Inaugural-Dissertation

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von
Diplom-Informatiker Michael Lampe
aus Heilbronn

Tag der mündlichen Prüfung: 6. 7. 2006

Parallele Visualisierung — ein Vergleich

Gutachter: Prof. Dr. Gabriel Wittum

Prof. Dr. Peter Bastian

Kurzfassung

Die wissenschaftliche Visualisierung stellt Methoden zur Interpretation von Daten bereit, wie sie etwa als Ergebnis einer Computersimulation entstehen. Mit den heute verfügbaren großen Parallelrechner-Installationen (mehrere hundert bis tausend Prozessoren, bis zu mehreren Terabyte Hauptspeicher) lassen sich praktisch bedeutsame Probleme in hoher Auflösung simulieren. Die traditionelle Vorgehensweise besteht in der Abspeicherung der Daten auf dem Rechner, auf dem die Simulation abläuft, dem Transfer der Daten auf eine Workstation nach Beendigung der Simulation und der dort abschließend stattfindenden Visualisierung. Während diese Vorgehensweise gut zum typischen Rechenzentrumsbetrieb passt — die Simulation läuft nicht-interaktiv im Batchmodus —, wird sie in jüngerer Zeit immer unpraktikabler: Die während der Simulation anfallenden enormen Datenmengen können im allgemeinen nicht einmal mehr auf dem Parallelrechner selbst abgespeichert werden, ein Transfer selbst über schnelle Netze würde viele Stunden oder sogar Tage dauern und natürlich wäre jede Workstation mit dieser Datenmenge hoffnungslos überfordert. Als Ausweg bietet sich an, die Daten sofort nach ihrer Entstehung auf dem Parallelrechner zu visualisieren. Dadurch steht einmal die nötige Rechenpower zur ihrer Verarbeitung zur Verfügung, zum anderen entfällt die Notwendigkeit, die Daten zu speichern und später zu übertragen. Folgerichtig ist das Thema dieser Arbeit die parallele Visualisierung. Die Möglichkeiten zur Parallelisierung des Visualisierungsprozesses werden anhand der typischen drei Schritte, die während dieses Prozesses ausgeführt werden, untersucht und bereits existierende Implementierungen diskutiert. Der Schwerpunkt liegt dabei auf dem letzten dieser drei Schritte, dem parallelen Rendering, da die Parallelisierung der ersten beiden im allgemeinen keine besonderen Probleme stellt. Abschließend erfolgt ein Vergleich der prinzipiellen Ansätze anhand des vom Autor parallelisierten Visualisierungssubsystems von UG, einem in der Arbeitsgruppe „Technische Simulation“ des Interdisziplinären Zentrums für wissenschaftliches Rechnen (IWR) der Universität Heidelberg entwickelten Simulations-Framework. Das wesentliche Ergebnis ist dabei, dass die Skalierbarkeit des Visualisierungsprozesses trotz immer wieder geäußerter anderer Ansichten nur durch seine vollständige Parallelisierung erreicht wird.

Abstract

Scientific visualization provides methods for the interpretation of data like those resulting from computer simulations. Current parallel computers (several hundreds or thousands of processors, up to several terabyte of main memory) are capable of simulating practically relevant problems at high resolution. The traditional procedure consists of saving the data on the machine the simulation runs on, transfering the data to a workstation after the simulation is finished, and finally visualizing it there. While this procedure fits the usual way computing centers are operated—simulations are run non-interactively in batch mode—, it is becoming more and more infeasible: the large-scale data that is produced by a simulation run can in general not even be saved on the parallel machine itself, the transfer even over fast networks would take many hours to days, and naturally no workstation would be able to handle it. One resort is to visualize the data on the spot on the parallel computer. This way one has enough computing power available for processing it and, at the same time, can do without saving and then transfering the data. Consequently, the subject of this thesis is parallel visualization. The options to parallelize the visualization process are investigated on the basis of the three typical steps involved in this process, and already existing implementations are discussed. The focus is on parallel rendering, the last of these three steps, because parallelizing the first two steps is generally not very challenging. Finally, the principal approaches are compared within UG's visualization subsystem parallelized by the author. UG is a framework for building simulation programs developed by the work group "Simulation in Technology" of the "Interdisciplinary Center for Scientific Computing" of the University of Heidelberg. The main conclusion from this comparison is that the visualization process only scales if it is parallelized entirely, despite prevalent opinion to the contrary.

These scientific products take of course some time to fructuate.

—Dr. Edgar H. Humbert

Inhaltsverzeichnis

Motivation

Heutige Supercomputer ermöglichen die Simulation „großer“ Probleme. In [OPR00] wird ein Beispiel aus der Meteorologie erwähnt. Auf einem Gitter mit $1000 \times 1000 \times 200$ Knoten wird ein dreidimensionales, zeitabhängiges Feld berechnet. In jedem Knoten sind pro Zeitschritt 23 Unbekannte zu bestimmen, insgesamt also 37 GByte pro Zeitschritt (bei 64 Bit pro Unbekannte). Die gesamte Simulation geht über 3600 Zeitschritte (zwölf Stunden Rechenzeit auf 400 Prozessoren einer SGI/Cray T3E) und liefert daher 3600×37 GByte $= 133$ TByte an Daten.

Der klassische Ansatz zur Auswertung von Rechenergebnissen (*Postprocessing*) besteht in der Abspeicherung und dem anschließenden Transfer der Daten auf die eigene Workstation oder eine ‚High End‘-Graphikmaschine, wo dann die Analyse der Daten mit üblichen Visualisierungsprogrammen durchgeführt wird.

Diese Vorgehensweise ist offensichtlich in dem genannten Beispiel zum Scheitern verurteilt. Während das Abspeichern der besagten Datenmenge auf dem Supercomputer noch möglich sein kann, dauert der Datentransfer, wenn man etwa eine Verbindung mit 1 GBit/s annimmt, ungefähr 300 Stunden. Ganz zu schweigen davon, dass die Zielmaschine dieser Datenmenge nicht gewachsen sein wird.

In dieser Arbeit werden Möglichkeiten diskutiert, mit denen diesem Problem begegnet werden kann, indem nämlich auch die Visualisierung der Daten auf dem Supercomputer durchgeführt wird.

Kapitel 1

Grundlagen

Dieses Kapitel vermittelt die wichtigsten Grundbegriffe zu den Themen *Wissenschaftliche Visualisierung*, *Computergraphik* und *Parallelrechner*.

1.1 Wissenschaftliche Visualisierung

1.1.1 Definition und Aufgabe

[Dom94] definiert *Wissenschaftliche Visualisierung* so: „*Visualisization* of *Scientific Data* describes the application of graphical methods to enhance interpretation and meaning of *Scientific Data*.“ *Wissenschaftliche Daten* können dabei das Resultat von Messungen oder von Computersimulationen sein. Statt von *Wissenschaftlicher Visualisierung* soll im Folgenden kurz von *Visualisierung* die Rede sein, statt von *wissenschaftlichen Daten* kurz von *Daten*.

Aufgabe der Visualisierung ist es also, Methoden und Werkzeuge zur Verfügung zu stellen, die eine bildhafte Darstellung von Daten erlauben, um so einen Zugang zu ihrer Bedeutung zu erlangen und ihre Interpretation zu ermöglichen.

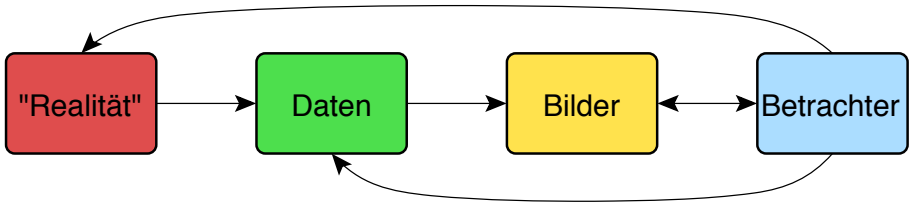

Abbildung 1.1: Abbildungen im Visualisierungskontext.

Visualisierung im Kontext kann man auch (vgl. Abb. 1.1 und [DG94]) als Folge von Abbildungen betrachten. Die „Realität“ wird durch Messung oder Simulation auf einen Datensatz abgebildet und dieser wiederum auf ein Bild. Am Ende der Kette steht der Betrachter dieses Bildes, der versucht, mit dessen Hilfe Einsichten über die Bedeutung der Daten bzw. das Wesen der „Realität“ zu gewinnen. Diese Rückschlüsse sind der Subjektivität des Betrachters — d.h. seinen individuellen Fähigkeiten, seiner Erfahrung, seiner Ausbildung, seinem kulturellen Umfeld, etc. — unterworfen.

Im Weiteren soll daher unter Visualisierung nur der rein technische Teil, also die Abbildung von Daten auf Bilder, verstanden werden.

1.1.2 Charakterisierung wissenschaftlicher Daten

Die folgende abstrakte Charakterisierung von Daten ist eine Vereinfachung der in [Bro92] angegebenen. Datensätze E (Entities) werden mit E_n^{mC} bezeichnet. Dabei bedeutet:

n Die Dimension des Gebiets, auf dem Daten definiert sind.

m Die Anzahl der gegebenen Datensätze nach C.

C Die Art des Datensatzes. S ist ein skalares Feld, V_k ein Vektorfeld der Dimension k.

Für $m = 1$ wird m nicht angegeben. Falls die Daten zeitabhängig sind, wird dies mit einem zusätzlichen Subskript t gekennzeichnet.

Beispiele: E_2^{3S} ist ein zweidimensionales Feld, bei dem jedem Raumpunkt drei skalare Werte zugeordnet sind, z.B. Dichte, Druck und Temperatur. $E_{3;t}^{V_3}$ bezeichnet ein dreidimensionales zeitabhängiges Vektorfeld, in dem jeder Vektor drei Komponenten hat, etwa ein Geschwindigkeitsfeld.

Diese Notation gibt nur die Struktur der Daten ohne Beachtung ihrer Bedeutung wieder und eignet sich daher, um Visualisierungstechniken ohne Bezug zu einem konkreten Beispiel zu betrachten.

1.1.3 Visualisierungstechniken

Visualisierungstechniken bezeichnen allgemeine Methoden zur Visualisierung eines Datensatzes. Nachfolgend einige Beispiele. Es sei jeweils f die durch die Daten definierte Funktion, die durch Interpolation auf das ganze Gebiet ausgedehnt ist.

2D

E_2^S Isolinien ($f(x,y) = \lambda$ für einige λ), Bilddarstellung ($f(x,y) \mapsto$ Punktfarbe), Graph ($z = f(x,y)$).

$E_{2;t}^S$ wie E_2^S für einige Zeitpunkte $t_1, \ldots, t_n$ und Animation.

$E_2^{V_2}$ Pfeildiagramm (vgl. Abb. 1.2), Bahnlinien ($\dot{x} = f_1(x,y)$, $\dot{y} = f_2(x,y)$ für einige Anfangswerte x, y)

$E_{2;t}^{V_2}$ wie $E_2^{V_2}$ für einige Zeitpunkte $t_1, \ldots, t_n$ und Animation.

3D

E_3^S Isoflächen ($f(x,y,z) = \lambda$ für einige λ, vgl. Abb. 1.3), Schnitte mit den Techniken für E_2^S.

$E_{3;t}^S$ wie E_3^S für einige Zeitpunkte $t_1, \ldots, t_n$ und Animation.

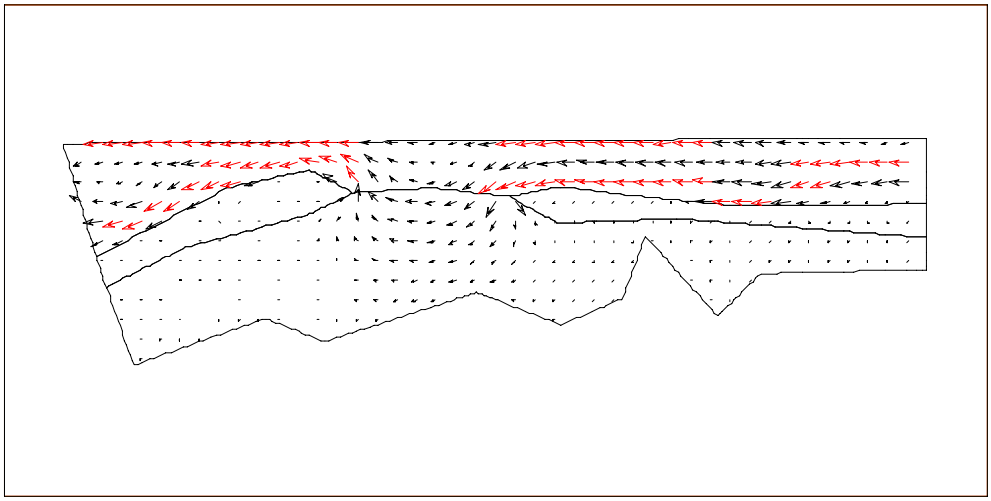

Abbildung 1.2: $E_2^{V_2}$: Vektorfeld als Pfeildiagramm mit UG [BBJ+97].

$E_3^{V_3}$ wie $E_2^{V_2}$, Schnitte mit den Techniken für $E_2^{V_2}$

$E_{3;t}^{V_3}$ wie $E_3^{V_3}$ für einige Zeitpunkte $t_1, \ldots, t_n$ und Animation.

Volume Rendering

Volume Rendering ist eine Visualisierungstechnik für E_3^S und $E_3^{V_k}$. Sie wird hier gesondert aufgeführt, weil sie im Gegensatz zu den klassischen Visualisierungstechniken versucht, die Information des *gesamten* Datensatzes in *einem* Bild darzustellen.

Dazu werden die Daten mittels *Transferfunktionen* auf optische Eigenschaften eines wolkenartigen Materials abgebildet (Transparenz, Farbe, eventuell Lichtemission und ähnliches), und die Effekte dieser optischen Eigenschaften längs jedem Strahl in der Blickrichtung des Betrachters aufintegriert.

In [Max95] werden diverse optische Modelle vorgestellt. Abb. 1.4 zeigt ein einfaches Beispiel, bei dem nur die Absorption des Lichts durch das Material modelliert wurde.

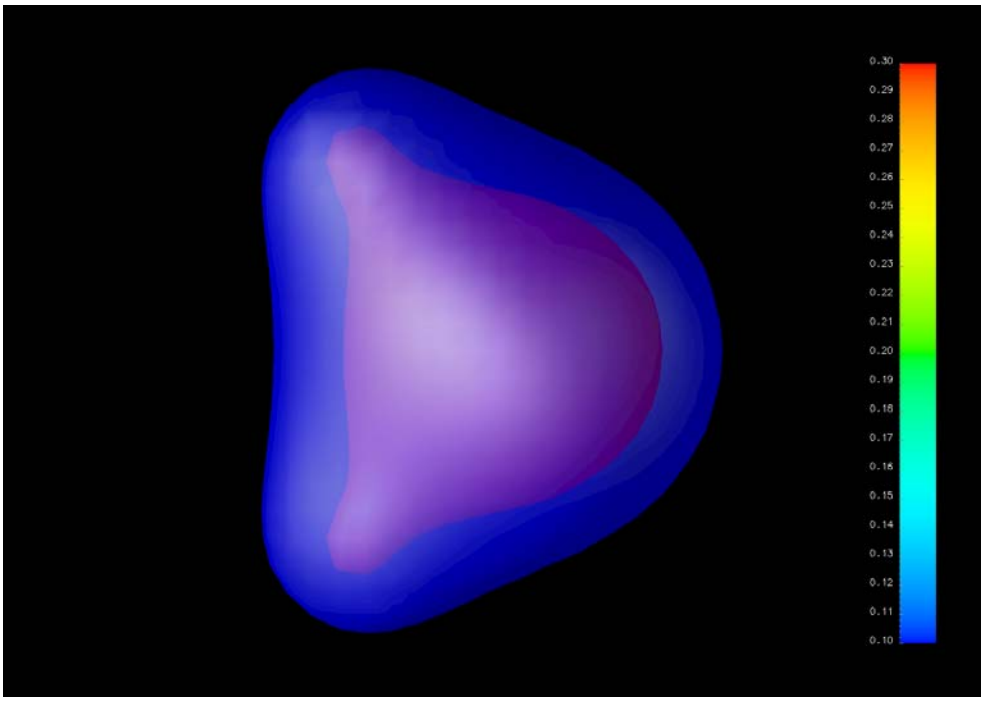

Abbildung 1.3: E_3^S: Isoflächen erstellt mit DataExplorer [AT95].

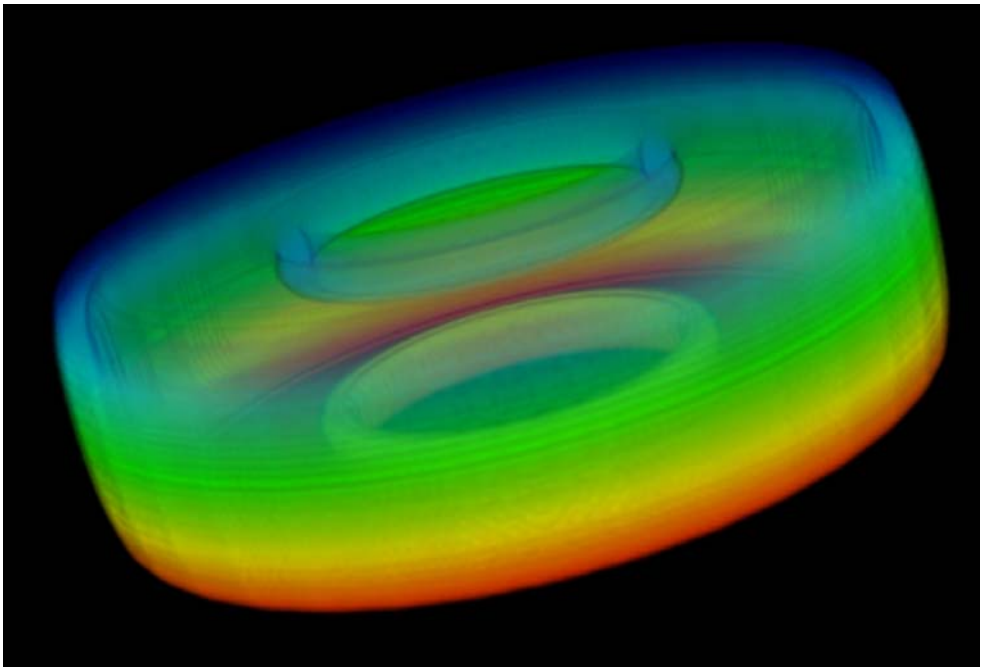

Abbildung 1.4: E_3^S: Volume Rendering erstellt mit dem VisIt Visualization Tool [VVT].

1.2 Computergraphik

1.2.1 Rendering

Unter *Rendering* versteht man die Konvertierung einer *Szene* (vgl. Abb. 1.5) bestehend aus geometrischen Objekten, Lichtquellen und Projektionsparametern in ein zweidimensionales aufgerastertes Bild. Im folgenden soll zur Abgrenzung von anderen Renderingtechniken (z.B. Raytracing, Radiosity, Volume Rendering) speziell von Polygonrendering die Rede sein. Geometrische Objekte sind in diesem Fall durch ebene polygonale Facetten begrenzt, die als Input für die im folgenden skizzierte *Renderingpipeline* (vgl. [FvDFH96]) dienen. Die Renderingpipeline zerfällt dabei in zwei größere Abschnitte, Geometrieverarbeitung und Rasterung.

Geometrieverarbeitung

- *Modelltransformation*
  Geometrische Objekte können in einem eigenen *Modellkoordinatensystem* definiert sein. In diesem Fall ist eine Transformation in das *Weltkoordinatensystem* — das Koordinatensystem, das für das Rendering der Szene benutzt wird — nötig.
- *Rückseiten entfernen*
  Rückseiten des Polygonmodells — solche, deren äußere Normale vom Betrachter wegweist — werden entfernt, da sie im projizierten Bild nicht sichtbar sind.
- *Beleuchtungsberechnungen*
  Die Einflüsse der Lichtquellen auf die geometrischen Objekte unter Berücksichtigung des Betrachterstandpunktes werden berechnet und einzelnen Punkten Farb- bzw. Helligkeitswerte zugeordnet.
- *Projektion*
  Abbildung der geometrischen Objekte aus dem dreidimensionalen Raum auf die Bildebene per Parallel- oder Perspektivtransformation.

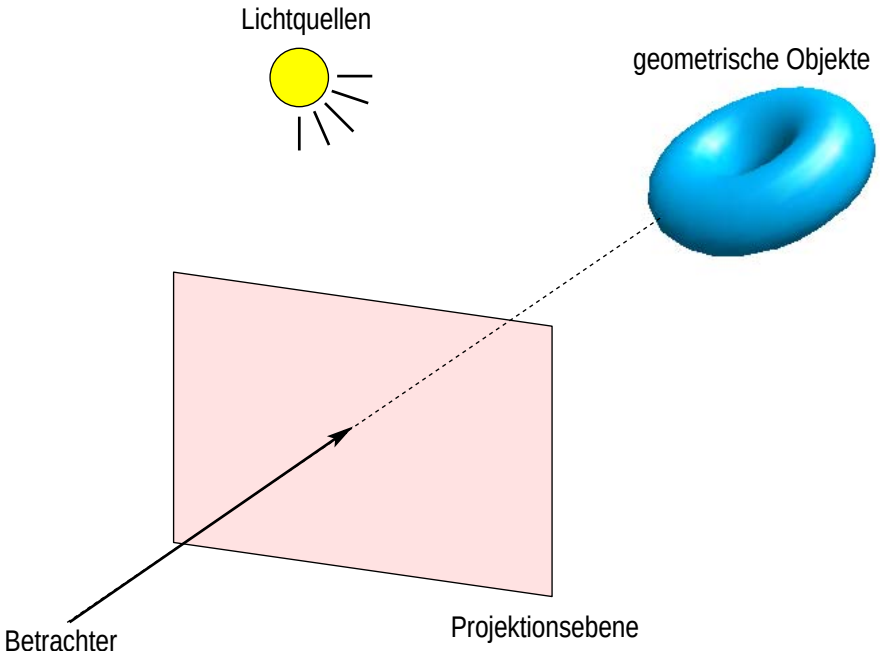

Abbildung 1.5: Eine Szene wird beschrieben durch geometrische Objekte, Lichtquellen und Projektionsparameter.

- *Clipping*

  Abschneiden jener Teile der projizierten Polygone, die außerhalb des sichtbaren Bereichs der Bildebene liegen.

Rasterung

- Scan Conversion

  Konvertierung der von projizierten Polygonen überdeckten Fläche in *Fragmente*. Fragmente sind einem Pixel zugeordnet und tragen Informationen wie Farbe, z-Wert, α-Wert, etc.

- Fragmentoperationen

  Ausgabe der Fragmente, z.B. durch Eintrag in den Frame Buffer der Graphikkarte, eventuell nach Bestehen vorheriger Tests (z.B. z-Wert kleiner als der des bereits gespeicherten Pixels), oder durch Verrechnung mit dem bereits gespeicherten Pixel (z.B. α-Blending).

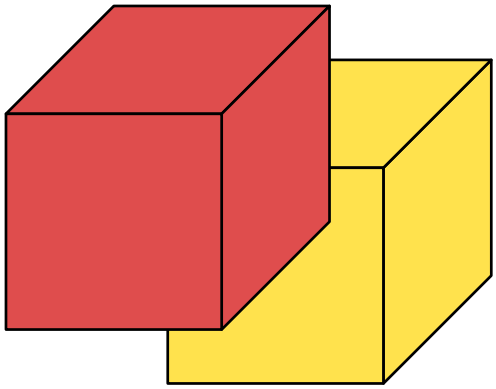

Abbildung 1.6: Illustration zum ‚Hidden Surface'-Problem. Der vordere rote Würfel verdeckt Teile des hinteren gelben.

1.2.2 ‚Hidden Surface'-Problem

Das *‚Hidden Surface'-Problem* ergibt sich aus der Tatsache, dass näher zum Betrachter liegende Objekte solche, die weiter entfernt sind, teilweise oder ganz verdecken können, vgl. Abb. 1.6. Die Aufgabe besteht also darin, die verdeckten Teile im gerenderten Bild auch korrekterweise nicht zu zeigen. (Die bereits erwähnte Entfernung der Rückseiten stellt keine allgemeine Lösung dieses Problems dar.)

Die verschiedenen Lösungen zum ‚Hidden Surface'-Problem lassen sich nach [SSS74] in folgende Klassen einteilen.

Objektraumalgorithmen

Objektraumalgorithmen arbeiten in dem Koordinatensystem, in dem die geometrischen Objekte definiert sind (Weltkoordinatensystem). Die sichtbaren Teile können mit der gleichen Genauigkeit berechnet werden, in der die Szenenbeschreibung vorliegt. Da man das ‚Hidden Surface'-Problem damit bereits gelöst hat, bevor die geometrischen Objekte die Renderingpipeline durchlaufen, sind diese Algorithmen unabhängig vom Ausgabegerät und dessen Eigenschaften.

Bildraumalgorithmen

Bildraumalgorithmen lösen das ‚Hidden Surface'-Problem erst während

der Rasterung mit gerade der nötigen Genauigkeit, indem sie entscheiden, was pro Pixel sichtbar ist. Sie arbeiten also als Teil der Renderingpipeline mit den diskreten Koordinaten eines Ausgabegerätes und sind auf dessen Auflösung zugeschnitten. (Bsp.: z-Buffer-Algorithmus)

‚List Priority'-Algorithmen

‚List Priority'-Algorithmen stellen eine Zwischenstufe der beiden vorgenannten Klassen dar. Dazu benötigen sie ein Ausgabegerät, das es erlaubt, bereits getätigte Ausgaben durch neue zu überschreiben (Bsp.: Bildschirm, Gegenbsp.: Plotter). Die Idee ist, dies auszunutzen, indem man die geometrischen Objekte der Szene von „hinten nach vorne" (vom Standpunkt des Betrachters) rendert. Die Berechnung der *Prioritätsliste* erfolgt im Objektraum, die Lösung des ‚Hidden Surface'-Problems ergibt sich quasi als Nebeneffekt mit dem Durchlaufen der Rendering-Pipeline.

1.2.3 Rendering-Systeme

Dieser Abschnitt gibt kurz die wesentlichen Eigenschaften zweier wichtiger Rendering-Systeme (X11 und OpenGL) wieder, die im weiteren noch von Bedeutung sind.

X Window System

Das X Window System (vgl. [SG92], kurz X11 genannt) ist das unter UNIX üblicherweise verwendete Fenster- und Graphiksystem. Es stellt nur Funktionen zum 2D-Rendering bereit, d.h. es implementiert von der in Abschnitt 1.2.1 skizzierten (3D-)Renderingpipeline nur die Rasterung. Das Besondere an X11 ist seine Client-/Server-Architektur. Eine Applikation (Client), die über X11 rendern will, ruft dazu Funktionen aus der X11-Bibliothek (`Xlib`) auf, die daraufhin mit dem X-Server (ein separater Prozess, der auf einem Rechner mit der notwendigen Hardware läuft) über das X-Protokoll kommuniziert. Das Rendering wird dann vom X-Server

durchgeführt. Das X-Protokoll kann sowohl sowohl über lokale Kommunikationsmechanismen transportiert werden (wenn Client und Server auf demselben Rechner laufen) als auch über das Netz per TCP/IP (falls Client und Server auf verschiedenen Rechnern laufen). Diese Netzwerktransparenz macht X11 interessant im Kontext der parallelen Visualisierung, da Parallelrechner und eigener Arbeitsplatz üblicherweise räumlich auseinanderliegen. Via X11 kann das Programm auf dem Parallelrechner direkt auf der Workstation am Arbeitsplatz ein Fenster öffnen und in dieses rendern. Selbst wenn das Rendering nicht per X11 oder OpenGL (s.u.) stattfinden soll, lässt sich X11 immer noch per `XPutImage` zur bequem handhabbaren Anzeige von auf dem Parallelrechner gerenderten Bildern verwenden. Andere Lösungen benötigen dazu ein separates Programm und ein eigenes Kommunikationsprotokoll.

X11-Implementierungen sind außer für UNIX-Maschinen für praktisch alle anderen Rechner und Betriebssysteme (Windows, MacOS, ...) verfügbar.

OpenGL

OpenGL (vgl. [NDW93]) ist ein Cross-Platform-API (API: Application Programming Interface) zum 3D-Rendering. Applikationen rufen dazu Funktionen aus der OpenGL-Bibliothek auf, die sich in zustandsverändernde Kommandos und Geometrie-Kommandos unterteilen. Zustandsverändernde Kommandos legen Renderingparameter fest, z.B. Transformationsparameter, Zeichenfarbe, Blending an/aus, etc. Die Auswirkungen dieser Parameter werden in der *OpenGL State Machine* festgehalten. Geometriekommandos beschreiben die zu rendernde Geometrie.

OpenGL definiert nur Renderingfunktionen. Um es konkret benutzen zu können (d.h. eine „Leinwand“ zu erhalten, auf der das Rendering dann stattfinden kann), ist noch eine Ankopplung an das jeweilige Ausgabe- oder Fenstersystem notwendig. Im Falle von X11 handelt es sich dabei um GLX (siehe [OGL93]). GLX definiert nicht nur Funktionen, um einen OpenGL-Renderingkontext (z.B. durch Assoziation mit einem Fenster) festzulegen, GLX erweitert auch das X11-Protokoll um die Möglichkeit, OpenGL-Kommandos zu transportieren. Dadurch wird OpenGL ebenso

wie X11 netzwerktransparent.

Viele System- oder Graphikkartenhersteller bieten OpenGL-Treiber an, die Hardware-beschleunigtes OpenGL ermöglichen, indem sie einen Großteil der OpenGL-Funktionalität direkt in Hardware erledigen. Moderne Graphikchips (auch im Consumer-Bereich — bis ca. 500 €) implementieren die komplette Renderingpipeline in Hardware.

OpenGL-Implementierungen sind außer für diverse UNIX-Varianten auch für Windows und MacOS verfügbar.

1.3 Parallelrechner

1.3.1 Klassifikation nach Flynn

Flynns Klassifikation von Parallelrechnern [Fly72] basiert auf der Interaktion von Befehls- und Datenströmen.

SISD *Single Instruction Single Data*. Ein Befehlsstrom arbeitet einen Datenstrom ab. Entspricht dem klassischen von Neumann-Rechner.

SIMD *Single Instruction Multiple Data*. Mehrere identische Verarbeitungseinheiten führen denselben Befehlsstrom auf unterschiedlichen Daten aus. Beispiele: Connection Machine [Hil85], MasPar [MPC91]. Diese Klasse hat heute keine Bedeutung mehr.

MISD *Multiple Instruction Single Data*. Diese Klasse kommt in der Praxis nicht vor.

MIMD *Multiple Instruction Multiple Data*. Mehrere Verarbeitungseinheiten können unabhängige Befehlsströme auf unterschiedlichen Datenströmen ausführen. Alle aktuellen Supercomputer fallen in diese Kategorie.

SPMD *Same Program Multiple Data*. Keine eigene Klasse in der Klassifikation von Flynn, sondern ein Programmiermodell auf der Basis von MIMD. Jeder Prozessor führt dasselbe Programm auf unterschiedlichen Daten aus — *Datenparallelität*. Synchrone Ausführung der Programmbefehle wird nicht benötigt. Aus naheliegenden Gründen der populärste Ansatz für Rechner mit sehr vielen Prozessoren.

