch nicht die Hauptaufgabe dieser Arbeit, die Übertragungsperformance zum Explorer-Modul zu optimieren.
- done** Ende der Übertragungsdaten, sowohl des Parameter-Template als auch der Bild- und Volumendaten. Bei den Bild- und Volumendaten wird dieses Schlüsselwort nur zur Konsistenzprüfung benutzt.

Da in einem MPL-Programm die Socket-Verbindung nicht aufgebaut werden kann, müssen die Daten an ein Unterprogramm geschickt werden, das auf dem Unix-Frontend der Maspar läuft. Die gesamte Kommunikation mit der Socket-Verbindung läuft auf diese Weise ab. Der Aufruf der Funktionen auf dem Frontend erfolgt transparent über in `explorer.h` definierte Makros, aber das Kopieren der Daten muß von Hand erledigt werden.

## 4.2 Beschreibung der einzelnen Handler

Im Folgenden werden sowohl die einzelnen Handlertypen als auch die jeweils implementierten Handler beschrieben. Die Sourcen sind in mehrere Unterverzeichnisse aufgeteilt, die auch README-Dateien mit zusätzlichen Informationen für die praktische Implementierung neuer Handler der jeweiligen Typen enthalten.

Alle Handler müssen in dem MPL-File `handlers.m` aufgelistet sein. Um die Auflistung möglichst einfach zu halten, habe ich dort einige Makros definiert und ihre Verwendung beschrieben. In dem File werden sowohl die Typen der einzelnen Handler festgelegt, als auch ihr Status bei Programmbeginn (aktiv/inaktiv) und die Information, ob der Handlertyp jeweils nur einen aktiven Handler erlaubt (Mutual Exclude).

### 4.2.1 Typ `RAY_H_INIT`: Initialisierung

Dieser Handlertyp war ursprünglich geplant, Initialisierungen gleichwelcher Art vor jeder Bildberechnung durchzuführen. Da jeder Handler zusätzlich zu seiner *Do*-Methode noch eine *Init*-Methode besitzt und die Größe des selbst allozierten Speichers sowieso bei jedem Aufruf getestet werden muß, habe ich bis jetzt noch keine Verwendung für diesen Handler gefunden.

Für die Zukunft wäre es denkbar, das komplette Option-Parsing einem Handler dieses Typs zu überantworten. Das hätte dann den Vorteil, daß es leicht möglich wäre, ein GUI für die Bedienung des Programms direkt an der Maspar-Konsole einzubinden, ohne den Explorer-Support zu verlieren.

### 4.2.2 Typ `RAY_H_EXIT`: Aufräumen

Dieser Handlertyp ist das Gegenstück zu `RAY_H_INIT`. Über hier eingebundene Handler können Aufräumarbeiten erledigt werden, die nach jeder Bildberechnung nötig sind.

Derzeit existiert genau ein Handler dieses Typs — *timing*. Dieser Handler gibt die in den User-Funktionen gemessene benötigte Zeit aller aktiven Handler auf *stdout* aus. Ist der Handler nicht aktiv, werden aus Effizienzgründen auch keine Zeiten gemessen.

### 4.2.3 Typ `RAY_H_INPUT`: Volumen einlesen

Die Input-Handler sind für das Einlesen des Volumens zuständig. Wurde das Hauptprogramm im Explorer-Modus gestartet, und es existiert eine Verbindung an den Input-Port des Kommunikations-Moduls, wird von diesem Modul automatisch der *explorerin*-Handler aktiviert, der sich das Volumen über den Datenport holt.

Die Input-Handler müssen bei jedem Aufruf ihrer *Do*-Methode den Status des Volumens in der Variablen `VolumeState` überprüfen und das Volumen gegebenenfalls neu laden, falls dieses irreversibel verändert wurde. Es kann jeweils nur ein Input-Handler aktiv sein.

Derzeit sind die folgenden Input-Handler implementiert:

**explorerin** Das Volumen wird über den Daten-Port vom Explorer-Kommunikationsmodul geholt.

**pseudopic** Es wird ein Pseudo-Volumen der Größe  $128^3$  generiert, das ein Kreuz sowie zwei zusätzliche Balken enthält. Dieses Volumen dient vor allen zu Testzwecken.

**rawin** Das Volumen liegt auf der lokalen Harddisk des Feldrechners und wird direkt von ihm geladen. Als Parameter können der Filename und die Ausdehnung in den drei Raumrichtungen eingegeben werden. Dieser Handler ist bisher der einzige, der Volumen mit nichtquadratischen Voxeln unterstützt. Die grundsätzlichen Strukturen und Berechnungsroutinen sind im Hauptprogramm bereits integriert, es müssen lediglich die anderen Input-Handler noch angepaßt werden. Da die für die Scherung benötigten Werte noch im Hauptprogramm und nicht in einer Handlerroutine berechnet werden, kann nach dem Einlesen eines neuen Volumens das erste Bild fehlerhaft berechnet werden. Von eins unterschiedliche Voxelgrößen in *x*- und in *y*-Richtung wurden noch nicht getestet.

**rawinscale** Da die verwendete Maspar nur wenig Speicher zur Verfügung hat, können Volumen der Größe  $256^3$  nicht mehr geladen werden. Um auch diese Volumen zumindest in geringerer Auflösung laden zu können, werden in diesem Modul die Volumenwerte in einem derzeit fest vorgegebenen Abstand von 2 neu gesampelt.

#### 4.2.4 Typ `RAY_H_OUTPUT`: Bild ausgeben

Nachdem ein Bild berechnet worden ist, möchte man das Ergebnis normalerweise anschauen oder zumindest speichern können. Mit den Output-Handlern können nun die Bilder in verschiedenen Formaten ausgegeben werden. Es ist allerdings nicht zwingend notwendig, daß ein Output-Handler die Bilddaten speichert. Genausogut kann er dazu dienen, den internen Zustand oder auch das geladene und transformierte Volumen zu speichern.

Wurde das Hauptprogramm im Explorer-Modus gestartet, und es existiert eine Verbindung an den Output-Port des Kommunikations-Moduls, wird automatisch zusätzlich der *explorerout*-Handler aktiviert. Es können zur gleichen Zeit selbstverständlich mehrere Output-Handler aktiv sein.

Die Output-Handler sind auch dafür verantwortlich, daß das Bild entsprechend dem im Stop-Handler berechneten Offset bei der Ausgabe verschoben wird. Wie ich in 4.1.2 auf

Seite 33 beschrieben habe, wird im Stop-Handler das Bild nur um einzelne Pixel verschoben, aber nicht von Prozessor zu Prozessor. Da die Output-Handler in der Regel sowieso die Daten umkopieren und in den meisten Fällen sogar sequentiallisieren müssen, ist das Verschieben der Daten über Prozessorgrenzen hinweg in den Output-Handlern nahezu kostenlos.

Derzeit sind folgende Output-Handler implementiert:

**explorerout** Das Bild wird über den Daten-Port zum Explorer-Kommunikationsmodul geschickt. In 4.1.4 ist ansatzweise das (unübliche) Format der übermittelten Daten beschrieben.

**mpddl** Dieser Handler benutzt die MPDDL-Bibliothek von Maspar für eine direkte Ausgabe auf einem X-Server. Voraussetzung hierfür ist natürlich, daß die DISPLAY-Variable gesetzt ist.

Leider können über MPDDL keine Daten mit prozessorabhängiger Größe ausgegeben werden, weshalb dieser Handler nicht ohne weiteres an die neue Datenstruktur angepaßt werden konnte. Aufgrund des flexiblen Handler-Konzepts konnte ich den Handler aber weiter im Programm lassen.

**ppm** Mit diesem Handler wird das aktuelle Bild als .ppm-File im Standard-P6-Binärformat gespeichert. Viele Programme unterstützen dieses Format, wie zum Beispiel xv.

**rawout** Dieser Handler schreibt das aktuelle Volumen in ein lokales File auf dem Feldrechner. Ich habe diese Funktion geschrieben, da der *rawinscale*-Handler sehr lange zum Einlesen eines großen Volumens braucht, obwohl nur ein Bruchteil der Information in dem Volumen benutzt wird.

#### 4.2.5 Typ RAY\_H\_TRANSFORM: Transformation des Volumens

Nach dem Einlesen des Volumens muß es gegebenenfalls noch transformiert werden. Liegt einer der Winkel gegenüber den Achsen zum Beispiel über  $\frac{\pi}{4}$ , muß das Volumen um  $\frac{\pi}{2}$  gedreht werden. Dabei ändert sich natürlich der Status des Volumens, da sich dieses nun nicht mehr im Originalzustand befindet. Die Veränderung ist in diesem Fall allerdings umkehrbar, so daß das Volumen nicht mehr neu geladen werden muß.

Ein Transformations-Handler muß beim Aufruf seiner *Do*-Methode über die Variable *VolumeState* testen, ob das Volumen in der Zwischenzeit neu geladen wurde oder ob es sich noch um das alte, von ihm gegebenenfalls bereits veränderte, Volumen handelt. Damit dieses Verfahren auch für mehrere aktive Transformations-Handler funktioniert, muß das Verfahren allerdings noch verfeinert werden.

Derzeit ist noch kein Transformations-Handler implementiert. Aus diesem Grund können in einigen der anderen Handler auch etliche *Assertions* fehlschlagen, wenn einer der Winkel gegenüber den Achsen  $\frac{\pi}{4}$  überschreitet.

