Hardware Accelerated Volume Visualization on PC Clusters

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Marcelo Eduardo Magallón Gherardelli

aus Santiago

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. H. Ruder
Tag der mündlichen Prüfung:	5.3.2004

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2004

“He’s in love,” said Gaspode. “It’s very tricky.”
— *Terry Pratchett, Moving Pictures*

Für Katrin.

Acknowledgements

This work would not have come to existence without the support of a lot of people. First and foremost my adviser, Thomas Ertl, who always provided me the confidence to try and explore new ideas, and who gave the word “Doktorvater” a meaning beyond “thesis supervisor”. To Hans Ruder, who in many occasions provided me with the support to make this work real. I am grateful to all the people I had the pleasure to work and share a life with during my four-year stay in Germany. To Matthias Hopf — one of the first people I met at the Visualization and Interactive Systems section at the University of Stuttgart — we shared many hours of fruitful and not-so-fruitful discussion, which directly influenced many of the ideas expressed in this work. To Peter Leinen at the University of Tübingen, who always had patience and comprehension, even if at times I seemed to be set on distracting him from his already busy work schedule. To my colleagues Klaus Engel, Stefan Röttger, Martin Kraus and Manfred Weiler, who always provided valuable advise, discussion and criticism, and who taught me more than they probably think they did. To Milan Ikits from the Computer Department at the University of Utah, whom I have never met in “real life”, a fact that has never prevented us from successfully working together. I am greatly indebted to two professors at the University of Costa Rica, who in different ways provided the motivation, resources and support for me to come to Germany in the first place: Jorge E. Páez, who motivated me to submit my application for a scholarship, and who has always shown patience and support, and who is eager to listen and discuss ideas; Roberto Magaña, who many years ago taught me about new avenues to explore, which eventually led to starting this work.

I wish to express many thanks to all my office mates at the VIS section, I know some of them had to put up with my bad moods more often than not. In chronological order: Sabine Iserhardt-Bauer, Matthias Ressel, Stefan Röttger, Dirc Rose (twice), Guido Reina, Joachim Diepstraten and Katrin Bidmon. The people at the VIS section gave me not only a workplace but also became a second family for me in Germany. From them I learned not only about computers and visualization, but also about the german way of life, culture and sense of humor. I am specially grateful to Katrin Bidmon, Dirc Rose, Guido Reina and Ulrike Ritzmann, friends who did not only teach me about themselves but also many things about me and my own life. I wish to thank Miguel Ángel, who always posed “difficult” questions; Ramón and Susana, who, without really knowing me, lend me a hand in a moment of need.

My sister Daniela and my parents Florencio and Isabel have always supported me during all of my life. During my stay in Germany, they always were there for me when I missed them the most, even if several thousand kilometers stood between us. This work would have never come to existence without them.

This work was financed by a scholarship granted by the German Service for Academic Exchange (DAAD) and the project SFB 382 of the German Research Foundation (DFG). I wish to express my gratitude towards Maria-Luise Nünning, my contact person at the DAAD, who was always ready to help.

Abbreviations

2D	two dimensional
3D	three dimensional
API	Application Program Interface
bpp	bits per pixel
COTS	commodity off the shelf
CPU	Central Processing Unit
e.g.	exempli gratia (for example)
et al.	et alia (and others)
Gb	gigabit (1024^3 bits)
GB	gigabyte (1024^3 bytes)
i.e.	id est (that is)
kB	kilobyte (1024 bytes)
Mb	megabit (1024^2 bits)
MB	megabyte (1024^2 bytes)
MHz	megahertz
MPI	Message Passing Interface
NOW	Network of workstations
OpenGL	Open Graphics Library
pixel	picture element
RGB	red, green and blue
RGBA	red, green, blue and alpha
SIMD	Single Instruction Multiple Data
texel	texture element
voxel	volume element

Contents

Abstract and chapter summaries **13**

Zusammenfassung und Kapitelüberblick **17**

1 Introduction **21**

1.1 Motivation 21
1.2 Contributions of this work 26

2 Direct volume visualization **29**

2.1 Visualization and computer graphics fundamentals 30
2.1.1 OpenGL fundamentals 31
2.1.2 Data representation 34
2.2 Volume visualization 34
2.2.1 Software-based volume rendering 35
2.2.2 Hardware-acceleration using dedicated hardware 35
2.2.3 Hardware-accelerated using commodity hardware 36
2.3 Physical basis of the algorithm 37
2.4 The transfer function 39
2.5 Hardware-accelerated volume rendering 40
2.6 State of the art 41

3 Parallel visualization **43**

3.1 State of the art in parallel rendering with PC clusters 43
3.2 Parallel volume rendering on clusters of PCs 45
3.3 Parallel compositing algorithm 45

4 Image compositing and volume rendering **51**

4.1 General image compositing 51
4.2 The compositing problem in volume rendering 52
4.3 Pre-multiplication 54
4.4 Pre-multiplication with OpenGL 55
4.4.1 Pre-multiplying in the transfer function 55
4.4.2 Pre-multiplying in the data itself 57

4.4.3 GL_EXT_blend_func_separate 57
4.4.4 GL_NV_register_combiners 58
4.4.5 GL_ARB_fragment_program 59

5 Software inter-brick compositing 61
5.1 Motivation 61
5.2 Compositing images in software 63
5.3 Compositing images in assembly 63
5.4 Discussion 67

6 System architecture 69
6.1 Message-Passing Interface 69
6.2 Network performance 69
6.3 Multithreading support 73
6.4 OpenGL integration 74
6.5 Access to the hardware 74

7 Application development with OpenGL 77
7.1 Motivation 77
7.2 Debugging OpenGL applications 77
7.3 Managing OpenGL extensions 79
7.3.1 Motivation 79
7.3.2 GLEW: The OpenGL Extension Wrangler 83

8 Remote access of visualization facilities 85
8.1 Motivation 85
8.2 OpenGL Graphics with the X Window System 86
8.3 Dynamic Linking 87
8.4 Implementation 89
8.5 Discussion 91
8.5.1 Results 92

9 Results and future perspectives 95
9.1 Implementation environment 95
9.2 Performance measurements 96
9.3 Conclusions 97

A Color plates 101

List of Figures

1.1 The visualization pipeline 22
1.2 Number of transistors used in consumer-grade GPUs 23

2.1 Basic 3D graphics pipeline. 30
2.2 From three-dimensional objects to pixels 30
2.3 OPENGL pipeline. 31
2.4 Rasterization stage in OPENGL 32
2.5 Per-fragment operations in OPENGL 32
2.6 Alignment of texture slices 37
2.7 Physical configuration for the general radiative transfer problem 38
2.8 Sorting of volume bricks. 41