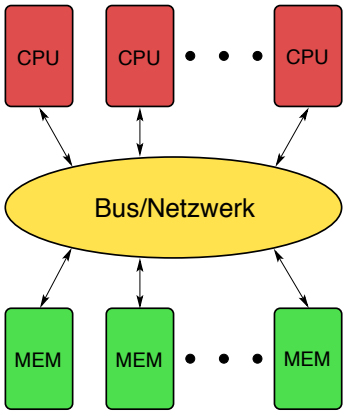

Abbildung 1.7: Schema Shared Memory Computer.

1.3.2 Shared Memory vs Distributed Memory

Rechner der Klasse MIMD lassen sich weiter nach der Koppelung zwischen Prozessoren und Speicher unterscheiden [HX98].

Shared Memory Computer

Shared Memory Computer (vgl. Abb. 1.7) zeichnen sich durch einen *globalen* Adressraum aus, auf den jeder Prozessor Zugriff hat. Kommunikation zwischen Prozessen findet über gemeinsam benutzte Variablen statt. Der Zugriff auf diese gemeinsamen Variablen wird durch spezielle Programmierkonstrukte geregelt, z.B. *Semaphore* (siehe etwa [Dij65] oder [Tan92]). Ein Semaphor ist ein opaker Datentyp, der einen Zähler enthält und über zwei atomare Operationen `P` (pausieren, down, wait) und `V` („vortfahren“, up, signal) manipuliert werden kann. Die `P`-Operation dekrementiert den Zähler

um eins. Falls er danach negativ ist, wird der ausführende Prozess auf eine mit dem Semaphor assoziierte Warteliste gesetzt und vom Betriebssystem in den Zustand „blocked“ überführt. Die V-Operation inkrementiert den Zähler um eins. Falls er danach nicht positiv ist, wird einer der dann auf der Warteliste des Semaphors stehenden Prozesse aus dieser entfernt und vom Betriebssystem wieder als „runnable“ markiert.

Gemeinhin wird die Meinung vertreten, dass Shared Memory Computer einfacher zu programmieren sind als die nachfolgend zu besprechenden Distributed Memory Computer.

Der Nachteil dieser Architektur ist die sehr hohe Anforderung an das Verbindungsnetzwerk, über das sämtliche Speicherzugriffe abgewickelt werden müssen. Eine Architekturvariante sind *Distributed Shared Memory Computer*, bei denen jedes Speichermodul einem Prozessor zugeordnet ist, auf das er lokal schnellen Zugriff hat. Lediglich der Zugriff auf den anderen Prozessoren zugeordneten Speicher geht über das Verbindungsnetzwerk. Der globale Adressraum bleibt erhalten, allerdings verändert sich die Charakteristik des Speicherzugriffs (Latenzzeit, Bandbreite) über den Adressraum — NUMA: Non Uniform Memory Access.

Beispiele für solche Maschinen sind die SGI Origin 2000 [LL97] oder die SGI Altix [ALX] mit jeweils bis zu 1024 Prozessoren.

Distributed Memory Computer

Distributed Memory Computer (vgl. Abb. 1.8) zeichnen sich dadurch aus, dass es *keinen* globalen Adressraum gibt. Einige wenige (typisch 1–16) CPUs bilden zusammen mit lokalem Speicher einen *Knoten*. Kommunikation zwischen Knoten ist nur durch expliziten Austausch von Nachrichten über das Verbindungsnetzwerk (SAN: System Area Network) möglich.

Zum De-Facto-Standard für die Programmierung von Distributed Memory Computern hat sich mittlerweile MPI (Message Passing Interface) [MPI94] entwickelt. Dabei rufen die auf den einzelnen Knoten laufenden Prozesse Funktionen der MPI-Bibliothek zum Versand und Empfang von Nachrich-

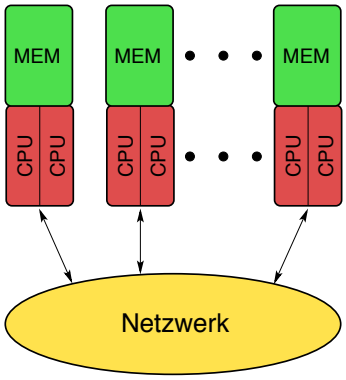

Abbildung 1.8: Schema Distributed Memory Computer.

ten auf. Diese Funktionen können entweder synchron sein, d.h. ein Prozess blockiert bis zur vollständigen Abarbeitung der Kommunikation, an der er beteiligt ist, oder asynchron, d.h. der Prozess läuft nach dem Absetzen des Kommunikationsauftrags weiter und muss sich über deren Verlauf selbst informieren.

Falls pro Knoten mehrere CPUs vorhanden sind, ist auch ein gemischtes Programmiermodell möglich, bei dem Prozesse, die auf dem gleichen Knoten laufen, die von Shared Memory Computern bekannten Kommunikationsmöglichkeiten nutzen.

Die überwiegende Zahl heutiger Supercomputer-Installationen fällt in die Klasse Distributed Memory Computer mit Systemen mit demnächst bis zu 65.536 Prozessoren (siehe [BGT02]). Im folgenden soll daher unter Parallelrechner, wenn nichts anderes gesagt ist, stets eine Maschine dieser Klasse verstanden werden.

1.3.3 Compute Cluster

Supercomputer der Mittelklasse werden schon seit einiger Zeit nach dem Commodity-Off-The-Shelf-Prinzip gebaut. Mann nimmt einige hundert bis einige tausend handelsübliche Rechner, meist relativ billige PCs, die dann über ein dediziertes Verbindungsnetzwerk miteinander verschaltet werden. Beim Netz kann man ebenfalls auf Standardprodukte, wie Fast-

oder Gigabit-Ethernet setzen, oder für etwas mehr Geld auf spezielle Lösungen, die mehr Bandbreite und geringere Latenzzeiten bieten, z.B. Myrinet (`www.myricom.com`, ca. 1100 $/Port) oder QsNet (`www.quadrics.com`, ca. 3200 $/Port). Die für den Betrieb eines Compute Clusters nötige Software (MPI-Implementierung, Batchsystem, Monitoring Tools, ...) ist frei verfügbar. Während man an der absoluten Spitze der Top 500-Liste der Supercomputer eher die klassischen teuren Komplettsysteme findet, dringen solche Compute Cluster regelmäßig bis in die Top 10 vor. Die Rechnungen für diese Arbeit wurden ebenfalls alle auf solchen Systemen gemacht, vgl. Kapitel 5.

1.3.4 Leistungsbewertung paralleler Algorithmen

Nachfolgend eine kurze Zusammenstellung der wichtigsten Maße zur Bewertung der Leistung von parallelen Algorithmen.

Speedup

Der erste Aspekt beim Einsatz von Parallelrechnern ist die Geschwindigkeitssteigerung (*Speedup*) $S(P)$, die man durch den Einsatz von P Prozessoren im Vergleich zur sequentiellen Variante ($P = 1$) auf demselben Rechner für ein Problem konstanter Größe erzielen kann. Dabei ist der Speedup als

$$S(P) := \frac{T(1)}{T(P)}$$

definiert. $T(P)$ ist die Laufzeit des parallelen Programms auf P Prozessoren, $T(1)$ wird manchmal als die Laufzeit des *schnellsten* sequentiellen Algorithmus definiert. Da dies schlicht unpraktisch ist, soll dies hier unterbleiben und $T(1)$ für die Laufzeit des zu bewertenden Algorithmus für $P = 1$ stehen. Da man normalerweise maximal $S(P) = P$ erwarten kann, definiert man die erzielte *Effizienz* $E(P)$ als

$$E(P) := \frac{T(1)}{P \cdot T(P)}.$$

Eine Effizienz von mehr als 100% kann in Ausnahmefällen erzielt werden, wenn etwa durch die lokal immer kleiner werdende Problemgröße die Caches der Prozessoren immer besser arbeiten. Generell fällt die Effizienz mit zunehmender Prozessorzahl immer weiter ab, da zum einen der Aufwand für die Kooperation zwischen den Prozessoren im Vergleich zur noch pro Prozessor zu leisteten Arbeit immer größer wird, zum anderen ist der Speedup auch durch die nicht-parallelisierbaren Anteile eines Programms limitiert (vgl. [Amd67], aber auch [Gus88] und [Shi96]).

Scaleup

Der zweite, meist wichtigere, Aspekt beim Einsatz von Parallelrechnern ist die Möglichkeit, mit ihnen große Probleme anzugehen, da mit jedem zusätzlichen Rechnerknoten nicht nur die Rechenkapazität sondern auch die Hauptspeicherkapazität (und im Idealfall auch die Kapazität aller anderen Ressourcen) wächst. Die Frage lautet also, ein um welchen Faktor größeres Problem lässt sich mit dem Übergang von einem auf P Prozessoren berechnen? Wenn $T(n, p)$ die Laufzeit des Programms auf p Prozessoren bei einer Problemgröße n bezeichnet und

$$T(N, 1) = T(M, P) \qquad (N < M)$$

für zwei Problemgrößen M und N ist, dann ist der erzielte *Scaleup* $SC(P)$ als

$$SC(P) := \frac{M}{N}$$

definiert. Von einem optimalen Algorithmus wird man wieder maximal $SC(P) = P$ erwarten können, d.h. mit P Prozessoren kann man maximal ein P-mal größeres Problem in der gleichen Zeit lösen. Als praktisch zugänglicheres Maß lässt sich unter Beachtung von

$$\frac{T(PN, 1)}{T(PN, P)} \approx \frac{P \cdot T(N, 1)}{T(PN, P)},$$

falls der Algorithmus in der Problemgröße linear ist, auch einen *Scaled Speedup* $S_s(P)$ durch

$$S_s(P) := \frac{P \cdot T(N, 1)}{T(PN, P)}$$

definieren. Wieder ist $S_s(P) \leq P$ zu erwarten. Effizienzen lassen sich entsprechend in beiden Fällen definieren.

Unter dem Aspekt des Scaleup lassen sich Parallelrechner mit vielen tausend Prozessoren effizient nutzen. Im Einklang mit der in der Motivation zu dieser Arbeit genannten Zielsetzung ist der Scaleup der zu betrachtende Hauptgesichtspunkt. Soweit nichts anderes gesagt ist, soll im weiteren unter *skalierbar* stets ein Algorithmus verstanden werden, dessen Scaleup (bzw. Scaled Speedup) im Bereich der betrachteten Prozessorzahlen in akzeptabler Nähe des Optimums liegt.

Kapitel 2

Parallele Visualisierung

2.1 Visualisierungspipeline und Parallelisierungsmöglichkeiten

Die Visualisierungspipeline (siehe Abb. 2.1) dient im Weiteren als Modell für die einzelnen Schritte im Visualisierungsprozess und erlaubt einen Überblick über die verschiedenen Parallelisierungsmöglichkeiten.

Am Anfang steht dabei eine parallele *Datenquelle*, die die zu visualisierenden Daten liefert: Der Parallelrechner, auf dem die Simulation ausgeführt wird bzw. der die Daten von einem parallelen Filesystem liest.

Unter *Filterung* wird die Gewinnung von *Extracts* verstanden. Extracts sind Felder, die auf einer Untermenge des Gebietes, auf dem der Datensatz

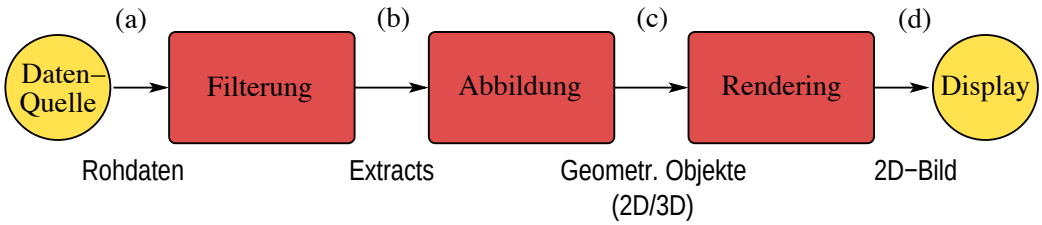

Abbildung 2.1: Visualisierungspipeline.

vorliegt, definiert sind. Näheres folgt in Abschnitt 2.1.2.

Der nächsten Schritt (*Abbildung*) bildet Extracts auf geometrische Objekte ab, aus denen in einem letzten Schritt (*Rendering*) ein Bild erzeugt und dieses dann angezeigt wird.

Die verschiedenen Parallelisierungsansätze ergeben sich nun daraus, an welcher Stelle in der Pipeline (in Abb. 2.1 mit (a)–(d) markiert) man den Parallelrechner verlässt und die noch verbleibenden Schritte sequentiell (oder moderat parallel) abarbeitet.

2.1.1 Traditioneller Ansatz

Der Parallelrechner wird sofort nach der Erzeugung der Daten verlassen. (Stelle (a) in Abb. 2.1). Die Daten werden in Files gespeichert und diese später auf eine Workstation oder einen speziellen Graphikcomputer transferiert. Dort findet die Visualisierung dann mit bekannten Programmen, wie z.B. AVS [UFK+89], IBM Data Explorer (DX) [AT95] oder Grape [NORS97] in einem Postprocessingschritt statt. Diese Programme arbeiten sequentiell oder bestenfalls moderat parallel auf Shared Memory-Computern. Zu DX gibt es allerdings eine erste noch experimentelle datenparallele Variante namens OpenDX-MPI (siehe [KKV+02] und [JKA+03]).

Das Problem mit diesem Ansatz wurde schon in der einführenden Motivation zu dieser Arbeit erwähnt: Mit aktuellen Supercomputern kann man problemlos Datensätze erzeugen, deren schiere Größe diese Vorgehensweise unpraktikabel bzw. unmöglich macht.

2.1.2 Parallele Extract-Erzeugung

Bei diesem Ansatz wird der Parallelrechner an der Stelle (b) in Abb. 2.1 nach der Erzeugung der Extracts verlassen.

Extracts sind (nach [Glo95]) Felder, die auf einem Teilbereich des Gebie-

tes, auf dem die Lösung berechnet wurde, definiert sind. Beispiele sind Schnittflächen, Isoflächen oder Bahnlinien mit angehefteten skalaren oder vektoriellen Feldgrößen.

Die Parallelisierung der Extract-Berechnung bietet sich aus zwei Gründen an. Zum einen benötigt dieser Schritt potentiell Zugriff auf das gesamte Lösungsgebiet, zum anderen sind die Algorithmen zur Berechnung der meisten Extract-Typen einfach zu parallelisieren. Beispielsweise arbeitet der bekannte ‚Marching Cubes'-Algorithmus zur Berechnung von Isoflächen [LC87] elementweise und erlaubt unmittelbar eine datenparallele Implementierung.

Extracts können deutlich kleiner sein als das ursprüngliche Lösungsgebiet, falls sie eine kleinere Dimension haben als dieses. Das ist z.B. der Fall für eine Schnittfläche durch ein reguläres 3D-Gitter (n Elemente pro Raumdimension). Während die Anzahl der Gitterelemente wie n^3 wächst, wächst die Anzahl der Polygone, aus denen der Schnitt besteht, nur wie n^2.

Durch die Dimensionsreduktion wird dieser Ansatz praktikabel. Die noch verbleibenden Schritte in der Visualisierungspipeline müssen nur noch auf einem viel kleineren Datenvolumen operieren und können daher auf einer Workstation sequentiell mit moderatem Speicherbedarf ausgeführt werden.

Zur Abbildung der Extracts auf geometrische Objekte und deren Rendering auf der Workstation benötigt man ein separates Programm, was die Möglichkeit bietet, lokal unterschiedliche Repräsentationen eines Extracts zu erzeugen und spezielle Graphik-Hardware auszunutzen. Dieses Programm kann entweder direkt mit der Extract-Erzeugung auf dem Parallelrechner verbunden sein (*On-Line-Visualisierung*) oder nachträglich verwendet werden, um abgespeicherte Extracts zu visualisieren, falls die Simulation im Batch-Betrieb läuft (*Off-Line-Visualisierung*).

Soweit die üblichen Argumente für den Extract-Ansatz, wie sie etwa auch in [MSLP02] propagiert werden. Kritisch ist aber vor allem die Frage, ob im allgemeinen Fall (unstrukturiertes lokal verfeinertes Gitter) das Argument der Dimensionsreduktion noch gilt. Betrachten wir dazu ein 2D-Modellproblem ("Fivespot") aus dem Bereich der porösen Medien am An-

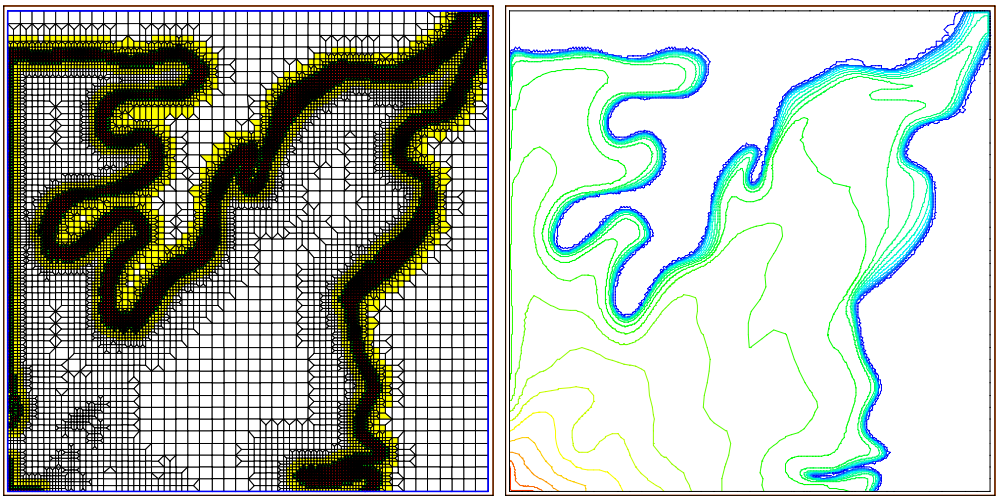

Abbildung 2.2: Modellproblem *Fivespot: adaptiertes Gitter (links) und Lösung (rechts) zu einem festen Zeitpunkt.*

wendungsbeispiel Erdölförderung. Details über die beschreibenden Gleichungen, Aspekte der Modellierung und Numerik, etc. finden sich z.B. in [Bas99]. Die konkrete Simulation, auf die hier Bezug genommen wird, entstammt [Lan01].

Aus einem Reservoir (modelliert als Quadrat, vgl. Abb. 2.2) wird oben rechts Erdöl abgepumpt und zusätzlich zur Erhöhung der Ausbeute unten links Wasser eingepumpt. Da sich die beiden Phasen nicht mischen, verdrängt das eingepumpte Wasser das Erdöl zunehmends. Zwischen den beiden Phasen bildet sich eine scharfe, von links unten nach rechts oben wandernde Front (äußerste blaue Isolinie in Abb. 2.2, rechtes Bild), deren genauer Verlauf durch die Permeabilitätsverteilung im Modellgebiet bestimmt ist und die durch Gitteradaption dynamisch aufgelöst wird (Abb. 2.2, linkes Bild, roter Bereich). Die Geometrie dieser Front (repräsentiert durch die äußerste blaue Isolinie) ist das zu diskutierende Extract.

Wenn man für die Länge der adaptierten Phasengrenze l Elemente und für ihre Dicke d Elemente ansetzt, so schneidet die Isolinie, die die Phasengrenze markiert, l Elemente. Verfeinert man das Gitter adaptiv entlang der Grenze einmal mehr, so bleibt d etwa konstant, während sich l verdoppelt. Entsprechend besteht die Isolinie dann aus etwa doppelt so vielen

Segmenten. In diesem Beispiel ist also die Extract-Berechnung *nicht* mit einer Dimensionsreduktion verbunden, das Extract wächst ungefähr in dem gleichen Maße wie das Gitter, da hier die Dimensionsreduktion schon mit der lokalen Gitteradaption vorweggenommen wird. Allerdings besteht der Sinn von lokaler Gitteradaption ja gerade in Möglichkeit, lokale Phänomene aufzulösen, die Rechenarbeit auf diese zu konzentrieren, um so große Probleme behandeln zu können.

Schließlich sei noch erwähnt, dass die über die einzelnen Knoten des Parallelrechners verteilten Extract-Stücke noch eingesammelt und kombiniert werden müssen, bevor sie an die Workstation verschickt werden können. Hier verbirgt sich ein potentieller Flaschenhals.

2.1.3 Parallele Erzeugung geometrischer Objekte

Bei diesem Ansatz wird der Parallelrechner an der Stelle (c) in Abb. 2.1 nach der Erzeugung geometrischer Objekte verlassen. Zusätzlich zur Extractgenerierung wird auch noch die Abbildung auf geometrische Objekte parallel ausgeführt. Der Abbildungsschritt ist relativ trivial. Extracts und geometrische Objekte teilen dieselben Koordinateninformationen. Während Extract-Knoten aber mit skalaren oder vektoriellen Feldgrößen verknüpft sind, tragen die Knoten von geometrischen Objekten die für das Rendering nötigen Informationen, z.B. Farbe, Normalenvektoren, Texturkoordinaten, usw.

Entsprechend ähneln auch die Eigenschaften dieses Ansatzes denen des Extract-basierten, insbesondere gilt auch hier das Argument der Dimensionsreduktion — oder auch nicht.

Wichtigste Unterschiede sind zum einen, dass die Notwendigkeit eines separaten Programms zur Ausgabe entfällt; man kann das Rendering der geometrischen Objekte direkt vom Parallelrechner durch Aufruf von X11 oder OpenGL, vgl. Abschnitt 1.2.3, Seite 10, erledigen. Die Ausnutzung von spezieller Graphik-Hardware ist also auch hier möglich. Zum anderen büßt man etwas an Flexibilität ein, da die Möglichkeit, Extracts lokal (auf der Workstation) weiter manipulieren zu können oder unterschiedliche

Darstellungen zu rendern, entfällt.

Erhalten bleibt die Möglichkeit zur Off-Line-Visualisierung, indem man die geometrischen Objekte in einem Metafileformat abspeichert und das Rendering zu einem späteren Zeitpunkt durchführt.

2.1.4 Parallele Bilderzeugung

Bei diesem Ansatz wird der Parallelrechner erst an der Stelle (d) in Abb. 2.1 nach der Erzeugung des Bildes verlassen, also der gesamte Visualisierungsprozess parallel abgearbeitet, insbesondere wird also das Rendering parallel ausgeführt. Die prinzipielle Vorgehensweise ist dabei, dass jeder Knoten des Parallelrechners ein Teilbild erstellt, aus denen dann schließlich das ganze Bild entsteht. Sowohl für das Rendern der Teilbilder als auch für das Zusammenfügen der Teilbilder stehen mehrere Methoden zur Verfügung deren genaue Besprechung das Thema von Kapitel 3 ist.

Die Parallelisierung des gesamten Visualisierungsprozesses bietet die besten Chancen auf gute Skalierbarkeit — auch in den Fällen, in denen das für die ersten beiden Ansätze zentrale Argument der Dimensionsreduktion nicht anwendbar ist. Ein Beispiel dazu wurde im Abschnitt 2.1.2 angegeben. Ein weiteres wäre Volume Rendering, da hier stets der gesamte Datensatz bearbeitet werden muss. Zwar muss man in der Regel auf Hardware-unterstützte Graphik verzichten (derartige Hardware ist in Supercomputern üblicherweise nicht vorhanden), andererseits sind aktuelle CPUs schnell genug, um auch per Software-Rendering beachtliche Performance zu liefern.

Auch hier ist zur Darstellung des Bildes nicht zwingend ein separates Programm nötig, da man im Falle der On-Line-Visualisierung zur Ausgabe wieder auf X11 (Stichwort `XPutImage`, vgl. Abschnitt 1.2.3, Seite 10) zurückgreifen kann. Im Off-Line-Fall kann man die erzeugten Bilder abspeichern und später einen beliebigen Image-Viewer verwenden.

2.2 Alternative Parallelisierungsansätze

Bisher wurde der Visualisierungsprozess unter dem Gesichtspunkt der Datenparallelität betrachtet, da dieser Ansatz mit dem SPMD-Modell einhergeht und auch als einziger Aussicht auf gute Skalierbarkeit gibt. Alternative Parallelisierungsansätze seien im Folgenden der Vollständigkeit halber auch erwähnt.

2.2.1 Aufgabenparallelität

Aufgabenparallelität ergibt sich z.B., wenn in einem Bild mehrere Visualisierungen eines Datensatz, der mehrere zusammengehörige Felder enthält (z.B. eine Konzentration, ein Temperatur- und ein Geschwindigkeitsfeld), dargestellt werden sollen. Statt die Visualisierungspipeline mehrmals sequentiell zu durchlaufen, kann man sie auch für die einzelnen Visualisierungsaufgaben parallel abarbeiten. Das Rendering erfolgt dabei entweder zentral oder ebenfalls verteilt, wobei im letzteren Fall die Techniken aus Kapitel 3 zum Einsatz kommen können.

Da die Anzahl der Visualisierungsaufgaben pro Datensatz beschränkt ist, ist von diesem Ansatz keine Skalierbarkeit zu erwarten. Zum Anderen trägt er auch nichts zur Handhabung großer Datensätze bei. Ein illustrierendes Beispiel, was mit diesem Ansatz zu erreichen ist, gibt Tab. 2.1. Hier werden fünf Visualisierungsaufgaben für ein Bild bei zentralem Rendering einmal sequentiell und einmal mit fünf Prozessoren bearbeitet.

Falls genügend Ressourcen vorhanden sind, lässt sich dieser Ansatz allerdings zusätzlich zur Datenparallelität verwenden, um so weitere Performancegewinne zu erzielen

1 Prozessor	5 Prozessoren	Speedup
23.0 s	6.8 s	3.4

Tabelle 2.1: Beispielperformance für Aufgabenparallelität aus [ALS+00].

2.2.2 Pipelineparallelität

Pipelineparallelität ist möglich, wenn für die Abarbeitung der einzelnen Stufen der Visualisierungspipeline unabhängige Ressourcen vorhanden sind, z.B. mehrere Prozessoren (pro Rechnerknoten) und/oder Graphik-Hardware. Pipelineparallelität lässt sich z.B. für einen zeitabhängigen Datensatz ausnutzen: Während eine Pipelinestufe den Datensatz zum Zeitpunkt t_n bearbeitet, läuft durch die vorige Stufe schon der Datensatz für den nächsten Zeitpunkt t_{n+1}. Skalierbarkeit kann man maximal bis zur Anzahl der Pipelinestufen erwarten, allerdings nur wenn die einzelnen Stufen gleich schnell durchlaufen werden. In Tab. 2.2 sind einige illustrierende Zahlen für ein Beispiel mit einer drei-stufigen Pipeline angegeben.

Wie im vorigen Abschnitt trägt auch dieser Ansatz nichts zur Visualisierung großer Datensätze bei, kann aber bei Vorhandensein geeigneter Hardware zusätzlich zur Datenparallelität angewandt werden.

1 Prozessor	3 Prozessoren	Speedup
271 s	194 s	1.4

Tabelle 2.2: Beispielperformance für Pipelineparallelität aus [ALS+00].

Kapitel 3

Paralleles Rendering

Die ersten beiden Schritte der Visualisierungspipeline (Filterung und Abbildung) sind, wie schon in Kapitel 2 angesprochen, in den allermeisten Fällen trivial parallelisierbar. Es bleibt die Parallelisierung des Renderings, die den Gegenstand dieses Kapitels bildet. Vorgestellt wird ein Klassifikationsschema des parallelen Renderings, eine Diskussion der verschiedenen Klassen, sowie bereits bestehende Implementierungungen mit und ohne speziellem Hardwaresupport als Beispiele.

3.1 Klassifikation Parallelen Renderings

Die folgende Klassifizierung des parallelen Renderings folgt dem Klassiker [MCEF94]. Ausgangspunkt ist das in Abb. 3.1 dargestellte Modell eines parallelen Renderingsystems, in dem sowohl für die Geometrieverarbeitung als auch für die Rasterung (vgl. Abschnitt 1.2.1, Seite 7) mehrere Prozessoren zur Verfügung stehen. Die Parallelisierung der Geometrieverarbeitung ergibt sich aus der Verteilung der geometrischen Objekte auf die vorhandenen Geometrieprozessoren (G), die des Rasterungsprozesses, indem man jedem Rasterungsprozessor (R) einen Teil der anfallenden Pixelberechnungen zuteilt.

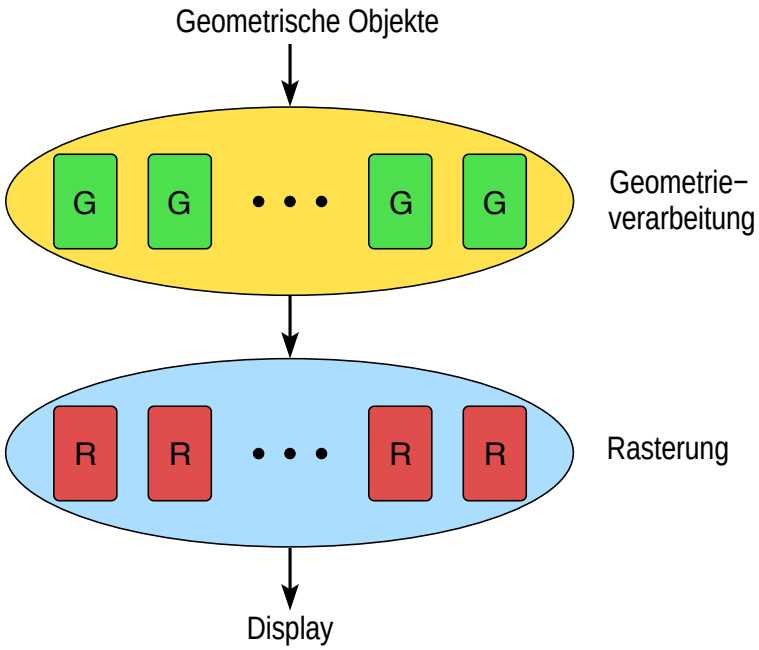

Abbildung 3.1: Modell eines parallelen Renderingsystems.

Nach [SSS74] kann man (Polygon-)Rendering (inkl. Lösung des ‚Hidden Surface'-Problems) als Sortiervorgang betrachten: Beim Rendering ist der Effekt jedes geometrischen Objekts auf jedes Pixel zu bestimmen. Abhängig von den Projektionsparametern kann aber jedes geometrische Objekt jedes beliebige Pixel beeinflussen — oder auch nicht. Die Aufgabe lässt sich also als „Einsortieren der geometrischen Objekte in den Bildraum" interpretieren. In einem parallelen Renderingsystem ist die Zuständigkeit für geometrische Objekte bzw. Pixel über verschiedene Prozessoren verteilt, sodass dieser Sortiervorgang zusätzlich eine Umverteilung von Daten zwischen den Prozessoren erfordert. Entsprechend Abb. 3.1 kommen dazu drei Möglichkeiten in Frage:

1. Vor der Geometrieverarbeitung (*Sort-First*). Es werden geometrische Objekte verteilt.
2. Zwischen Geometrieverarbeitung und Rasterung (*Sort-Middle*). Es werden Bildraumobjekte verteilt.
3. Nach der Rasterung (*Sort-Last*). Es werden Pixel oder Samples verteilt.