#### 4.2.6 Typ RAY\_H\_START: Initialisierung vor Raycast-Schleife

Dieser Handlertyp wird unmittelbar vor dem Eintritt in die Raycast-Schleife aufgerufen. Der derzeit vorhandene *stdstart*-Handler ruft dabei die *Start*-Methoden aller aktiven Handler auf und alloziert den benötigten Speicher für die *Computation Area* und das *Pixel Array*.

Man könnte das Aufrufen aller *Start*-Methoden auch als User-Funktion implementieren und das Allozieren der entsprechenden Speicherbereiche in andere Handler verschieben. In diesem Fall würde dieser Handlertyp wegfallen. Durch die *Start*-Methoden haben die übrigen Handler auch eine mächtigere Möglichkeit, sich zu initialisieren.

#### 4.2.7 Typ RAY\_H\_STOP: Aufräumen nach Raycast-Schleife

Die Stop-Handler werden unmittelbar nach Beenden der Raycast-Schleife aufgerufen. In diesen Handlern wird das berechnete Bild zu seinem Ursprung zurückgeschoben beziehungsweise nur der entsprechende Offset berechnet (siehe auch 4.2.4). Anschließend wird das Bild auf die richtige Größe skaliert. Der Handler muß darauf achten, daß die Werte im *Output Array* den Wertebereich  $[0, PMAXVALUE]$  nicht überschreiten, da der Integral-Handler den Bereich im *Pixel Array* möglicherweise überschritten hat.

Diese Aktionen könnten auch ohne Probleme über eine neue *Stop*-Methode realisiert werden. In diesem Fall könnte man die beiden Schritte auch hervorragend voneinander trennen.

Bei der Skalierung fallen auf jeden Prozessor unterschiedlich große Teilbilder an, die sich um maximal ein Pixel in *x*- und in *y*-Ausdehnung unterscheiden. Mir ist allerdings erst sehr spät aufgefallen, daß diese Ausdehnungen für die vier Teilbilder, die durch die hierarchische Speicherung der Daten entstehen, unterschiedlich sein können. Die verwendeten Variablen und Datenstrukturen ziehen sich jedoch durch viele Handler hindurch, so daß eine Änderung zu diesem Zeitpunkt aufwendig ist.

Da ich die Reste bei der nötigen Integer-Division nur für einen Quadranten ermittele, aber alle Quadranten gleich skaliere, können unter Umständen im Bild zwei aufeinander senkrechte Grenzen erscheinen, an denen durch den Übergang von einem Quadranten in den nächsten Artefakte auftreten. Ich habe bis jetzt noch keine solche Artefakte erkennen können, also kann der Effekt nicht sehr groß sein, aber er ist theoretisch vorhanden. Wird noch die *z*-Achsen-Rotation implementiert, erhöht sich die Anzahl an Teilbildern auf 9, wodurch sich der Effekt theoretisch auf jeweils zwei Linien parallel zu den beiden Koordinatenachsen ausweiten würde.

Die in diesem Handler verwendeten Algorithmen sind theoretisch sehr leicht zu verstehen, doch ihre praktische Implementierung auf einem Feldrechner erwies sich unter anderem durch die prozessorabhängige Bildgröße und Nebeneffekte bei der parallelen Programmierung als relativ schwierig.

Derzeit sind die folgenden Stop-Handler implementiert:

**stdstop** In diesem Handler wird das Bild über ein Lookup-Verfahren skaliert.

**stdstopint** Dieser Handler benutzt zur Bildskalierung bilineare Interpolation benachbarter Pixelwerte.

Es kann jeweils nur ein Stop-Handler aktiv sein.

#### 4.2.8 Typ RAY\_H\_POS: Ermittlung eines Volumenwertes

Die Volumenwerte, über die das Integral gebildet werden soll, werden über einen *Pos*-Handler ermittelt. Er schreibt in alle Einträge der *Computation Area* den entsprechenden Wert.

Für spätere Versionen ist dieser Handler auch dafür vorgesehen, Gradienten in einem Voxel zu ermitteln und in der *Computation Area* zu speichern.

Derzeit existieren zwei Pos-Handler:

**posfast** Es wird in einem einfachen Lookup-Verfahren der Wert des nächsten Nachbarn verwendet.

**posbilin** Es wird zwischen den Werten der vier nächsten Nachbarn bilinear interpoliert. Durch die Struktur des Shear-Warp-Algorithmuses kann — wie in 2.3.1 gezeigt — auf trilineare Interpolation verzichtet werden. Damit für die bilineare Interpolation keine Werte über das XNet beschafft und somit die Grenzen als Sonderfälle behandelt werden müssen, greift der Handler auf Volumenwerte zu, die — wie in 4.1.2 beschrieben — überlappend gespeichert sind. Für diesen Handler muß deshalb OVERLAP\_RIGHT mindestens gleich eins sein.

Es kann jeweils nur ein Pos-Handler aktiv sein.

#### 4.2.9 Typ RAY\_H\_TRANSFER: Transferfunktion

Die Transfer-Handler haben die Aufgabe, aus dem Lookup-Wert des Pos-Handlers einen RGBA-Farbwert zu ermitteln, den der Integral-Handler für seine Berechnung verwenden kann.

Es sind die folgenden Transfer-Handler implementiert:

**collookup** Der Lookup-Wert wird als Offset in einer Farbtabelle interpretiert und der Farbwert nachgeschlagen. Da ein paralleler Zugriff auf den ACU-Speicher sehr lange dauert, muß die Farbtabelle auf alle PEs kopiert werden. Dabei wird die

Farbtabelle gleich tiefenkorrigiert, also der Wert der Exponentialfunktion der Gleichung (11) für jeden Farbwert mit dem normierten Opazitätswert multipliziert. Aus Geschwindigkeitsgründen werden von den ersten 256 Prozessoren je ein Wert berechnet und die Ergebnisse anschließend verteilt. Da die Farbtabelle auf allen PEs Speicher benötigt, ist es je nach Speicherausbau nötig, nur mit *uchar*-Farbwerten zu arbeiten.

**collookupint** Dieser Handler enthält den gleichen Algorithmus wie der *collookup*-Handler, nur werden die Farbwerte noch bilinear interpoliert.

**scale** Dieser Handler begrenzt und skaliert die Lookup-Werte mit den Werten, die durch Parameter gegeben sind und ergibt so einen monochromen Farbton.

Es kann jeweils nur ein Transfer-Handler aktiv sein.

#### 4.2.10 Typ **RAY\_H\_INTEGRAL**: Integration

Ein Integral-Handler integriert die in der Transferfunktion berechneten RGBA Werte. Unter Integration muß nicht notwendigerweise ein diskretisiertes Integral verstanden werden, es ist jede Interpretation der Transferdaten möglich. Aus verständlichen Gründen kann jeweils nur ein Integral-Handler aktiv sein.

Derzeit existieren die folgenden Integral-Handler:

**emissabsorp** Dieser Handler implementiert das in 2.2.3 beschriebene Emission-Absorption-Modell ohne Scattering-Effekte und ohne lokale Beleuchtungsrechnung.

**mip** Dies ist der in 2.2.2 beschriebene Mip-Integrationshandler.

**roentgen** Dies ist der in 2.2.1 beschriebene Röntgen-Integrationshandler.

#### 4.2.11 Typ **RAY\_H\_STEP**: Eine Ebene weiter in der Raycast-Schleife

Die Raycast-Schleife durchläuft alle Ebenen des Volumens von vorne ( $z = 0$ ) nach hinten ( $z = \text{Maximum}$ ). Nach jeder Ebene muß — wie in 4.1.2 beschrieben — das *Pixel Array* verschoben werden, um die Scherung zu realisieren. Dies geschieht im Step-Handler. Dies könnte zwar ohne weiteres in das Hauptprogramm übernommen werden, aber das Handlerkonzept bietet geradezu an, hierfür einen eigenen Typ zu implementieren.

Da dieser Typ extrem durch den verwendeten Shear-Warp-Algorithmus geprägt ist und wenig Spielraum für Änderungen läßt, existiert derzeit nur ein Step-Handler.

#### 4.2.12 Typ RAY\_H\_RECOLOR: Abschließende Transformation des Bildes

Dieser Handertyp ist komplett optional und arbeitet nur auf dem bereits fertig gerechneten Bild. Der hier implementierte *pseudocolor*-Handler benutzt die Farbwerte eines monochromatischen Bildes, um über eine Farbtabelle ein Falschfarbenbild zu generieren. Er wurde im wesentlichen für Testzwecke implementiert.

### 4.3 Explorer Kommunikationsmodul RaycastMaspar

Das Kommunikationsmodul ist für den Start des Hauptprogramms auf der Maspar und für das Weiterreichen aller benötigten Daten an die angeschlossenen Module sowie an das Hauptprogramm zuständig. Zu diesem Zweck baut es eine Datenverbindung mit dem Hauptprogramm auf, wie in 4.1.4 beschrieben. Das Modul benötigt vom Benutzer die Rechneradresse, den Usernamen, das entsprechende Passwort sowie den Programmpfad. Außerdem muß dem Modul der Prompt der auf dem Parallelrechner benutzten Shell bekannt sein, damit es das erfolgreiche Ende des Einlog-Vorgangs erkennen kann. Für all diese Parameter stehen Default-Werte zur Verfügung, mit Ausnahme der Rechneradresse.