3.1 General program flow for the controlling and rendering processes. 46
3.2 Integration of the volume renderers into a PC cluster environment. 47
3.3 Dataset partitioning for parallel rendering 47
3.4 Total compositing time as a function of image size using 16 processing elements. 49
3.5 Total compositing time as a function of the number of nodes participating in the composition 50

4.1 Opacity values interpreted as "coverage" information for a grey value of 0.5. . . . 52
4.2 Three slices to be composited together in associative fashion 53
4.3 Register combiner setup for multiplying the alpha component into the color components before the blending stage 58
4.4 Comparison between the usual non-pre-multiplied blending operations and the inter-slice blending operations performed using register combiners 60

5.1 Comparison of images composited in hardware using the traditional single-brick path and in software using multiple bricks. 64
5.2 Performance comparison for software blending across different systems. 66

6.1 MPICH layered architecture. 70
6.2 Measured network bandwidth as a function of message size 71
6.3 Transfer speed across four different nodes as a function of image size 72
6.4 Architecture of the interprocess communication. 75

7.1 Graphical user interface for `spyGLass` 80

8.1 GLX architecture 87
8.2 System architecture for remotely accessing OpenGL applications 89
8.3 Hardware-accelerated remote rendering using VNC 92
8.4 Performance measurements for hardware-accelerated remote rendering 93

A.1 Color version of figure 4.4 on page 60 102
A.2 Color version of figure 5.1 on page 64 102
A.3 Subset of the visible female dataset 103
A.4 Jet stream dataset 104

List of Tables

4.1 Some of the blending functions available in OpenGL 53

5.1 Performance for framebuffer read and write operations 62
5.2 Comparison between two alternative paths for performing image composition in hardware 62

8.1 Modified GLX functions 90

9.1 Time to render a single frame of the visible female and jet stream datasets. 98

Abstract and chapter summaries

Chapter 1: Introduction

The primary goal of this chapter is to motivate the subject in this thesis. The parallelization possibilities on the visualization pipeline are explained, and how some of this possibilities have been exploited. The evolution of commodity graphics hardware is also discussed, and serves as motivation for working on the parallelization of the last step of the visualization pipeline, namely the rendering of images. A comparison between “high end” proprietary visualization systems and commodity hardware is provided and serves as motivation for working on the parallelization of the rendering using clusters of PCs equipped with commodity graphics hardware. The characteristics of visualization applications are analyzed in the context of cluster-based computing, paying special attention to the composition of partial results for its display on a single device. The different approaches to this problem are discussed and it is argued that using general purpose processors instead of special purpose hardware is a viable alternative, even if distributed memory architectures have high inter-processor latencies and slow inter-connections. Finally a short summary of the contributions of this work is provided.

Chapter 2: Direct volume visualization

This chapter provides a brief introduction to volume visualization in general and direct volume visualization in particular. First a review of basic computer graphics concepts is provided, along with an introduction to the OpenGL graphics API. In particular the OpenGL pipeline is discussed, paying special attention to the rasterization and per-fragment operations stages. This provides the grounds for discussing the visualization of volumetric datasets using parallel rendering algorithms. Existing work in three different areas is reviewed: software-based rendering, hardware-accelerated rendering using dedicated hardware and hardware-accelerated rendering using commodity hardware.

A derivation of the essential algorithm for direct volume rendering using texture mapping hardware is provided, starting from physical concepts. After explaining the concepts of *transfer functions* and *classification*, the hardware-accelerated volume rendering technique using texture mapping is explained and its bottlenecks are discussed, and its parallelization is introduced. After this, the current state of the art in hardware-accelerated volume rendering is reviewed.

Chapter 3: Parallel visualization

This chapter introduces concepts common to all parallel visualization algorithms and then proceeds to make a review of the current state of the art in general purpose parallel visualization algorithms. The parallelization of the volume rendering algorithm is presented, and special attention is paid to the parallelization of the compositing stage: two well known algorithms for parallel compositing, direct send and binary swap, are discussed and compared and a justification for choosing one over the other is provided.

Chapter 4: Image compositing and volume rendering

This chapter discusses in more detail image composition in the context of parallel volume rendering. In particular the requirement for storing the opacity information along with color information is discussed. The manner in which the opacity information has to be computed in order to fulfill the associativity requirement of the parallelization is discussed, and this serves to introduce the concept of *alpha pre-multiplication*. Different pre-multiplication methods in the context of OPENGL are discussed. It is also shown why performing the pre-multiplication in the transfer function is not correct.

Chapter 5: Software inter-brick compositing

This chapter presents a software-based implementation for compositing partial results of the distributed volume rendering. First, a justification for not performing the composition in hardware is given. Then some basic remarks regarding in-software compositing are made. After this, assembler implementations of the compositing operation are given, one for the Intel's x86 architecture, and one for AMD's AMD64. The performance characteristics of this implementations are discussed.

Chapter 6: System architecture

This chapter discusses in more detail the architecture of the system developed during this work. First, a justification for the choice of MPI as a communication medium is provided, and the consequences of this choice are examined. Then the performance characteristics of the underlying network are discussed. Some low-level implementation details are discussed since the problems faced, as well as the decisions and solutions taken are non-evident, in particular how to integrate OPENGL applications into the cluster environment. Finally a discussion of the actual implementation as well as remotely accessing the facility is provided.

Chapter 7: Application development with OpenGL

This chapter focuses on a topic that was always present during the course of this work: application development using OPENGL, in particular debugging applications at the OPENGL level as well as accessing non-standarized functionality.

Chapter 8: Remote access of visualization facilities

In this chapter remote access of generic visualization facilities is presented. First a review of other solutions is provided, and then a detailed description of a generic method for remotely accessing visualization applications is provided. The only assumption that is made is that the application uses OPENGL and that it uses the X11 protocol for communication with the rendering hardware. After discussing the generic implementation, some optimizations specific to the distributed volume visualization are presented.

Chapter 9: Results and future perspectives

This chapter presents the results of this work, and makes an attempt to look into the future with respect to scalability of the developed solution. Attention is paid again to the compositing step of the distributed visualization process, and how this might become less of an issue in the coming years.