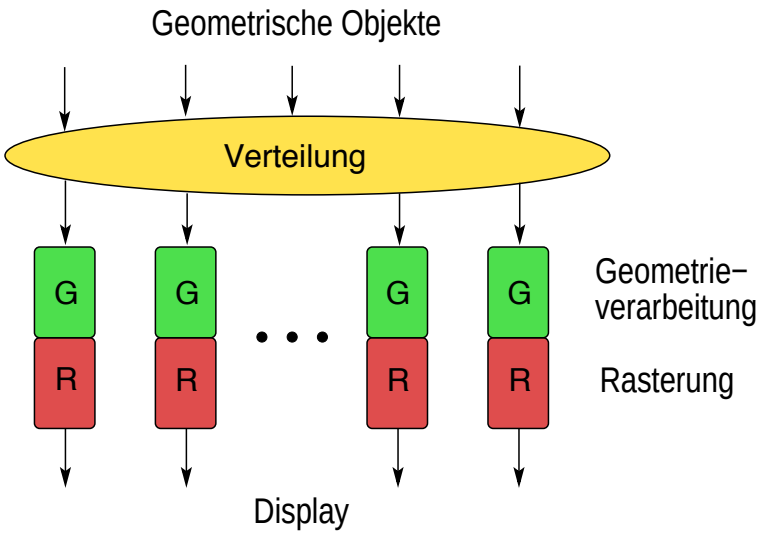

Abbildung 3.2: Sort-First-Rendering.

Nachfolgend sind diese drei Varianten näher beschrieben.

3.1.1 Sort-First-Rendering

Bei Sort-First (vgl. Abb. 3.2) werden die geometrischen Objekte ganz zu Beginn auf die vorhandenen Renderer (Paare aus Geometrieprozessor (G) und Rasterungsprozessor (R)) verteilt. Dafür wird eine (prinzipiell beliebige) Partitionierung des Bildraumes (des „Bildschirms“) verwendet. Jeder Renderer führt dann den gesamten Renderingprozess für seine Partition durch.

Zur Ermittelung der Zielpartitionen für die geometrischen Objekte ist eine *Vortransformation* derselben notwendig. Da ein Objekt in mehrere Partitionen fallen kann, muss es eventuell mehrfach verschickt und auch gerendert werden. Dieser Overhead ist offensichtlich umso kleiner, je größer die Bildraumpartitionen im Vergleich zu den sich aus den geometrischen Objekten ergebenden Bildraumobjekten sind. Der Kommunikationsaufwand selbst ist im wesentlichen proportional zur Anzahl der geometrischen Objekte.

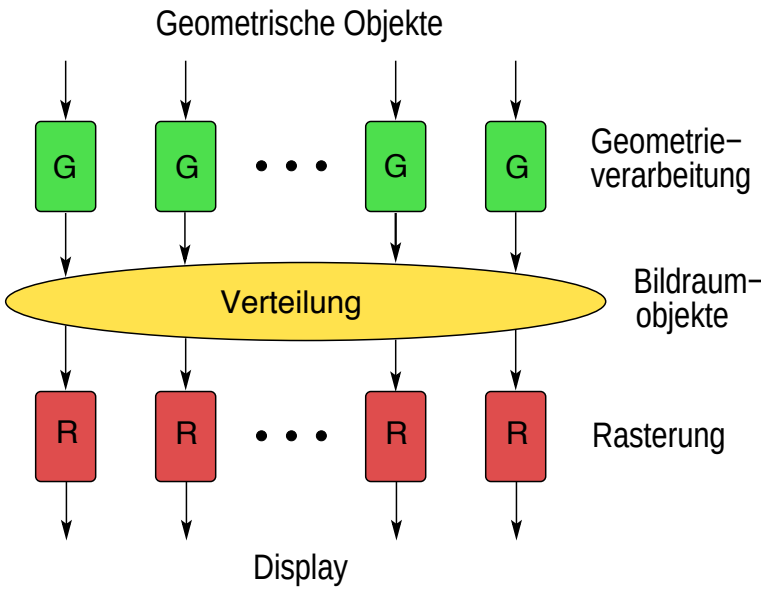

Abbildung 3.3: Sort-Middle-Rendering.

Falls sich die geometrischen Objekte nicht gleichmäßig auf die Partitionen verteilen, entsteht das Problem des Lastungleichgewichts unter den Renderern. Für reine Softwarelösungen nach dem Sort-First-Prinzip gibt es die Option, die Partitionierung dynamisch anzupassen, z.B. [Mue95]. Seine wesentliche Bedeutung hat das Sort-First-Prinzip aber heute im Zusammenhang mit *Tiled Displays*, großen Displays, die aus vielen kleinen aufgebaut sind, von denen jedes von einem Renderer angesteuert wird. Siehe Abschnitt 3.4.1.

3.1.2 Sort-Middle-Rendering

Sort-Middle (vgl. Abb. 3.3) verteilt Bildraumobjekte wie sie nach der Geometrieverarbeitung entstehen. Für die Zuordnung der Bildraumobjekte zu den Rasterungsprozessoren benötigt man wieder eine Partitionierung des Bildraums wie bei Sort-First. Allerdings erfolgt jetzt die Umverteilung an der naheliegendsten Stelle. Eine Vortransformation wie bei Sort-First ist nicht notwendig.

Da Bildraumobjekte wieder in mehrere Partitionen fallen können, ergibt

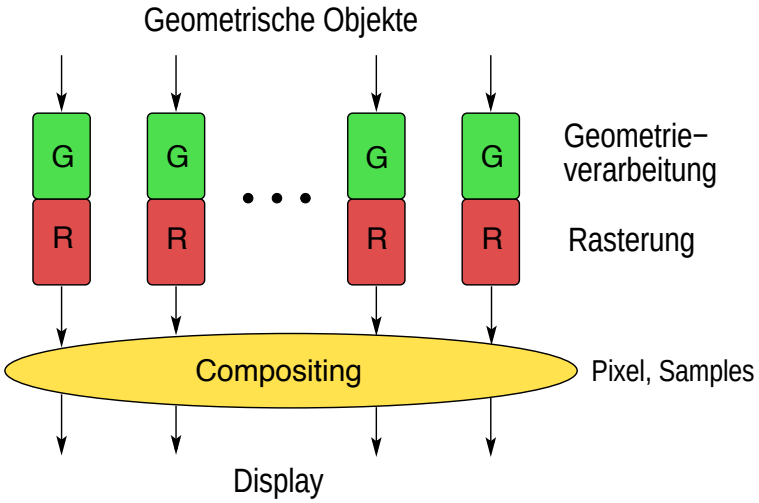

Abbildung 3.4: Sort-Last-Rendering.

sich u.U. wieder ein Overhead durch mehrfaches Verschicken an die Rasterprozessoren. Software-Implementierungen zerlegen daher meist die Bildraumobjekte in geeigneter Form, bevor sie verschickt werden, siehe Abschnitt 3.4.2. Der Kommunikationsaufwand erhöht sich entsprechend der Anzahl der Teilstücke. Falls Lastungleichgewichte unter den Rasterprozessoren auftreten, bietet sich wie bei Sort-First die Möglichkeit der dynamischen Anpassung der Partitionierung. Hardware-Implementierungen verwenden oft ein schnelles Broadcast-fähiges ‚On Chip'-Netzwerk, um die geometrischen Objekte an alle Rasterer zu verschicken. Sind n Rasterprozessoren vorhanden, dann rastert jeder nur jede n-te Spalte. Siehe z.B. [Ake93] und [MBDM97].

3.1.3 Sort-Last-Rendering

Bei Sort-Last (vgl. Abb. 3.4) wird der Sortiervorgang erst ganz zum Schluss nach der Rasterung durchgeführt. Jeder Renderer bearbeitet einen (im Idealfall beliebigen — bei Compositing über z, siehe Abschnitt 3.2) Teil der geometrischen Primitive unabhängig davon, wo sie im Bildraum zu liegen kommen. Danach werden die von den einzelnen Renderern erzeugten Teilbilder in einem *Compositing* genannten Schritt zu einem Gesamtbild kombiniert.

Das Lastverteilungsproblem ist im wesentlichen gelöst, wenn alle Renderer gleich viele geometrische Objekte zu verarbeiten haben. Lediglich in der Rasterungsphase können noch Ungleichgewichte auftreten, da der Aufwand für die Rasterung mit der Größe der Bildraumobjekte zunimmt.

Der Kommunikationsaufwand für das Compositing hängt nur von der für den Bildraum gewählten Auflösung ab, hingegen nicht von der Anzahl der zu rendernden geometrischen Objekte, was Sort-Last attraktiv für die Visualisierung sehr großer Datensätze bei moderater Bildgröße macht. Weiterhin lässt sich das Compositing in Software als Reduktionsoperation implementieren, während die Kommunikation bei Sort-First und Sort-Middle vom Typ $m : n$ ist. Ganz generell fügt sich der Sort-Last-Ansatz am glattesten in nach dem SPMD-Konzept geschriebene Programme ein. Für reine und Hardware-unterstützte Software-Implementierungen siehe die Abschnitte 3.4.3 und 3.4.4. Komplette Hardware-Implementierungen von Sort-Last wurden ebenfalls versucht, siehe z.B. [MEP92] und [EMP+97].

3.2 Compositing

Compositing bezeichnet das Zusammenfügen mehrerer unabhängig berechneter Bilder A, B, C, ... zu einem neuen Bild $A \oplus B \oplus C \oplus \cdots$, das sich aus der pixelweisen Verknüpfung der Ausgangsbilder ergibt. Dazu werden außer den Farbkomponenten (r, g, b) der Pixel noch weitere Komponenten benötigt. Hier werden die Fälle einer zusätzlichen Tiefenkomponente z bzw. eines α-Kanals betrachtet.

Für den Fall einer Tiefenkomponente z (z-Achse weist vom Betrachter weg) und zwei Pixel $p_1 = (r_1, g_1, b_1, z_1)$ und $p_2 = (r_2, g_2, b_2, z_2)$ ist der Compositing-Operator $\oplus$ so definiert:

$$p_1 \oplus p_2 := \begin{cases} p_1 & \text{falls } z_1 \leq z_2; \\ p_2 & \text{falls } z_1 > z_2. \end{cases}$$

Diese Verknüpfung $\oplus$ (vgl. Abb. 3.5) ist kommutativ (mit Ausnahme des

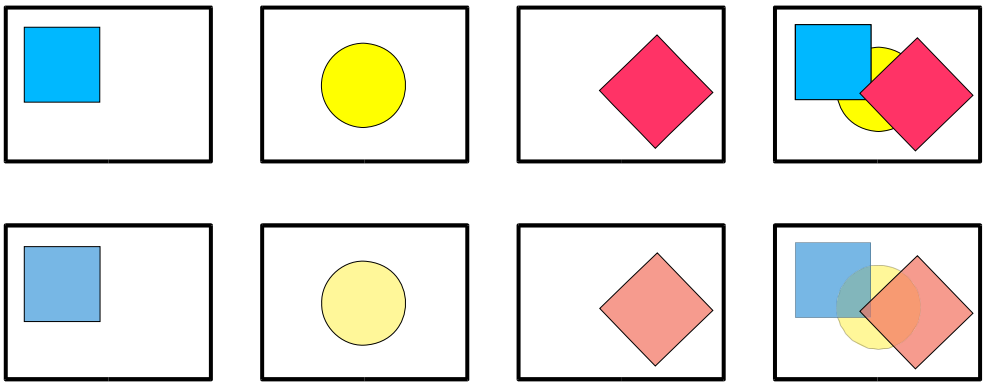

Abbildung 3.5: Compositing: Bilder oben über z, Bilder unten über α.

Falles $z_1 = z_2$) und assoziativ. Die Berechnung von $A \oplus B \oplus C \oplus \cdots$ lässt sich also in jeder beliebigen Reihenfolge vornehmen; das stellt den oben für Sort-Last angegebenen Idealfall dar, bei dem die Lastverteilung beliebig ist. Für konkrete Implementierungsmöglichkeiten des Compositings über z siehe Abschnitt 3.5

Für den Fall eines α-Kanals ($0 \leq \alpha \leq 1$ gibt die Opazität eines Pixels an; für $\alpha = 0$ ist das Pixel durchsichtig, für $\alpha = 1$ opak) kann man zunächst folgende Überlegung anstellen. Ein Pixel $p_1 = (r_1, g_1, b_1)$ mit α-Wert soll über ein zweites Pixel $p_2 = (r_2, b_2, g_2)$ gezeichnet werden, wobei der Compositing-Operator jetzt mit over bezeichnet wird:

$$p_1 \text{ over } p_2 = \alpha p_1 + (1 - \alpha) p_2.$$

Dieser Definitions-Versuch, der dem in OpenGL [NDW93] meistens verwendeten Blending entspricht, berücksichtigt allerdings nicht, dass auch mit p_2 eine Opazität assoziiert ist und auch für das Zielpixel eine solche definiert werden muss. Der over-Operator nach Porter und Duff (siehe [PD84] und [Bli94]), der all dies berücksichtigt und den obigen Versuch als Speziallfall enthält, lässt sich am einfachsten durch Farbkomponenten mit *premultiplied alpha* formulieren:

$$p = (r, g, b, \alpha) \quad \rightarrow \quad p' = (r', g', b', \alpha) \quad \text{mit } c' = \alpha c,$$

wobei c für die Farbkomponenten r, g und b steht. Der over-Operator ist dann für $p'_i = (r'_i, g'_i, b'_i, \alpha_i)$ als

$$p'_1 \text{ over } p'_2 := \alpha_1 p'_1 + (1 - \alpha_1) p'_2$$

definiert. Dieser over-Operator (vgl. Abb. 3.5) ist *nicht* kommutativ aber assoziativ, d.h. bei einer Berechnung von A over B over C over $\cdots$ ist die vorgegebene Reihenfolge einzuhalten, die Klammerung jedoch beliebig. Mehr noch: Das gesamte Rendering ist gemäß einer Prioritätsliste für *alle* geometrischen Objekte durchzuführen, da normalerweise auch die Teilbilder durch Pixeleintrag mit dem over-Operator entstehen. Eine gültige Lastverteilung für Sort-Last ordnet etwa, wenn n Renderer vorhanden sind, dem ersten das erste n-tel dieser Liste zu, dem zweiten das zweite n-tel, usw. Das Compositing muss dann ebenfalls in dieser Reihenfolge stattfinden.

3.3 Bildausgabe

Wenn man entweder den Parallelrechner selbst oder einen an diesen angekoppelten Spezial-Rechner zur Visualisierung benutzt, stellt sich das Problem der Bildausgabe, da der eigene Arbeitsplatz sich üblicherweise an einem anderen Ort befindet. Während die gemeinhin verfügbare Netzbandbreite sicher besonderen Anforderungen wie extreme Bildauflösung oder Interaktion bei hohen Bildraten nicht genügt, lassen sich moderatere Anforderungen an Bildauflösung und Bildrate erfüllen. Die schon in Kapitel 1 und 2 angesprochene Netzwerktransparenz von X11 und OpenGL ist z.B. ein naheliegender Lösungsansatz ebenso wie die Verwendung eigener Protokolle zur Bildübertragung. Einige Beispiele für letzteres sind in den Abschnitten 3.4.2 und 3.4.3 angeführt.

3.4 Beispiele für parallele Renderingsysteme

3.4.1 WireGL

WireGL (siehe [HBEH00] und [HEB+01]) ist ein System zur Ansteuerung von *Tiled Display Walls*, bei denen mehrere exakt ausgerichtete Beamer durch Rückprojektion gemeinsam ein großes hochauflösendes Bild auf eine Leinwand werfen, vgl. [SFF+00]. Jeder Beamer wird von einem eigenen Rechner, einem *Rendering-Server*, über eine Graphikkarte angesteuert. Untereinander und mit dem Frontend-Rechner, auf dem die (sequentielle) Applikation läuft, sind sie über ein Netzwerk gekoppelt, vgl. Abb. 3.6. Alternativ lässt sich eine Tiled Display Wall auch aus LCD-Monitoren mit möglichst dünnem Rahmen aufbauen.

WireGL in der hier beschriebenen Version befasst sich nur mit der Skalierbarkeit der Displayauflösung. Zur Skalierbarkeit der Datensatzgröße trägt es nichts bei. Es illustriert aber den wesentlichen Einsatzbereich des Sort-First-Ansatzes.

Architektur

WireGL implementiert das OpenGL-API. Unveränderte OpenGL-Applikationen, die nicht im Quelltext vorliegen müssen, linken dynamisch zur Laufzeit den WireGL-Ersatz für die `libGL.so` des Systems. Diese Ersatzbibliothek verteilt die OpenGL-Kommandos der Applikation nach dem Sort-First-Prinzip an die Rendering-Server, die dann ihren Anteil über die OpenGL-Bibliothek des Systems ausführen.

Implementierung

Die WireGL-Bibliothek wertet die von der Applikation gesendeten OpenGL-Kommandos getrennt nach Geometrie- und zustandsverändernden Kommandos aus. Geometriekommandos werden in einem Geometriepuffer zwi-

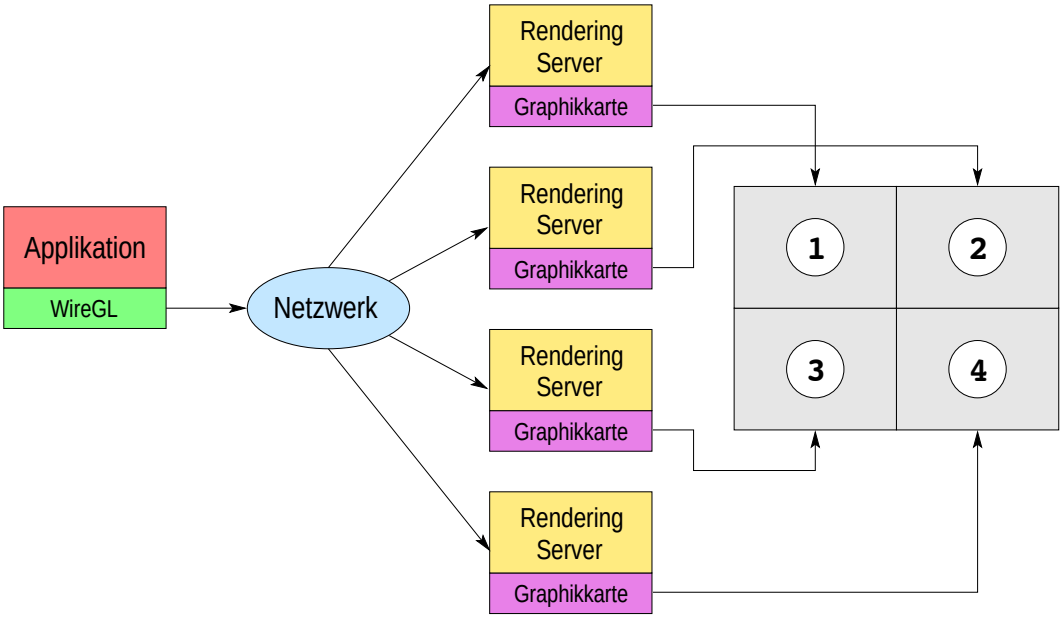

Abbildung 3.6: Schematische Darstellung des WireGL-Systems: OpenGL-Kommandos der Applikation werden via Sort-First an Rendering-Server zur Ansteuerung einer Tiled Display Wall verteilt.

schengepuffert und die Bounding Box (im Objektraum) der im Puffer befindlichen Kommandos aktualisiert. Wenn der Geometriepuffer geleert werden muss, wird die Bounding Box in den Bildraum transformiert und der Puffer in die Ausgangsqueue aller Rendering-Server kopiert, deren Ausgabebereich mit der transformierten Bounding Box überlappt. Dabei wird auf den räumlichen Zusammenhang aufeinanderfolgender geometrischer Primitive vertraut, um den Puffer nicht an zu viele Rendering-Server schicken zu müssen. Jedes einzelne geometrische Objekt separat zu verfolgen wäre jedoch zu aufwendig.

Zustandsverändernde Kommandos (z.B. z-Buffering an/aus, Blendingmodus oder Beleuchtungsparameter ändern) können nicht wie Geometriekommandos behandelt werden, da sie keine räumliche Ausdehnung haben und außerdem potentiell jeden Rendering-Server betreffen. WireGL löst dieses Problem, indem es sowohl den Zustand der Applikation als auch den aller Rendering-Server mitverfolgt. Zustandsverändernden Kommandos ändern dabei nur eine von der WireGL-Bibliothek geführte Datenstruktur. Der

Geometriepuffer muss daraufhin geleert werden, da die Zustandsänderung nachfolgende Geometriekommandos betrifft und nicht die bereits gepufferten. Dazu wird die Zustandsdifferenz zwischen der Applikation und den Rendering-Servern, die diesen Puffer zugestellt bekommen, ermittelt, der mitgeführte Zustand dieser Rendering-Server in der Datenstruktur aktualisiert und nur diese Differenz zusammen mit den Geometriekommandos in die Ausgangsqueue der betroffenen Rendering-Server gestellt. Zur optimierten Berechnung von OpenGL-Zustandsdifferenzen siehe [BHH00].

Performance

Es ist seit geraumer Zeit bekannt, siehe z.B. [ISH98], dass eine einzelne CPU Schwierigkeiten hat, moderne Graphikhardware mit deren voller Geschwindigkeit zu betreiben. Das gilt auch im Falle des *Direct Rendering* (siehe [KBH95]), bei dem die Applikation direkt auf der Maschine mit der Graphik-Hardware läuft, und diese (mit etwas Kernelhilfe und Kooperation mit dem X Server) direkt aus dem User Space angesteuert wird. Im Falle des *Remote Rendering*, das WireGL betreibt, ist dieses Problem noch deutlich verstärkt zu erwarten. In [HH99] sind z.B. für einen Benchmark auf einem (nicht mehr aktuellen) SGI InfiniteReality System bei Direct Rendering 870.000 Dreiecke/s und bei Indirect Rendering über GLX [OGL93] 170.000 (lokal) bzw. 110.000 Dreiecke/s (remote, 100 MBit Ethernet) angegeben. Das WireGL-Protokoll ist nach [HBEH00] effizienter als das traditionelle GLX, nach [Pau04] aber nicht sehr. Das Updaten der Bounding Box macht nach [HBEH00] einen erstaunlich hohen Anteil der in der WireGL-Bibliothek beim Verarbeiten von Geometrie-Kommandos benötigten Zeit aus.

Für das in [HBEH00] verwendete Beispielsystem (32 Rendering-Server mit je zwei Pentium III, 800 MHz, Myrinet als Verbinungsnetzwerk, GeForce2 Graphikkarten) ist die Renderingleistung durch das Netzwerk limitiert. Trotzdem skaliert das System sehr gut bis 32 Tiles. Die relative Framerate verglichen mit nur einem Tile liegt bei 50–90%, je nach Anwendung.

Folgende Optimierungsmöglichkeiten zur Verbesserung der Renderingleistung des Systems drängen sich auf:

- Verwendung eines schnelleren Netzwerks.
- Parallelisierung der Applikation, um mehr OpenGL-Kommandos pro Zeiteinheit zu erzeugen und damit die Rendering-Server besser auszulasten.

Ersteres ist eine Frage des Geldes und der verfügbaren Technologie. Letzteres erfordert eine parallele Erweiterung des OpenGL-APIs, wie sie in [ISH98] vorgeschlagen wurde, und die im WireGL-Nachfolger Chromium (siehe Abschnitt 3.4.4) implementiert ist. Durch die Parallelisierung der Applikation wird dann auch wieder ein Beitrag zur Skalierbarkeit in der Datengröße geleistet.

3.4.2 PGL

PGL (siehe [Cro97], [Cro94] und [CO93]) ist eine parallele Graphikbibliothek zum Rendern von Polygonen auf Distributed Memory Computern mit Message Passing-Interface. PGL ist eine reine Softwarelösung, die ohne Grafikhardware auskommt. Ziel ist sowohl Skalierbarkeit in der Größe des Datensatzes als auch in der Auflösung der Bildausgabe. Gerenderte Bilder können entweder im Dateisystem des Parallelrechners gespeichert oder zur On-Line Visualisierung an eine Workstation geschickt werden.

Architektur

PGL implementiert ein eigenes API, das eine Mischung aus Immediate Mode und Retained Mode darstellt. Die Idee ist dabei, größere Agglomerate zu definieren, die dann in einem Durchgang gerendert werden. Dies ist wünschenswert, da PGL nach dem Sort-Middle-Prinzip arbeitet und alle Prozessoren sowohl an der Geometrieverarbeitung als auch an der Rasterung beteiligt sind. Das Rendern großer Agglomerate in einem Arbeitsgang erlaubt es, die nötige Kommunikation asynchron im Hintergrund auszuführen. und so größtenteils zu verstecken. Zur Bildausgabe bei On-Line Visualisierung ist ein spezielles Programm auf der Workstation nötig.

Implementierung

PGL verschränkt den Bildraum zeilenweise über die einzelnen Prozessoren. Damit wird nicht nur die für Sort-Middle benötigte Partitionierung des Bildraumes geschaffen, sondern auch die Skalierbarkeit der Auflösung erreicht. Das gilt vor allem für die Maschinen, für die PGL ursprünglich entwickelt wurde, wie z.B. die Intel Paragon, die für heutige Verhältnisse relativ wenig Hauptspeicher pro Knoten hatten. Auch auf aktuellen Maschinen würde man den Speicher für einen kompletten Frame Buffer in Megapixelauflösung nicht auf jedem Prozessor für die Graphik opfern wollen.

PGL implementiert nicht exakt den Sort-Middle-Ansatz aus Abschnitt 3.1.2, sondern führt den ersten Teil der Rasterung, die Berechnung der *Spans*, d.h. der Schnitte eines projizierten Polygons mit den Bildzeilen, noch vor der Kommunikation aus. Jeder Span besteht aus den Koordinaten der Endpunkte zuzüglich Farb- und z-Werte. Der Vorteil der Versendung von Spans ist, dass man keine zusätzlichen Berechnungen vornehmen muss, die in der Renderingpipeline nicht sowieso anfallen. Andere Lösungen, die entweder das projizierte Polygon an alle Prozessoren verschicken, mit deren Bildraumanteil es überlappt, oder die vor dem Verschicken die Schnitte der Polygone mit den Bildraumanteilen berechnen (z.B. der PGL-Vorgänger [CO93]) erzeugen ein geringeres Kommunikationsvolumen, führen aber zusätzliche Berechnungen durch, die normalerweise nicht in der Renderingpipeline vorkommen. PGL versucht daher die Kommunikation zur Verteilung der Spans asynchron auszuführen und mit der Berechnung zu überlappen. Dazu ist es notwendig, eine größere Anzahl geometrischer Objekte, die PGL dazu übrigens nicht kopiert, sondern die durch Pointer auf die Benutzerdatenstrukturen gegeben sind, in einem Durchgang zu rendern. In der Renderingschleife erfolgt zunächst die Berechnung einiger Spans, die in die Ausgabepuffer der zuständigen Prozessoren kopiert werden. Für gefüllte Puffer wird ein nicht-blockierendes Senden angestoßen. Dann wird ebenfalls nicht-blockierend nachgefragt, ob Spans in den eigenen Eingabepuffern bereit stehen und diese gegebenenfalls gerastert. Für leere Eingangspuffer wird dann ein nicht-blockierender Empfang eingeleitet und schließlich zum Anfang der Renderingschleife zurückgesprungen. Wieviele Spans jeweils

zu berechnen sind, bevor Empfang und Rasterung von wievielen Spans anderer Prozessoren versucht wird, und wieviele Spans vor dem Abschicken zwischenzupuffern sind, ist offensichtlich von den Eigenschaften der verwendeten Maschine abhängig. In [Cro94] sind einige Zahlen angegeben.

Zur Bildausgabe definiert PGL ein eigenes Dateiformat namens YMF (*Y Movie Format*). Das *Y* bezieht sich darauf, dass jede Bildzeile mit ihrer Zeilennummer markiert wird, die Bildzeilen also in beliebiger Reihenfolge in der Datei stehen können. *Movie* spielt auf die Tatsache an, dass nur das erste Bild einer Sequenz vollständig gespeichert wird. Für Folgebilder ist jeweils nur die Differenz zum nächsten gespeichert. Einzelne (Differenz-) Bilder werden zudem mit einer Lauflängenkodierung komprimiert. Das Format ist ein Kompromiss zwischen Kompressionseffizienz und Rechenaufwand. YMF-Dateien lassen sich entweder abspeichern oder als „Live Stream“ auf einer Workstation betrachten. Für letzteres bietet PGL einen Dämon, der auf der Workstation läuft und automatisch für jeden eingehenden Stream ein Fenster öffnet, in dem er angezeigt wird.

Performance

In [Cro94] wird für einen einzelnen Prozessor einer Intel Paragon XP/S eine Renderingleistung von 4.200 Dreiecken/s und für 192 Prozessoren eine Renderingleistung von 292.000 Dreiecken/s angegeben, was einer parallelen Effizienz von 39% entspricht. Für einen weiteren Benchmark wird eine Effizienz von 59% reklamiert. In [Cro98] sind 10.7 MDreiecke/s auf 128 Prozessoren einer T3E-600 und 24.8 MDreiecke/s auf 512 Prozessoren, jeweils für eine Bildgröße von 800x640, genannt. Eine parallele Effizienz lässt sich daraus nicht ermitteln, dem Schluss „These results demonstrate that software-based renderers for large-scale parallel systems can deliver performance which compares favorably with that of high-end graphics workstations“ kann man aber zustimmen. Zur Skalierbarkeit in der Bildgröße erwähnt [Cro94], dass mit PGL routinemäßig Bilder bis zur Größe von 4096×4096 für Publikations- und Präsentationszwecke erzeugt wurden.

3.4.3 PMESA

PMESA[1] (siehe [MC98]) ist wie PGL eine parallele Graphikbibliothek für Distributed Memory Computer mit Message Passing-Interface, die ebenfalls ohne Graphik-Hardware auskommt, also eine reine Software-Lösung darstellt. Skalierbarkeit in der Datensatzgröße wird angestrebt, Skalierbarkeit in der Bildauflösung (im Gegensatz zu PGL) nicht. Gerenderte Bilder lassen sich entweder lokal im Dateisystem des Parallelrechners speichern oder zur Anzeige an eine Workstation verschicken.

Architektur

PMESA ist ein Zusatz zur Mesa-Bibliothek (siehe [MES]), einer Open Source-Implementierung des OpenGL-APIs in Software. Die Erweiterung besteht lediglich aus der Realisierung eines Compositing-Schrittes nach dem Sort-Last-Ansatz zur Erzeugung des Gesamtbildes aus den Einzelbildern und einem Mechanismus zur Ausgabe des gerenderten Gesamtbildes.

Implementierung

Mesa rendert normalerweise per Software in einen Pixelpuffer im Hauptspeicher von der Größe des zu erzeugenden Bildes. Nachdem ein Bild fertig gestellt ist, wird es über X11 in einem Fenster angezeigt (über `XPutImage`). PMESA verwendet eine Mesa-eigene Erweiterung zum Offscreen Rendering (*OSMesa*), bei der der Anzeigeschritt via X11 entfällt. Während des Renderings bearbeitet jeder Prozessor die ihm zugewiesen geometrischen Objekte völlig unabhängig von den anderen genauso wie in einer sequentiellen Anwendung. Für den abschließenden Compositing-Schritt über z müssen dann diese Teilbilder ausgelesen werden. Da der Pixelpuffer eine opake Datenstruktur der Mesa-Bibliothek ist, wird zum Auslesen der Farb- und z-Werte `glReadPixels` verwendet. Dadurch bleibt PMESA unabhängig

[1] Nicht zu verwechseln mit dem gleichnamigen inzwischen aufgegebenen Sourceforge-Projekt, das das Software-Rendering in Mesa für SMP-Rechner optimieren wollte.

von den Interna der Mesa-Bibliothek bzw. der verwendeten Mesa-Version. Prinzipiell ließe sich die Idee auch auf andere OpenGL-Implementierung übertragen, die statt der Mesa-eigenen Offscreen-Erweiterung die neueren offiziellen *pbuffers* (OpenGL-Extension für offscreen Pixelpuffer) mitbringen. Das Compositing selbst findet dann in Software über das Verbindungsnetzwerk des Parallelrechners nach einer der in Abschnitt 3.5.1 vorgestellten Methoden statt. Für die Berechnung der Teilbilder steht zwar das volle OpenGL-API zur Verfügung, da das Compositing aber über z stattfindet, führen alle Renderingtechniken, die damit nicht auskommen, zu Fehlern im Gesamtbild.