Username und Rechneradresse können wie in Abbildung 15 ersichtlich in ein Widget eingetragen beziehungsweise an den entsprechenden Explorer-Port geleitet werden. Wird nur ein Rechnername eingegeben, nimmt das Programm den Usernamen des aktuellen Benutzers. Muß das Hauptprogramm unter einer fremden UID gestartet werden, kann ein Username mit einem Klammeraffen vor der Rechneradresse eingegeben werden. Soll das Programm von Hand gestartet werden, muß '@' als Rechnername benutzt werden. Der Port, der für die Datenkommunikation vom Modul bereitgehalten wird, wird beim Start des Moduls im Log-Fenster bekanntgegeben.

Als Defaultwert für den Programmpfad wird `~/prog/raycast` angenommen, für den Prompt '>'. Wird kein Passwort benötigt, weil der aktuelle Rechner in der `.rhosts`-Datei des Parallelrechners eingetragen ist, bleibt dieses Feld leer. Ansonsten kann hier der Name eines Files angegeben werden, der das Passwort enthält. Da es im Explorer nicht direkt möglich ist, ein Passwort-Widget zu erzeugen, das eingetippte Zeichen nicht auf dem Bildschirm ausgibt, habe ich mich für diese unkonventionelle Lösung entschlossen.

Mit einem weiteren Schalter kann angegeben werden, wieviel Informationen `Raycast-Maspar` in das Log-Fenster ausgeben soll. Im Debug-Modus wird die gesamte Kommunikation mit dem Parallelrechner mitprotokolliert, unter anderem auch der Einlog-Vorgang.

Die Verbindung wird aufgebaut, sobald der *Connect*-Schalter aktiviert wird. Klickt man auf den Schalter, wenn bereits eine Verbindung besteht, wird die alte Verbindung gestoppt und eine neue aufgebaut. Ist als Rechnername '@' eingegeben worden, muß nun das Hauptprogramm mit der Option `'explorer'` — wie in Abschnitt 4.1.4 beschrieben — gestartet werden. Das Programm sendet sofort nach der Veränderung eines Eingabeports oder -widgets das Kommando zur Neuberechnung eines Bildes, wenn der *Mode*-Schalter auf *Run* gestellt ist. Ein wiederholtes Drücken des *Mode*-Schalters veranlaßt ebenfalls die Berechnung eines neuen Bildes. Die Stellung *Auto* hat derzeit noch den gleichen Effekt wie die Stellung *Run*, ist aber für eine intelligente Analyse der Änderung vorgesehen. Zum Beispiel könnte beim Hauptprogramm nachgefragt werden, ob eine bestimmte Parameter-Änderung eine Änderung des Bildes zur Folge hat.

### 4.3.1 Startup und Kommunikation mit dem Hauptprogramm

Zum Start des Hauptprogramms wird in einem Sohnprozeß, dessen *stdio* über Pipes an das Explorer-Modul weitergeleitet wird, ein *rlogin*-Kommando ausgeführt. Ist die Verbindung aufgebaut, wird gegebenenfalls das Passwort übertragen und auf das Erscheinen des Prompts gewartet. Falls die Verbindung nicht aufgebaut werden kann, wird das Explorer-Modul wieder in seinen Ausgangszustand gebracht. Im anderen Fall wird das Kommando übermittelt und gestartet.

Dannach wartet das Modul darauf, daß das Hauptprogramm eine Verbindung mit dem Datenport aufbaut. Ist dies geschehen, analysiert das Modul solange die einkommenden Daten, bis das Bereitschaftsschlüsselwort 'ok\n' erscheint. Jetzt können die Parameteroptionen übermittelt und das erste Bild berechnet werden.

### 4.3.2 Kommunikation mit anderen Explorer-Modulen

Damit andere Explorer-Module sowohl die benötigten Volumen-Daten senden als auch die erhaltenen Bilder verarbeiten können, sind einige Explorer-Ports zur Kommunikation vorhanden:

**Input** An diesen Port kann ein uniformes 3D-Lattice mit den Volumendaten geschickt werden. Der Wertebereich ist  $[0, 255]$ , der Datentyp wird bei Bedarf automatisch in `uchar` umgewandelt. Ist kein Modul mit diesem Port verbunden, wird angenommen, daß die Volumendaten vom Hauptprogramm auf andere Weise eingelesen werden.

**Options** Über diesen Port können die Parameter des Hauptprogramms angepaßt werden. Optionen werden in dem in 4.1.3 beschriebenen Format geschickt. Es können mehrere Module angeschlossen werden, alle Optionen werden gesendet. Im allgemeinen wird man diesen Port mit dem `RayOptions`-Modul verbinden.

**Transform** Das uniforme 2D-Lattice, das an diesen Port geschickt werden kann, wird als Abbildungsmatrix interpretiert und an das Hauptprogramm geschickt. Es werden nur homogene  $4 \times 4$ -Matrizen unterstützt.

**Colormap** Wird dieser Port an ein `GenerateColormap`-Modul angeschlossen, wird die übergebene Farbtabelle an das Hauptprogramm gesendet. Derzeit wird nur eine Farbtabelle unterstützt.

**Output** Das berechnete Bild wird über diesen Port als uniformes 2D-Lattice ausgegeben. Es kann zum Beispiel mit `DisplayImg` im Explorer dargestellt werden. Ist kein Modul an diesen Port angeschlossen, nimmt `RaycastMaspar` an, daß das Bild direkt vom Hauptprogramm abgespeichert wird.

**Template** Über diesen Port wird das Parameter-Template ausgegeben. Das `RayOptions`-Modul wertet dieses aus, um dem Benutzer alle möglichen Optionen und ihre Werte anbieten zu können.

Des weiteren sind selbstverständlich alle oben angegebenen Widgets auch über Ports erreichbar.

## 4.4 Explorer Optionsmodul `RayOptions`

Die Wirkungsweise jedes einzelnen Handlers des Hauptprogramms wird von etlichen Parametern beeinflusst. Um dem Benutzer zu erlauben, alle Parameter zu ändern, müssen diese dem User-Interface bekannt sein und dieses muß für jeden Parameter eine Einstellmöglichkeit bereithalten. Laufen das User-Interface und das Hauptprogramm wie in diesem Fall auf unterschiedlichen Rechnern, ist es sehr schwierig und umständlich, diese beiden Programmteile bei Veränderungen konsistent zu halten. Ich habe mich deshalb dazu entschlossen, einen unkonventionelleren Weg einzuschlagen und das User-Interface selbstadaptiv zu halten. Werden dem Hauptprogramm zusätzliche Parameter hinzugefügt oder alte verändert, paßt sich das Optionsmodul automatisch an. Die Flexibilität ist natürlich insofern eingeschränkt, als daß sich auf diese Weise nur einfache Parameter anpassen lassen. Umfangreiche „Parameter“ wie zum Beispiel Farbtabelle oder auch das Eingabevolumen werden deshalb direkt vom Kommunikationsmodul behandelt und an das Hauptprogramm weitergeleitet.

### 4.4.1 Prinzip

In Abbildung 16 ist das User-Interface des Optionsmoduls `RayOptions` zu sehen. Der aktuelle Handler kann mit einem Select-Widget ausgewählt werden. Ursprünglich war es geplant, hier ein ListView-Widget einzusetzen, aber dynamisch veränderliche ListView-Widgets sind anscheinend auch in der Explorer-Version 3 noch mit Fehlern behaftet, da die Hauptfunktion des Moduls nach einer Änderung der ListView-Einträge nicht mehr aufgerufen wurde.

Der ausgewählte Handler kann aktiviert und deaktiviert und seine Parameter ausgewählt werden. Eine Kurzbeschreibung mit Versionsnummer des Handlers beziehungsweise des Parameters wird in einem Textfeld ausgegeben. Abhängig vom ausgewählten Parameter sind einige Widgets auswählbar, mit denen der Parameter aktiviert, deaktiviert und verändert werden kann. Die Ober- und Untergrenzen für reelle und ganzzahlige Parameter können verändert werden, auch wenn dies nicht empfohlen wird.

Mit *Cancel* können alle Parameter auf den Stand im Hauptprogramm zurückgesetzt werden, mit *Accept* werden Parameteroptionen für alle geänderten Parameter generiert und an das Kommunikationsmodul geschickt. Ist dieses im *Run*-Modus, wird automatisch ein Bild mit den neuen, geänderten Parametern berechnet.