Zusammenfassung und Kapitelüberblick

Kapitel 1: Einführung

Das primäre Ziel dieses Kapitels ist es, das Thema dieser Arbeit zu motivieren. Sowohl die Parallelisierungsmöglichkeiten der Visualisierungspipeline als auch die Methoden, wie sie bisher ausgenutzt worden sind, werden erläutert. Die zeitliche Entwicklung der gebräuchlichen Standard-Graphikhardware wird besprochen und die Motivation für die Parallelisierung des letzten Schrittes der Visualisierungspipeline wird eingeführt, nämlich das Erzeugen von Bilder (*Rendering*). Ein Vergleich zwischen "hochleistenden" proprietären Visualisierungssystemen und gebräuchlicher Hardware wird dargestellt, und dient als Motivation, um mit der Parallelisierung des *Renderings* auf mit Graphikhardware ausgestatteten PC-Clustern zu arbeiten. Die Merkmale von Visualisierungsanwendungen im Kontext von Cluster-basierten Algorithmen mit Betonung auf das *Compositing* (Zusammenfassung) von Zwischenergebnisse werden analysiert. Verschiedene Ansätze für die Lösung dieses Problems werden erörtert und es wird argumentiert, dass es eine praktikable Möglichkeit ist, handelsübliche Prozessoren anstatt dedizierter Hardware einzusetzen, obwohl *distributed memory* (verteilte Speicher) Architekturen größere Latenzzeiten und langsamere Verbindungen zwischen Prozessoren haben. Schließlich wird eine kurze Zusammenfassung der Beiträge dieser Arbeit gegeben.

Kapitel 2: Direkte Volumenvisualisierung

Dieses Kapitel führt kurz in die allgemeine und direkte Volumenvisualisierung ein. Zuerst wird ein Überblick über grundlegende Computergraphik-Konzepte gegeben, ebenso eine Einführung in das OPENGL Graphik-API. Die OPENGL *Pipeline* wird dargestellt, wobei besonders auf die Rasterisierung und die *pro-Fragment* Operationen eingegangen wird. Dies bildet die Grundlage, um über die Visualisierung von volumetrischen Datensätzen unter Verwendung paralleler Rendering-Algorithmen zu diskutieren. Vorherige Arbeiten auf drei verschiedenen Gebieten werden besprochen: Software-basiertes Rendern, hardware-unterstütztes Rendern mit dedizierter Hardware und hardware-unterstütztes Rendern mit handelsüblicher Hardware.

Ausgehend von physikalischen Konzepten wird eine Herleitung des grundlegenden Algorithmus zum direkten Volumenrendern mit *Texture-mapping* Hardware gegeben. Das Konzept der Transferfunktionen und die Klassifierung wird erklärt, anschließend wird die Technik des hardware-unterstützten Volumenrenderns mit *Texture-mapping* erläutert, dessen Grenzen dargelegt und in dessen Parallelisierung eingeführt. Schließlich wird der aktuelle Stand des hardwareunterstützten Volumenrenderns dargelegt.

Kapitel 3: Parallele Visualisierung

Dieses Kapitel führt in Konzepte ein, die alle parallelen Visualisierungsalgorithmen gemeinsam haben und gibt einen Überblick über die aktuellen allgemein üblichen Algorithmen zur parallelen Visualisierung. Die Parallelisierung des Algorithmus zum Rendern von Volumina wird vorgestellt. Besondere Beachtung findet dabei die Parallelisierung des *Compositing*: zwei allgemein bekannte Algorithmen zum parallelen *Compositing*, *direct send* und *binary swap* werden besprochen und verglichen, und eine Begründung für die getroffene Wahl wird gegeben.

Kapitel 4: Bild-*Compositing* und Volumenrendern

In diesem Kapitel wird eine detalliertere Erörterung des Bild-*Compositing* im Kontext parallelen Volumenrenderns gegeben. Insbesondere wird die Notwendigkeit die Opazität zusammen mit der Farbinformation zu speichern, erläutert. Die Methode, um die Opazität so zu berechnen, dass die notwendige Assoziativität der Parallelisierung gegeben ist, wird vorgestellt. Dies dient dazu, das Konzept des vormultiplizierten Alphawertes einzuführen. Unterschiedliche Methoden zur Vormultiplikation mit OpenGL werden erläutert. Ebenso wird gezeigt, warum es nicht korrekt ist, die Werte in der Transferfunktion vorzumultiplizieren.

Kapitel 5: Software *inter-brick compositing*

Dieses Kapitel stellt eine software-basierte Implementierung zum *Compositing* von Zwischenergebnissen des verteilten Volumenrenderns vor. Zuerst wird eine Begründung gegeben, warum die Durchführung des *Compositing* nicht in Hardware stattfindet. Anschließend werden einige grundsätzliche Bemerkungen bezüglich des *Compositing* in Software gemacht, und Assembler-Implementierungen des *Compositing* vorgestellt: Eine für die Intel x86 Architektur und eine für die AMD AMD64 Architektur. Die Performanzmerkmale dieser Implementierungen werden diskutiert.

Kapitel 6: Systemarchitektur

Die Architektur des in dieser Arbeit entwickelten Systems wird in diesem Kapitel genauer beschrieben. Zuerst wird eine Begründung dafür gegeben, warum MPI als Kommunikationsschnittstelle ausgewählt wurde, und die Konsequenzen dieser Wahl werden untersucht. Die Performanzmerkmale des darunterliegenden Netzwerkes werden beschrieben. Einige Implementierungsdetails werden genauer erläutert, da sowohl die aufgeworfenen Probleme ebenso wie die getroffenen Entscheidungen als auch deren Lösung nicht offensichtlich sind; inbesondere die Integration von OPENGL-Anwendungen in die Cluster-Umgebung. Schließlich folgt eine Erläuterung der tatsächlichen Implementierung und des entfernten Zugangs zum Cluster.

Kapitel 7: Anwendungsentwicklung mit OPENGL

Dieses Kapitel hat als Schwerpunkt den Entwicklungsprozess von OPENGL-Anwendungen, insbesonders das Debugging auf OPENGL-Ebene und den Zugriff auf nicht-standarisierte Funktionalität.

Kapitel 8: Entfernter Zugang zu Visualisierungsmöglichkeiten

Dieses Kapitel stellt eine generische Methode für den entfernten Zugang zu Visualisierungsanwendungen vor. Zuerst wird ein Überblick über existierende Lösungen gegeben und anschließend wird eine generische Methode genauer beschrieben, mit der auf Visualisierungsanwendungen entfernt zugegriffen werden kann. Die einzige Annahme die gemacht wird ist, dass die Anwendung auf OPENGL basiert und das X11-Protokoll für die Kommunikation mit der Hardware verwendet wird. Anschließend werden Optimierungen, die speziell die verteilte Volumenvisualisierung betreffen, vorgestellt.

Kapitel 9: Ergebnisse und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit vorgestellt, und ein Ausblick im Hinblick auf die Skalierbarkeit der entwickelten Lösung wird gegeben. Besonders wird auf den *Compositing*-Schritt des Prozesses zur verteilten Visualisierung eingegangen, und darauf wie dies in den nächsten Jahren weniger kritisch sein wird.