Performance

Für einen Polygondatensatz bestehend aus knapp einer Million Dreiecken nennt [MC98] eine Renderingleistung von 8.570 Dreiecken/s auf einem Prozessor einer Intel Paragon XP/S und 1.049 MDreiecke/s auf 256 Prozessoren, was eine parallele Effizienz von 48% ergibt. Gemäß einer zum Zeitpunkt der Abfassung dieser Arbeit nicht mehr auffindbaren Webseite der Sandia National Labs wurde PMESA seinerzeit zur Visualisierung einer der ersten großen Simulationen (angeblich sogar dem Abnahmetest) auf ASCI Red verwendet: Dem Einschlag eines Kometen auf Long Island — PMESA heißt seitdem TNT_PMESA.

Im Vergleich zu PGL sieht PMESA gut aus. Außer einer eher besseren Performance hat PMESA noch folgende Pluspunkte aufzuweisen:

- PMESA bietet ein wohlbekanntes (wenn auch nicht ganz vollständiges) Standard-API, während PGL aus technischen Gründen ein eigenes und eher ungewöhnliches API verwenden muss.
- Das PMESA-Addon ist auch in der Implementierung trivial: 719 Zeilen Code (inkl. Kommentare) für zwei zusätzliche Funktionen. PGL ist ungleich komplexer.

PGL hat als Plus „nur“ die Skalierbarkeit in der Bildauflösung zu bieten.

3.4.4 Chromium

Chromium (siehe [HHN+02] und [CHR], Cr: Cluster Rendering) hat im Gegensatz zu den bisher vorgestellten parallelen Renderingsystemen keine festgefügte Gestalt, sondern versteht sich als Framework zum parallelen Rendering auf Clustern aus Standardkomponenten und zwar inklusive Graphikkarten. Unterstützt werden innerhalb dieses Frameworks Sort-First-, Sort-Last-, sowie prinzipiell auch hybride Konfigurationen — wobei für letzteres noch der Nachweis einer sinnvollen Anwendung innerhalb des Chromium-Frameworks aussteht. Sort-Middle wird explizit nicht unterstützt, da jede Graphikkarte innerhalb des Clusters eine komplette Renderingpipeline darstellt: Für Sort-Middle müsste man die Graphikkarten quasi „aufschneiden."

Chromium ist der offizielle Nachfolger von WireGL, d.h. es enthält viele seiner Ideen und verallgemeinert sie. Insbesondere lässt sich eine Konfiguration mit der WireGL-Funktionalität erstellen.

Ziel ist der Aufbau von Konfigurationen, die in der Renderingrate, in der Displayauflösung sowie in der Datensatzgröße skalieren — nicht unbedingt in allen drei Dimensionen gleichzeitig.

Architektur

Zentral für Chromium ist der Begriff des *Streams*. Ein Stream ist eine potentiell unendliche Folge von Kommandos. Programme zum *Stream Processing* bearbeiten diese Kommandos unmittelbar nachdem sie eintreffen, und verwenden dazu nur endlich viele insbesondere von der Länge des Streams unabhängige Ressourcen. ‚Immediate Mode-'Rendering in OpenGL ist das eine hier interessierende Beispiel. Das Graphiksystem verarbeitet die ankommenden Kommandos eines oder mehrerer Streams unter Verwendung eines endlichen Frame Buffers sowie von endlich viel Speicher für den Zustand der OpenGL State Machine. Das andere ist eine Komponente, die unter derselben Randbedingung (endlicher, von der Länge des Input Streams unabhängiger Ressourcenverbrauch) als Ergebnis der Verarbeitung

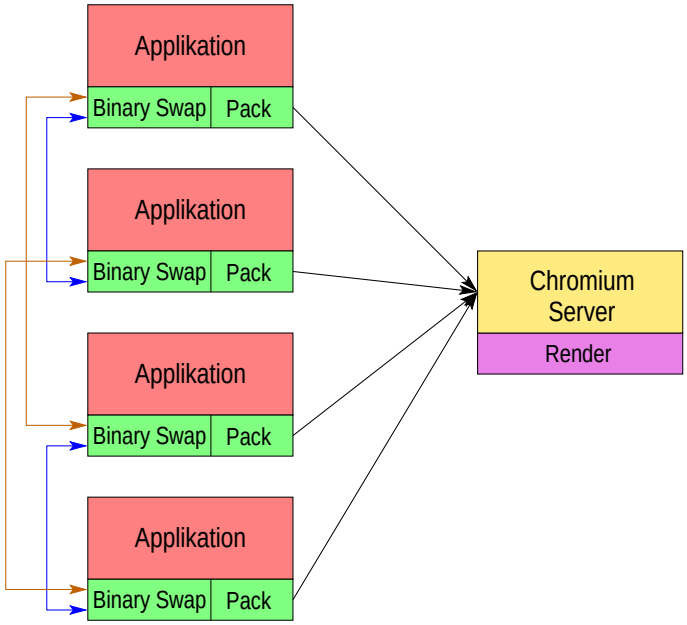

Abbildung 3.7: Chromium-Konfiguration zum Sort-Last-Rendering. Jeder Client rendert seinen Anteil an den geometrischen Objekten mit lokaler Graphik-Hardware. Über die Binary Swap-SPU findet das Compositing (out-of-band) statt, wonach jeder Client seinen Abschnitt von Gesamtbild zur Ausgabe verschickt.

eines oder mehrerer Input Streams einen oder mehrere neue Output Streams erzeugt. Chromium spricht in beiden Fällen von *Stream Processing Units* (SPUs) und Streams sind immer OpenGL-Streams. Damit lassen sich Renderingsysteme als (gerichteter azyklischer) Graph mit Rechnern als Knoten und Streams als Kanten aufbauen, wobei auf den Rechnern eine oder mehrere in Reihe geschalteter SPUs die einlaufenden Streams verarbeiten. Rechnerknoten, in die keine Kanten einmünden, werden als Clients bezeichnet. Auf ihnen wird der OpenGL-Stream von einer Applikation erzeugt (Quellen). Rechnerknoten, die eine oder mehrere eingehende Kanten haben, werden als Server bezeichnet. Sie verbrauchen ihre einkommenden Streams, üblicherweise durch Rendering (Senken), oder sie erzeugen neue, die dann weitergesendet werden. Eine Beispielkonfiguration zum Sort-Last-Rendering ist in Abb. 3.7 zu sehen, eine zum Sort-First-Rendering in Abb. 3.9.

Implementierung

Jeder Serverknoten in einem Chromium-Graphen besteht aus einer Serialisierungskomponente und einer Transformationskomponente. Die Serialisierungskomponente ist mit dem Scheduler eines Betriebssystems vergleichbar, indem sie aus den ankommenden Streams einen auswählt und ihn solange in den Output-Stream kopiert, bis der Input-Stream abbricht oder aufgrund der in Abschnitt 3.4.4 zu diskutierenden Kommandos zur Synchronisation von Streams "blockiert". Dann wird soweit möglich ein anderer Input-Stream weitergereicht. Die Transformationskomponente dispatcht als erstes die einkommenden Kommandos des Streams auf entsprechenden Funktionen der SPU. Dazu enthält jede SPU eine Tabelle, in der zu jeder Kommandonummer ein Pointer auf die dieses Kommando verarbeitende Prozedur steht. Falls mehrere SPUs in Reihe geschaltet sind, werden sie beginnend bei der letzten initialisiert. Jede SPU liefert bei der Initialisierung eine Liste all der Kommandos, für die sie Prozeduren anbietet. Falls eine SPU nur einige wenige Kommandos bearbeiten will, kann sie für alle anderen den Pointer auf die Prozedur der ihr in der Kette nachfolgenden SPU als eigenen zurückgeben. Dadurch wird Indirektion vermieden und es entsteht kein unnötiger Overhead. Im Regelfall ist die letzte (meist einzige) SPU auf einem Serverknoten die Rendering-SPU (s.u.), die die eintreffenden Kommandos mit der OpenGL-Bibliothek des Systems rendert („Rendering-Server", Senke). Falls trotzdem noch weitere Aufgaben anfallen, kann die Pack-SPU (s.u.) nachgeschaltet werden, die einen neuen Stream erzeugt und diesen über das Netzwerk an den nächsten Server weiterleitet, d.h. ein Knoten kann sowohl Server als auch Client sein.

Auf Client-Knoten entfällt der Serialisierer. Seine Funktion wird von der Applikation übernommen, die den ursprünglichen Stream erzeugt, der direkt in die SPUs eingespeist wird. Da Clients im Chromium-Modell kein Rendering ausführen, sondern ihren Stream auf jeden Fall an einen Server weiterleiten, ist die Funktionalität der Pack-SPU (s.u.) implizit. Entsprechend sind in den Abb. 3.7 und 3.9 die die Streams repräsentierenden Kanten von der SPU auf dem Client zum Server, der den Serialisierer enthält gezeichnet.

Zur Programmierung von eigenen SPUs sei gesagt, dass eine SPU nicht das

komplette OpenGL-API implementieren muss. Stattdessen kann sie von einer bereits vorhandenen SPU nach dem ‚Single Inheritance'-Model abgeleitet werden. Das ist übrigens nichts anderes als das oben beschriebene In-Reihe-Schalten von SPUs auf einem Knoten unter einem etwas anderen Gesichtspunkt.

Nachfolgend eine unvollständige Auswahl von SPUs, die mit zum Chromium-Framework gehören:

- Render-SPU

  Die Render-SPU rendert ihren Input-Stream über die OpenGL-Bibliothek des Systems in ein Fenster. Praktisch jede Chromium-Konfiguration enthält die Render-SPU oder eine von ihr abgeleitete SPU.

- Tilesort-SPU

  Die Tilesort-SPU implementiert die WireGL-Funktionalität in Chromium, indem sie ihren Input-Stream entsprechend einer Zerlegung des Bildraums in Tiles auf ihre Output-Streams verteilt. Ein Beispiel für eine solche Sort-First-Konfiguration mit zwei Clients und vier Servern ist in Abb. 3.9 skizziert.

- Readback-SPU

  Die Readback-SPU ist von der Render-SPU abgeleitet. Sie überschreibt allerdings deren `SwapBuffer`-Methode durch eine eigene Implementierung, die stattdessen aus dem Frame Buffer mittels `glReadPixels` die Farbwerte und eventuell z- bzw. α-Kanal ausliest und daraus mittels `glWritePixels` einen Output-Stream erzeugt. Die Readback-SPU wird z.B. in Sort-Last-Konfigurationen verwendet, in denen ein weiterer Server das Compositing übernimmt.

- ‚Binary Swap'-SPU

  Die ‚Binary Swap'-SPU ist wie die Readback-SPU von der Render-SPU abgeleitet. Sie überschreibt ebenfalls deren `SwapBuffer`-Methode und liest den Frame Buffer mittels `glReadPixels` aus. Im Gegensatz zur Readback-SPU werden die Teilbilder aber nicht sofort an den nächsten Server geschickt, sondern das Compositing erledigen die ‚Binary Swap'-SPUs untereinander parallel und out-of-band (d.h. nicht

über die im Chromium-Graphen über die Kanten vorgegebenen Kommunikationswege) mit dem in Abschnitt 3.5.1 zu beschreibenden Binary Swap-Algorithmus. Danach besitzt jede der n ‚Binary Swap'-SPUs ein n-tel des Gesamtbildes, das dann wieder über einen `glWritePixels`-Stream an einen Server zur Anzeige geschickt wird, vgl. Abb. 3.7.

- Saveframe-SPU

  Ebenfalls von der Render-SPU abgeleitet ist die Saveframe-SPU. Wieder wird die `SwapBuffer`-Methode überschrieben, diesmal durch eine Funktion, die nach dem Auslesen des Frame Buffers diesen in eine Datei schreibt. Danach wird die `SwapBuffer`-Methode der Rendering-SPU (der Ober-SPU) aufgerufen. Damit lassen sich z.B. Screenshots der Einzelbilder einer Applikation machen, auch wenn diese dies nicht unterstützt, ohne sie zu verändern.

- Pack-SPU

  Die Pack-SPU verschickt ihren Input-Stream über das Chromium-Protokoll (s.u.) an den nächsten Server. Auf Clients ist diese Funktionalität implizit enthalten, ein Server, der seinen Input-Stream nicht bis zur Anzeige bringt und deshalb weiterschicken will, muss die Pack-SPU als letzte in seiner SPU-Reihe verwenden.

- Print-SPU

  Die Print-SPU verarbeitet ihren Input-Stream, indem sie eine textuelle Repräsentation desselben auf die Konsole oder in eine Datei schreibt. Sie eignet sich zum Debugging von Chromium- oder auch ganz prinzipiell von beliebigen OpenGL-Applikationen.

- Passthrough-SPU

  Die Passthrough-SPU kopiert ihren Input-Stream unverändert in den Output-Stream. Für sich alleine nutzlos ist sie zur Abbleitung eigener SPUs gedacht. Ein einfaches Beispiel wäre eine Grayscale-SPU zum Thema *Stylized Drawing*, die alle `glColor`-Kommandos mit einer Version überschreibt, die aus den RGB-Werten einen Grauwert berechnet und dann diesen als Zeichenfarbe setzt.

Zur Übertragung von Streams zwischen Rechner-Knoten wird im wesentlichen genauso vorgegangen wie schon bei WireGL beschrieben. Geometriekommandos werden senderseitig zwischengepuffert und en block übertragen. Für zustandsverändernde Kommandos verfolgt jeder Client seinen eigenen Zustand sowie den aller Server, mit denen er kommuniziert, sodass nur noch Kommandos zur Realisierung dieser Zustandsdifferenz verschickt werden müssen. Auf Serverknoten ist entsprechen ein eigener Zustand für jeden Client-Stream vorhanden. Wechselt ein Server seinen Input-Stream muss er entsprechend auch den Zustand seines OpenGL-Kontextes ändern. Beides lässt sich mit dem optimierten OpenGL State Tracker aus WireGL (siehe wieder [BHH00]) effizient lösen. Zum Zugriff auf das Netz verwendet Chromium eine eigene Abstraktionsschicht, die für TCP/IP und Myrinet GM implementiert ist.

Um die OpenGL-Streams der Applikationen umzuleiten, verwendet auch Chromium den Trick, die `libGL.so` des Systems durch eine eigene Variante zu ersetzen.

Die Konfiguration eines Chromium-Systems, d.h. die Unterteilung von Rechnerknoten in Clients und Server, die Zuordnung der SPUs zu den Rechnerknoten sowie die Verschaltung der Knoten zu einem Graphen, ist programmierbar und wird durch ein Python(siehe [PYT])-Skript vorgenommen.

Paralleles API

Dieser und der nächste Abschnitt befassen sich speziell mit zwei weiteren Aspekten, die sich vor allem beim Ansteuern einer Tiled Display Wall nach dem Sort-First-Modell ergeben, und die bisher ignoriert wurden.

Das OpenGL-API garantiert, dass OpenGL-Kommandos in der Reihenfolge, in der sie die Applikation absetzt, abgearbeitet werden. Das ist für Renderingtechniken wichtig, die auf diese Garantie angewiesen sind, z.B. beim Eintrag von Pixeln mit dem over-Operator. Wenn im Sort-First-Modell mehrere Clients eine Applikation parallel ausführen, können bei jedem Rendering-Server mehrere Streams ankommen. Nach dem bisher gesagten steht es jedem Rendering-Server frei, diese Streams nach Belieben

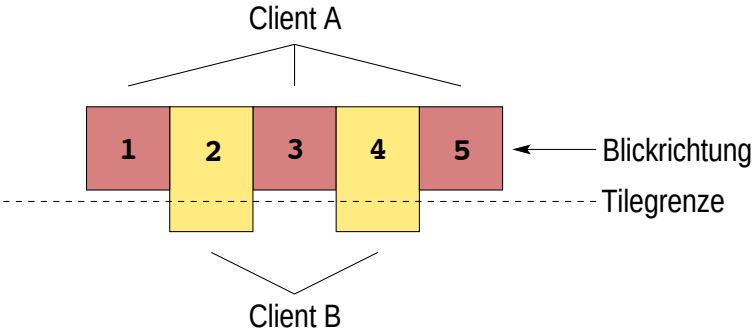

Abbildung 3.8: Zur Illustration des parallelen APIs in Chromium: Fünf Blöcke verteilt auf zwei Clients, sollen von „hinten nach vorne“, d.h. in der Reihenfolge 1–5, gerendert werden.

auf seine Graphikkarte zu multiplexen. Um auch bei mehreren parallelen OpenGL-Streams applikationsseitig eine eindeutige Reihenfolge definieren zu können, ist eine Erweiterung des OpenGL-APIs notwendig, wie sie in [ISH98] vorgeschlagen wurde. Die Idee ist, dass eine Applikation durch Einfügen der bekannten Synchronisationsprimitive aus dem Bereich der Interprozesskommunikation, Semaphor und Barrier (siehe z.B. [Dij65] und [Tan92]), in ihre Streams den Rendering-Servern die gewünschte Abarbeitungsreihenfolge mitteilt bzw. globale Synchronisationspunkte (etwa zum Front-/Back Buffer Swap) setzt. Der wesentliche Punkt dabei ist, dass diese Semaphore und Barrier nicht die Applikation auf den Clients blockieren, sondern die Abarbeitung der OpenGL-Streams auf den Rendering-Servern. Bei genügend Pufferkapazität könne die Clients den Servern „vorauseilen“. Ein einfaches Beispiel zur Verwendung dieser Synchronisationsprimitive ist in Abb. 3.8 illustriert. Eine Applikation auf zwei Clients erzeugt parallel eine Visualisierung eines Datensatzes, bestehend aus fünf Blöcken, die wie gezeigt lastverteilt sind, und die in der angegebenen Reihenfolge, d.h. von Block 1 nach Block 5, gerendert werden sollen. Mit den Bezeichnungen P und V für die Semaphor-Operationen, vgl. Abschnitt 1.3.2, Seite 13, fünf Semaphoren 1–5, deren Zähler auf Null initialisiert sind, und $R(i)$ für das Erzeugen der OpenGL-Kommandos für Block i, lässt sich das Problem durch das Erzeugen folgender Streams lösen.

Client A:

$$R(1); V(1); P(2); R(3); V(3); P(4); R(5).$$

Client B:

$$P(1); R(2); V(2); P(3); R(4); V(4).$$

Da man zur Verwaltung der Semaphore ohne eine zentrale Instanz auskommen will, muss über diese auf allen Rendering-Servern Buch geführt werden, und die Semaphor-Operationen selbst müssen an alle Server verschickt werden. Im obigen Beispiel erhalten die Server folgende Streams.

Oberer Rendering-Server:

$$R(1); V(1); P(2); R(3); V(3); P(4); R(5).$$
$$P(1); R(2); V(2); P(3); R(4); V(4).$$

Unterer Rendering-Server:

$$V(1); P(2); V(3); P(4).$$
$$P(1); R(2); V(2); P(3); R(4); V(4).$$

Barrier schließlich sind generell für jede parallele Renderingapplikation nützlich. Die übliche Grundstruktur einer solchen Applikation zum Zeichnen eines Frames sieht etwa so aus:

1. Der Master-Prozess löscht den Frambuffer.
2. Alle Prozesse rendern ihren Anteil an einer Szene.
3. Der Master-Prozess veranlasst die Bildausgabe, z.B. durch `SwapBuffers`.

Zur Synchronisation dieser Schritte ist zwischen Schritt 1 und 2 sowie zwischen Schritt 2 und 3 ein Barrier einzufügen.

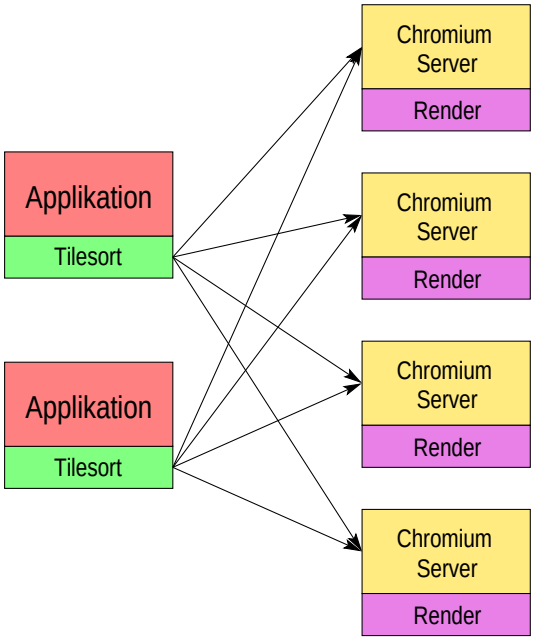

Abbildung 3.9: Chromium-Konfiguration, bei der zwei Clients parallel nach dem Sort-First-Prinzip eine 2x2 Tiled Display Wall ansteueren.

DMX

Bisher war nur von OpenGL die Rede, was aber vernachlässigt, dass kein Programm ausschließlich OpenGL benutzt. Fensterdekorationen und GUI-Elemente werden über X11 gezeichnet. Für Tastatur- und Mauseingaben ist ebenfalls der X-Server verantwortlich. Für Chromium in der WireGL-Konfiguration (und WireGL selbst) bedeutet das, da X11-Aufrufe im Gegensatz zu OpenGL-Aufrufen nicht abgefangen werden, dass auf dem Rechner, auf dem die Applikation läuft, ein leeres Fenster erscheint, und nur der Inhalt dieses Fenster, soweit er mit OpenGL erzeugt wird, auf der Tiled Display Wall angezeigt wird. Interaktion (z.B. mit der Maus) über das leere Fenster auf dem Applikations-Rechner ist allerdings trotzdem möglich, wenn auch wahrscheinlich nicht sehr intuitiv. Auch ergibt sich ohne die Koordination zwischen X11 und OpenGL die Einschränkung auf ein einzelnes Fenster, in das mit OpenGL gezeichnet werden kann.

DMX (Distributed Multihead X, siehe [DMX]) schließt diese Lücke, indem es die hinter den einzelnen Tiles stehenden X-Server zu einem einzelnen X-Server zusammenfasst, der die ganze Tiled Display Wall umfasst. Diese Abstraktion wird durch einen *Front-End X-Server* erreicht, der als Proxy für die *Back-End X-Server* auf den Rendering-Servern fungiert, und der von der Applikation statt eines lokalen X-Servers angesprochen wird. Da der Front-End X-Server mit den Back-End X-Servern über das gewöhnliche X-Protokoll kommuniziert, lassen sich damit (via GLX) auch OpenGL-Aufrufe transportieren. Da es, wie die Beschreibung von WireGL bzw. der Tilesort-SPU von Chromium gezeigt hat, nicht trivial ist, die OpenGL-Aufrufe effizient an die Rendering-Server zu verteilen, benutzt DMX hierfür einfach einen Broadcast. Dieser letzte Schwachpunkt wurde durch die naheliegende Integration von Chromium und DMX beseitigt. Damit lassen sich (serielle) Applikationen auf einer Tiled Display Wall genauso ausführen wie auf einem gewöhnlichen Bildschirm.

Performance

Aufgrund des Framework-Charakters von Chromium lassen sich über die Performance keine allgemeinen Aussagen machen, einige der häufig verwendeten Konfigurationen sollen jedoch betrachtet werden. Die WireGL-Konfiguration von Chromium entspricht weitgehend dem originalen WireGL, so dass das dort zur Performance gesagte auch für Chromium gilt. Die beiden anderen interessanten Konfigurationen sind die Sort-First-Variante mit mehreren parallelen Clients (Abb. 3.9) und die Sort-Last-Konfiguration (Abb. 3.7).

Auch in Chromium sind Sort-First-Konfigurationen generell durch das Netzwerk limitiert. In [Pau02] sind Zahlen für eine parallele Konfiguration mit mehreren Clients und einem Rendering-Server angegeben. Für hinreichend rechenintensive Anwendungen steigt die Renderingleistung fast linear in der Zahl der Clients ($n \leq 32$). Für weniger rechenintensive Anwendungen, also solche bei denen jeder Knoten immer mehr OpenGL-Kommandos pro Zeiteinheit absetzen kann, stagniert die Renderingleistung erwartungsgemäß, sobald die Netzwerkverbindung zum Rendering-Server gesättigt ist.

Kombinationen aus n Rendering-Servern und m Clients skalieren nach derselben Quelle wieder gut, soweit die Rendering-Last sich gleichmäßig über die Rendering-Server verteilt. Auch wenn man davon im allgemeinen eher nicht ausgehen kann, so kann man immer noch einem pragmatischen Ansatz folgen, nach dem man die n Rendering-Server eben braucht, um n Tiles anzusteuern, und die m Clients, um einen entsprechend großen Datensatz zu verarbeiten. Sort-First-Konfigurationen skalieren demnach in der Displaygröße und der Datensatzgröße, jedoch nicht unbedingt in der Visualisierungs-Leistung. Ähnliches kann man auch [BHPB03] entnehmen.

Eine Sort-Last-Konfiguration wie in Abb. 3.7 skaliert unter der Bedingung einer optimalen Lastverteilung der geometrischen Objekte bis zum Compositing-Schritt stets ideal. Diese optimale Lastverteilung lässt sich i.a. viel einfacher herstellen als im Falle von Sort-First, da sie unmittelbar im Objektraum stattfindet, also unabhängig von den Projektionsparametern ist. Bleibt noch der Aufwand für das Compositing und die Bildausgabe, was in Abschnitt 3.5 besprochen wird. In [HHN+02] ist ein Beispiel zum parallelen Volume Rendering angegeben, bei dem die Visualisierungs-Leistung (in Bildern pro Sekunde) nahezu konstant bleibt, wenn man sukzessive die Datensatzgröße und die Zahl der Renderer verdoppelt. (Allerdings nur bis 16 Knoten angegeben.) Sort-Last skaliert in der Datensatzgröße und der Renderingleistung, allerdings nicht in der Displaygröße.

3.5 Beispiele für Compositing-Implementierungen

3.5.1 Compositing in Software

In diesem Abschnitt sollen Compositing-Implementierungen besprochen werden, die lediglich die vorhandenen CPUs und das Verbindungsnetzwerk des Parallelrechners verwenden. Da der Compositing-Operator (vgl. Abschnitt 3.2) pixelweise definiert ist, liegt die Implementierung für das Compositing zweier Bilder auf einem Rechnerknoten als Schleife über alle Pixel der beiden Bilder fest. Man kann also im wesentlichen nur noch durch geschickte Wahl des Kommunikationsschemas zwischen den Knoten etwas

erreichen. Die beiden wesentlichen Schemata, die beide von einem assoziativen aber nicht notwendigerweise kommutativen Compositing-Operator $\oplus$ ausgehen, sollen im Folgenden beschrieben werden. Dabei wird von einem geswitchten voll-duplex-fähigen Verbindungsnetzwerk ausgegangen und der Einfachheit halber angenommen, dass die Anzahl der Prozessoren eine Zweierpotenz ist.

Zur Analyse der Algorithmen werden folgende Bezeichnungen verwendet.

- T_{CPU} CPU-Rechenzeit für das Compositing
- T_{com} Zeit für die Kommunikation während des Compositings
- T_{lat} Latenzzeit des Netzwerks
- T_{pix} Zeit um ein Pixel zu übertragen
- $T_{\oplus}$ Zeit um zwei Pixel per $\oplus$ zu verknüpfen
- A Grösse des Bildes in Pixel
- N Anzahl der Prozessoren (Zweierpotenz)

Log-Reduce Tree Compositing

Die einfachste Idee ist, das Compositing in der Art einer simplen Reduktionsoperation auszuführen, wie man sie etwa von dem Problem, die Summe von über die einzelnen Knoten verteilten Instanzen einer skalaren Variablen zu berechnen, kennt. Wenn etwa $P_0 \oplus P_1 \oplus \cdots \oplus P_7$ zu berechnen ist, wobei jedes Bild P_i auf Prozessor i liegt, so kann man unter Ausnutzung der Assoziativität und folgender Klammerung

$$\Big(\big((P_0 \oplus P_1) \oplus (P_2 \oplus P_3)\big) \oplus \big((P_4 \oplus P_5) \oplus (P_6 \oplus P_7)\big)\Big)$$

die Teilbilder parallel kombinieren, vgl. Abb. 3.10. Im ersten Schritt senden die Prozessoren 1, 3, 5 und 7 ihre Teilbilder parallel an die Prozessoren 0, 2, 4 und 6, die dann den ersten Compositing-Schritt parallel ausführen. Im zweiten Schritt senden 2 und 6 ihr Zwischenergebnis parallel an 0 und 4, die den zweiten Compositing-Schritt parallel ausführen. Im letzten Schritt

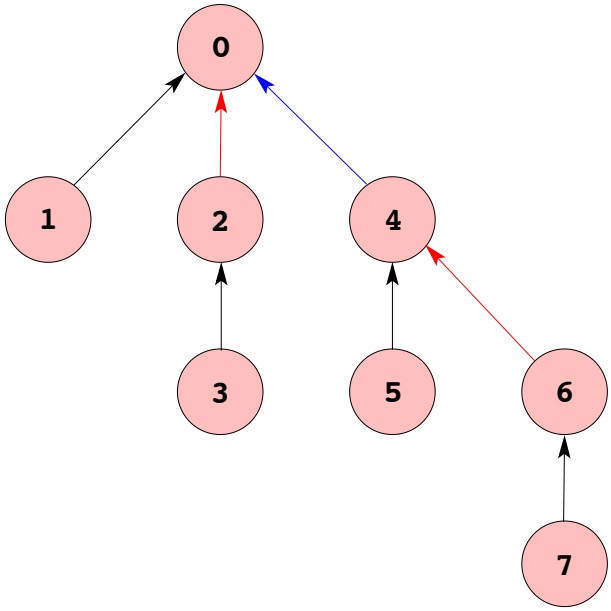

Abbildung 3.10: Beispiel zum Log-Reduce Tree Compositing für acht Prozessoren.

sendet 4 an 0 und 0 erledigt den letzten Compositing-Schritt. Für allgemeines N sind log N Schritte auszuführen, wobei in jedem Schritt 2^{k-1} parallele Sendevorgänge und dann 2^{k-1} parallele Compositing-Vorgänge ausgeführt werden, $k = \log N, \cdots, 1$. Daraus ergeben sich folgende Werte für den Rechenzeitbedarf T_{CPU} und die Kommunikationszeit T_{com}.

$$T_{CPU} = \sum_{k=1}^{\log N} AT_{\oplus} = AT_{\oplus} \log N$$

$$T_{com} = \sum_{k=1}^{\log N} (T_{lat} + AT_{pix}) = (T_{lat} + AT_{pix}) \log N$$

Beide Größen sind im wesentlichen linear in der Bildgröße und proportional zu log N. Ersteres spiegelt die prinzipielle Eigenschaft des Compositings wieder und ist der Grund, warum Sort-Last nicht in der Displaygröße skaliert. Letzteres ist die wesentliche Eigenschaft des ‚Log-Reduce Tree'-Schemas.