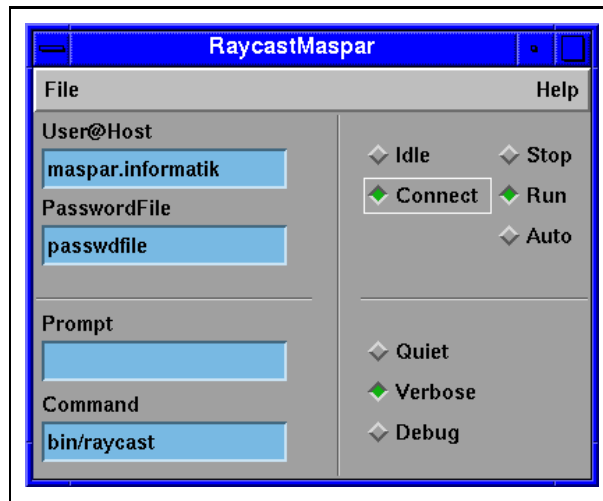


Abbildung 15: User-Interface des Kommunikationsmoduls RaycastMaspar

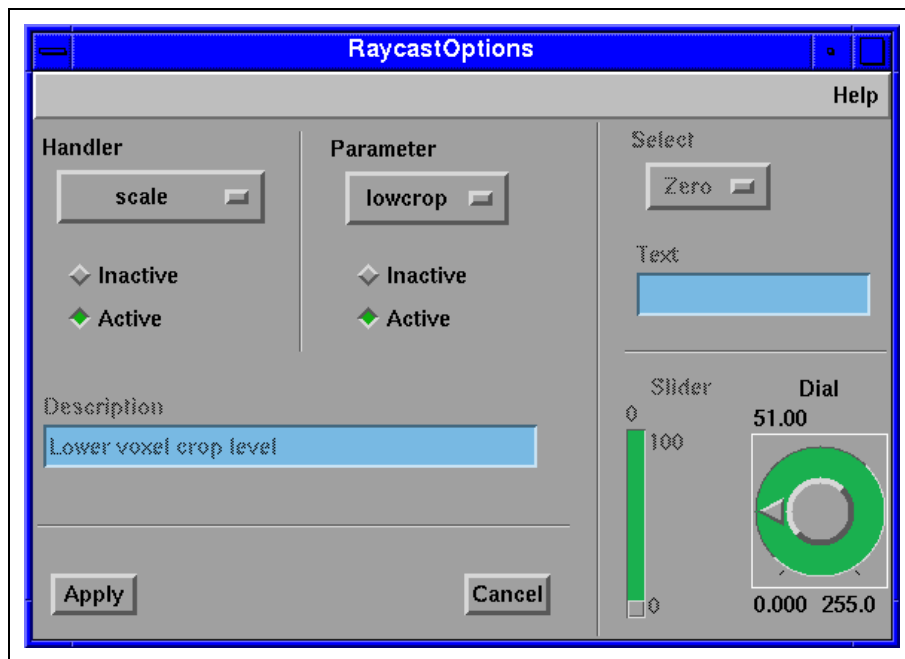


Abbildung 16: User-Interface des Optionsmoduls RayOptions

Im Moment schickt das Hauptprogramm über das Kommunikationsmodul nach jedem Schritt das komplette Parameter-template an das Optionsmodul. Es wäre sinnvoll, nur beim ersten Start das vollständige Template und später ebenfalls nur Änderungen zu schicken, die sich dadurch ergeben, daß zum Beispiel ein Handler aktiviert wurde, der in einer Handlerklasse nur allein aktiv sein darf (Mutual Exclude) oder ein Parameter vom Handler verändert wurde, weil er sich außerhalb zulässiger Grenzen befand. Derzeit ist dies aber noch nicht implementiert.

#### **4.4.2 Realisierung als Explorer-Modul**

Beim Aufruf der Hauptfunktion überprüft das Modul sämtliche Explorer-Ports, ob sich die Daten geändert haben und ruft die entsprechenden Funktionen auf. Ändert sich das Optionstemplate, wird dieses analysiert und die interne Datenbank neu aufgebaut. Wurde ein Widget aktiviert, werden die interne Datenbank und die Widgetinhalte aktualisiert, beziehungsweise die Parameteroptionen generiert. Bei Veränderung der Parameter werden nur die Widgetinhalte verändert und somit neu gezeichnet, die sich unter Umständen verändert haben könnten. Auf diese Weise werden unnötige Redraws vermieden.

Erwähnenswert ist noch die Tatsache, daß sich die Reihenfolge der Parameter in der internen Datenbank natürlich jederzeit ändern kann, da die Reihenfolge im Template nicht definiert ist. Um hier die Anzeige konsistent zu halten, werden nicht nur die aktuelle Handler- und Parameternummer in Betracht gezogen, die von den Explorer-Ports der entsprechenden Widgets zurückgeliefert werden, sondern auch die Namen der aktuellen Handler und Parameter gespeichert und bei jedem Aufruf der Hauptfunktion verglichen. Stimmen sie nicht überein, werden die Anzeigen zurückgesetzt und vom Benutzer eine erneute Auswahl erwartet.

## 5 Ergebnisse

### 5.1 Verifikation der erhaltenen Bilder

Die erhaltenen Bilder lassen sich nicht ohne weiteres durch einfache, automatisierbare Verfahren verifizieren, es sei denn, man hat eine direkte Vergleichsmöglichkeit. Eine Alternative stellt die heuristische Verifikation; dabei unternimmt man mehrere Probeläufe mit unterschiedlichen Parametern und einem bekannten Datensatz — Abbildung 17 zeigt den implementierten Testdatensatz — und vergleicht die erhaltenen Bilder mit den erwarteten.

Die Hauptfehlerquellen der verwendeten Algorithmen können in die folgenden Klassen unterteilt werden:

- Kommunikation der Prozessoren untereinander

Kommunikation spielt in den Handlern *stdstop*, *stdstopint* und *shiftpix* eine wichtige Rolle. Eine fehlerhafte Implementierung erkennt man an Artefakten im berechneten Bild, die an für den Fehler typischen Stellen parallel zu den Bildachsen oder nur bei einer Rotation in eine bestimmte Richtung auftreten. Diese Art Fehler wird zwar leicht bemerkt, es kann allerdings zeitaufwendig sein, die fehlerhafte Programmstelle zu finden.

- Unter-/Überschreiten der Bildgrenzen in Schleifen

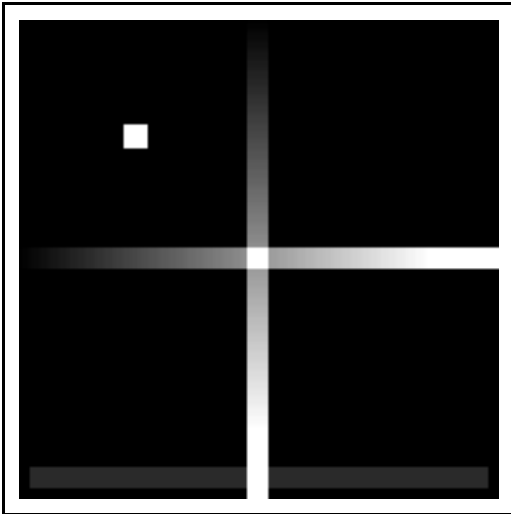
Ein Unterschreiten der tatsächlichen Bildgrenzen während der Berechnung kann an einem mosaik-ähnlichen Aufbau der erhaltenen Bilder erkannt werden. Werden die Bildgrenzen hingegen überschritten, können durch das Überschreiben anderer Variablen seltsame Effekte auftreten. In der Regel läßt sich jedoch ein Fall konstruieren, in dem das Überschreiten der Grenzen eine Speicherschutzverletzung hervorruft.

- Nichtinitialisierte Variablen

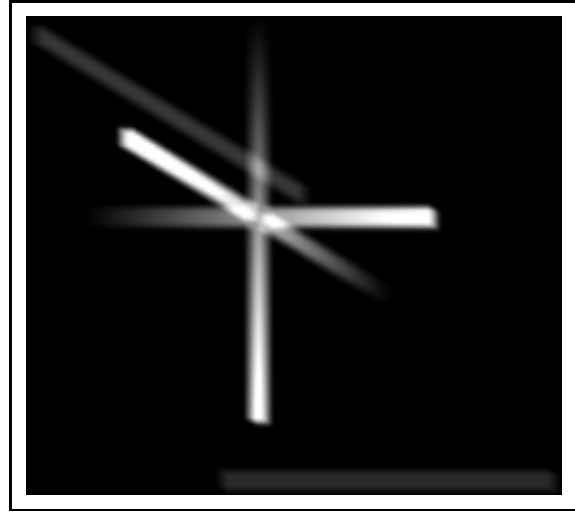
Durch das Handlerkonzept sinkt die Gefahr, auf nichtinitialisierte Variablen zuzugreifen, da jeder Handler vor seinem Aufruf explizit initialisiert wird. Bei einer Änderung der Volumenparameter müssen allerdings alle Handler selbst dafür sorgen, daß ihr innerer Zustand mit den Volumenparametern konform bleibt. Wird zum Beispiel erst ein Volumen der Größe  $128^3$  eingelesen und dargestellt und anschließend eines der Größe  $256^3$ , befindet sich in der derzeitigen Implementierung irgendein Handler noch in seinem alten Zustand und das resultierende Bild ist fehlerhaft.

- Implementierung der Berechnungsvorschriften

Die Werte für die einzelnen Operationen werden nach den in Abschnitt 2 beschriebenen Formeln berechnet. Bei der Implementierung können im wesentlichen Tipfehler die Berechnungsvorschriften verfälschen; diese führen meist zu völlig unsinnigen Bildern.



Frontalansicht



Diagonalansicht

Abbildung 17: Das vom *pseudopic*-Handler bereitgestellte Testvolumem. Bei der Berechnung waren die Handler *posbilin*, *scale*, *roentgen* und *stdstopint* aktiv.

An etlichen Stellen in den einzelnen Handlern müssen Floating-Point-Zahlen in Integer-Werte umgewandelt werden. Nach den IEEE-Richtlinien soll ein Cast nach `int` eine Fließkommazahl immer in Richtung des Ursprungs (0) runden. Die Implementation in der `libmpl` scheint diese Spezifikation aber anscheinend nicht vollständig zu beachten. Die in mehreren Handlern verwendete lineare Interpolation benötigt die exakte Aufteilung eines Wertes in Vor- und Nachkommastellen; eine erste Implementierung mit Casts hat zu merkwürdigen Verzerrungen und Farbstichen im resultierenden Bild geführt. Wurden die Casts durch eine Aufruf von `fp_floor` ersetzt, verhielten sich die Routinen wie vorgesehen. In den meisten Fällen, in denen eine Umwandlung nötig ist, ist das exakte Verhalten der Umwandlung allerdings irrelevant und ein einfacher Cast genügt.