Chapter 1

Introduction

1.1 Motivation

Researchers and engineers, that is, the end-users of visualization techniques, strive for better, faster and easier ways of *understanding* their data. Visualization is *not* about “pretty pictures”, its ultimate purpose is comprehension. Whether this comprehension arrives from simple diagrams or complex interactive simulations is only an “implementation detail”, it is not a goal *per se*. Thus, visualization specialists are faced with the problem that their users are producing, on a daily basis, larger datasets which they wish to visualize. The visualization community’s answer to that problem is *interactive* visualization, and the commonly accepted definition is that the user is able to directly manipulate the subject under study, with the representation on the display system being updated at a rate of at least 10 times a second. When the size of the datasets exceeds the processing capabilities of the available hardware, one way of achieving this goal is to employ multiple processing elements in parallel. Looking at the visualization pipeline (figure 1.1), one can recognize multiple points where parallelization might yield benefits: going from raw data to “visualization data” is a filtering process which is at least directly proportional to the size of the input data, and in general well paralellizable; the mapping of data structures to visualization structures can also be subject to parallel processing, and this was the focus of many research papers of the last decade, which dealt with reduction of the datasets in order to make them more amenable to treatment with interactive computer graphics techniques. The last step in the pipeline, the rendering of the data, has been also approached with parallelization in mind. Textbook examples of this are raytracing and radiosity-based algorithms, with processors working in unison in order to generate a single final image or working in an interleaved fashion in order to generate different frames of a single animation. In this case scalability in two areas is being sought after: scalability on the size of the input data and scalability on the update rates of the rendering itself. An orthogonal approach is to distribute the rendering process across processing groups, with each group generating images for a single display device, which when observed all together make for a single coherent visualization of the data. Here scalability on the final image resolution is being targeted. In this regard, hardware accelerated visualization techniques are also a form of parallelization: by recognizing the implicitly parallel nature of the

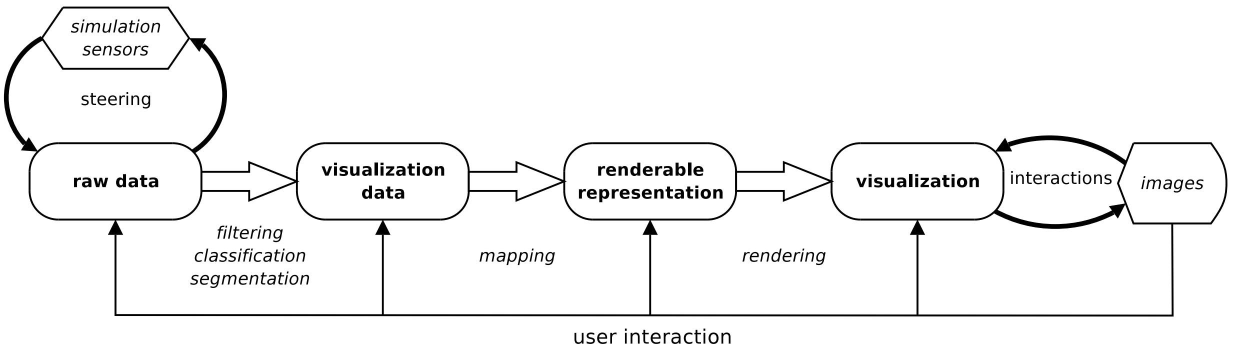


Figure 1.1: The visualization pipeline

rasterization process, hardware designers are able to improve on the design and performance of the graphics chips, to the point where these have become highly-pipelined highly-specialized SIMD processors.

Commodity graphics hardware is of particular interest. Its development has been pushed forward over the last few years by the PC gaming industry. As a result, today's commodity graphics hardware has reached and in many aspects surpassed the capabilities of so-called high-end solutions. The initial efforts were geared towards improving performance alone, with little or no advance in the area of programmability of the hardware. During a long period of time, graphics card makers concentrated themselves on providing hardware implementations of functions exposed by the graphics APIs such as OPENGL, that is, they implemented a fixed-function pipeline in hardware. NVIDIA was one of the first hardware vendors to develop and implement *extensions* to the basic APIs which introduced small amounts of programmability into them. This flexibility was welcomed by the programmer community, which started to use the features and therefore implicitly or explicitly *recommend* certain hardware vendors over others, giving them reason to research and develop even more extensions – proprietary in most cases – to the fixed-function pipeline. This trend has continued until today, when almost fully programmable GPUs are available on the market. Thanks to standardization efforts from several parties, these features are now accessible in hardware- and platform-independent fashions. The evolving complexity of these components reflects itself as an exponential increase in the number of on-chip transistors. This exponential behaviour is dubbed *Moore's law* in reference to a 1965 paper by Gordon Moore [Moo65], in which he provided arguments as to why the reduced costs of integrated electronics would continue to provide advantages and thus fuel the development of larger and more complex components. The wide-spread formulation of Moore's law states that the number of transistors on a CPU doubles every 18 months. As seen in figure 1.2, the development pace of GPUs seems to be faster than that of CPUs.

The advantages brought by this kind of flexibility have not remained in the gaming realm. By exploiting the capabilities of flexible graphics pipelines, researchers in different areas such as volume rendering, photorealistic and non-photorealistic rendering have been able to improve not