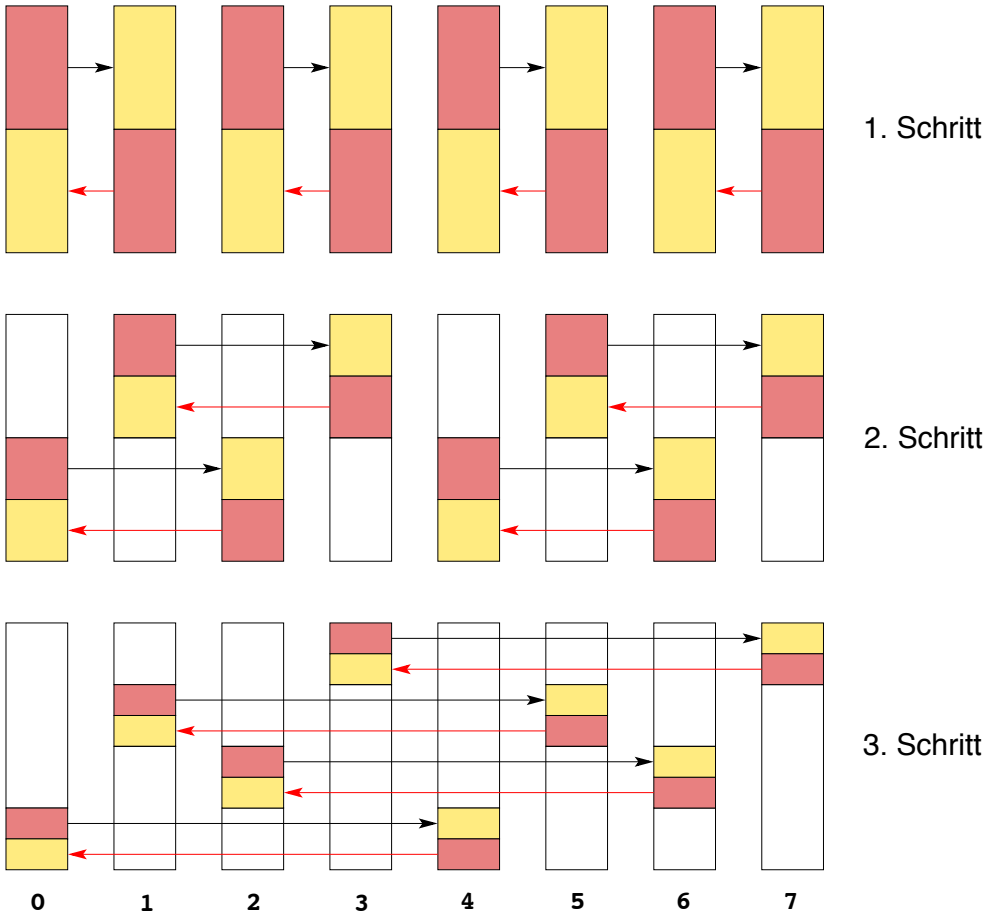

Abbildung 3.11: Beispiel zum Binary-Swap Compositing für acht Prozessoren.

Binary-Swap Compositing

Der offensichtlich Nachteil im ‚Log-Reduce Tree'-Schema ist, dass die Anzahl der Prozessoren, die noch am Compositing beteiligt sind, sich pro Schritt halbiert. Während das Schema für die Reduktion einer einzelnen skalaren Variable trotzdem optimal ist, bietet die Tatsache, dass hier ganze Bilder pixelweise kombiniert werden, weitere Möglichkeiten zur effizienteren Parallelisierung. Die Grundidee beim Binary-Swap-Schema (zuerst in [MPHK94] vorgeschlagen) ist, dass ein Prozessor nicht alle seine Pixel an einen anderen Prozessor schickt und diesen dann das Compositing alleine machen lässt, sondern dass beide Prozessoren jeweils die Hälfte ihres

Pixelbereichs mit dem anderen tauschen und dann jeder die Hälfte der Compositing-Arbeit übernimmt. Im Falle von zwei Teilbildern auf zwei Prozessoren hat dann jeder eine Hälfte des Gesamtbildes errechnet. Zur Berechnung von $P_0 \oplus P_1 \oplus P_2 \oplus P_3$ auf vier Prozessoren bildet man zwei Gruppen zu je zwei Prozessoren, von denen die erste Gruppe wie oben je zur Hälfte $P_0 \oplus P_1$ und die zweite Gruppe je zur Hälfte $P_2 \oplus P_3$ berechnet. Im nächsten Schritt wird umgruppiert. Die Prozessoren, die sich um dieselbe Hälfte ihrer Teilaufgaben (d.h. $P_0 \oplus P_1$ bzw. $P_2 \oplus P_3$) gekümmert haben, bilden neue Gruppen, in denen sie dann gemeinsam das Gesamtbild für ihre Hälften berechnen, d.h. nach dem zweiten Schritt hat jeder Prozessor ein Viertel des Gesamtbildes. Für acht Prozessoren ist der Binary-Swap-Algorithmus in Abb. 3.11 veranschaulicht. Für allgemeines N sind $\log N$ Schritte auszuführen, wobei in jedem Schritt N parallele Sendevorgänge der Größe $A/2^k$ und N parallele Compositing-Vorgänge für $A/2^k$ Pixel ausgeführt werden, $k=1, \cdots, \log N$, und jeder Prozessor zum Schluss ein N-tel des Gesamtbildes berechnet hat. Für den Rechenzeitbedarf T_{CPU} und die Kommunikationszeit T_{com} ergibt sich deshalb:

$$T_{CPU} = \sum_{k=1}^{\log N} \frac{A}{2^k} T_{\oplus} \approx A T_{\oplus}$$

$$T_{com} = \sum_{k=1}^{\log N} \left(T_{lat} + \frac{A}{2^k} T_{pix} \right) \approx T_{lat} \log N + A T_{pix}$$

Beide Größen sind also wieder im wesentlichen proportional zu A, der $\log N$-Term taucht aber nur noch in der Kommunikationszeit mit der (sehr kleinen) Latenzzeit als Faktor auf. In der Praxis wurde beobachtet, dass die Gesamtzeit für das Binary-Swap Compositing bei konstanter Bildgröße ab einer bestimmten Anzahl von Prozessoren praktisch konstant bleibt, vgl. z.B. [MPHK94] und [MC98].

Ein Nachteil des Binary-Swap-Schemas ist, dass das Gesamtbild auf alle N Prozessoren verteilt ist. Falls man, was den Normalfall darstellt, nicht auf einen Multi-Port Frame Buffer (siehe Abschnitt 3.5.3) zurückgreifen kann, muss man das Gesamtbild erst noch einsammeln. Die einfache Methode,

bei der jeder Prozessor sein Teilbild an einen ausgewählten schickt, benötigt einen Kommunikationsaufwand von $T_{com} = T_{lat}N + AT_{pix}$. Falls man das Einsammeln hierarchisch organisiert, kommt man mit $T_{com} = T_{lat} \log N + AT_{pix}$ aus.

Optimierungen

Optimierungsmöglichkeiten ergeben sich einmal aus der Tatsache, dass nur „nicht-leere“ Pixel (solche, die nicht die Hintergrundfarbe haben, in die also gezeichnet wurde) zum Gesamtbild beitragen, sowie daraus, dass man durch Komprimieren der Teilbilder vor dem Verschicken CPU-Leistung gegen effektive Übertragungsrate eintauschen kann.

Zur Realisierung der ersten Idee bietet es sich an, die minimale Bounding Box der nicht-leeren Pixel zu verwenden. Das spart sowohl Kommunikationsvolumen als auch während des Compositings CPU-Zeit. Soweit sich die Bounding Box nicht während des Renderns als Nebenprodukt ergibt, ist der Aufwand dafür proportional zur Anzahl der Pixel in der zu verschickenden Region. Allerdings wird dieser Aufwand nur zu Beginn des ersten Compositing-Schrittes fällig, danach lässt sich die Bounding Box eines kombinierten Bildes aus denen der Ausgangsbilder leicht berechnen. Unter der Voraussetzung, dass die auf den einzelnen Knoten gerenderten geometrischen Objekte räumlich zusammenhängen, kann man zumindest in den ersten Schritten, in denen die größten Teilbildbereiche verschickt werden, einen Einsparungseffekt erwarten. Die Teilbilder in späteren Schritten enthalten dagegen soviele Anteile von verschiedenen Prozessoren, dass nur noch wenig Einsparungen zu erhoffen sind. In [MPHK94] werden für diese Optimierung Werte für bis zu 512 Prozessoren auf einer Connection Machine CM5 angegeben, nach denen die Gesamtzeit für das Compositing mit steigender Prozessorzahl bei gleichem Datensatz sogar deutlich abnimmt. In [TIH03] wird auf das Problem eines möglichen Lastungleichgewichts zwischen den jeweiligen Partnern in einem Compositing-Schritt hingewiesen, wenn man die ‚Bounding Box‘-Optimierung zusammen mit einer blockweisen Halbierung der Teilbilder verwendet und eine zeilenweise Aufteilung der Teilbilder vorgeschlagen. Der eine Partner kümmert sich um die geraden,

der andere um die ungeraden Zeilen.

Ob sich der zweite Ansatzpunkt, Teilbilder vor dem Versenden zu komprimieren, zusätzlich zur ‚Bounding Box'-Optimierung lohnt, hängt vom Verhältnis der CPU-Leistung im Vergleich zur Netzwerkgeschwindigkeit ab. Bei einem langsamen Verbindungsnetzwerk (etwa 10-100 MBit/s Ethernet) können sich relativ aufwendige Verfahren wie Ziv-Lempel-Kompression in optimierter Implementierung (z.B. [Wil91]) noch lohnen, bei schnelleren Netzen (etwa Myrinet) wird man sich mit einfachen schnellen Verfahren wie z.B. RLE (Run Length Encoding) bescheiden. Für den ersten Fall (Workstations der 90er, 10 MBit-Ethernet) ist in [MPHK94] für Ziv-Lempel ein Kompressionsfaktor von bis zu vier und eine bis zu 80% schnellere Kommunikation (nach Abzug des Overheads durch die Kompression) angeführt. Für den zweiten Fall (aktuelle PCs, Myrinet 2000) lassen sich nach [YYC01] und [TIH03] noch gewisse Einsparungen durch RLE erreichen.

3.5.2 Compositing in Hardware

Ein PC-Cluster mit Graphikkarten, der nach dem Sort-Last-Prinzip betrieben wird, skaliert prinzipiell bis zur Compositing-Phase optimal. Um auch diese Lücke noch zu füllen, liegt der Versuch nahe, das Compositing komplett mit spezieller Hardware durchzuführen. Als Nebeneffekt wird dadurch auch die CPU entlastet. Einige Ansätze dazu wurden in den letzten Jahren verfolgt, z.B. Lightning-2 (siehe [SEP+01]) an der Stanford University und Sepia (siehe [MSH99] und [LMS+01]) bei Compqa/HP. Beide Systeme sind als PCI-Karten, die in die Renderer zusammen mit der Graphikkarte eingebaut werden, ausgelegt und wurden mit FPGAs (Field-Programmable Gate Arrays) realisiert. Für das Compositing wird ein eigenes Netzwerk verwendet. Sie befinden sich noch immer im Prototypenstatus und sind nicht im Handel erhältlich.

Bildakquisition

Damit die Compositing-Hardware ihren Zweck erfüllen kann, muss sie als erstes an die auf den Renderern erzeugten Teilbilder, die in den Frame Buffern der Graphikkarten liegen, herankommen. Dafür kommen unter den gegebenen Rahmenbedingungen folgende Möglichkeiten in Frage.

1. Auslesen der Teilbilder mit `glReadPixels` in den Hauptspeicher. Von dort werden sie per DMA an die Compositing-Hardware auf den PCI-Karten geschickt.
2. Direkter Transfer der Teilbilder aus dem Frame Buffer der Graphikkarte per DMA über den PCI-Bus zur Compositing-Hardware auf den PCI-Karten.
3. Ausgabe der Teilbilder über den DVI(Digital Visual Interface)-Ausgang der Graphikkarten. Die Signale werden per Loopback-Kabel in die Compositing-Hardware eingespeist.

Bei den beiden ersten Varianten ist zu berücksichtigen, dass der Frame Buffer eine opake Datenstruktur der Graphikhardware bzw. ihres Treiber ist, seine innere Struktur nicht dokumentiert und keinerwegs trivial ist (Stichwort: Hierarchischer (vgl. [GKM93]) und/oder komprimierter z-Buffer) und sich jederzeit ändern kann. Variante 1 umgeht dieses Problem durch Verwenden einer wohldefinierten Schnittstelle. Allerdings ist auch mit aktueller Hardware das Lesen aus dem Frame Buffer in den Hauptspeicher eine sehr langsame Operation. In [HHN+02] sind für eine GeForce3-Graphikkarte ca. 37 MB/s für RGBA angegeben (z-Buffer ist noch langsamer). Das ist etwa nur ein Drittel der Bandbreite des dort verwendeten Netzwerkes. Tatsächlich ist die Compositing-Leistung in diesem Fall durch die Geschwindigkeit, mit der die Teilbilder von der Graphikkarte zurückgelesen werden können, schon bei reinem Software-Compositing limitiert. Variante 2 umgeht das Performance-Problem, benötigt aber dafür die schon angesprochenen intimen Kenntnisse über den Aufbau des Frame Buffers. Da darüber nur der Hersteller des Graphikchips Bescheid weiß, kommt diese Variante eigentlich nur in Frage, wenn derselbe Hersteller auch die Compositing-Hardware fertigt.

Variante 3 mag auf den ersten Blick etwas verwunderlich erscheinen, ist aber trotzdem momentan am beliebtesten. DVI ist ein standardisiertes Interface mit hoher Datenrate und die Ausgabe über DVI seitens der Graphikkarte darauf optimiert, parallel ablaufende Renderingvorgänge möglichst nicht zu stören. Vor allem aber ist damit die Compositing-Hardware von der Graphik-Hardware soweit entkoppelt, dass sie auch noch mit dem alle sechs Monate erscheinenden Nachfolgemodell zusammenarbeitet. Demgegenüber steht der gravierende Nachteil, dass von der Graphikkarte über DVI natürlich nur die RGB-Komponenten ausgegeben werden, nicht aber die für das Compositing benötigten α- und z-Komponenten. Als Workaround werden zwei Möglichkeiten verwendet. Die erste arbeitet mit doppelter Auflösung in der Horizontalen und bringt nach der Fertigstellung eines Frames die α- oder z-Komponente in der rechten Bildhälfte unter. Während man für α dies mit Standard-OpenGL-Befehlen direkt im Frame Buffer erledigen kann, gibt es für z keine solche Lösung, so dass man doch wieder die langsame Methode über den Hauptspeicher mit `glGetPixels` und `glPutPixels` gehen muss. Einige NVIDIA-Graphikkarten kennen allerdings eine OpenGL-Extension, mit der man den z-Buffer doch direkt auf der Karte umkopieren kann. Die andere Möglichkeit ist, RGB und α bzw. z bei normaler Auflösung eventuell unter Anhebung der Wiederholfrequenz alternierend zu senden. Dabei ist allerdings zu beachten, dass nur RGB und α double buffered sind, z ist auf jeden Fall single buffered, d.h. es kann während der Übertragung des RGB-Signals aus dem Front Buffer noch kein Rendering in den Back Buffer stattfinden, da dabei der z-Buffer zerstört würde. Außerdem muss die Compositing-Hardware einen kompletten RGB-Buffer zwischenspeichern können.

Sepia

Von den oben genannten Ansätzen zum Hardware-Compositing soll hier HPs Sepia (momentan in der Inkarnation Sepia-2a) als konkretes Beispiel dienen. Sepia besteht aus einer PCI-Karte mit drei FPGAs, Pufferspeicher und einem ServerNet-II Chip (siehe [HGKH98]). Von den FPGAs dient einer zur PCI-Anbindung, einer zur Anbindung des Netzwerkchips und einer zur Durchführung des Compositing-Vorgangs. Die einzelnen Sepia-Karten

sind über einen Switch verbunden. Servernet-II erlaubt eine real erzielbare Transferrate von 180 MB/s pro Richtung bei einer Latenzzeit von ca. 10 µs. Für das Compositing sind die einzelnen Karten als Pipeline verschaltet. Der jeweils nächste Knoten in der Pipeline führt das Compositing mit den eigenen Pixeln ohne den Empfang eines kompletten Teilbildes abzuwarten sofort durch und schickt die Pixel des neue Teilbild ebenfalls sofort weiter. Dadurch entsteht lediglich eine kleine Latenzzeit proportional zur Anzahl der Renderingknoten, die aber nur bei sehr großen Konfigurationen von Bedeutung sein sollte. Per Default sind im Compositing-FPGA Kombination der Pixel über z und den over-Operator implementiert. Es sollen sich aber mit Standard-Tools für FPGAs auch eigene Verknüpfungs-Operatoren programmieren lassen. Für den over-Operator macht sich auch der auf den ersten Blick überflüssige Switch bezahlt, da sich über ihn die Reihenfolge der Knoten in der Compositing-Pipeline bei einer Änderung des Betrachterstandpunkts dynamisch ändern lässt. Zur Bildakquisition lassen sich alle drei der im vorigen Abschnitt erwähnten Methoden verwenden, die ersten beiden da es sich ja um eine PCI-Karte handelt, die dritte über eine zusteckbare Erweiterungskarte für DVI-IO. Über DVI sollte das Compositing bei einer Auflösung von 1280x1024 bei 60 Hz (faktisch 2560x1024 bei 60 Hz, wegen zusätzlichem z- oder α-Kanal) bei den genannten Eckdaten problemlos möglich sein. Daneben wird die Bildakquisition durch Umkopieren in den Hauptspeicher in naher Zukunft mit der Einführung von PCI Express (PCIe) wahrscheinlich ebenfalls attraktiv werden, da PCIe im Gegensatz zu AGP auch einen schnellen Rückkanal bietet.

Detaillierte Werte zur Performance von Sepia wurden bisher nicht veröffentlicht. In [LMS+01] wird immerhin erwähnt, dass ein System aus acht Renderingknoten Volume Rendering für Datensätze der Größe 512^3 mit ca. 25 Bildern/s berechnen kann. Dabei wurde für das Volume Rendering auf den Knoten allerdings spezielle Hardware verwendet. Zeiten für die einzelnen Phasen des Berechnungsvorgangs sind nicht angegeben. Eine neuere Quelle [FK04] enthält ebenfalls nichts konkretes zur Compositing-Leistung von Sepia.

Als Nebenprodukt der Sepia-Entwicklung ist außerdem ein erstes allgemeines Compositing-API (siehe [CMP]) entstanden, mit dem sich die größtenteils hier gar nicht genannten Details auch programmtechnisch angehen

lassen.

Schnelles Hardware-Compositing würde eine offensichtliche Lücke beim Aufbau von Hochleistungs-Graphikrechern aus Standardkomponenten ausfüllen. Da fertige Produkte noch nicht erhältlich sind, ist es allerdings fraglich, inwieweit sie sich preislich in den Commodity-Off-The-Shelf-Gedanken einfügen werden.

3.5.3 Multi-Port Frame Buffer

Bei einem Multi-Port Frame Buffer handelt es sich um ein über mehrere Ports an das Verbindungsnetzwerk eines Parallelrechners angeschlossenes Gerät, das neben dem Frame Buffer noch mindestens über die Hardware zur Ansteuerung eines oder mehrerer Displays verfügt. Es ermöglicht allen Knoten eines Parallelrechners einen schnellen Zugang zum Frame Buffer. Während in den 90ern Multi-Port Frame Buffer noch als Extra z.B. zur Connection Machine CM-5 zu erwerben waren, hat ihre Bedeutung seitdem stark abgenommen. In jüngerer Zeit hat IBM unter dem Namen Scalable Graphics Engine (SGE, siehe [KKV+02] und [PJ01]) dieses Konzept in Verbindung mit dem hochauflösenden Display T221 (3840x2400, 22", 204 dpi) wiederbelebt. Nur sehr wenige Graphikkarten können dieses Display direkt ansteuern und wenn, dann nur mit sehr niedriger Bildwiederholrate. Die SGE besitzt genügend mehrfach verschränkten Speicher, um bis zu 16M Pixel doppelt zu puffern, 16 Gigabit Ethernet-Uplinks und acht synchronisierte DVI-Ausgänge. Die einzige wesentliche Operation, die die SGE unterstützt, ist das Eintragen von rechteckigen Pixelbereichen, die über die Uplinks gesendet werden, in ihren Frame Buffer und das Generieren der Ausgangssignale entsprechend dem Inhalt des Frame Buffers. Ein T221-Display lässt sich damit über vier DVI-Links mit dem möglichen Maximum von 41 Hz betreiben. Neuerdings gibt es allerdings auch Graphikkarten, die ebenfalls über mehrere synchronisierte DVI-Ausgänge verfügen und dieses Display direkt ansteuern können.

An Software gehört zur SGE ein speziell angepasster X-Server sowie eine Tunnel-Bibliothek, mit der man direkt in den Frame Buffer schreiben kann.

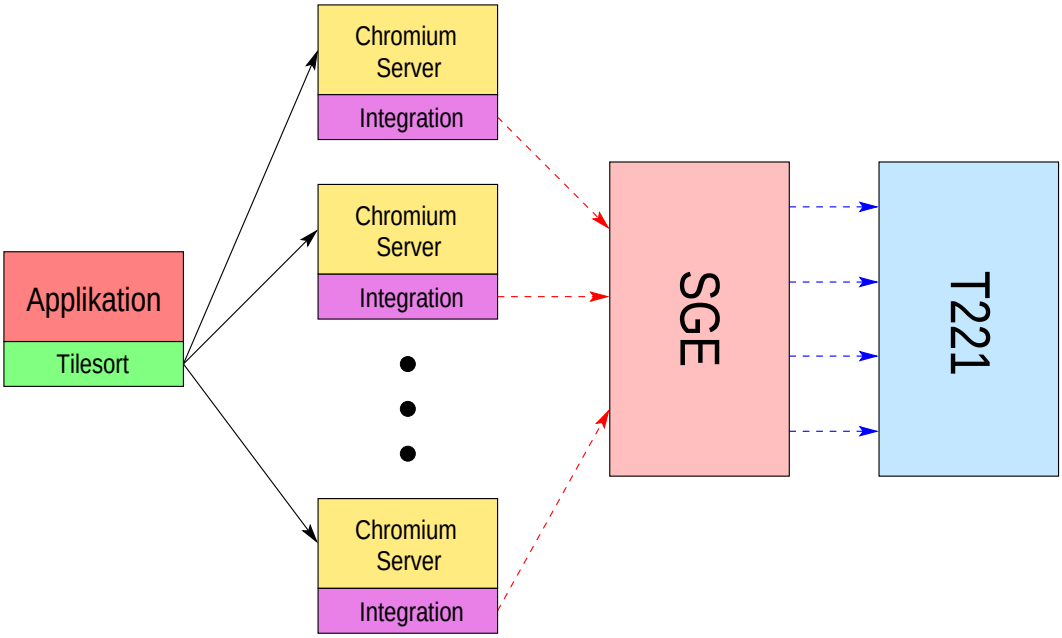

Abbildung 3.12: Ansteuerung eines hochauflösenden T221-Displays mit der SGE und Chromium.

Applikationen benutzen den auf einem Knoten laufenden X-Server zur Darstellung von Fenstern und GUI-Elementen, während das Rendering parallel auf den Knoten ausgeführt wird und die Pixel direkt unter Umgehung des X-Servers über die Tunnel-Bibliothek in den Frame Buffer geschrieben werden. Damit eignet sich die SGE vor allem zur Realisierung von Sort-First-Konfigurationen. Der Begriff Compositing wird aber oft weitergefasst als in Abschnitt 3.2 eingeführt, so dass er z.B. auch *Image Reassembly* umfasst, weshalb die SGE hier im Kontext des Hardware-Compositings erwähnt wird. Auch die im vorigen Abschnitt behandelte Sepia-Hardware bietet Funktionen zum Image Reassembly.

Eine Chromium-Konfiguration, die die SGE zur Ansteuerung des T221-Displays verwendet, ist in Abb. 3.12 zu sehen. Neu ist dabei im Vergleich zur klassischen WireGL-Konfiguration aus Abschnitt 3.4.4 die Integration-SPU. Sie ist von der Render-SPU abgeleitet und überschreibt deren `SwapBuffers`-Methode mit einer Version, die die auf den Rendering-Servern erzeugten Pixel mit `glReadPixels` ausliest und dann out-of-band über die Tunnel-Bibliothek per Gigabit-Ethernet in den Frame Buffer der SGE

schreibt. Die Tilesort-SPU ist entsprechend dem oben genannten Betriebsmodell für die SGE so konfiguriert, dass sie den Bereich eines X-Fensters in einzelne Tiles für die Rendering-Server zerlegt. Nach [KKV+02] lässt sich mit dieser Konfiguration auch nur als Binärcode vorliegende kommerzielle Software direkt benutzen.

Im Vergleich zu einer WireGL-Konfiguration, in der jeder Rendering-Server automatisch für einen festgelegten Teil des Gesamtbildes verantwortlich ist, lassen sich mit der SGE auch leicht Algorithmen realisieren, in denen der Bildraum zur Lastverteilung dynamisch und nicht-regulär zerlegt wird, denn jeder Rendering-Server hat beliebigen Zugriff auf den gesamten Frame Buffer. Auch im Zusammenhang mit Sort-Last-Konfigurationen ließe sich die SGE einsetzen, z.B. wenn das Compositing über den ‚Binary Swap'-Algorithmus erfolgt ist, bei dem jeder der n Renderer zum Schluss ein n-tel des Gesamtbildes hat. Das abschließende Einsammeln des Gesamtbildes würde entfallen, da jeder Renderer direkten parallelen Zugriff auf den Frame Buffer hätte. Ganz generell würden sicher viele Ansätze von der Möglichkeit des direkten Frame Buffer-Zugriffs profitieren können.

Die SGE hat zur Zeit in der Version 3 nach [JKA+03] in etwa die Größe eines 16-Port Ethernet-Switches. Frühere Prototypen (zu sehen in [KKV+02]) waren erheblich voluminöser. Im freien Handel ist die SGE3 ebenso wie Sepia-2a nicht zu erhalten.

Kapitel 4

Ein paralleles Visualisierungssubsystem

Dieses Kapitel beschreibt das Simulationssystem UG zur Lösung partieller Differenzialgleichungen in 2D und 3D. Der Schwerpunkt liegt dabei auf dem parallelen Visualisierungssubsystem, das insbesondere die letzten beiden der in Kapitel 2 genannten Parallelisierungsansätze implementiert, nämlich die parallele Erzeugung geometrischer Objekte und die parallele Bilderzeugung.

4.1 Simulationssystem UG

UG (Abkürzung für Unstructured Grids, vgl. [BBJ+97]) ist ein Framework zur Simulation von Problemen, die sich durch partielle Differentialgleichungen beschreiben lassen. UG wird seit etwa 1994 entwickelt und gepflegt und realisiert den State-of-the-Art seiner Zunft: Parallele adaptive Mehrgitterverfahren auf Distributed Memory Computern.

4.1.1 Allgemeines

Da partielle Differentialgleichungen sich im allgemeinen nicht analytisch lösen lassen, werden in der Praxis numerische Methoden verwendet. Dazu

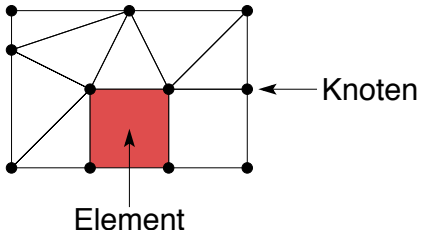

Abbildung 4.1: Unstrukturiertes Gitter in 2D.

ist zuerst das Gebiet, auf dem die Rechnung durchgeführt werden soll, in Teilgebiete, die *Elemente*, zu zerlegen. Als Elemente werden in 2D Dreiecke und Vierecke (vgl. Abb. 4.1), in 3D Tetraeder, Pyramiden, Prismen und Hexaeder verwendet. Unstrukturierte Gitter helfen einmal bei der Approximation komplexer Geometrien und ergeben sich andererseits durch lokale Gitteradaption (siehe nächsten Abschnitt) sowieso. Unbekannte werden meistens in den Knoten angesetzt, möglich ist auch der Ansatz auf Elementen, Elementkanten oder (in 3D) auf Elementseiten. In jedem Fall ist damit eine Diskretisierung des Raumes erreicht, aus der sich nach der Methode der Finiten Differenzen, Finiten Elemente oder Finiten Volumen am Ende auch eine Diskretisierung der partiellen Differentialgleichungen in Form eines linearen Gleichungssystems $Kx = f$ ergibt. Die Matrix K dieses linearen Gleichungssystems ist dünn besetzt, d.h. bei n Unbekannten sind auch nur $O(n)$ Einträge ungleich Null vorhanden, sodass sie sich effizient speichern lässt. Um diese Eigenschaft von K während des Lösungsvorgangs nicht zu zerstören, setzt man iterative Löser ein. Diese haben allerdings die Eigenschaft, nur die (relativ zur Maschenweite) hohen Frequenzen des Fehlers der Iterierten schnell zu reduzieren (*Glätter*). Deshalb wurden Verfahren entwickelt, die auf folgender Heuristik beruhen:

1. Führe einige Schritte des elementaren Verfahrens aus, um den Fehler zu glätten.

2. Beschränke den momentanen Zustand des Problems auf eine Teilmenge der Gitterpunkte (*Grobgitter*) und löse das sich ergebende projizierte Problem.

3. Interpoliere die Lösung auf dem Grobgitter zurück auf das ursprüngliche Gitter und führe noch einige Schritte des elementaren Verfahrens aus.

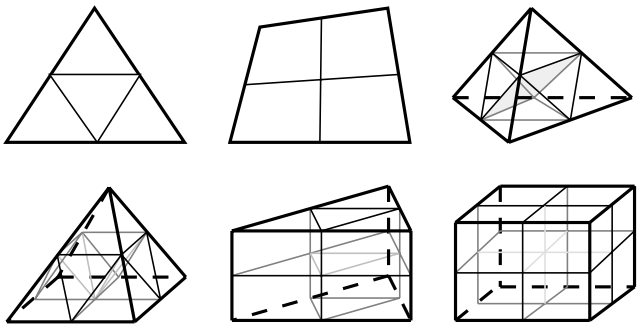

Abbildung 4.2: Reguläre Verfeinerungsregeln in 2D und 3D.

Schritt 1 heißt *Vorglättung*, Schritt 3 *Nachglättung*. Führt man die Schritte 1–3 der Reihe nach aus, erhält man ein Zweigitterverfahren. Wenn man das Verfahren im Schritt 2 rekursiv anwendet, erhält man ein *Mehrgitterverfahren*. Die Rekursion endet dabei auf dem gröbsten der verwendeten Gitter, auf dem man das verbleibende kleine Gleichungssystem dann direkt lösen kann. Mehrgitterverfahren sind in vielen Fällen optimal, d.h. der Rechenaufwand bei n Unbekannten ist von der Ordnung $O(n)$. Vgl. z.B. [Hac93], [BBC+94] und [Bas96].