Die Verifikation der einzelnen Handler wurde im Allgemeinen schon während der Implementierung der Teilschritte durchgeführt. Der vom *pseudopic*-Handler erzeugte Datensatz eignet sich hierfür deshalb so gut, weil er das ganze Volumen in allen drei Raumrichtungen ausfüllt, nahezu alle Datenwerte mindestens einmal vorkommen und die genaue Geometrie des Datensatzes durch seine Konstruktionsvorschrift bekannt ist. So konnte durch die Berechnung der Abbildung 20 die Funktionalität des *collookup*-Handlers getestet werden.

Ein wichtiger Test bestand darin, die gleichen Ansichten des Volumens durch interpolierende Handler und Standard-Handler zu berechnen und zu vergleichen. Ergeben die Standard-Handler subjektiv bessere Bilder als die interpolierenden, ist wahrscheinlich der interpolierende Handler fehlerhaft. Da die Standard-Handler immer wesentlich einfacher aufgebaut sind als die interpolierenden Handler, sind erstere meist sofort fehlerfrei implementiert. Ein Vergleich der Bilder durch diese unterschiedlichen Handler-Typen ist in Abschnitt 5.3 zu finden.

## 5.2 Geschwindigkeitsvergleich und Speedup-Messungen

Die Maspar MP-I ist in der heutigen Zeit selbst in großen Ausbaustufen in den meisten Anwendungen kaum schneller als eine Hochleistungs-Workstation; im Vergleich zu modernen Parallelrechnern wie der SGI Power Challenge ist die Leistung einer MP-I mit nur 1024 Prozessoren natürlich nicht zu vergleichen. Allerdings ist das SIMD-Programmiermodell sehr einfach und effizient zu handhaben und es existieren im Bereich der Feldrechner nur wenige schnellere Architekturen. Gerade bei der Volumenvisualisierung gibt es allerdings einige Ansätze, die Berechnungszeit durch *Early Ray Termination* und ähnliche Ansätze drastisch zu reduzieren. Diese Ansätze haben alle gemeinsam, daß sie sich nicht auf ein SIMD-Programmiermodell abbilden lassen. Trotzdem ist es gelungen, eine Implementierung des Shear-Warp-Algorithmuses zu erreichen, dessen Ausführungszeiten mit denen anderer Algorithmen oder Implementierungen bei gleichem Hardware-Einsatz konkurrieren können.

Die Zeitmessung der einzelnen Handler-Routinen hatte auf die Laufzeit des Programms dabei eine so starke Auswirkung, daß die eigentlichen Werte erst nach mehrfacher Ausführung des Programms mit aktiver und inaktiver Zeitmessung hochgerechnet werden konnten. Sehr interessant ist, daß der Overhead, der durch die Handleraufrufe hervorgerufen wird, mit circa 5–50 Millisekunden im Vergleich zu den Berechnungszeiten sehr gering ist.

Die Zeiten des *shiftpix*-Handlers schwanken stark zwischen dem Minimum beim Berechnen einer Frontalansicht — siehe Abbildung 17 links — und dem Maximum beim Berechnen einer Diagonalansicht — siehe Abbildung 17 rechts. Negativ auf die Zeiten hat sich das Fehlen einer pluralen *memmove*-Funktion ausgewirkt. Die Funktion ist zwar in den Manuals der MP-I beschrieben, war aber in den Bibliotheken nicht zu finden. Die zum Vergleich herangezogene, aber nicht verwendbare *memcpy*-Funktion war um den Faktor 2–3 schneller als der selbstprogrammierte *memmove*-Ersatz. Damit dürfte sich die maximale Ausführungszeit des *shiftpix*-Handlers noch um 30–50% reduzieren. Da das Programm je nach Volumengröße und Blickwinkel bis zu 60% der Ausführungszeit im *shiftpix*-Handler verbringt, wird sich eine Verbesserung deutlich auf die Gesamtzeit auswirken.

Es wurden Messungen der Gesamtberechnungszeit für zwei verschiedene Handler-Konfigurationen durchgeführt:

- Konfiguration A: *stdstart, posfast, scale, roentgen, shiftpix, stdstop*
- Konfiguration B: *stdstart, posbilin, collookup, emissabsorp, shiftpix, stdstopint*

In Tabelle 1 sind alle Vergleichswerte und die Skalierbarkeit in Abhängigkeit von der verwendeten Konfiguration aufgelistet. Angegeben sind jeweils der Minimalwert bei Frontalansicht und der Maximalwert bei Diagonalansicht in Millisekunden. Alle aufgeführten Zeiten wurden durch Mittelwertbildung der Resultate von 25 Ausführungen erstellt.

Für diese beiden Konfigurationen findet man zusätzlich in der Tabelle 2 eine Aufstellung, wieviel Zeit in den jeweiligen Handlern auf einer MP-I mit 16384 Prozessoren bei der Berechnung eines Bildes zu einem Volumens der Größe  $256^3$  verbraucht wird. Overhead bezeichnet

Konfiguration	A	A	B	B
Anz. Prozessoren	1024	16384	1024	16384
$128 \times 128 \times 54$	435–1016	82–171	1000–1519	126–222
$128^3$	977–2368	148–366	1963–3296	213–424
$256 \times 256 \times 109$	—	285–718	—	523–953
$256^3$	7147–15830 *	606–1626	13839–22366 *	1033–2054
Skalierbarkeit bei $128^3$	0,41n–0,40n		0,58n–0,49n	
Skalierbarkeit bei $256^3$	0,74n–0,61n		0,84n–0,68n	

\* Auf der MP-I mit 1024 Prozessoren konnte diese Zeitmessung wegen Speichermangel nur mit einer Spezialversion des Programms gemessen werden, die nur für eine Ebene des Volumens Speicher allozierte und daher keine brauchbaren Bilder lieferte. Die gemessenen Zeiten wurden davon allerdings nicht beeinflusst. Die Skalierbarkeit gibt die Berechnungsbeschleunigung in Abhängigkeit von der Prozessoranzahl an. Der Wert  $1,0n$  stellt mit volllinearer Skalierbarkeit die theoretische Obergrenze dar.

Tabelle 1: Vergleich der Gesamtausführungszeiten für unterschiedliche Volumengrößen und Rechnerkonfigurationen

die Zeit, die für Handlerrufe benötigt wird. Mit Timing wurde die Zeit bezeichnet, die zusätzlich zu den angegebenen Gesamtzeiten benötigt wurde, um die Handlerzeiten zu messen. Alle Zeiten beziehen sich auf ihren Worst Case.

Noch unbefriedigend sind die benötigten Zeiten für die Übertragung eines berechneten Bildes zum Explorer. Läuft `raycast` auf einem entfernten Rechner — in meinem Fall in Stuttgart —, sind Übertragungszeiten von bis zu 40 Sekunden für ein Bild der Größe 1 MByte nicht außergewöhnlich. Tests haben jedoch ergeben, daß der Hauptteil der benötigten Zeit nicht in den Kommunikationsroutinen verbraucht wird, sondern in der Linearisierung des Bildes. Die Systembibliotheken bieten zwar Funktionen zur Linearisierung und gleichzeitigen Übertragung zum Frontend an, allerdings sind diese Funktionen für diesen Fall nicht mächtig genug. Durch die unterschiedliche Teilbildgröße auf den einzelnen Prozessoren können diese Routinen nicht eingesetzt werden. Die Übertragung der Bilder zum Explorer kann sicher noch um Größenordnungen beschleunigt werden, allerdings hätte dies den Rahmen dieser Studienarbeit gesprengt.

Die einzigen mir bekannten Vergleichswerte stammen aus [4]. Dabei handelt es sich um die Implementierung eines Raycasting-Verfahrens, unter anderem auf der Maspar MP-I. Es handelt sich zwar nicht um das gleiche Verfahren, aber zumindest die Größenordnungen der Ergebnisse dürften interessant sein. In Tabelle 3 sind einige Ergebnisse im direkten Vergleich aufgeführt.

Konfiguration	A	B	Worst Case bei
<i>stdstart</i>	1 (0%)	15 (1%)	<i>collookup</i> aktiv
<i>stdstop</i>	43 (3%)	—	Frontalansicht
<i>stdstopint</i>	—	130 (6%)	Frontalansicht
<i>posfast</i>	106 (6%)	—	konstant
<i>posbilin</i>	—	294 (14%)	konstant
<i>scale</i>	215 (13%)	—	konstant
<i>collookup</i>	—	203 (10%)	konstant
<i>roentgen</i>	276 (16%)	—	konstant
<i>emissabsorp</i>	—	388 (19%)	konstant
<i>shiftpix</i>	1000 (59%)	1000 (48%)	Diagonalansicht
Overhead	50 (3%)	50 (2%)	konstant
Gesamt	1691 (100%)	2080 (100%)	
Timing	1040 (+62%)	1040 (+50%)	

Alle Zeiten dieser Tabelle wurden mit einer Programmversion ermittelt, die mit Debugging-Information und Asserts kompiliert wurde. Die Gesamtausführungszeit einer Programmversion mit ausgeschaltetem Debugging liegt bei einer Volumengröße von  $256^3$  um circa 100 Millisekunden niedriger.