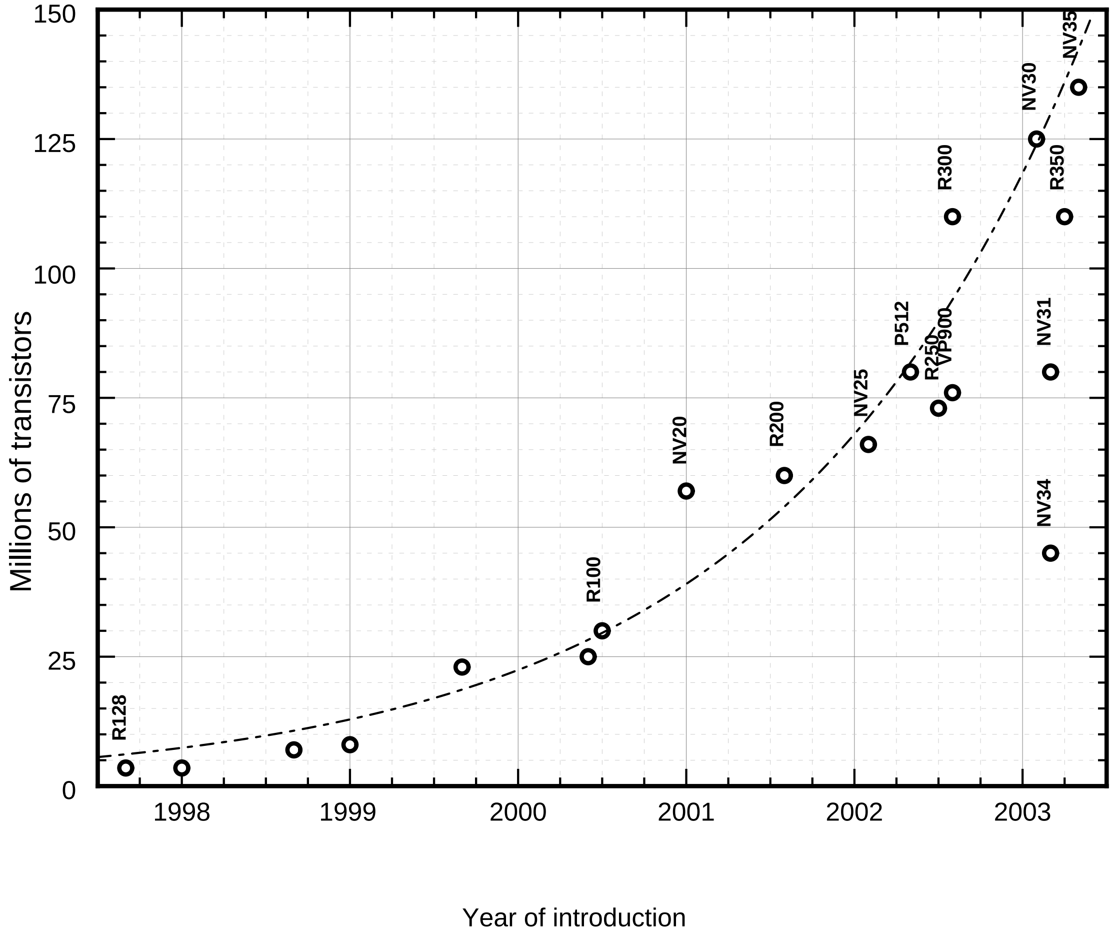


Figure 1.2: Number of transistors used in consumer-grade GPUs. The dashed curve is an exponential fit $N = 4.25 \times 2^{\frac{4}{5}(t-1997)}$, i.e. a duplication of the number of transistors every 10 months. This fit does not take into account the introduction of products targeted towards lower-end markets, such as the NV34. R100, R200, R250, R300 and R350 are several generations of the Radeon chip produced by ATI. Chips labeled with the letters NV are several generations of the GeForce product line by NVIDIA, as well as the Riva 128 (R128) chip. P512 is the Parhelia-512 product from Matrox. VP900 is the Wildcat VP product from 3Dlabs. Source: Product press releases from the respective vendors.

only the performance but also the quality of the images they produce (some exemplary papers are [EKE01], [SW01], [FMS02]).

Looking at the latest generation of consumer-oriented graphics hardware, it is remarkable that it not only provides a much larger degree of programmability, but also better accuracy and more on-board memory, making the comparison against “high-end” hardware hard to avoid. Looking at the specifications of SGI’s InfiniteReality4 (IR4 for short), it is easy to spot two areas where it still holds an advantage. The first of them is the amount of on-board memory that can be dedicated to textures: the IR4 can manage up to 1 GB of RAM per pipe, whilst current offerings from ATI and NVIDIA are still limited to 256 MB of on-board memory, with 64 and 128 MB configurations being commonplace. The reason for this limitation is clear: adding more memory would not only increase the power requirements of the board but also its street price and therefore its visibly. There is also an architectural problem to consider: these boards are still designed with Intel x86 class hardware in mind, where memory addresses are limited to 32 bits. Because of the architecture design, the graphics memory is usually mapped into the normal 32 bit address space[1], thus limiting the amount of system memory usable by individual processes. The other interesting aspect of the IR4 is that it can drive displays of up to 133 million pixels, in contrast to PC hardware, which is still limited to resolutions one order of magnitude smaller. A question poses itself naturally then: how can we take advantage of the much faster development pace and much more flexible architecture of consumer grade graphic boards while keeping the desirable characteristics of systems like the IR4?

SGI’s answer to that question is to integrate commodity graphics hardware into their own systems, as seen in their “UltimateVision” product line. The problem is that SGI is limiting consumers to comparatively small configurations, and the vendor lock-in effect is still present. Faced with this, the most natural way of achieving the desired result is using small clusters of PCs and install graphics adapters on each node. The idea of clustering *commodity off the shelf* (COTS) components in order to achieve supercomputer-like performance took off after it was initially demonstrated at the Center of Excellence in Space Data and Information Sciences at NASA Goddard Space Flight Center and subsequently published by Becker et al [BSS$^+$95]. At that time, PC-class hardware had not only fallen down to prices that made the idea attractive, but their performance had risen up to levels that balanced out the latency introduced by using low speed networks as intercommunication medium. This configuration is known as a *network of workstations* (NOW) or a Beowulf-like cluster (in reference to the name that Becker and his team gave to their system). Nowadays PC clusters are widely accepted as a good alternative to more expensive proprietary supercomputers[2]. Their application domains overlap, but scalability

[1]For example, on a x86 Linux system with NVIDIA hardware and the vendor supplied drivers, the graphics card memory is mapped into the same region where shared memory segments and shared libraries reside. This region starts at address `0x40000000` and ends somewhere below `0xbffff0000`. This is the same region used for allocating large chunks of contiguous memory. This means that, in effect, installing a graphics card with 1 GB of on-board memory reduces the total system memory usable by a graphics application as data space by 1 GB. Because of the design of the Linux kernel on this architecture, this leaves less than 2 GB usable for the application. For practical purposes, this problem vanishes on 64-bit architectures.

[2]`http://www.top500.org/` and `http://clusters.top500.org/` contain performance and configuration data for existing installations, the former including also supercomputers, the later being dedicated only to clusters.

of cluster-based solutions is still an issue because of the limited communication bandwidth across nodes. Nevertheless, systems with more than a thousand nodes have been demonstrated and are in use. With the advent of 64-bit computing on the desktop, much more powerful systems are being built around architectures like Intel's Itanium and AMD's Opteron. Even vendors like SGI are shifting towards the use of commodity processors for their systems. Their Altix line uses Itanium CPUs along with their proprietary Cray-bus for interconnection, which, being an specialized solution, still offers better communication performance and scalability.

Visualization applications are unique in the context of cluster-based computing. First and foremost, as argued before, visualization is an *interactive* task, it cannot be performed *off-line*, in batch mode. It has as a rule higher I/O demands than numerical simulations, the typical application domain of PC clusters. It also has different access patterns: while numerical simulations tend to have a high degree of locality, visualization applications, by nature, need to access more widespread data. Data partitioning is also in general of a different nature, since many of the algorithms are *view-dependent*, a component not present in simulations. In general, a high CPU load exists, because graphics primitives need to be recomputed on a frame-per-frame basis. Except in a subset of cases, partial images need to be aggregated, which means the computational cost is proportional to the image size, which is a factor that cannot be ignored, since effective visualizations in this context call for millions of pixels per frame. On top of that lies the problem of image transport: since it is often impractical or even impossible to perform the visualization task at the installation place of the cluster, there is a need for transporting images over a distance, whilst still keeping interactivity.