4.1.2 Gitteradaption

Die im vorigen Abschnitt erwähnte Hierarchie von Gittern wird normalerweise ausgehend von einem Anfangsgitter (*Grobgitter*) iterativ durch Verfeinerung der Elemente (*Gitteradaption*) erzeugt. Die durch Verfeinerung einer Gitterebene neu entstandenen Elemente bilden die nächste Gitterebene. Die regulären Regeln für die Elementverfeinerung in UG sind in Abb. 4.2 für 2D (Dreieck und Viereck) und 3D (Tetraeder, Pyramide, Prisma und Hexaeder) skizziert. Wendet man diese Regeln auf alle Elemente eines Gitters an (*uniforme Verfeinerung*), so erhält man sehr schnell sehr viele Elemente und entsprechend viele neue Unbekannte. In 2D enthält eine durch uniforme Verfeinerung entstandene neue Gitterebene viermal und in 3D achtmal so viele Elemente wie die alte Gitterebene, was den Rechenaufwand um denselben Faktor erhöht. Da die Lösung des zu berechnenden Problems üblicherweise nicht auf dem gesamten Rechengebiet „interessant“ ist, d.h. Phänomene aufweist, die eine hohe räumliche Auflösung zu ihrer

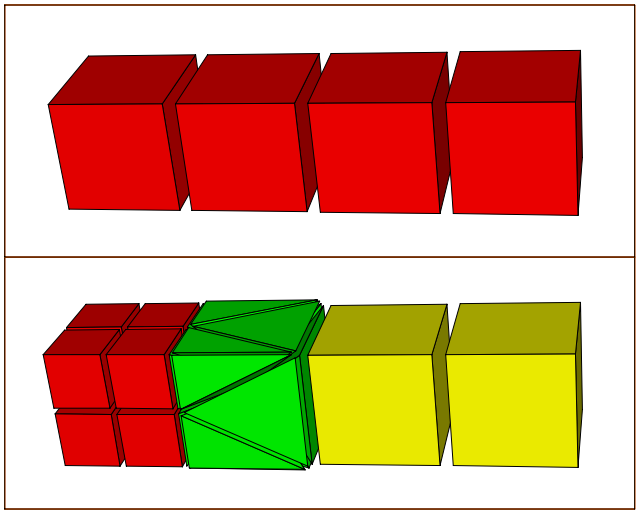

Abbildung 4.3: Beispiel für lokale Gitteradaption. Oben: Grobgitter, unten: nächste Verfeinerungsstufe.

Erfassung erfordern (Singularitäten, Wirbel, ...), liegt es nahe, die Gitteradaption (und den sich ergebenden zusätzlichen Rechenaufwand) auf solche „interessanten“ Bereiche zu beschränken (*lokale Gitteradaption*). Dazu ist ein problemspezifischer *Fehlerschätzer* notwendig, der solche Bereiche erkennen kann. Abb. 4.3 zeigt hierzu ein Beispiel. Für ein Grobgitter aus vier Hexaedern hat der Fehlerschätzer entschieden, nur das am weitesten links liegende Element zu verfeinern. Nachdem dieses Element regulär verfeinert wurde, ist allerdings das Oberflächengitter (definiert durch die jeweils höchste Verfeinerungsstufe an jedem Ort) nicht mehr *konsistent*. Dabei heißt ein Gitter konsistent, wenn der Schnitt zweier verschiedener Elemente entweder leer ist oder aus einem Knoten oder einer Kante oder (in 3D) einer Seite besteht. Da diese Eigenschaft von etlichen Diskretisierungsverfahren benötigt wird, wird sie durch an den eigentlichen Verfeinerungsbereich anschließende irreguläre Verfeinerung wieder hergestellt (*Grid Closure*, in Abb. 4.3 grün markiert). Schließlich werden noch einige daran anschließende Elemente des gröberen Gitters (in Abb. 4.3 gelb markiert) auf die nächste Ebene kopiert, um die guten Kovergenzeigenschaften des Mehrgitterlösers zu erhalten. Die neue Gitterebene muss dabei nicht das ganze Gebiet überdecken. In [Lan01] ist eine detailliertere Beschreibung der Zusammenhänge zu finden.

Es sei ebenfalls noch erwähnt, dass bei instationären Problemen auch eine Vergröberung des Gitters möglich ist, nämlich dann, wenn der Fehlerschätzer anzeigt, dass sich die „interessanten“ Bereiche der Lösung an eine andere Stelle bewegt haben.

4.1.3 Architektur von UG

Auf der höchsten Abstraktionsebene besteht ein auf UG aufbauendes Programm aus drei Komponenten, vgl. auch [BBJ+97] und [LW04].

UG-Bibliothek

Die UG-Bibliothek stellt allgemeine, vom konkreten Problem unabhängige Mechanismen bereit. Dazu gehören geometrische und algebraische Datenstrukturen und darauf aufbauend Prozeduren zur Manipulation von Gittern, numerische Algorithmen, ein User Interface und nicht zuletzt diverse Visualisierungsmöglichkeiten.

Problemklassenbibliothek

Die Problemklassenbibliothek liefert Diskretisierung, Fehlerschätzer und eventuell einige problemspezifische Löser für eine konkrete partielle Differentialgleichung.

Applikation

Der Applikationsteil stellt die für ein konkretes Problem benötigten Parameter bereit. Das sind z.B. eine Beschreibung des Rechengebiets, der Randbedingungen, Koeffizientenfunktionen, etc.

Von der Gesamtzahl der Codezeilen eines mit UG realisierten Programms entfallen üblicherweise bis zu 90% auf die problemunabhängige UG-Bibliothek, die vom Programmierer nicht modifiziert werden muss.

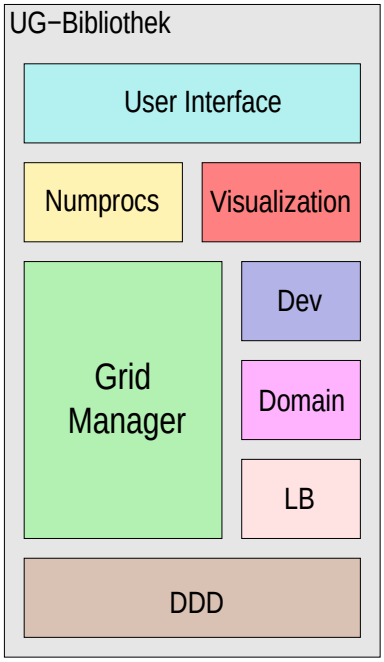

Abbildung 4.4: Subsysteme der UG-Bibliothek.

Aufbau der UG-Bibliothek

Die UG-Bibliothek zerfällt in zahlreiche Subsysteme, von denen die wichtigsten in Abb. 4.4 skizziert sind und im folgenden kurz beschrieben werden.

User Interface

Das User Interface von UG besteht im wesentlichen aus einer Shell-artigen Skriptsprache, über die eine Simulation initialisiert und gesteuert werden kann. Ebenfalls über diese Skriptsprache wird das Visualisierungssubsystem angesprochen. Daneben lassen sich Visualisierungen auch interaktiv manipulieren (Zoom, Pan, Rotate, ...).

Visualization

Das Visualisierungssubsystem wird in Abschnitt 4.2 detailliert.

Device Manager (Dev)

Der Device Manager stellt eine einheitliche Schnittstelle zur Ein-/Ausgabe bereit. Ein Default Device namens *Screen* erlaubt zumindest

zeichenbasierte Ein-/Ausgae für das User Interface. Das Screen Device ist für die C-Bibliothek, X11 und MaxOS implementiert. Für X11 und MacOS verfügt das Screen Device auch über eine interaktive graphische Ein-/Ausgabe. Daneben gibt es noch ein Postscript-, ein Metafile- und ein PPM[1]-Device, die nur Ausgabe erlauben.

Numprocs

Dieses Subsystem stellt numerische Algorithmen bereit, etwa BLAS-Routinen, lineare und nichtlineare Löser, Standardfehlerschätzer, Operatoren zum Transfer zwischen den Ebenen eines Mehrgitters und vieles mehr. Zusätzlich sind hier noch Hilfsmittel zur Diskretisierung (Berechnung von Teilkontrollvolumen, Quadraturformeln, ...) untergebracht.

Grid Manager

Der Grid Manager ist für die Verwaltung der Datenstrukturen für die UG-Gitterhierarchie und der zugehörigen ‚Sparse Matrix'-Datenstruktur zuständig. Dazu gehört insbesondere die Durchführung der Gitteradaption auf unstrukturierten Gittern, wie sie in Abschnitt 4.1.2 angedeutet ist.

Domain

Das Domain-Subsystem erlaubt die Beschreibung des Rechengebietes, der Randbedingungen und von Koeffizientenfunktionen für das Innere des Rechengebietes. Das Rechengebiet wird dabei durch eine Beschreibung seines Randes definiert, die entweder analytisch durch Abbildungen gegeben ist, die einzelne Patches beschreiben, oder in diskreter Form mittels Dreiecksfacetten.

Load Balancing (LB)

Das ‚Load Balancing'-Subsystem kümmert sich um die Verteilung einer Mehrgitterhierarchie auf eine gegebene Anzahl von Prozessoren. Ziel ist, die Rechenlast für alle Prozessoren möglichst gleich zu halten, und das Kommunikationsaufkommen zu minimieren.

[1] Portable Pixmap: ein einfaches Dateiformat für Pixelgraphik

DDD

Das Subsystem DDD (Dynamic Distributed Data, siehe [BB94]) stellt das parallele Programmiermodell zur Verfügung, mit dessen Hilfe die oben genannten Subsysteme parallelisiert sind. DDD's Programmiermodell beruht auf der Vorstellung eines verteilten Graphen und ermöglicht durch diese Abstraktion überhaupt erst die Parallelisierung von UG, die sonst aufgrund der Komplexität der Aufgabe mit einem einfachen Message Passing-Ansatz sicher fehlgeschlagen wäre.

4.2 UGs Visualisierungssubsystem

4.2.1 Randbedingungen

Aus der Tatsache, dass das UG-Visualisierungssubsystem nur Teil eines Gesamtsystems ist, ergeben sich einige Anforderungen bzw. Einschränkungen an seine Arbeitsweise. Dies betrifft insbesondere den Hauptspeicherverbrauch. Während separate Visualisierungsprogramme zum Postprocessing die gesamten Ressourcen eines Rechners für sich alleine haben (und davon auch ausgiebig Gebrauch machen), sollte die UG-Graphik nur moderate Speicheranforderungen stellen. Das bedeutet einmal die Verwendung eines prozeduralen Interfaces zum Zugriffs auf die zu visualisierenden Daten (`GetFirstXXX`, `GetNextXXX`) und schließt zum anderen das Kopieren in eigene Datenstrukturen aus. Ebenfalls nicht statthaft sind unter diesem Aspekt Algorithmen, die in großem Stil neue geometrische Objekte einführen, die außerhalb der Graphik keinen Nutzen bringen, z.B. durch Zerlegung von Elementen.

Da die Aufgabe von UG in erster Linie darin besteht, eine numerische Simulation durchzuführen, sind auch dem Zeitverbrauch durch die Visualisierung gewisse Grenzen gesetzt. Einige Sekunden für eine Visualisierung im Vergleich zu einigen Minuten etwa für einen Zeitschritt der Simulation sind akzeptabel. Komplexere Visualisierungstechniken wie z.B. Volume Rendering oder komplexere Renderingmethoden wie z.B. Raytracing eigen sich daher kaum. Elementare Visualisierungstechniken zusammen mit Po-

lygonrendering sind hingegen gut geeignet.

Zum Thema Load Balancing bietet sich der ‚Intelligent Memory'-Ansatz an, d.h. der Speicher der Rechnerknoten wird in erster Linie eben gebraucht, um die Daten (zu visualisierende und andere) aufzunehmen. Die CPU kommt nur ins Spiel, wenn diese Daten (oder auch nur Teile davon) visualisiert werden sollen. Oder anders formuliert: Eine eigene Lastverteilung für die Graphik findet im Vertrauen darauf, dass die für die Rechnung benutzte auch für die Visualisierung gut ist, nicht statt. Allerdings wäre eine eigene Lastverteilung für die Graphik auch wegen der Komplexität des Themas und der untergeordneten Rolle des Visualisierungssubsystems im Gesamtkontext nicht zu rechtfertigen.

Schließlich ist noch die Zielplattform von UG zu beachten: große Parallelrechner an Universitäten oder Rechenzentren. Allen solchen Installationen ist gemein, dass sie keine Unterstützung für den Visualisierungsprozess in Form von Hardware, wie z.B. Graphikkarten auf den Rechnerknoten, bieten. Soweit also das Rendering ebenfalls parallel stattfinden soll, bleibt nur reines Softwarerendering übrig.

4.2.2 Aufbau des UG-Visualisierungssubsystems

Datenstrukturen

Dieser Abschnitt gibt einen kurzen Überblick über die wichtigsten Datenstrukturen und Begriffe der UG-Graphik.

Plotobject

Plotobjects sind abstrakte Beschreibungen geometrischer Objekte, für die eine Mehrgitterhierarchie als Datenbasis dient. Beispiele für Plotobjects und damit auch für die in der UG-Graphik verfügbaren Visualisierungsmethoden sind in Tab. 4.1 zu finden.

Plotobject	Beschreibung
Grid	Gitterdarstellung als Netz (in 3D optional mit Schnitt)
EScalar	Skalarfeld in Farbdarstellung (in 3D auf Schnittfläche)
EVector	Vektorfeld in Pfeildarstellung (in 3D auf Schnittfläche)
Isosurface	Isofläche durch skalares Feld (nur 3D)

Tabelle 4.1: Beispiele für Plotobjects/Visualisierungsmethoden in UG.

Viewedobject

Viewedobjects sind von Plotobjects abgeleitet und erweitern diese um alle erforderlichen Parameter, die für das Rendering erforderlich sind. In der Terminologie von Abschnitt 1.2.1, Seite 7 definieren Viewedobjects eine Szene.

Picture

Pictures sind von Viewedobjects abgeleitet. Sie verbinden ein Viewedobject mit einer konkreten „Zeichenstelle“ in einem Ugwindow (s.u.), das Platz für mehrere Pictures bieten kann.

Ugwindow

Ein Ugwindow beschreibt eine konkrete Ausgabemöglichkeit für Pictures. Beispiele sind ein Fenster unter X11 oder MacOS oder etwa eine geöffnete Datei zur Ausgabe von Postscript.

Work

Die Datenstruktur Work beschreibt die „Arbeiten“, die man an einem Plotobject, Viewedobject bzw. Picture verrichten lassen kann. In objektorientierter Terminologie würde man von Methoden sprechen. Beispiele sind *Draw Work* (rendert ein Viewedobject), *Findrange Work* (Ermittelt das Intervall, in dem eine skalare Größe liegt) oder *Select Element* (Erlaubt das interaktive Markieren eines Elements).

Drawingobject

Drawingobjects entstehen als Ergebnis des Aufrufs einer Draw Work-Methode. Es handelt sich dabei um serialisierte Zeichenbefehle z.B. für Linien und Polygone. Durch das abschließende Rendering der Drawingobjects ergibt sich schließlich die Visualisierung.

Modulstruktur

Dieser Abschnitt gibt die Organisation des Visualisierungssubsystems in Modulen wieder und skizziert deren Funktionalität.

Window/Picture Manager

Aufgabe des Window/Picture Managers ist die Verwaltung der im letzten Abschnitt erwähnten Ugwindows und Pictures (Öffnen, Schließen, Platzieren, ...), sowie die Initialisierung und Konfiguration von Plotobjects und Viewedobjects.

Work On Picture

Das Work On Picture-Modul implementiert die Aktionen, die für ein Paar aus Plotobject/Viewedobject/Picture und Work auszuführen sind. Im Fall einer Draw Work-Methode (darauf wollen wir uns im folgenden beschränken) wird dazu über eine ausgewählte Teilmenge der Elemente der Gitterhierarchie mit Hilfe des prozeduralen Interfaces iteriert und als Resultat Drawingobjects produziert, die im sequentiellen Fall direkt verwertet werden. Im parallelen Fall werden sie entweder eingesammelt und vom Masterprozess über den Device Manager (siehe Abschnitt 4.1.3) für das betreffende Gerät umgesetzt (Methode der parallelen Erzeugung geometrischer Objekte, Abschnitt 4.2.3) oder über das Bullet-Modul (s.u.) direkt parallel gerendert (Methode der parallelen Bilderzeugung, Abschnitt 4.2.4). Generell sieht die Hauptschleife dieses Moduls wie in Abb. 4.5 aus.

```
for (alle relevanten Elemente E)
{
    Evaluate(E, DO);
    Execute(DO);
}
```

Abbildung 4.5: Hauptschleife in Work On Picture: Evaluate erzeugt aus Element `E` ein Drawingobject `DO`, Execute „führt“ es aus.

Bullet

Das Bullet-Modul enthält einen einfachen Softwarerenderer, der Linien und Polygone rendert. In 3D wird zur Lösung des ‚Hidden Surface'-Problems der z-Buffer-Algorithmus verwendet. Dieses Modul wird für die Methode der parallelen Bilderzeugung verwendet.

4.2.3 Methode der parallelen Erzeugung geometrischer Objekte

Der erste im UG-Visualisierungssubsystem implementierte Ansatz ist die parallele Erzeugung geometrischer Objekte, d.h. Filerungs- und Abbildungsschritt (vgl. Abschnitt 2.1.3, Seite 23) werden auf dem Parallelrechner ausgeführt, das Rendering hingegen findet sequentiell statt. Unterstützt werden von dieser Methode die Ausgabe über X11 oder in ein Meta- oder Postscriptfile. Da alle drei Ausgabemöglichkeiten nur 2D-Graphik unterstützen, ist sowohl die Geometrieverarbeitung (erster Teil der (3D-)Renderingpipeline aus Abschnitt 1.2.1, Seite 7) als auch eine Lösung zum ‚Hidden Surface'-Problem (vgl. Abschnitt 1.2.2, Seite 9) dem abschließenden (2D-)Rendering vorzuschalten. Während die selbstauszuführende Geometrieverarbeitung trivial ist, macht eine parallele Lösung zum ‚Hidden Surface'-Problem etwas mehr Mühe.

Lösung des ‚Hidden Surface'-Problems

Der gewählte Weg verfolgt die Lösung des ‚Hidden Surface'-Problems mit Hilfe eines ‚List Priority'-Verfahrens, d.h. es wird von „hinten nach vorne" gezeichnet, vgl. Abschnitt 1.2.2, Seite 9. Dazu seien alle Elemente auf allen Prozessoren gemäß einer globalen Prioritätsliste für das gesamte Gitter nummeriert (*Plot ID*). Dann kann man folgenden Algorithmus verwenden: Jeder Prozessor durchläuft lokal die Hauptschleife in Work On Picture aus Abb. 4.5 in aufsteigender Reihenfolge der Plot IDs seiner Elemente. An die Drawingobjects, die von der Evaluate-Funktion erzeugt werden, wird die Plot ID des erzeugenden Elements angehängt. Die Execute-Funktion führt jetzt die Zeichenbefehle in den Drawingobjects nicht mehr direkt aus, sondern realisiert ein Übertragungsprotokoll, das einmal das Einsammeln der

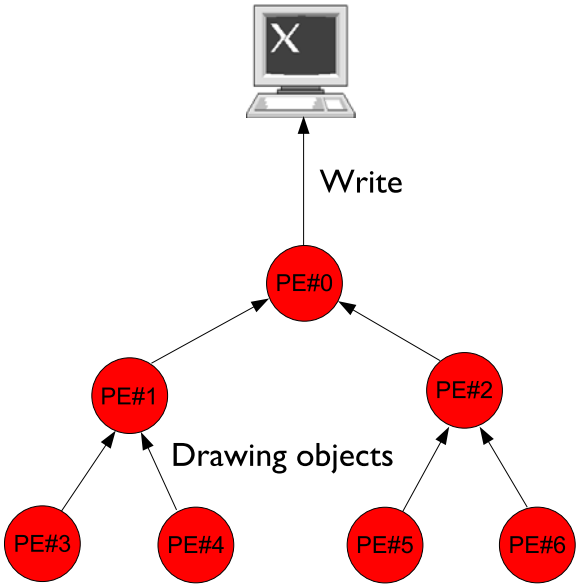

Abbildung 4.6: Baum zur Übertragung der Drawingobjects à la Mergesort.

Drawingobjects von allen Prozessoren auf dem Master-Prozessor erledigt und zum anderen dafür sorgt, dass sie dort auch in global aufsteigender Plot ID-Reihenfolge ankommen. Dazu gibt in einer Baumstruktur wie in Abb. 4.6 jeder Prozessor das Drawingobject weiter, das von den von ihm erzeugten und von ihm empfangenen mit der kleinsten Plot ID markiert ist. Das Verfahren entspricht dem Verschmelzen von Läufen in Mergesort (siehe z.B. [Knu73b]). Der Master-Prozessor kann dann die Drawingobjects über den Device Manager „ausführen“, d.h. es werden entweder X11-Aufrufe abgesetzt oder es wird in ein Meta- oder Postscriptfile geschrieben. Implementierungsdetails und Optimierungen wie das Packen von Drawingobjects zu größeren Einheiten sind in [Lam97] zu finden.

Berechnung der Prioritätsliste

Zur Berechnung einer Proritätsliste für ein konvexes Gitter verwendet man üblicherweise den MPVO-Algorithmus (Meshed Polyhedra Visibility Ordering, siehe [Wil92]). Dazu wandelt man das Gitter in einen Graphen um, in dem die Elemente des Gitters die Knoten sind und gerichtete Kanten zwischen den Knoten benachbarter Elemente von jeweils dem Knoten ausge-

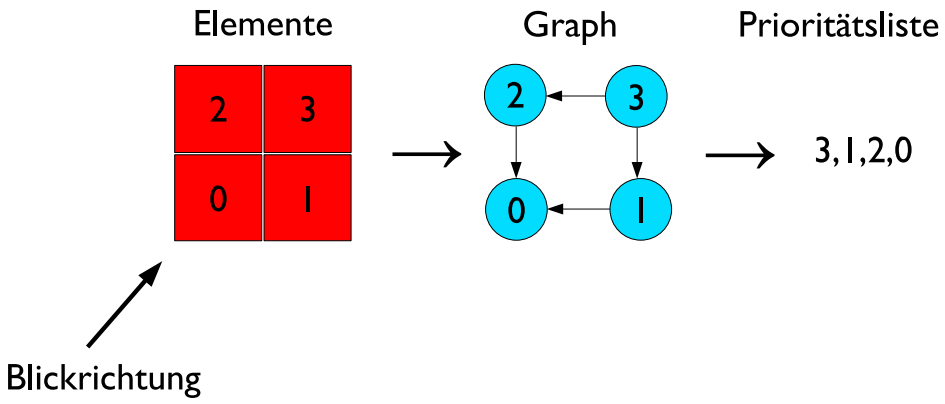

Abbildung 4.7: Beispiel zum MPVO-Algorithmus.

hen, dessen Element die gemeinsame Seite als Vorderseite hat. Die dadurch definierte Relation lässt sich als „muss vorher gezeichnet werden“ interpretieren. Der Graph ist also von den Projektionsparametern abhängig. Eine Prioritätsliste erhält man durch topologisches Sortieren (siehe [Knu73a]) dieses Graphen, d.h. es ist ein lineare Anordnung der Knoten zu bestimmen, bei der stets ein Knoten bzw. Element e_1 vor e_2 steht, wenn es eine Kante (e_1, e_2) im Graphen gibt. Abb. 4.7 zeigt ein einfaches Beispiel. Man beginnt mit Knoten, in die keine Kante einmündet ($Eingangsgrad = 0$), die man an den Anfang der Liste schreibt. Danach vermindert man den Eingangsgrad aller Knoten, die mit den gerade weggeschriebenen mit einer Kante verbunden sind um eins und wendet das Verfahren iterativ an. Der MPVO-Algorithmus benötigt linearen Aufwand in der Anzahl der Elemente. Der Speicherverbrauch ist minimal, da in UG die Informationen über Elementnachbarschaften sowieso schon vorhanden sind und man für die Information, welche Elementseiten Vorder- bzw. Rückseiten sind, maximal sechs Bit pro Element braucht.

Für nichtkonvexe Gitter funktioniert MPVO nicht, da nur Elementnachbarschaften berücksichtigt werden und diese über „Löcher“ und „Einbuchtungen“ die „muss vorher gezeichnet werden“-Relation nicht vermitteln. Die meisten bekannten Algorithmen versuchen daher, den MPVO-Graphen vor dem topologischen Sortieren durch zusätzliche Kanten so zu erweitern, dass auch Prioritäten über diese „Löcher“ und „Einbuchtungen“ hinweg korrekt wiedergegeben werden. Dabei ist die Beobachtung wichtig, dass zusätzliche

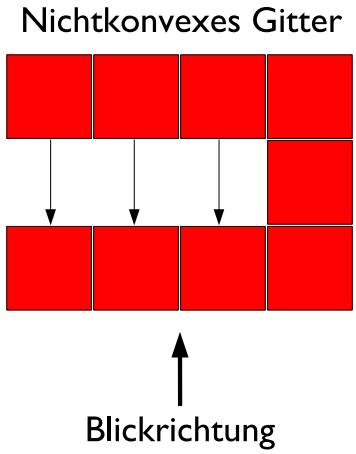

Abbildung 4.8: Zusätzliche Kanten erweitern den MPVO-Graphen für nichtkonvexe Gebiete.

Kanten nur von Randelementen mit einer Vorderseite auf Randelemente mit einer Rückseite nötig sind, vgl. Abb. 4.8. In [SMW98] etwa ist ein Algorithmus namens XMPVO angegeben, der durch Prüfen aller Kombinationen von Vorder- und Rückseiten auf dem Rand die zusätzlichen Kanten in den Graphen einfügt. Entsprechend ist der Aufwand für ein Gitter mit n Elementen und b Randseiten $O(n + b^2)$. Die UG-Implementierung aus [Lam97] realisiert dieselbe Idee, testet aber nicht kombinatorisch jedes Paar von Vorder- und Rückseiten[2]. Dazu werden die *Bounding Boxen* (kleinste umfassende achsenparallele Rechtecke) der Projektionen der Randrückseitenseiten auf die Bildebene in eine geometrische Datenstruktur namens *Box Sort*, vgl. [Hou87], eingefügt, die schnelle Bereichsanfragen unterstützt, d.h. zu einer gegebenen Box (*Query Box*) werden alle mit dieser überlappenden Boxen (*Target Boxen*) gefunden. Die zusätzlichen Kanten ergeben sich, indem man nacheinander die Bounding Boxen der Projektionen der Randvorderseiten als Query Box verwendet. Zwischen den zur Query Box und den Target Boxen gehörigen Elementen sind potentiell neue Kanten einzufügen. Ob dies wirklich der Fall ist, ist noch zu überprüfen,

[2]Tatsächlich erwähnt [SMW98] auch eine Möglichkeit, die zusätzlichen Kanten mit Aufwand $O(b \log b)$ einzufügen. Allerdings wurde diese von den Autoren wegen der Komplexität ihres Vorschlags nie implementiert. Der Folgeartikel [CKM+99] verfolgt eine ähnliche Idee wie hier beschrieben, verwendet allerdings einen *Binary Space Partitioning Tree*, was für UG wegen des Zerschneidens von Elementen nicht in Frage kommt.

da für die Bereichsanfragen ja nur die Bounding Boxen verwendet werden. Für eine Bereichsanfrage ist der Aufwand $O(\log b)$, wobei $O(b)$ solche Anfragen durchzuführen sind. Insgesamt hat man also einen Aufwand von $O(b \log b)$ für das Einfügen der zusätzlichen Kanten. Ebenfalls $O(b \log b)$ ist der Aufwand für das Aufbauen der Box Sort-Datenstruktur. Der Gesamtalgorithmus, der hier mit MPVO++ bezeichnet werden soll, ist also von der Ordnung $O(n + b \log b)$ bzw. $O(n)$, wenn man $b \approx n^{2/3}$ ansetzt. Auf einem Rechner mit PPC 603-Prozessor (233 MHz) können Gitter mit ca. 130.000 Elementen/s sortiert werden, vgl. [Lam97].

Hierarchisches Sortieren einer Mehrgitterhierarchie

In einer Mehrgitterhierarchie, die wie in Abschnitt 4.1.2 beschrieben entstanden ist, nehmen die durch Verfeinerung eines Elements sich ergebenden Elemente (*Söhne*) denselben Raum ein wie dieses. Das lässt sich zur Berechnung der Prioritätsliste ausnutzen, da es genügt, die Söhne eines Elements innerhalb diesem anzuordnen. Mann muss den MPVO++-Algorithmus also nur auf das gröbste Gitter anwenden, für die Söhne der Elemente auf der nächsten Verfeinerungsstufe kann man den MPVO-Algorithmus verwenden, da Elemente stets konvex sind. So erhält man iterativ schließlich eine Prioritätsliste für das gesamte Mehrgitter.

Parallelisierung

Weder der MPVO- noch der MPVO++-Algorithmus bietet erkennbare Ansätze für eine Parallelisierung nach dem SPMD-Modell. Allenfalls eine moderat parallele Implementierung auf Shared Memory-Rechnern scheint möglich. Die UG-Graphik sammelt deshalb für MPVO++ einen das Grobgitter beschreibenden Graphen auf dem Master-Prozessor ein, führt die Sortierung dort sequentiell aus und verteilt das Ergebnis dann. Das ist akzeptabel, da ein Grobgitter nur aus relativ wenigen Elementen besteht und der MPVO++-Algorithmus hinreichend schnell ist. Für das hierarchische Sortieren per MPVO ist zu beachten, dass jeweils nur die Söhne eines Elements sortiert werden müssen. Für jeden Satz Söhne, die aus demsel-

ben Element entstanden sind, kann die Sortierung also unabhängig und parallel erfolgen, falls sie auf demselben Prozessor liegen. Dies stellt den Normalfall dar, da auch für die numerische Simulation eine Lastverteilung mit räumlich zusammenhängenden Partitionen wünschenswert ist. Falls diese Voraussetzung doch nicht erfüllt ist, z.B. auf einer Partitionsgrenze, dann wird wieder auf einem der beteiligten Prozessoren der MPVO-Graph eingesammelt, dort die Sortierung ausgeführt und das Ergebnis verteilt. Der was die Anzahl der zu bearbeitenden Elemente angeht größte Teil der Bestimmung der Priotitätsliste läuft also im wesentlichen parallel ab. Implementierungsdetails sind wieder in [Lam97] zu finden.

4.2.4 Methode der parallelen Bilderzeugung

Der zweite im UG-Visualisierungssubsystem implementierte Ansatz ist die parallele Bilderzeugung, d.h. alle drei Schritte der Visualisierungspipeline (Filterung, Abbildung und Rendering, vgl. Abschnitt 2.1.4, Seite 24) werden auf dem Parallelrechner ausgeführt. Es entsteht ein komplettes aufgerastertes Bild, das sich entweder über das X11-Device per `XPutImage` ausgeben oder über das PPM-Device in eine Datei schreiben lässt.