Tabelle 2: Ausführungszeiten nach Handlern aufgeschlüsselt

Programm	raycast	raycast	aus [4]	aus [4]
Interpolation	Aus	An	Aus	An
1024 Prozessoren	1315–2705	1963–3296	1390	—
4096 Prozessoren	—	—	566 (1812) *	816
16384 Prozessoren	158–378	213–424	238	—
Skalierbarkeit	$0,52n-0,45n$	$0,58n-0,49n$	$0,37n$	—

\* Der in Klammern angegebene Wert bezieht sich auf eine Bildgröße von  $256^2$ , im Gegensatz zu dem vorangestellten Wert, der sich auf eine Bildgröße von  $128^2$  bezieht. *raycast* erzeugt in der Diagonalansicht — dem Worst Case — ebenfalls Bilder der Größe  $256^2$ . Die Diskrepanz der beiden angegebenen Werte gibt einen ersten Eindruck der Schwierigkeit, die angegebenen Zeiten zu vergleichen.

Tabelle 3: Direkter Vergleich mit den in [4] angegebenen Ausführungszeiten. Die Parameter-Unterschiede der jeweiligen Messungen sind in Tabelle 4 wiedergegeben.

Parameter	raycast	aus [4]
Methode	Shear-Warp	Segmented Raycasting *
Volumengröße	$128^3$ *	$128 \times 128 \times 112$
Bildgröße	$181^2 - 256^2$ *	$128^2$
Beleuchtungsberechnung	Emission-Absorption	Lokale Beleuchtungsberechnung *

\* Der Parameter wirkt sich gegenüber dem Vergleichsparameter negativ auf die Berechnungszeit aus.

Tabelle 4: Unterschiede der Parameter bei den Messungen

### 5.3 Nutzen der Interpolationsfunktionen

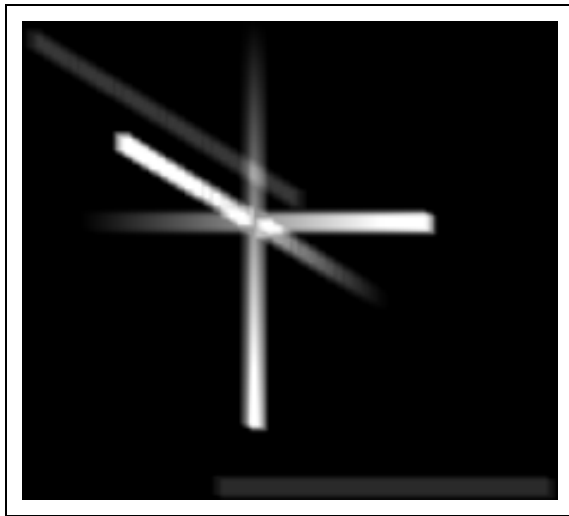
Bei der Implementierung des Shear-Warp-Algorithmuses haben sich drei Teilschritte ergeben, die diskrete Wertetabellen in nichtganzzahligen Abständen auslesen (sampeln) müssen; diese Teilschritte können nun entweder als Lookup-Verfahren implementiert werden oder über eine Interpolationsfunktion realisiert werden:

- Auslesen der Volumenwerte: *posfast* / *posbilin*
- Ermitteln der Farbwerte: *collookup* / *collookupint*
- Abschließende Skalierung des Bildes: *stdstop* / *stdstopint*

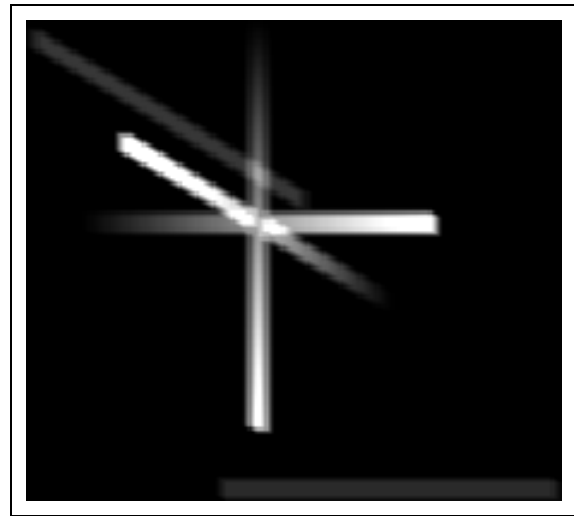
Die Interpolation bei der Ermittlung der Farbwerte hat sich als unbedeutend herausgestellt; es können visuell keinerlei Unterschiede festgestellt werden.

Erste Tests mit dem Testdatensatz ließen vermuten, daß Interpolation bei der Skalierung des Bildes einen deutlich stärkeren Einfluß auf die resultierenden Bilder hat als Interpolation bei den Volumenwerten. Abbildung 18 zeigt den Testdatensatz aus Abbildung 17 mit je einer ausgeschalteten Interpolationsfunktion.

Allerdings haben spätere Berechnungen mit einem realen Datensatz — die MNR-Tomographie eines Kopfes, siehe auch Abbildung 21 — die Vermutung ins Gegenteil umschlagen lassen. Wie Abbildung 23 im Vergleich zeigt, wirkt sich ein Abschalten der Interpolation bei der Bildskalierung nur negativ auf die Qualität aus, während die Interpolation der Volumenwerte essentiell für die Darstellung aller Details aus jedem Blickwinkel ist.



Interpolation der Volumenwerte  
ausgeschaltet  
(*posfast* anstelle von *posbilin* aktiv)



Interpolation bei der Bildskalierung  
ausgeschaltet  
(*stdstop* anstelle von *stdstopint* aktiv)

Abbildung 18: Das Testvolumen mit unterschiedlichen Status der Interpolationsroutinen

## 6 Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde ein Shear-Warp-Algorithmus aus der Klasse der Volume-Raycasting-Verfahren zur direkten Volumenvisualisierung für das SIMD-Programmiermodell der Maspar MP-I entwickelt. Das Programm wurde über zwei Module in das IRIS Explorer Visualisierungspaket transparent eingebunden.

Für das Programm wurde ein Handler-Konzept entwickelt, das dem Benutzer erlaubt, während der Laufzeit einzelne Handler zu aktivieren und zu deaktivieren sowie ihre Parameter zu ändern. Die einzelnen Handler kapseln jeweils einen Teilschritt des implementierten Algorithmuses und verbergen die jeweilige Implementierung. Durch das Handlerkonzept ist das Programm leicht erweiterbar, ohne bereits existierende Handler entfernen zu müssen. Zeitmessungen haben ergeben, daß das Konzept sich nur unwesentlich negativ auf die Berechnungsdauer auswirkt. Über eines der implementierten Explorer-Module kann der Benutzer sämtliche Parameter aller Handler beeinflussen. Änderungen an der Parameterstruktur des Hauptprogramms werden dabei automatisch an das Modul weitergeleitet.

Der implementierte Shear-Warp-Algorithmus hat sich als effizient und gut skalierbar herausgestellt. Das SIMD-Programmiermodell der Maspar ist einfach zu benutzen und bildet eine gute Abstraktion von der Hardware. Ansätze zur Berechnungszeitverkürzung wie *Early Ray Termination*, die auf MIMD-Architekturen die Berechnung deutlich beschleunigen, können aufgrund der SIMD-Architektur allerdings nicht eingesetzt werden.

Es wurden Tests mit einem algorithmischen Testdatensatz (Abbildung 20) und dem Datensatz einer MNR-Tomographie eines Kopfes (Abbildung 21) durchgeführt. Die Erstellung einer guten Farbtabelle für den realen Datensatz (Abbildung 22) erwies sich als schwierig, da zum Beispiel für das Gehirn typische Werte auch teilweise in der Haut vorkommen. Für eine gute Klassifikation der Daten ist eine einfache Transferfunktion wie das implementierte Table-Lookup-Verfahren anscheinend nicht ausreichend; aufwendigere Verfahren, die auch den lokalen Gradienten hinzuziehen, sind wahrscheinlich geeigneter.

Die abschließende  $z$ -Achsen-Rotation ist der nächste Teilschritt, der implementiert werden sollte. Dabei ist zu beachten, daß sich durch die Rotation die Bilddarstellung im Speicher ändert. Die *Output*-Handler müssen dementsprechend angepaßt werden; die benötigte Theorie wurde bereits in dieser Arbeit behandelt. Es muß noch überdacht werden, ob eine Verlagerung dieses Schrittes auf das Frontend oder einen anderen Rechner sinnvoll ist.

Dannach sollte der noch fehlende Transformations-Handler implementiert werden. Durch diesen wird es dann möglich sein, Volumen auch unter Winkeln größer als  $\frac{\pi}{4}$  zu betrachten. Es muß getestet werden, ob der Globale Router eine für diesen Zweck genügende Übertragungsleistung bietet. Andernfalls sollte eine Lösung mit XNet-Kommunikation in Betracht gezogen werden.

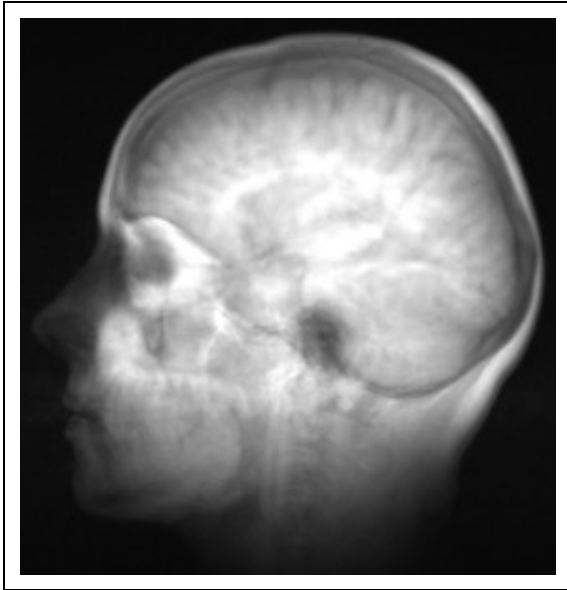
Die Übertragung der berechneten Bilder zum Explorer-Modul kann noch deutlich beschleunigt werden. Ein interessanter Aspekt wäre die parallele Komprimierung der Bilddaten vor ihrer Linearisierung. Allerdings ist dies durch die unterschiedliche Teilbildgröße auf den

einzelnen Prozessoren keine triviale Aufgabe. Die Linearisierung der Teilbilder kann teilweise mit der Übertragung zum Frontend gekoppelt und über Betriebssystemroutinen beschleunigt abgewickelt werden. Außerdem kann die Übertragung der Bilder zum Explorer asynchron zur Linearisierung erfolgen.