The problems deriving from the small amounts of texture memory available on COTS graphics hardware can be overcome by developing techniques to perform adequate data partitioning and distribution across processing elements. The implication of the data partitioning is, as several authors [SFLS00, MHE01, HHN$^+$02] have discussed, that rendering on PC clusters calls for hybrid sort-first sort-last algorithms. In turn, the post rendering sorting calls for high performance image compositing methods. Some groups, e.g. Heirich and Moll [HM99], Stoll et al [SEP$^+$01], have developed custom hardware solutions which address this necessity. The problem with this kind of solution is that it falls out of the category "commodity hardware": these are custom-made parts which are expensive and complex to produce and deploy, and generally not found on the market. Every other component in this approach has become a commodity: CPUs, in the broadest sense of the word, are mass-marketed and nowadays it is easy to find "desktop" PCs that surpass the performance of the "high-end" workstations of the 90s for just a fraction of the price. The last five years are witness to an explosive development in the area of PC-based entertainment, taking some market-share even from the more mature gaming console sector. This growth reflects itself as high-performance graphics processors becoming a commodity, too. The third component which is also a commodity part is the interconnection network: high-speed low-latency networks are still much more expensive than their low-end counterparts, but the increasing demand for higher data transfer rates, particularly in the context of distributed computing and data warehouse environments, has had a two-fold effect: On one hand, high-end interconnection solutions such as Myrinet and Quadrics are becoming faster and at the same time less expensive, and solutions based on Infiniband are gaining momentum, their performance is getting comparable to that of proprietary technologies [Coh03] and they are likely to become more affordable; on the other hand, the low-cost alternatives, based around the Ethernet standard, are becoming faster, more widespread and more reliable. By using commodity parts for these components, it is possible to take advantage of the mass market benefits: open competition among vendors leads to faster, featureful and more reliable products; if a part goes bad, it is possible to replace it with a new one which is likely to be easy to find, thus minimizing downtime; it is also possible to plan for piecewise upgrades, at a pace dictated by the consumer instead of the producer of the hardware; reusing parts also becomes possible, once state-of-the-art components in the visualization cluster can be relocated for other uses, which might not require high-end hardware, but can benefit from it. With this in mind, a software-based approach to image compositing has been considered, namely using the available CPU power of the cluster to perform this task. The advantage of a software approach is that it is more flexible and reduces the maintenance cost. The downside is the higher CPU and network utilization. As CPUs and network technologies evolve, this problem diminishes.

1.2 Contributions of this work

Instead of trying to develop a general parallelization framework for visualization applications, this work focuses on the parallelization of *direct volume visualization on uniform meshes*. Direct volume visualization allows the user to have a global perception of the data while still being able to focus on specific features. This is achieved by assigning opacity values to all data points. It is a very versatile algorithm that allows for example for the extraction of isosurfaces from the data, working with multimodal datasets, and interactive clipping and slicing of the data, while maintaining constant or almost constant display framerates. Its major disadvantage is that implementations that exploit hardware acceleration require large amounts of texture memory. It also demands very high rasterization rates and high memory bandwidth, due to the complexity of the algorithm being proportional to the number of data points. It has already been shown that these effects can be counteracted by using multiple rasterizers on shared memory architectures (e.g. Li et al [LWMT97]). The downside to this solution is that the cost of such hardware is at least twenty times more expensive when compared to the price of a cluster of PCs with the same aggregated computing power. In addition to the better price-to-performance ratio, using clusters of PCs is also attractive because of a lower total cost of ownership, better technology tracking, better modularity and flexibility and higher scalability as discussed before.

Moving from shared-memory architectures to PC clusters shifts the performance bottleneck from the rendering itself to the compositing step. The “binary swap” algorithm as presented by Ma et al [MPHK94] has been used traditionally for this purpose, even if it is theoretically less efficient than the alternative “direct swap” algorithm presented by Neumann [Neu93]. The reason for this is that the binary swap algorithm puts a lower load on network topologies that are not fully switched. With fully-switched full-duplex networks the direct send algorithm can be implemented in more efficient fashion. Orthogonally to this, by exploiting the SIMD operations available on modern CPUs, this work shows how to achieve compositing rates that surpass those imposed by the need for an interactive visualization system. Additionally, the problems arising from partitioning the volume rendering operation across several nodes and then compositing the intermediate results are discussed.

During the course of this work, other problems were attacked. The first and foremost goal is gaining remote access to high performance visualization facilities and eventually providing access via web-based services. A second problem was dealing with the fact that high-performance graphics programming APIs in general, and OPENGL in particular, are in a state of constant evolution, which introduces problems related to writing truly portable applications based on them. These problems and their solutions are described in the later chapters.

Chapter 2

Direct volume visualization

Kaufman et al define [KHKS94] *volume visualization* as:

> “[. . .] a method of extracting meaningful information from volumetric data sets through the use of interactive graphics and imaging. It addresses the representation, manipulation, and rendering of volumetric data sets, providing mechanisms for peering into structures and understanding their complexity and dynamics. Typically, the data set is represented as a 3D regular grid of volume elements (voxels) and stored in a volume buffer (also called cubic framebuffer), which is a large 3D array of voxels. However, data is often defined at scattered or irregular locations that require using alternative representations and rendering algorithms.”

This definition encompasses all the important aspects of modern theory and practice in the field. First and foremost, volume visualization practitioners continuously strive for higher interactivity. Over the years the working definition of “interative framerates” has evolved along with the rendering algorithms.

Kraus argues regarding the acceptance of the algorithm [Kra03]:

> “While volume graphics, i.e., the rendering of volumetric objects, has found its way into movie and video game productions, volume visualization has only found some niches, although it has been successfully applied in almost all sciences [. . .]
>
> However, volume visualization is by far less popular than one might expect from the large range of potential applications. In particular, direct volume visualization is frequently rejected for quantitative analyses because of the fuzzy nature of the resulting images. In these cases, isosurfaces are often strongly preferred. Thus, the visualization of medical scans, e.g., CT (computer tomography) or MR (magnetic resonance) scans, is still by far the most important field of application of direct volume visualization.
>
> There are many reasons for this limited popularity of volume visualization apart from the mentioned “fuzziness”: the insufficient support by standard graphics hardware, a lack of non-commercial (sic) tools, uncomfortable and inefficient interfaces of

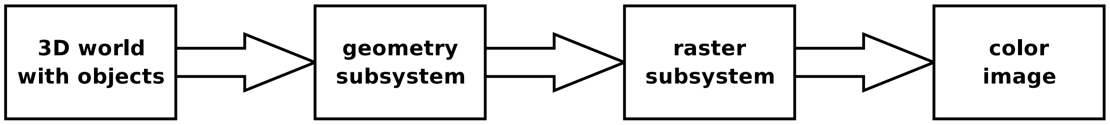


Figure 2.1: Basic 3D graphics pipeline.