Wahl der parallelen Renderingmethode

Unter den in Abschnitt 4.2.1 aufgeführten Randbedingungen sieht die Auswahl unter den in Kapitel 3 vorgestellten Möglichkeiten zum parallelen Rendering so aus: Sort-First scheidet aus, da es eine eigene Lastverteilung erfordern würde. Sort-Middle und Sort-Last kommen prinzipiell in Frage. Beide können mit jeder beliebigen Lastverteilung arbeiten. Wie der Vergleich in Kapitel 3 zwischen PGL (Abschnitt 3.4.2) und PMesa (Abschnitt 3.4.3) gezeigt hat, bieten Software-Implementierungen von Sort-Middle gegenüber solchen von Sort-Last keine besonderen Vorteile. Die Fähigkeit von Sort-Middle auch in der Bildauflösung zu skalieren, da auch der Bildspeicher über die Prozessoren verteilt ist, ist nur sekundär; dies ist definitiv keines der Ziele der UG-Graphik. Demgegenüber überzeugt Sort-Last durch die Einfachheit und Klarheit des Ansatzes und der Implementierung, ganz im

Gegensatz zu Sort-Middle. UG verwendet deshalb einen Sort-Last-Ansatz konzeptionell ganz ähnlich dem von PMesa. Das Compositing findet über das Verbindungsnetzwerk des Parallelrechners in Software statt.

Implementierung

Das Bullet-Modul (vgl. Abschnitt 4.2.2) implementiert sowohl einen Softwarerenderer für Linien und Polygone mit z-Buffering zur Lösung der ‚Hidden Surface'-Problems als auch das Compositing in 779 Zeilen inklusive Kommentaren. Das deckt bereits den allergrößten Teil der Funktionalität ab, die von der UG-Graphik benötigt wird. (Ausnahme: keine Textausgabe.) Pro Pixel werden je zwei Byte für die Pixelfarbe (ein Byte Index in die Farbtabelle des Ausgabegerätes entsprechend UGs Device-Abstraktion, ein Byte für die Intensität, in der diese Farbe gezeichnet werden soll) und vier Byte für den z-Wert verwendet. Multipliziert mit der Bildauflösung ergibt sich daraus der einzige zusätzliche Speicherbedarf pro Prozessor.

Für das Compositing über z kommt das in Abschnitt 3.5.1, Seite 54 beschriebene Log-Reduce Tree Compositing zum Einsatz. Wie das folgende Kapitel zeigen wird, erreicht man damit auf aktuellen Parallelrechnern mit hinreichend schnellem Verbindungsnetzwerk bereits ohne jede weitere Optimierung Compositing-Zeiten im Subsekundenbereich.

Die Integration des Softwarerenderers ist ebenfalls höchst einfach. Die in Abb. 4.5 skizzierte Hauptschleife des ‚Work On Picture'-Moduls (vgl. 4.2.2) ist exakt die gleiche wie im sequentiellen Fall und lediglich mit einem Aufruf der Compositing-Funktion und der Ausgabe des Bildes abzuschließen.

Kapitel 5

Vergleich der Parallelisierungsansätze

In diesem Kapitel werden die drei prinzipiellen Ansätze zur parallelen Visualisierung, wie sie in Kapitel 2 vorgestellt wurden, anhand eines Experimentes innerhalb von UG miteinander verglichen. Als Vertreter der Extract-basierten Methode kommt eine Anbindung von pV3 an UG zum Einsatz, die beiden im letzten Kapitel vorgestellten Methoden des UG-Visualisierungssubsystems repräsentieren die parallele Erzeugung geometrischer Objekte bzw. die parallele Bilderzeugung.

5.1 Das Experiment

Das für den Vergleich der Visualisierungsmethoden benutzte Experiment ist durch eine 3D-Variante des schon in Abschnitt 2.1.2, Seite 20 erwähnten Fivespot-Modellproblems, das Buckley-Leverett-Problem (siehe [BL42]), motiviert. Bei diesem strömt in einen mit Öl gefüllten quaderförmigen Kanal an einem Ende Wasser ein, das das Öl verdrängt. Der Verdrängungsprozess ist durch die Dichten und die Viskositäten der beiden Phasen bestimmt. Unter geeigneten Bedingungen können sich aus kleinen Oberflächenstörungen am Phaseninterface einzelne Ausstülpungen herausbilden (siehe [KF88]). Dieses Phänomen ist als viskoses Fingering bekannt. Im Sinne von Abschnitt 4.1.2, Seite 68 sind diese Finger das „interessante“

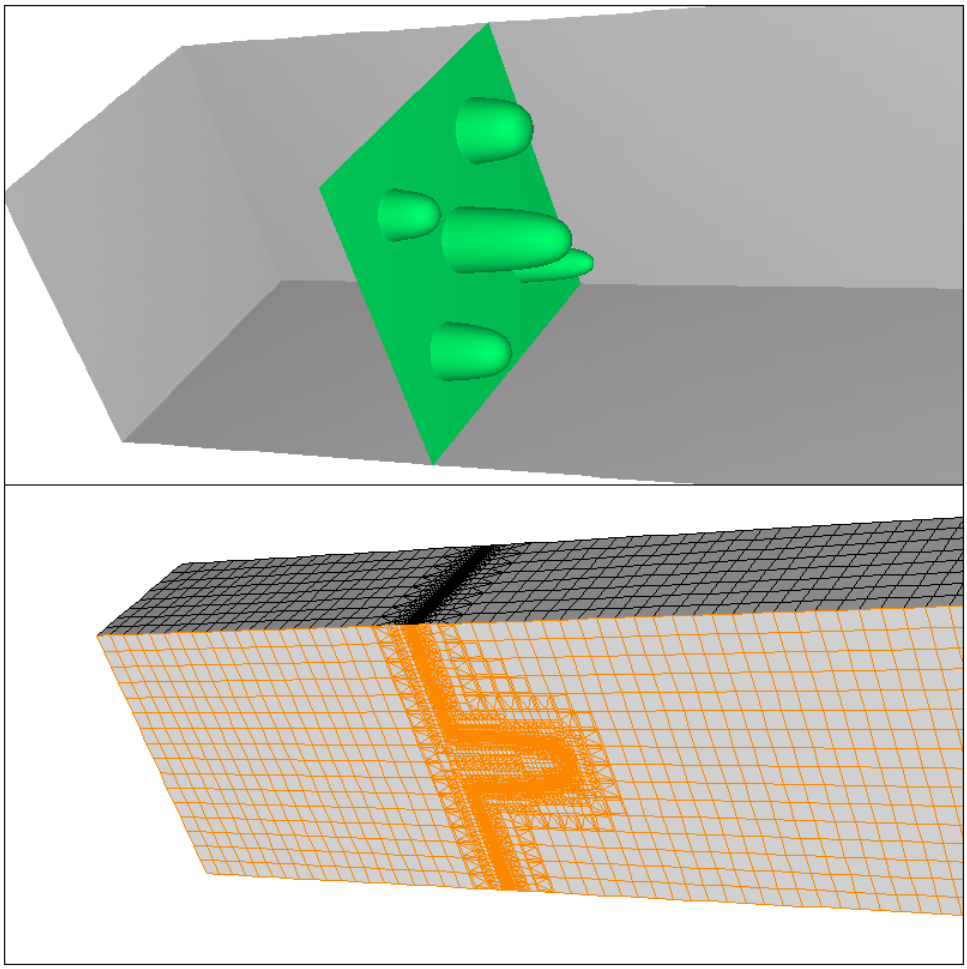

Abbildung 5.1: Visualisierung des Experiments mit UG. Oben: vorgegebene Front, unten: Schnitt durch entsprechend der Front adaptiertes Gitter.

Phänomen, das mittels Gitteradaption aufgelöst und auf das die Rechenarbeit konzentriert werden soll. Obwohl dieses Problem schon erfolgreich mit UG gerechnet wurde, vgl. z.B. [Lan01], ist der Aufwand in Puncto Rechenzeit nicht unerheblich. Auch das Finden von Einstellungen, insbesondere für den Fehlerschätzer, die für alle Prozessorzahlen zu einem bestimmten Zeitpunkt eine bis auf die Gitterauflösung vergleichbare Konfiguration ergeben, ist schwierig. Daher wurde ein Experiment entworfen, in dem eine dem Buckley-Leverett-Problem vergleichbare Situation nicht berechnet sondern vorgegeben wird. Dazu beschreibt eine Funktion $x = \phi(y, z)$ (die x-Achse zeigt in den Kanal hinein) analytisch eine Front, an der sich Finger gebildet

haben. Vermittels $\Phi(x, y, z) = x - \phi(y, z)$ lässt sich damit eine skalares Feld im gesamten Kanal definieren, wobei $\Phi = 0$ gerade der vorgegeben Front entspricht. Diese Bedingung gibt sowohl das Kriterium für den Indikator („Fehlerschätzer") zur Gitteradaption als auch die Visualisierungsaufgabe vor: eine Isofläche durch das gegebene Feld zum Wert $\lambda = 0$, siehe Abb. 5.1.

Als Lastverteilung ist in allen Fällen eine statische gewählt, die Partitionen bestehen aus „Balken", die in Längsrichtung des Kanals angeordnet sind. Das ist zumindest für eine Front ohne Finger optimal und zeigt mit Fingern auch die Auswirkungen des ‚Intelligent Memory'-Ansatzes, vgl. Abschnitt 4.2.1, Seite 73.

Das Experiment läuft jeweils folgendermaßen ab: Das Grobgitter aus vier Hexaedern in Reihe wird viermal uniform verfeinert. Dabei entstehen in der yz-Ebene 16x16 Elemente — ausreichend, um die oben angegebene Lastverteilung für bis zu 256 Prozessoren zu realisieren. Danach wird mit Hilfe des Indikators lokal verfeinert, von dreimal (ein Prozessor) bis siebenmal (256 Prozessoren). Die Prozessorzahl wird mit jedem zusätzlichen Level vervierfacht, entsprechend der Tatsache, dass sich dabei die Zahl der Elemente jeweils ebenfalls vervierfacht, da exakt entlang der Front adaptiert wird. Das Experiment stellt damit den Worst-Case für die Extract-basierte Methode und die Methode der parallelen Erzeugung geometrischer Objekte nach und versucht, die Skalierbarkeit in der Datensatzgröße zu überprüfen.

Als Parallelrechner für den Vergleich kamen zwei Compute Cluster unter Linux des IWR der Universität Heidelberg zum Einsatz. Einmal der Rechner der Arbeitsgruppe „Simulation in Technology", SPEEDO, bestehend aus 80 Rechnern mit Pentium II (400 MHz), 512 MByte Speicher pro CPU und geswitchtem 100 MBit/s Fast-Ethernet als Verbindungsnetzwerk. Zum anderen das „Heidelberg Linux Cluster System", HELICS, mit 512 Athlon-XP Prozessoren (1,4 GHz), 1 GByte Speicher pro CPU und Myrinet 2000 als System Area Network.

5.2 pV3

5.2.1 Architektur

pV3 (*Parallel Visual3*, siehe [Hai94], [HE97], [HJ01]) ist ein paralleles Visualisierungssystem zur interaktiven On-Line-Visualisierung oder für den Batchbetrieb. Es realisiert den Extract-basierten Ansatz aus [Glo95], bei dem der Filterungsschritt in der Visualisierungspipeline noch parallel abgearbeitet wird, der Rest aber sequentiell. Wie in Abschnitt 2.1.2, Seite 20 beschrieben geschieht dies mit der Begründung, dass einmal die Berechnung von Extracts potentiell Zugriff auf den gesamten Datensatz benötigt, die berechneten Extracts aber wegen der Dimensionsreduktion klein genug sind, um sequentiell auf einer Workstation abgebildet und gerendert zu werden. Verschiedene Abbildungen eines Extracts auf geometrische Objekte sowie Änderungen der Projektionsparameter (Zoom, Pan, Rotate, ...) für das Rendering können ohne erneuten Zugriff auf den Parallelrechner realisiert werden. Entsprechend besteht pV3 aus einem Back-End (Clients), das auf dem Parallelrechner läuft und die Extract-Erzeugung erledigt und einem Front-End (Server), das auf einer Workstation läuft und die Abbildung der Extracts und das Rendering der geometrischen Objekte übernimmt. Verbunden sind Front- und Back-End über ein Netzwerk. pV3 wurde ursprünglich mit PVM (Parallel Virtual Machine, siehe [GBD+94]) als Message Passing API realisiert, da PVM auch heterogene Ansammlungen von Rechnern unterstützt. Ebenfalls sehr nützlich sind *PVM Groups*, denen Prozesse dynamisch beitreten und sie wieder verlassen können. Damit ist es z.B. möglich, dass der pV3-Server sich nur zeitweise an die Clients ankoppelt, etwa um eine langlaufende Simulation zu überwachen. Auf der anderen Seite hat sich MPI (siehe [MPI94]) schon lange zum De Facto Standard für Parallelrechner entwickelt. pV3 verwendet deshalb einen besonderen Prozess, den *Konzentrator*, der sowohl auf das interne Verbindungsnetzwerk des Parallelrechners über MPI Zugriff hat, als auch am externen Netz hängt, über das er mit dem Server via PVM kommuniziert. Entsprechend dieser Vermittlungsrolle erledigt der Konzentrator das Einsammeln von Extracts (daher der Name) und das Austeilen von Befehlen vom Front-End, vgl. Abb. 5.2.

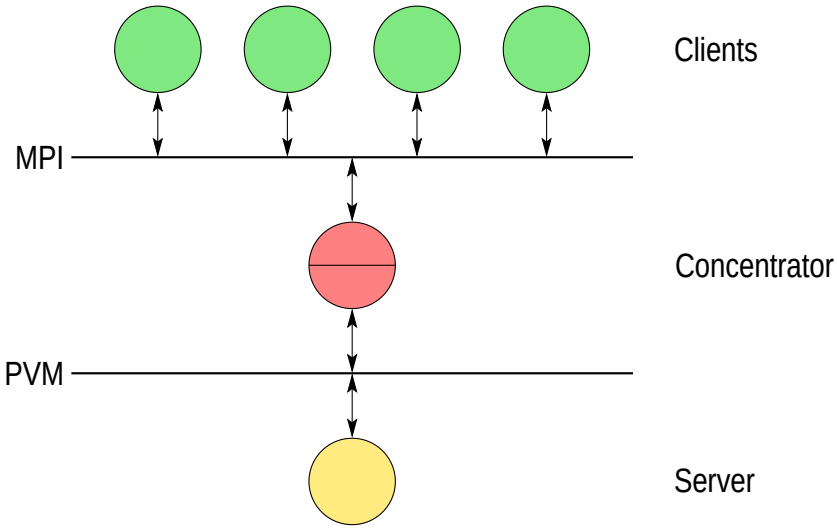

Abbildung 5.2: Netzwerkbasierte Architektur von pV3.

5.2.2 Integration in UG

pV3 ist zusammen mit umfangreicher Dokumentation kostenlos von seinem Autor zu erhalten, siehe [PV3], allerdings nicht im Sourcecode, sondern nur in Form von vorkompilierten Binaries für verschiedene Architekturen. Mitgeliefert werden im wesentlichen zwei Server, ein interaktives Serverprogramm zur On-Line-Visualisierung und einer für den Batchbetrieb, der eine vordefinierte Folge von Extracts anfordert und abspeichert, ein Extract-Viewer für den Batchserver und eine Bibliothek für die Clientseite, die in die Simulationssoftware zu integrieren ist. Dazu sind einmal einige Callback-Funktionen bereit zu stellen, über die das pV3-Back-End Zugriff auf die Geometrie des Gitters (Knotenkoordinaten, Verbindung der Knoten zu Elementen, Randseiten, Partitionsränder, usw.) und die berechneten Lösungsvektoren erhält. Zum anderen ist regelmäßig eine Routine `PV_UPDATE` aufzurufen, damit die Clients auch aktiv werden. Die Funktionsweise der Callback-Funktionen ist der erste Schwachpunkt von pV3, da sie sämtliche Informationen für das gesamte Gitter in eigene Datenstrukturen umkopieren, mit entsprechendem Speicherverbrauch. Der zweite problematische Punkt ist der Konzentrator. Ob überhaupt ein Rechnerknoten mit den benötigten Eigenschaften vorhanden ist, hängt vom verwendeten Parallelrechner ab. Falls ja, erfordert es mindestens installationsspezifische

Procs	1	4	16	64
Level	4+3	4+4	4+5	4+6
Elemente	374121	1497023	5942375	23845652
Speicher	138 M	150 M	129–214 M	87–419 M
Zeit	<1 s	2 s	8 s	34 s

Tabelle 5.1: Werte für pV3 auf Rechner SPEEDO.

Kenntnisse, um in jeder Situation aus den etwa von einem Batchsystem zugewiesenen Knoten den richtigen zu bestimmen — falls überhaupt einer darunter ist.

5.2.3 Ergebnisse des Experiments

Ausgeführt wurde das Experiment wegen der im letzten Abschnitt genannten Schwierigkeiten mit der Wahl des Konzentrators nur auf dem Rechner SPEEDO mit dem interaktiven pV3-Server. Alle beteiligten Rechner waren über geswitchtes Fast-Ethernet verbunden. Abb. 5.3 zeigt das Ergebnis, Tabelle 5.1 die gemessenen Werte, vgl. auch Abschnitt 5.3.1. Da der Sourcecode von pV3 nicht zugänglich ist, konnte er auch nicht instrumentiert werden, weshalb die angegebenen Laufzeiten für die Durchführung des Experiments von Hand gestoppt sind und sämtliche Arbeitsschritte zur Lösung der Visualisierungsaufgabe bis zum fertigen Bild enthalten. Die Speicherangaben pro Rechnerknoten beziehen sich ausschließlich auf UG und nicht auf den zusätzlichen Verbrauch durch pV3, der nicht zu bestimmen war.

Die Laufzeiten spiegeln die zu erwartende Performance des Extract-basierten Ansatzes bei diesem Experiment wieder: Es findet keine Dimensionsreduktion statt, sodass das Hauptargument für diesen Ansatz nicht greift. Die Zeit für die Visualisierung nimmt ziemlich genau um den gleichen Faktor zu wie die Anzahl der Elemente. Dabei spielt auch das zunehmende Lastungleichgewicht bis zu 64 Prozessoren keine Rolle mehr. Der Flaschenhals ist der Konzentrator, der die Datenströme von bis zu 64 Prozessoren auf eine Leitung mit der gleichen Bandbreite multiplexen muss. Tatsächlich wurde während des Experiments ein dauernder maximaler Datenstrom zwi-

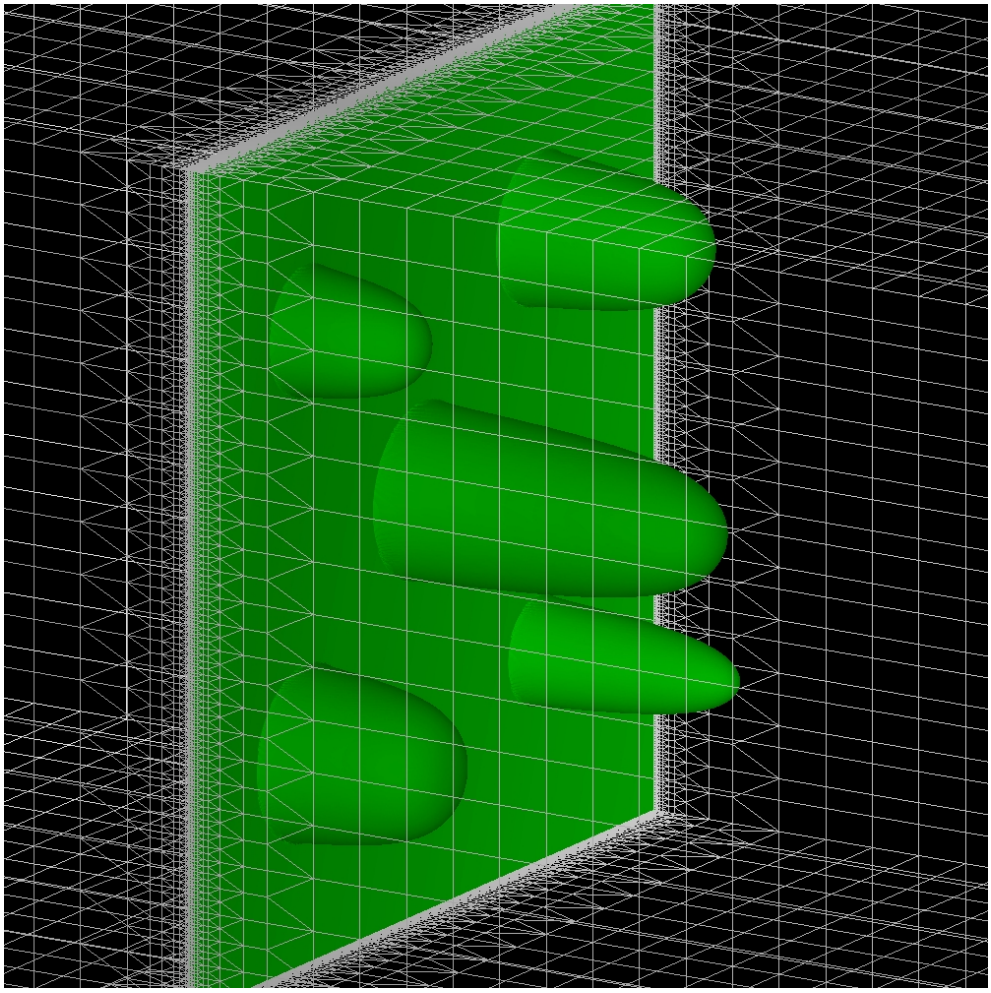

Abbildung 5.3: Visualisierung des Experiments mit pV3.

schen dem Konzentrator und der Workstation, auf der der pV3-Server lief, gemessen. Entsprechendes gilt erst recht für aktuellere Rechner als SPEEDO, denn externe Netzanbindungen sind dann auf jeden Fall langsamer als das interne Verbindungsnetzwerk des Parallelrechners.

Insgesamt erweist sich der Extract-basierte Ansatz bei diesem Experiment als nicht skalierbar in der Datensatzgröße. Wegen des Flaschenhalses Konzentrator kann er es generell nicht sein. Der Extract-basierte Ansatz ist nur gut, wenn das Argument der Dimensionsreduktion greift und das Extract klein genug ist. Z.B. sollte man für eine Isofläche durch ein strukturiertes Gitter mit 10^9 Elementen größenordnungsmäßig 10^6 Polygone erhalten, die

man mit obiger Hardware in ca. 1–2 s visualisieren könnte.

5.3 UG: „List Priority“ und „Bullet“

5.3.1 Ergebnisse des Experiments

Für die beiden Methoden des UG-Visualisierungssubsystems wurde der Code instrumentiert, um die Laufzeiten einiger interessanter Routinen zu messen. Dazu wurde der *Time Stamp Counter* der CPU (ab Pentium vorhanden) benutzt, ein 64 Bit breites Register, das von der CPU in jedem Taktzyklus um eins hochgezählt wird. Alle Zeiten sind also Wall Clock-Zeiten. Da die Läufe im Batchbetrieb absolviert wurden, kamen das Metafile- bzw. das PPM-Device der UG-Graphik zum Einsatz. Im einzelnen wurden folgende Werte gemessen:

Elemente

Die Anzahl der Elemente, aus denen das Gitter besteht.

Level

$n + m$ bedeutet, n mal uniform verfeinert, lastverteilt und dann m mal adaptiv verfeinert, vgl. Abschnitt 5.1.

Polygone

Die Anzahl der Polygone, aus denen die Isofläche besteht.

Speicher

Der pro Rechnerknoten verbrauchte Speicher. Sind Bereiche angegeben, so geben sie das maximale Lastungleichgewicht unter den Knoten wieder.

Preprocess

Eine Schleife über alle Elemente der Mehrgitterhierarchie, die alle Elemente testet und diejenigen markiert, die von der Isofläche geschnitten werden.

Evaluate

Akkumulierte Zeiten für das Erzeugen der Drawingobjects aus den markierten Elementen. Die Zeiten für den Iterator sind *nicht* enthalten. Zwei Subroutinen sind aufgeschlüsselt: `Extract` für das „Ziehen“ der Isoflächenpolygone nach der Tetra-Cubes-Methode (siehe [CSK96]), `Light` für die Beleuchtungsberechnung.

Procs	1	4	16	64
Level	4+3	4+4	4+5	4+6
Elemente	374121	1497023	5942375	23845652
Polygone	156416	614640	2448336	9764032
Speicher	138 M	150 M	129–214 M	87–419 M
Preprocess	0,87 s	0,92 s	1,31 s	2,54 s
Evaluate	0,70 s	0,70 s	0,98 s	1,96 s
Extract	0,14 s	0,14 s	0,20 s	0,41 s
Light	0,14 s	0,14 s	0,19 s	0,37 s

Tabelle 5.2: Beiden Methoden gemeinsame Werte auf Rechner SPEEDO.

Procs	1	4	16	64
Vorderseiten	2,82 s	3,01 s	4,35 s	8,66 s
Elemente sortieren	2,98 s	4,07 s	7,62 s	17,99 s
Polygone sortieren	5,27 s	5,44 s	7,55 s	15,16 s
Protokoll	0,00 s	22,88 s	105,42 s	435,17 s
Schreiben	1,09 s	4,31 s	17,11 s	68,76 s

Tabelle 5.3: Zeiten für „List Priority“ auf Rechner SPEEDO.

Procs	1	4	16	64
Render	0,38 s	0,39 s	0,44 s	0,85 s
Composite	0,00 s	0,39 s	1,16 s	1,93 s
Schreiben	4,0 s			

Tabelle 5.4: Zeiten für „Bullet“ auf Rechner SPEEDO.

Procs	1	4	16	64	256
Level	4+3	4+4	4+5	4+6	4+7
Elemente	374121	1497023	5942375	23845652	95247702
Polygone	156416	614640	2448336	9764032	39024840
Speicher	138 M	150 M	129–214 M	87–419 M	90–630 M
Preprocess	0,58 s	0,63 s	0,87 s	1,76 s	2,61 s
Evaluate	0,19 s	0,19 s	0,27 s	0,54 s	0,80 s
Extract	0,03 s	0,03 s	0,05 s	0,09 s	0,14 s
Light	0,02 s	0,02 s	0,03 s	0,05 s	0,08 s

Tabelle 5.5: Beiden Methoden gemeinsame Werte auf Rechner HELICS.

Procs	1	4	16	64	256
Vorderseiten	1,28 s	1,39 s	2,00 s	4,00 s	5,99 s
Elemente sortieren	1,96 s	2,48 s	4,13 s	9,51 s	15,01 s
Polygone sortieren	1,31 s	1,36 s	1,88 s	3,61 s	5,51 s
Protokoll	0,00 s	5,13 s	22,53 s	91,03 s	385.96 s
Schreiben	0,30 s	1,16 s	4,86 s	20,83 s	125,08 s

Tabelle 5.6: Zeiten für „List Priority“ auf Rechner HELICS.

Procs	1	4	16	64	256
Render	0,08 s	0,08 s	0,09 s	0,17 s	0,25 s
Composite	0,00 s	0,04 s	0,10 s	0,18 s	0,28 s
Schreiben	0,8–1,0 s				

Tabelle 5.7: Zeiten für „Bullet“ auf Rechner HELICS.

Für die Methode der parallelen Erzeugung geometrischer Objekte „List Priority“:

Vorderseiten

Die Zeit für die Klassifikation aller Elementseiten in Vorder- und Rückseiten.

Elemente sortieren

Die Zeit für die Berechnung der globalen Prioritätsliste für alle Elemente der Mehrgitterhierarchie.

Polygone sortieren

Die akkumulierte Zeit für die Berechnung von Prioritätslisten für die gezogenen Polygone innerhalb der Elemente.

Protokoll

Die Zeit für die Übertragung der Drawingobjekts nach aufsteigender Plot ID, vgl. Abschnitt 4.2.3, Seite 77.

Schreiben

Die Zeit für die Ausgabe der auf dem Master-Prozessor eingesammelten Drawingobjects durch Ausgabe in eine Datei.

Für die Methode der parallelen Bilderzeugung „Bullet“:

Render

Die Zeit für das Rendern der Drawingobjects im Bullet-Modul bei einer Bildgröße von 800×400 Pixeln, vgl. Abb. 5.1, oberes Bild.

Composite

Die Zeit für das Compositing für 800×400 Pixel.

5.3.2 Diskussion der Resultate

Zunächst kann man festhalten, dass sowohl die Gesamtzahl der Elemente als auch die Zahl der daraus gezogenen Polygone mit jeder Verfeinerungsstufe um ziemlich genau Faktor vier wächst. Da auch die Zahl der Prozessoren jeweils vervierfacht wurde, würden die gemessenen Zeiten im Idealfall etwa konstant bleiben. (Optimaler Scaleup, vgl. Abschnitt 1.3.4, Seite 16.) Wenn man das Lastungleichgewicht zwischen den einzelnen Läufen berücksichtigt, nehmen die Zeiten für `Preprocess` und `Evaluate` aus den Tabellen 5.2 und 5.5, `Vorderseiten` und `Polygone sortieren` aus den Tabellen 5.3 und 5.6, sowie `Render` aus den Tabellen 5.4 und 5.7 auch recht genau um den Faktor zu, der sich aus diesem ergibt (Verhältnis des maximalen Speicherbedarfs zu vergleichender Läufe. Bsp.: Beim Übergang von 64 zu 256 Prozessoren in Tabelle 5.7 nimmt die Zeit für `Render` um den Faktor $0,25/0,17 = 1,47$ zu. Das entspricht dem Verhältnis der maximalen Prozessorlasten $630/419 = 1,50$) Die Zeiten für `Elemente sortieren` aus den Tabellen 5.3 und 5.6 nehmen etwas stärker zu, da hier mit größerer Prozessorzahl auch der Kommunikationsbedarf zunimmt. Insgesamt skalieren diese Schritte fast optimal in der Datensatzgröße.

Die Schritte `Protokoll` und `Schreiben` aus den Tabellen 5.3 und 5.6 hingegen nehmen jeweils um etwas mehr als Faktor vier zu, entsprechend der Vergrößerung des Datensatzes. „List Priority“ skaliert aus denselben Gründen wie pV3 nicht in der Datensatzgröße, weder unter den besonders ungünstigen Bedingungen dieses Experiments noch generell.

„Bullet“ schließt neben den sehr gut skalierenden Schritten noch `Composite` und `Schreiben` aus den Tabellen 5.4 und 5.7 ein. Für ersteres ist der Aufwand proportional zur Bildgröße und bei der in UG gewählten einfachen Implementierung proportional zum Logarithmus der Prozessorzahl. Auf HELICS sind beide Zeiten gering, auf SPEEDO wegen des langsamen Netzwerks schon größer. Bei Bedarf lassen sich die in Abschnitt 3.5.1, Seite 53 genannten Optimierungen anwenden. Für UG erscheint das kaum lohnenswert, wenn man diese Zeiten im Gesamtkontext betrachtet. Insgesamt skaliert „Bullet“ sehr gut in der Datensatzgröße. Die reine Renderingleistung beträgt 156 MPolygone/s (unbeleuchtet) bzw. 118 MPolygone/s (beleuchtet) für den Lauf auf 256 Prozessoren des Rechners HELICS, bei günstigerer

Lastverteilung darf man etwa das Vierfache erwarten.