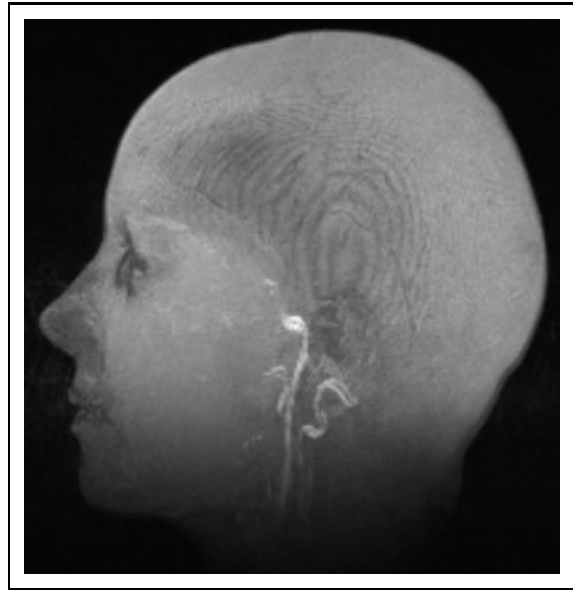
Für interaktive Bildraten ist eine direkte Darstellung der erhaltenen Bilder an der Maspar-Konsole interessant. Die im mpddl-Handler implementierte Ausgabe mit Hilfe der `libmpddl` ist durch die abschließende Skalierung nicht mehr verwendbar, da diese Bibliothek nur verteilte Bilder mit identischer Teilbildgröße auf allen Prozessoren unterstützt.

Um noch aussagekräftigere und realistischere Bilder zu erhalten, muß ein besseres Beleuchtungsmodell benutzt werden. Die Erweiterung des implementierten Emission-Absorption-Modells um lokale Beleuchtungsberechnung ist nicht weiter schwierig; allerdings müssen die *Pos*-Handler zusätzlich zum entsprechenden Volumenwert auch den lokalen Gradienten ermitteln. Hierfür muß auch die Struktur der *Computation Arrea* erweitert werden. Da die Ermittlung des Gradienten mehr Rechenzeit benötigt und dieser im Emission-Absorption-Modell nicht verwendet wird, sollte sie nicht in die bestehenden Handler integriert werden; durch das Handler-Konzept kann hierfür leicht ein neuer Handler erstellt werden.

Entwickelt wurde das Hauptprogramm auf der MP-I des Lehrstuhls VII des IMMD der Universität Erlangen-Nürnberg, bei dem ich mich herzlich für die Unterstützung dieser Arbeit bedanken möchte. Für die Performance-Tests hat uns die Universität Stuttgart freundlicherweise ihre Maspar MP-I mit 16384 Prozessoren zur Verfügung gestellt.