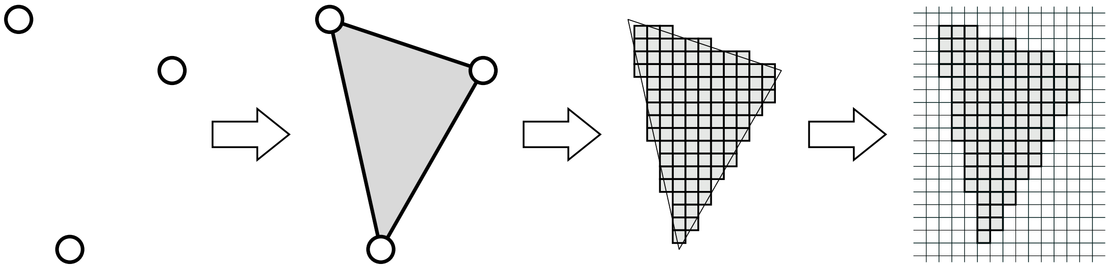


Figure 2.2: From three-dimensional objects to pixels. An object is described in terms of vertices and their connectivity which are then passed as primitives to the graphics API. These primitives are then rasterized and placed on the framebuffer as images.

> existing tools, the rare use of volume visualization for publications outside of the visualization community, and the lack of education about volume visualization, to name just a few. Unfortunately, some of these reasons reinforce each other."

Nevertheless, volume visualization has proved itself as an effective technique for the exploration of large and complex data sets, even if its main application domain is still medical visualization.

2.1 Visualization and computer graphics fundamentals

Working with "3D graphics" on a two-dimensional display means mapping three-dimensional representations of objects to two-dimensional raster images. Looking at figure 2.1, one starts with a digital representation of the world, which consists of objects made up of points, curves, surfaces and volumes with associated material properties, which is then converted to a geometrical representation more appropriate for the underlying graphics subsystem, e.g. using only primitive objects such as points, lines and triangles, as illustrated in figure 2.2. Working with this representation, the geometric subsystem applies *geometric transformations* and clips the results to the visible image area in order to obtain primitives described in *normalized device coordinates*. At this stage, vertices are assigned colors according to the illumination model in use. These transformed and lighted primitives are then handed to the rasterization subsystem. Here "fragments" are generated from the primitives, which are then shaded and later stored as pixels of the final image.

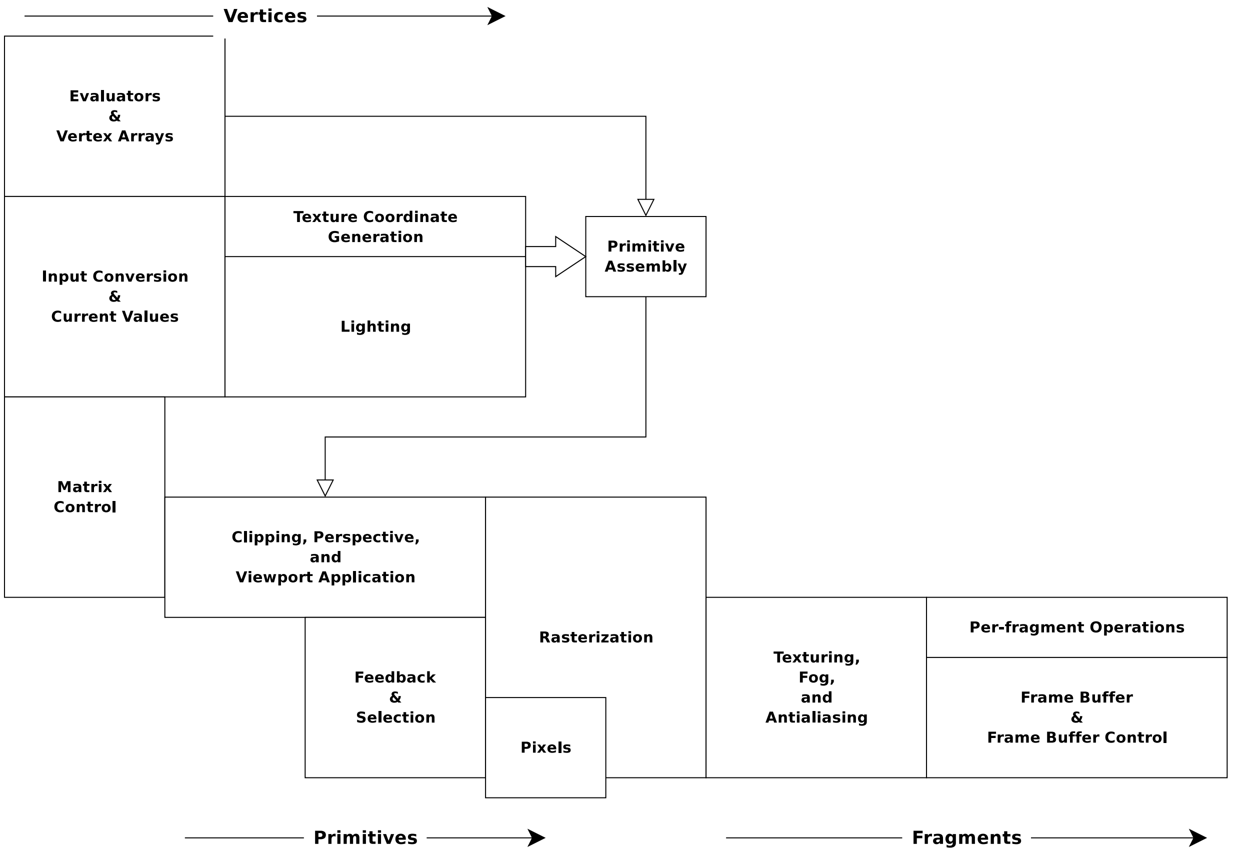

Vertices
Evaluators & Vertex Arrays
Input Conversion & Current Values
Texture Coordinate Generation
Lighting
Primitive Assembly
Matrix Control
Clipping, Perspective, and Viewport Application
Rasterization
Feedback & Selection
Pixels
Texturing, Fog, and Antialiasing
Per-fragment Operations
Frame Buffer & Frame Buffer Control
Primitives
Fragments

Figure 2.3: OpenGL pipeline. This is an overview of the OpenGL state machine as presented in [Kil96].