Im Vergleich der beiden Rechner untereinander zeigt sich, dass Operationen, die einmal quer durch den Speicher iterieren, wie `Preprocess` aus den Tabellen 5.2 und 5.5 oder `Vorderseiten` und `Elemente sortieren` aus den Tabellen 5.3 und 5.6 beim Übergang auf den schnelleren Rechner um teilweise nicht einmal Faktor zwei schneller werden. Operationen mit lokalem Speicherzugriff werden hingegen vier- bis fünfmal schneller ausgeführt. Das Schreiben der Ausgabefiles dauert auf SPEEDO mehr als dreimal so lange wie auf HELICS, obwohl das benutzte Filesystem beide Male auf einem NFS-Server, der mit 100 MBit/s angebunden ist, liegt. Das bestätigt eventuell die oft vertretene Meinung, dass der Linux NFS-Server, der auf SPEEDO verwendet wird, nichts taugt. Der SUN-NFS-Server von HELICS erreicht bei der Ausgabe jedenfalls die etwa zu erwartenden 10 MByte/s.

5.4 Maximum Intensity Projection

Bisher wurde die parallele Visualisierung nur unter dem Gesichtspunkt der Skalierbarkeit in der Datensatzgröße betrachtet. Das ist auch naheliegend, denn wer einen kleinen bis moderat großen Datensatz visualisieren will, verwendet dazu keinen teuren Parallelrechner sondern eine Graphik-Workstation. Manche Visualisierungstechniken, z.B. Volume Rendering (siehe etwa [MPHK94], [LMS+01] und [FK04]), sind allerdings sehr rechenintensiv und können den Einsatz von Parallelrechnern dadurch rechtfertigen. Für eine dem Volume Rendering verwandte Technik, die *Maximum Intensity Projection*, soll hier deshalb auch noch der mit der Methode „Bullet“ bei diesem Problem zu erzielende Speedup ermittelt werden.

Maximum Intensity Projektion (MIP) ist eine Renderingtechnik, bei der ein Datensatz vom Typ E_3^S (vgl. Abschnitt 1.1.2, Seite 3) visualisiert werden soll. Jedem Pixel auf der Projektionsebene wird der größte Wert, den das Feld entlang eines Strahls in Blickrichtung annimmt, zugewiesen und dann auf eine Farbe (Graustufen oder Spektrum) abgebildet. MIP wird z.B. in der medizinischen Bildverarbeitung für durch Magnetresonanztomographie erhaltene Datensätze eingesetzt. Der hier benutzte Datensatz

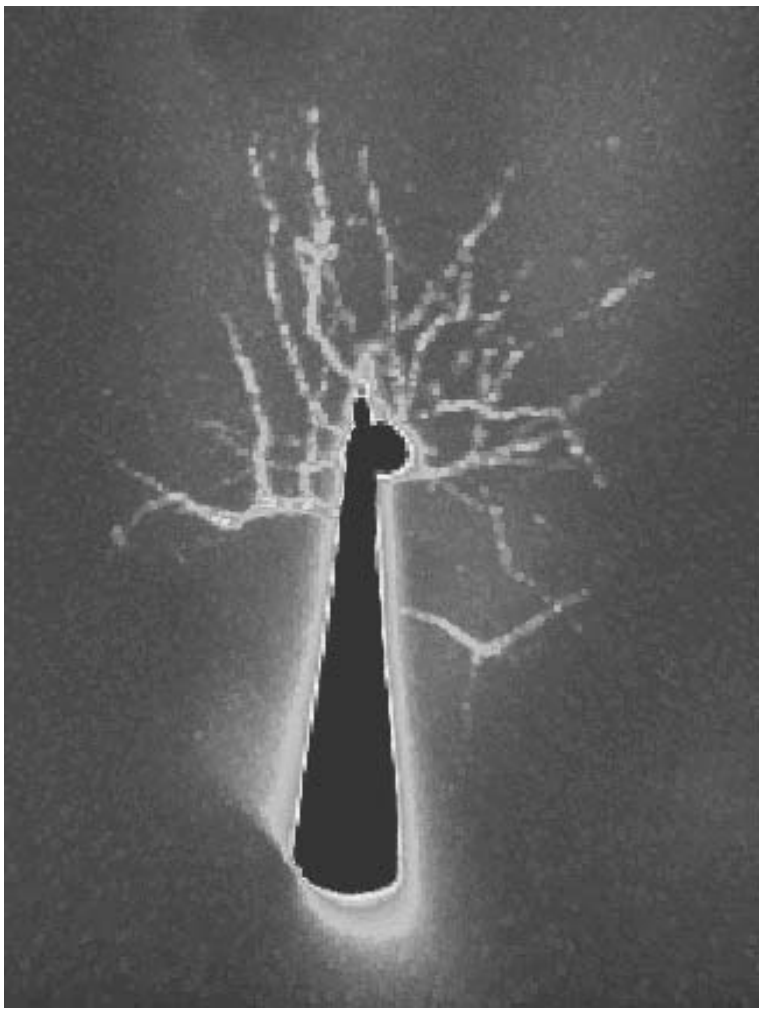

Abbildung 5.4: MIP der Nervenzelle (mit Pipette um Zellkörper).

ist durch Zwei-Photonen-Mikroskopie einer Nervenzelle aus dem Gehirn einer lebenden Maus entstanden und stammt vom Max-Planck-Institut für medizinische Forschung in Heidelberg. Zwei-Photonen-Mikroskopie ist ein Lasermikroskopieverfahren, das hochauflösende, dreidimensionale Bilder lebender Organismen ermöglicht. Die Bilder enthalten einen relativ hohen Anteil an Hintergrundrauschen, da man nur mit geringen Laserenergien arbeiten kann, um die Organismen nicht zu beschädigen. Die Idee ist, mittels MIP die Gestalt der Nervenzelle vom Rauschen zu unterscheiden. Die sich aus der Methode ergebenden Fehler nimmt man in Kauf. Abb. 5.4 zeigt das Ergebnis.

Procs	1	2	5	10	20	50	100
Render	41,44 s	22,47 s	11,35 s	4,62 s	2,33 s	0,94 s	0,47 s
Composite	0,00 s	0,01 s	0,02 s	0,03 s	0,04 s	0,07 s	0,08 s
Speedup	1	1,8	4,6	8,9	17,4	41	74
Effizienz	100%	92%	91%	89%	87%	82%	74%

Tabelle 5.8: Werte für MIP der Nervenzelle.

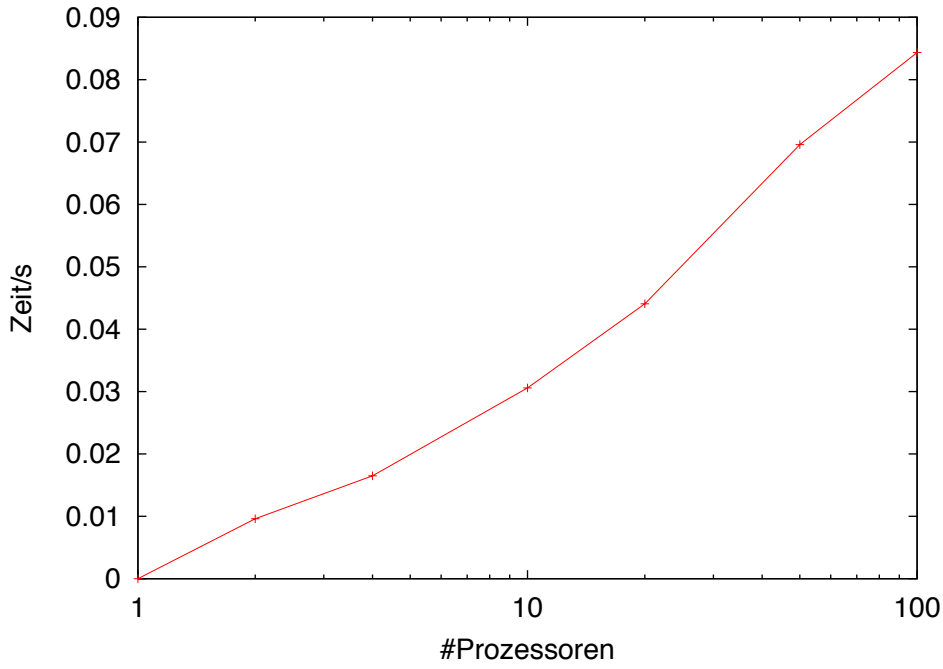

Abbildung 5.5: Compositing-Zeiten für MIP.

Der verwendete Bilderstapel besteht aus 100 Schnitten zu je 256×256 Pixeln, wobei der Abstand der einzelnen Bilder im Stapel größer ist, als der Pixelabstand der Bilder. Der Stapel ist fast würfelförmig. Zum Rendering wird um jedes Pixel im Stapel eine Box (*Voxel*) zentriert, so dass die Voxel den Raum ausfüllen. In jedem Voxel wird die zum zugehörigen Pixel gehörende Intensität als konstant angenommen und dann die Vorderseiten des Voxels mit dieser Intensität via „Bullet“ bei einer Auflösung von 800×800 Pixeln gerendert. Die Intensität dient gleichzeitig als z-Wert. Hier werden 8 Bit für die Intensität verwendet. Zur Ausgabe wird die Intensität auf das Spektrum von blau nach rot abgebildet. Die Lastverteilung erfolgt durch gleichmäßiges Verteilen der Schnitte unter den Prozessoren.

Tabelle 5.8 enthält die gemessenen Werte auf HELICS. Der Grund für die abnehmende Effizienz bei steigender Prozessorzahl dürfe sein, dass der Aufwand für den Eintrag in den Pixel Buffer davon abhängt, ob ein Fragment den z-Test besteht oder nicht, im ersten Fall ist ein Schreibzugriff auf den Pixel Buffer notwendig, im letzteren nicht. Dadurch entsteht ein Lastungleichgewicht. Auch wird der Anteil des Compositings bei Erhöhung der Prozessorzahl immer stärker. Die Zeiten für das Compositing sind in Abb. 5.5 noch einmal graphisch dargestellt. Die leichte Abweichung von dem in Abschnitt 3.5.1, Seite 54 berechneten logarithmischen Aufwand dürfte davon herrühren, dass die Prozessorzahlen keine Zweierpotenzen sind, das Schema also nicht glatt aufgeht. Das oben über das mögliche Lastungleichgewicht beim Pixeleintrag gesagte gilt entsprechend auch für jeden einzelnen Compositingschritt. Trotzdem ist die parallele Effizienz akzeptabel.

Es sei noch erwähnt, dass der hier verwendete MIP-Algorithmus ein reiner ‚Brute Force'-Algorithmus ist. Eine bessere sequentielle reine Software-Implementierung ist in [CKG99] beschrieben, eine OpenGL-basierte (Stichwort: 3D-Texturen) lässt sich aus [WGW94] ableiten. Letztere benötigt aber Hardware-Support in Form von Graphikkarten, bei ersterer ist zu bedenken, dass die direkte Übertragung von Optimierungen für den sequentiellen Fall im Parallelen meistens nur zu einem erheblichem Lastungleichgewicht führt.

Resümee und Ausblick

Die Motivation zu dieser Arbeit war der Wunsch, große Datensätze zu visualisieren, wie sie auf heutigen Supercomputern produziert werden können. Dazu werden in Kapitel 2 die Parallelisierungsmöglichkeiten des Visualisierungsprozesses anhand der Visualisierungspipeline (Filterung $\mapsto$ Abbildung $\mapsto$ Rendering) untersucht und drei prinzipielle Ansätze vorgestellt: die parallele Extract-Erzeugung, die nur die Filterung parallel ausführt, die parallele Erzeugung geometrischer Objekte, die sowohl Filterung als auch Abbildung parallel ausführt und die parallele Bilderzeugung, bei der alle drei Schritte parallel ausgeführt werden. Da die ersten beiden Ansätze auf eine Dimensionsreduktion während der Filterung setzen und nur unter dieser Annahme zur Visualisierung von großen Datensätzen in der Lage sind, liegt das Hauptaugenmerk auf der einzigen vollständig parallelen Methode: der der parallelen Bilderzeugung. Nach der Erkenntnis, dass sowohl Filterung als auch Abbildung in der Regel trivialerweise parallelisierbar sind, verbleibt als wesentliche Aufgabe die Parallelisierung des Renderings, welches den Gegenstand von Kapitel 3 bildet. Kapitel 4 beschreibt das parallele Visualisierungssubsystem des Simulations-Frameworks UG und dessen vom Autor durchgeführte Parallelisierung. Kapitel 5 schließlich vergleicht die beiden Ansätze aus UG (parallele Erzeugung geometrischer Objekte und parallele Bilderzeugung) und den Extract-basierten, der durch das externe Visualisierungssystem pV3 vertreten ist, miteinander und bestätigt, dass vor allem bei der Benutzung adaptiver Gitter tatsächlich nur mit der parallelen Abarbeitung der gesamten Visualisierungspipeline die Visualisierung großer Datensätze effizient gelingen kann.

In der Praxis werden große Parallelrechner allerdings — schon immer und wohl auch in Zukunft — im Batchmodus betrieben. Der Einsatz eines sol-

chen Rechners zur Visualisierung bedeutet daher den Verlust der Interaktivität im Vergleich zum traditionellen Postprocessing. Niemand wird freiwillig diesen Weg gehen, solange er wegen der Datensatzgröße nicht schlicht dazu gezwungen ist. Dem Autor sind bisher nur wenige mit UG gemachten Simulationen bekannt, die den Einsatz des UG-Visualisierungssubsystems zwingend erfordert hätten. Immerhin ist UG damit für die Zukunft gerüstet, für die immer größere und leistungsfähigere Parallelrechner abzusehen sind.

Bei den bereits bestehenden sehr großen Parallelrechner-Installationen, z.B. denen der US-amerikanischen National Laboratories, zeichnet sich der Trend ab, das klassische Postprocessing doch wieder zu ermöglichen, indem man die während der Simulation anfallenden Daten parallel auf mit hoher I/O-Bandbreite ansteuerbare Filesysteme ausgibt. Diese Filesysteme (z.B. PVFS [CIRT00] und LUSTRE [Bra03]) sind auf RAID-Systemen mit hinreichend hoher Kapazität untergebracht, womit zumindest die Voraussetzungen gegeben sind, die Daten eines umfangreichen Simulationslaufs abspeichern zu können. Die Visualisierung selbst kann dann wieder unabhängig nach dem Postprocessing-Schema stattfinden. Da dazu ebenfalls Zugriff auf das schnelle parallele Filesystem und wenigstens moderate Rechenpower nötig ist, wird hierfür gerne ein eigener Visualisierungscluster oder eine spezielle Partition des Parallelrechners benutzt, typischerweise einige zehn bis einige hundert Rechner; der ganze Rechner gilt als „zu wertvoll“ für die Visualisierungsaufgabe. Falls ein eigener Visualisierungscluster benutzt wird, ist dieser wenigstens mit Graphikkarten und eventuell mit einem System zum Hardware-Compositing ausgestattet, vgl. dazu Kapitel 3. Die geringere Zahl der Knoten im Visualisierungscluster kann gerechtfertigt sein, da nicht alle während einer Simulation anfallenden Daten zum Postprocessing gebraucht werden, z.B. die Diskretisierungsmatrizen. Im Zweifelsfall wird man sich mit dem Thema „Out-of-Core Rendering“ beschäftigen müssen. Softwareseitig werden Programme wie ParaView [PaV] oder VisIt [VVT] eingesetzt, die beide auf dem Visualization ToolKit VTK [VTK] bzw. dessen paralleler Version Parallel VTK [ALS+00] aufbauen. Parallel VTK realisiert dabei neben den in Kapitel 2 kurz erwähnten und wenig versprechenden Ansätzen der Aufgaben- und Pipelineparallelität ebenfalls den Sort-Last-Ansatz zum parallelen Rendering.

Literaturverzeichnis

[Ake93] Kurt Akeley, *Realityengine graphics*, Proceedings of SIGGRAPH, 1993, pp. 109–116.

[ALS+00] James Ahrens, Charles Law, Will Schroeder, Ken Martin, and Michael Papka, *A parallel approach for efficiently visualizing extremely large, time-varying datasets*, Tech. Report LAUR-00-1620, Los Alamos National Laboratory, 2000.

[ALX] *SGI Altix*, http://www.sgi.com/products/servers/altix.

[Amd67] Gene Amdahl, *Validity of the single processor approach to achieving large-scale computing capabilities*, Proceedings of the AFIPS Spring Joint Computer Conference, 1967, pp. 483–485.

[AT95] Greg Abram and Lloyd A. Treinish, *An extended data-flow architecture for data analysis and visualization*, Proceedings of IEEE Visualization, 1995, pp. 263–270.

[Bas96] Peter Bastian, *Parallele Adaptive Mehrgitterverfahren*, Teubner Skripten zur Numerik, B. G. Teubner, Stuttgart, 1996.

[Bas99] Peter Bastian, *Numerical computation of multiphase flow in porous media*, Habilitationsschrift, Universität Kiel, 1999.

[BB94] Klaus Birken and Peter Bastian, *Dynamic distributed data (DDD) in a parallel programming environment—specification and functionality*, Forschungs- und Entwicklungsberichte RUS-22, Rechenzentrum der Universität Stuttgart, Germany, September 1994.

[BBC+94] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*, 2nd ed., SIAM, Philadelphia, PA, 1994.

[BBJ+97] Peter Bastian, Klaus Birken, Klaus Johannsen, Stefan Lang, Nikolas Neuß, Henrik Rentz-Reichert, and Christian Wieners, *UG—A flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science **1** (1997), no. 1, 27–40.

[BGT02] The BlueGene/L Team, *An overview of the BlueGene/L supercomputer*, Supercomputing Technical Papers, 2002.

[BHH00] Ian Buck, Greg Humphreys, and Pat Hanrahan, *Tracking graphics state for networked rendering*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, 2000.

[BHPB03] E. Wes Bethel, Greg Humphreys, Brian Paul, and J. Dean Brederson, *Sort-first, distributed memory parallel visualization and rendering*, Proceedings of the 2003 IEEE Symposium on Parallel and Large Data Visualization and Graphics, 2003.

[BL42] S. E. Buckley and M. C. Leverett, *Mechanism of fluid displacement in sands*, Transactions of the American Institute of Mining, Metallurgical, and Petroleum Engineers **146** (1942), 107–116.

[Bli94] James F. Blinn, *Compositing, Part I: Theory*, IEEE Computer Graphics and Applications **14** (1994), no. 5, 83–87.

[Bra03] Peter J. Braam, *The lustre storage architecture*, 2003.

[Bro92] Ken Brodlie, *Scientific visualization—techniques and applications*, Springer Verlag, 1992.

[CHR] *Chromium Homepage*, http://chromium.sf.net.

[CIRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur, *PVFS: A parallel file system for linux clusters*, Proceesings of the 4th Annual Linux Showcase and Conference, USENIX Association, 2000, pp. 317–327.

[CKG99] Balazs Csebfalvi, Andreas König, and Eduard Gröller, *Fast maximum intensity projection using binary shear-warp factorization*, Tech. Report TR-186-2-99-02, Vienna University of Technology, 1999.

[CKM+99] Joao Comba, James T. Klosowski, Nelson Max, Joseph S.B. Mitchell, Claudio T. Silva, and Peter L. Williams, *Fast polyhedral cell sorting for interactive rendering of unstructured grids*, Proceedings of EUROGRAPHICS, 1999.

[CMP] *Compositing Network API Reference Guide*, http://www.hp.com/techservers/hpccn/sci_vis/resources.html.

[CO93] Thomas W. Crokett and Tobias Orloff, *A MIMD rendering algorithm for distributed memory architectures*, Proceedings of the 1993 Parallel Rendering Symposium, IEEE Computer Society Press, 1993, pp. 35–42.

[Cro94] Thomas W. Crokett, *Design considerations for parallel graphics libraries*, Tech. Report 94-49, Institute for Computer Applications in Science and Engeneering, 1994.

[Cro97] Thomas W. Crockett, *PGL: A parallel graphics library for distributed memory applications*, PGL home page, http://www.compsci.wm.edu/PGL, February 1997.

[Cro98] Thomas W. Crockett, *High performance parallel rendering on the Cray T3E*, http://www.icase.edu/docs/hilites/tom/T3E.html, May 1998.

[CSK96] Bernardo Piquet Carneiro, Claudio T. Silva, and Arie E. Kaufman, *Tetra-cubes: An algorithm to generate 3D isosurfaces based upon tetrahedra*, Proceedings of SIBGRAPI, 1996, pp. 205–210.

[DG94] Gitta O. Domik and B. Gutkauf, *User modeling for adaptive visualization systems*, Proceedings of IEEE Visualization, 1994, pp. 217–223.

[Dij65] Edgar W. Dijkstra, *Cooperating sequential processes*, EWD123, University of Eindhoven, The Netherlands, 1965.

[DMX] *Distributed Multihead X Project*, http://dmx.sf.net.

[Dom94] Gitta O. Domik, *Scientificic visualization: A tutorial*, University of Colorado at Boulder, HPSC Course Notes, 1994.

[EMP+97] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, and Nick England, *Pixelflow: The realization*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, 1997, pp. 57–68.

[FK04] Susan Frank and Arie Kaufman, *Massive volume rendering on a volume visualization cluster*, Tech. report, Center For Visual Computing and Department of Computer Science, Stony Brook University, 2004.

[Fly72] Michael J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computers **C** (1972), no. 21, 948–960.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer graphics: Principles and practice in C (2nd ed.)*, Addison Wesley, 1996.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, *PVM: Parallel virtual machine. A users' guide and tutorial for networked parallel computing*, MIT Press, 1994.

[GKM93] Ned Greene, Michael Kassy, and Gavin Miller, *Hierarchical z-buffer visibility*, Proceedings of SIGGRAPH, 1993, pp. 231–238.

[Glo95] Al Globus, *A software model for visualization of time dependent 3-d computational fluid dynamics results*, AIAA Paper 95-0115, 1995.

[Gus88] John L. Gustafson, *Reevaluating Amdahl's law*, Communications of the ACM **31** (1988), no. 5, 532–533.

[Hac93] Wolfgang Hackbusch, *Iterative Lösung grosser schwachbesetzter Gleichungssysteme*, Teubner, 1993.

[Hai94] Robert Haimes, *pV3: A distributed system for large-scale unsteady CFD visualization*, AIAA Paper 94-0321, 1994.

[HBEH00] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan, *Distributed rendering for scalable displays*, Proceedings of IEEE Supercomputing, 2000.

[HE97] Robert Haimes and David E. Edwards, *Visualization in a parallel processing environment*, AIAA Paper 97-0348, 1997.

[HEB+01] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan, *WireGL: A scalable graphics system for clusters*, Proceedings of SIGGRAPH, 2001.

[HGKH98] Alan Heirich, David Garcia, Michael Knowles, and Robert Horst, *Servernet-II: A reliable interconnect for scalable high performance cluster computing*, Tech. report, Compaq Tandem Labs, 1998.

[HH99] Greg Humphreys and Pat Hanrahan, *A distributed graphics system for large tiled displays*, Proceedings of IEEE Visualization, 1999, pp. 215–223.

[HHN+02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski, *Chromium: A stream-processing framework for interactive rendering on clusters*, Procedings of SIGGRAPH, 2002, pp. 693–702.

[Hil85] D. Hillis, *The connection machine*, MIT Press, 1985.

[HJ01] Robert Haimes and Kirk E. Jordan, *A tractable approach to understanding the results from large-scale 3D transient simulations*, AIAA Paper 2001-0918, 2001.

[Hou87] Piet Houthuys, *Box Sort, a multidimensional binary sorting method for rectangular boxes, used for quick range searching*, The Visual Computer **3** (1987), no. 4, 236–249.

[HX98] K. Hwang and Z. Xu, *Scalable parallel computing—technology, architecture, programming*, McGraw-Hill, 1998.

[ISH98] Homan Igehy, Gordon Stoll, and Pat Hanrahan, *The design of a parallel graphics interface*, Proceedings of SIGGRAPH, 1998, pp. 141–150.

[JKA+03] Thomas Jackman, Peter Kirchner, Gregory Abram, James Klosowski, and Christopher Morris, *The deep view project: Scalable, network-centric visualization with the commodity cluster*, Internal IBM Research Report, 2003.

[KBH95] Mark J. Kilgard, David Blythe, and Deanna Hohn, *System support for OpenGL direct rendering*, Proceedings of Graphics Interface, 1995, pp. 116–127.

[KF88] Bernard H. Kueper and Emil O. Frind, *An overview of immiscible fingering in porous media*, Journal of Contaminant Hydrology **2** (1988), no. 2, 95–110.

[KKV+02] James T. Klosokowski, Peter D. Kirchner, Julia Valuyeva, Greg Abram, Christopher J. Morris, Robert H. Wolfe, and Thomas Jackman, *Deep View: High-resolution reality*, IEEE Computer Graphics and Applications **22** (2002), no. 3, 12–15.

[Knu73a] Donald E. Knuth, *The art of computer programming, Vol. 1: Fundamental algorithms*, 2nd ed., Addison-Wesley, Reading, Massachusetts, 1973.

[Knu73b] Donald E. Knuth, *The art of computer programming, Vol. 3: Sorting and searching*, Addison-Wesley, Reading, Massachusetts, 1973.

[Lam97] Michael Lampe, *Parallelisierung eines Graphik-Subsystems in einem Paket zur numerischen Lösung partieller Differentialgleichungen*, Diplomarbeit Nr. 1501, Fakultät für Informatik, Universität Stuttgart, 1997.

[Lan01] Stefan Lang, *Parallele Numerische Simulation instationärer Probleme mit adaptiven Methoden auf unstrukturierten Gittern*, Ph.D. thesis, Institut für Wasserbau, Universität Stuttgart, 2001.

[LC87] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, Computer Graphics **21** (1987), no. 4, 163–169.

[LL97] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA highly scalable server*, Proceedings of the 24th International Symposium on Computer Architecture (Philadelphia), 1997.

[LMS+01] Santiago Lombeyda, Laurent Moll, Mark Shand, David Breen, and Alan Heirich, *Scalable interactive volume rendering using off-the-shelf components*, Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics, 2001, pp. 115–121.

[LW04] Stefan Lang and Gabriel Wittum, *Large-scale density-driven flow simulation using parallel unstructured grid adaptation and local multigrid methods*, IWR/SFB-Preprints, Univ. Heidelberg, 2004.

[Max95] Nelson Max, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics **1** (1995), no. 2, 99–108.

[MBDM97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal, *Infinitereality: A real-time graphics system*, Proceedings of SIGGRAPH, 1997, pp. 293–302.

[MC98] Tulika Mitra and Tzi-cker Chiueh, *Implementation and evaluation of the parallel mesa library*, Tech. report, State University of New York at Stony Brook, 1998.

[MCEF94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs, *A sorting classification of parallel rendering*, IEEE Computer Graphics and Applications **14** (1994), no. 4, 23–32.

[MEP92] Steven Molnar, John Eyles, and John Poulton, *PixelFlow: High-speed rendering unsing image composition*, Computer Graphics **26** (1992), no. 2, 231–240.

[MES] *The Mesa 3D Graphics Library*, http://mesa3d.org.

[MPHK94] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh, *Parallel volume rendering using binary-swap image composition*, IEEE Computer Graphics and Applications **14** (1994), no. 4, 59–68.

[MPC91] MasPar Computer Co., *MasPar system overview, v2.0*, March 1991.

[MPI94] Message Passing Interface Forum, *MPI: A message-passing interface standard*, Tech. Report UT-CS-94-230, 1994.

[MSH99] Laurent Moll, Mark Shand, and Alan Heirich, *Sepia: Scalable 3D compositing using PCI Pamette*, Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999.

[MSLP02] Amirudh Modi, Nilay Sezer-Uzol, Lyle N. Long, and Paul E. Plassmann, *Scalable computational steering systems for visualization of large-scale cfd simulations*, Proceedings of AIAA Fluid Dynamics Meeting, 2002.

[Mue95] Carl Mueller, *The sort-first rendering architecture for high-performance graphics*, Proceedings of the 1995 Symposium on Interactive 3D Graphics, 1995, pp. 75–84.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo, *OpenGL programming guide*, Addison-Wesley, 1993.

[NORS97] Ralf Neubauer, Mario Ohlberger, Martin Rumpf, and Ralph Schwörer, *Efficient visualization of large-scale data on hierarchical meshes*, Visualization in Scientific Computing, Springer, 1997.

[OGL93] OpenGL Architecture Review Board, *OpenGL reference manual: The official reference document for OpenGL, Release 1*, Addison-Wesley, 1993.

[OPR00] Stephan Olbrich, Helmut Pralle, and S. Raasch, *Efficient volume visualization using parallelization and 3D streaming techniques in a high-performance computing and networking scenario*, Tech. Report 20000426, University of Hannover, 2000.

[Pau02] Brian Paul, *Scalable rendering with Chromium*, Presentation at the University of Utah computer science department, March 2002, http://www.tungstengraphics.com/chromium/chromium.html.

[Pau04] Brian Paul, *Chromium vs GLX protocol*, X.Org Mailing List, July 2004.

[PaV] *ParaView*, http://www.paraview.org.

[PD84] Thomas Porter and Tom Duff, *Compositing digital images*, Computer Graphics **18** (1984), no. 3, 253–259.

[PJ01] Kenneth A. Perrine and Donald R. Jones, *Parallel graphics and interactivity with the scaleable graphics engine*, Proceedings of the ACM/IEEE Conference on Supercomputing, 2001.

[PV3] *pV3 Home Page*, http://raphael.mit.edu/pv3/pv3.html.

[PYT] *The Python Programming Language*, http://python.org.

[SEP+01] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan, *Lightning-2: A high-performance display subsystem for PC clusters*, Proceedings of SIGGRAPH, 2001, pp. 141–149.

[SFF+00] Daniel S. Schikore, Richard A. Fischer, Randall Frank, Ross Gaunt, John Hobson, and Brad Whitlock, *High-resolution multiprojector display walls*, IEEE Computer Graphics and Applications **20** (2000), no. 4, 38–44.

[SG92] Robert W. Scheifler and James Gettys, *X window system (3rd ed.)*, Digital Press, 1992.

[Shi96] Yuan Shi, *Reevaluating Amdahl's law and Gustafson's law*, http://joda.cis.temple.edu/ shi/docs/amdahl/amdahl.html, 1996.

[SMW98] Claudio T. Silva, Joseph S. B. Mitchell, and Peter L. Williams, *An exact interactive time visibility ordering algorithm for polyhedral cell complexes*, Proceedings of the 1998 IEEE Symposium on Volume Visualization, 1998, pp. 87–94.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacher, *A characterization of ten hidden-surface algorithms*, Computing Surveys **6** (1974), no. 1, 1–55.

[Tan92] Andrew S. Tanenbaum, *Modern operating systems*, Prentice-Hall, 1992.

[TIH03] Akira Takeuchi, Fumihiko Ino, and Kenichi Hagihara, *An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors*, Parallel Computing **29** (2003), no. 11-12, 1745–1762.

[UFK+89] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The application visualization system: A computational environment for scientific visualization*, IEEE Computer Graphics and Visualization (1989), 30–41.

[VTK] *The VTK Homepage*, http://www.vtk.org.

[VVT] *VisIt Visualization Tool*, http://www.llnl.gov/VisIt/.

[WGW94] Orion Wilson, Allen Van Gelder, and Jane Wilhelms, *Direct volume rendering via 3D textures*, Tech. Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

[Wil91] Ross N. Williams, *An extremely fast ziv-lempel compression algorithm*, Proceedings of IEEE Computer Society Data Compression Conference, 1991.

[Wil92] Peter L. Williams, *Visibility ordering meshed polyhedra*, ACM Transactions on Graphics **11** (1992), no. 2, 103–126.

[YYC01] Don-Lin Yang, Jen-Chih Yu, and Yeh-Ching Chung, *Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers*, Journal of Supercomputing **18** (2001), no. 2, 201–220.