Integrationshandler *roentgen*



Integrationshandler *mip*

Abbildung 19: Beispiele zu unterschiedlichen Integrationshandlern

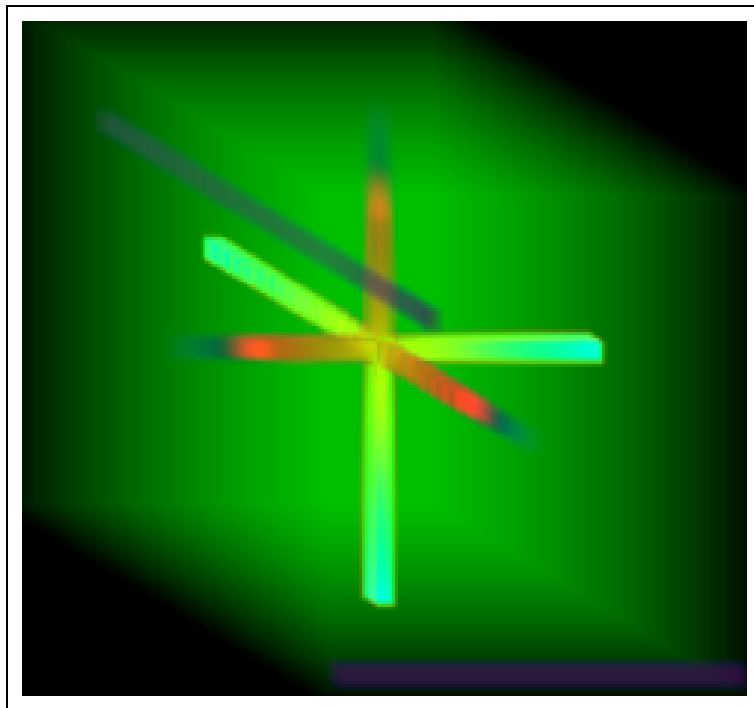


Abbildung 20: Der Testdatensatz mit dem Emission-Absorption-Modell berechnet

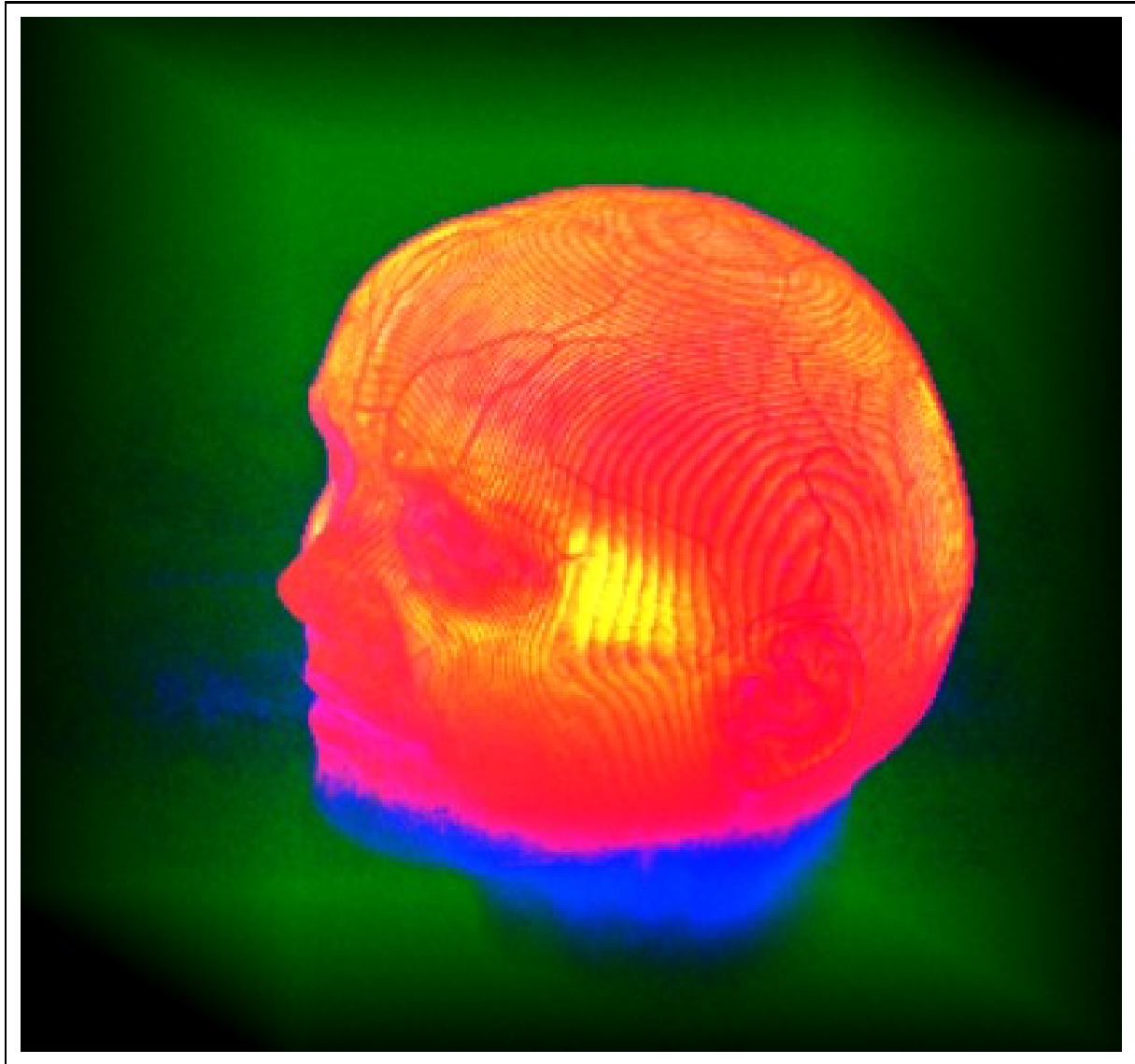


Abbildung 21: Der MNR-Datensatz mit dem Emission-Absorption-Modell berechnet

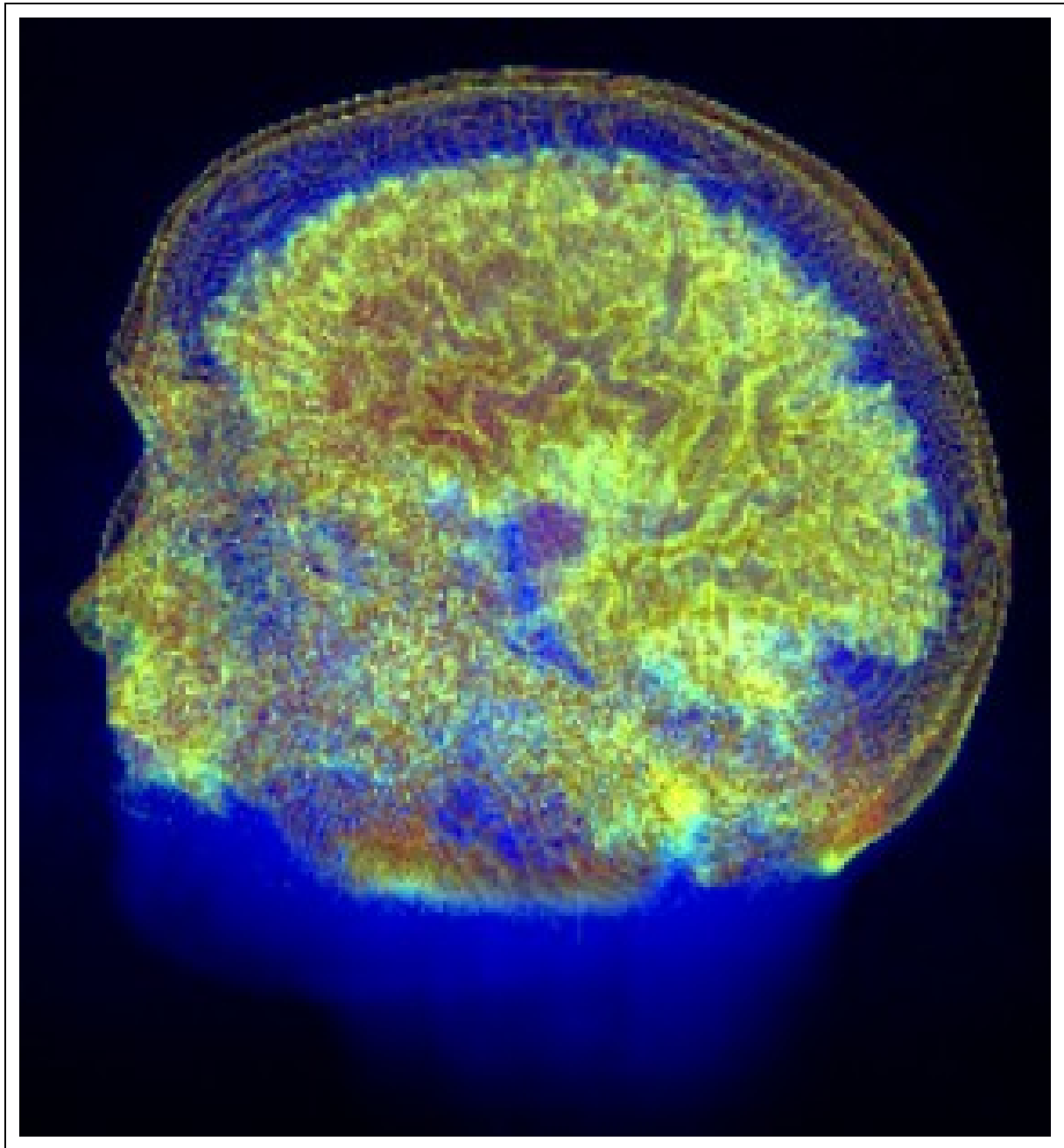
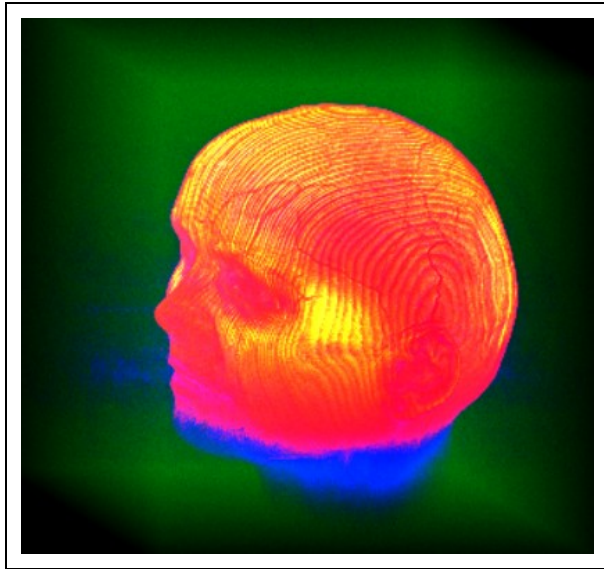
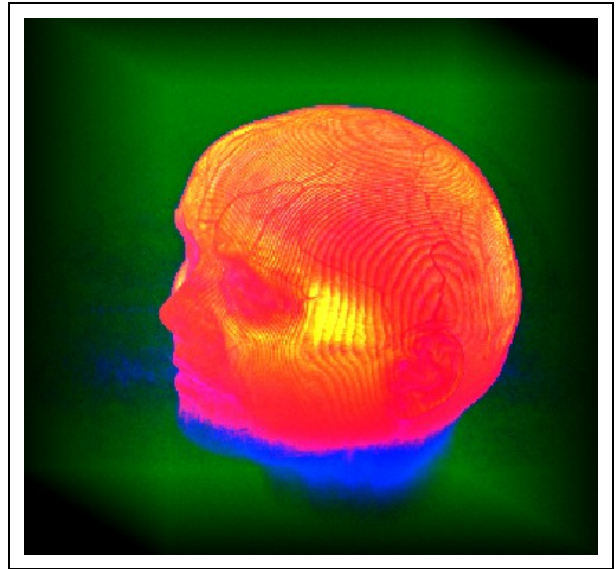


Abbildung 22: Das MNR-Volumen mit einer anderen Farbtabelle berechnet



Interpolation der Volumenwerte  
ausgeschaltet  
(*posfast* anstelle von *posbilin* aktiv)



Interpolation bei der Bildskalierung  
ausgeschaltet  
(*stdstop* anstelle von *stdstopint* aktiv)

Abbildung 23: Das MNR-Volumen mit unterschiedlichen Status der Interpolationsroutinen

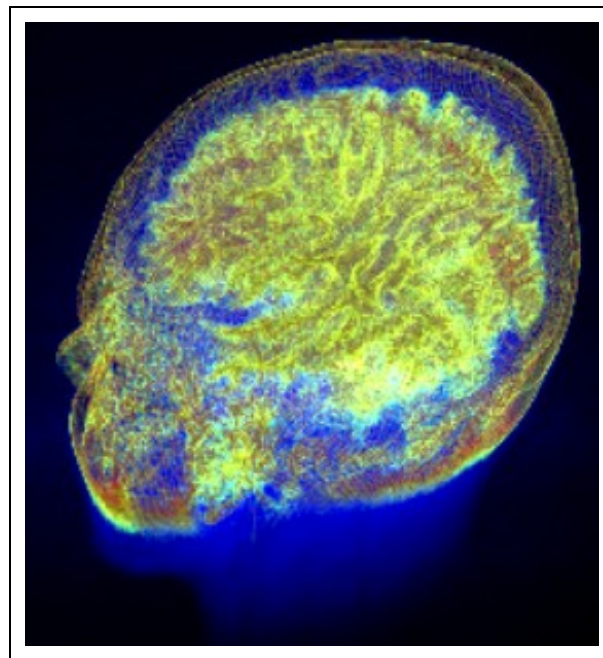


Abbildung 24: Eine andere Ansicht des MNR-Volumens

## Literatur

- [1] Mines B. Amin, Ananth Grama, and Vineet Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Parallel Rendering Symposium*, pages 7–14, Atlanta GA USA, 1995. ACM.
- [2] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice*. Addison-Wesley, second edition, 1993.
- [3] H. C. Hege, T. Höllerer, and D. Stalling. Volume rendering.
- [4] William M. Hsu. Segmented ray casting for data parallel volume rendering. In *IEEE*, pages 7–14, 1993.
- [5] Philippe Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. In *Parallel Rendering Symposium*, pages 15–22, Atlanta GA USA, 1995. ACM.
- [6] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation.
- [7] Maspar Computer Corporation, Sunnyvale, California 94086. *MasPar MP-1 Installation and Service Manual*.
- [8] Maspar Computer Corporation, Sunnyvale, California 94086. *MPL Reference Manual*.
- [9] Hans-Peter Meinzer. Räumliche Bilder des Körperinneren. *Spektrum der Wissenschaft*, pages 56–65, July 1993.
- [10] Peter Schröder and Wolfgang Krüger. Data parallel volume-rendering algorithms for interactive visualization. In *The Visual Computer*, pages 405–416. Springer-Verlag, 1993.
- [11] Silicon Graphics Inc., Mountain View, California. *IRIS Explorer Module Writer's Guide*, 1993.
- [12] Wolfgang Krüger and Peter Schröder. Data parallel volume rendering. In *Scientific Visualization*, pages 37–52. Academic Press Ltd, 1994.

# 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.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 02. April 1996

(Matthias Hopf)