2.1.1 OpenGL fundamentals

The OpenGL graphics system is a software interface to graphics hardware. The API and behaviour is defined by a standard, which is in turn regulated by a multi-vendor group of graphics hardware manufacturers called the *architecture review board* (ARB). In this sense, the development of OpenGL itself is open: vendors are free to research and develop new functionality, integrate it into their hardware and propose modifications to the standard in order to support it, which can then be adopted by other vendors. The OpenGL API provides access to 2D and 3D operations, and has been implemented on a wide range of platforms and it is independent of the underlying windowing system.

In the specific case of OpenGL the transformation and rasterization pipeline from figure 2.1 has the form depicted in figure 2.3. Of particular interest for hardware accelerated volume rendering are the rasterization (figure 2.4) and per fragment operations (figure 2.5) stages.

In the rasterization stage is where texture application occurs. Texturing maps a portion of one (*single-texturing*) or more (*multi-texturing*) images onto each primitive for which texturing

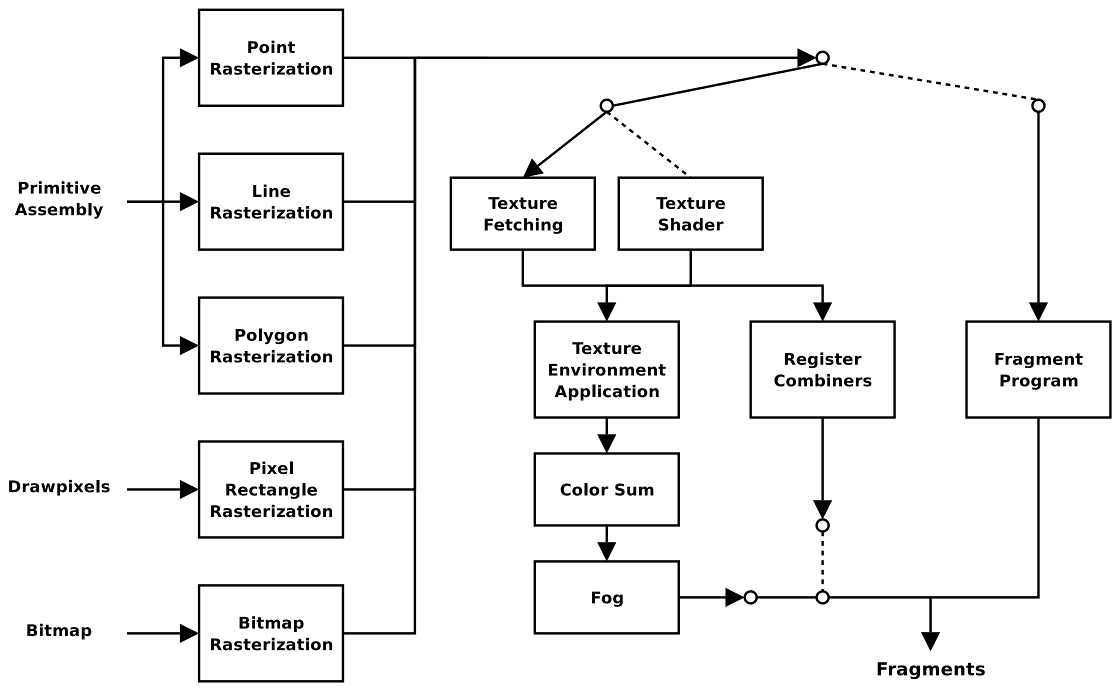


Figure 2.4: Rasterization stage in OPENGL. This corresponds to figure 3.1 in [SA03], expanded with information from the `NV_register_combiners` [Kil02], `NV_texture_shader` [Kil03] and `ARB_fragment_program` [Lip03] specifications.

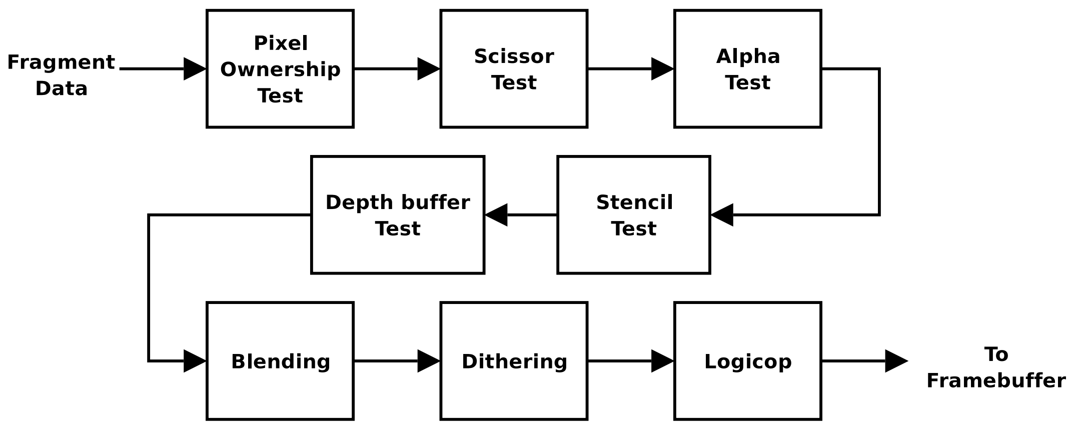


Figure 2.5: Per-fragment operations in OPENGL. Corresponds to figure 4.1 in [SA03].

is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's *texture coordinates* to modify the fragment's primary RGBA color. These coordinates are specified on a per-vertex basis and are interpolated for each fragment. The dimensionality of the images used for texturing is not directly constrained by the standard, but only the one-, two- and three-dimensional cases are defined. Texture image data can be specified in a variety of formats, which is then converted to RGBA data before application. The actual manner in which textures are applied to fragments can be controlled via a fixed set of *texture functions* indicated by the *texture environment*. For example, the fragment's RGBA value can be replaced by that of the texture, or it can be modulated by it. This is also used to specify the way in which textures interact with each other for the multi-texturing case.

In the per-fragment processing stage (figure 2.5) is where several tests take place, all of which can be enabled or disabled on an individual basis. If a fragment does not pass a given test, it is discarded and the framebuffer's and auxiliary buffers' contents remain unchanged. For example, a fragment can be discarded based on its depth, alpha or stencil values. This is also the stage where blending of incoming data and framebuffer data is performed according to the current *blend equation* and *blend factors*, selected out of a fixed set. The blend equation determines the operations performed between the two operands, and the factors determine which components are used and how.

OpenGL defines an *extension mechanism* which allows hardware manufacturers to support functionality not foreseen at the time the specification was written. Extensions do not change the default state of the state machine, that is, a program running on a platform that does not support a specific extension will run the same on another platform that does support it. Programs need to be modified in order to take advantage of the new functionality. In figure 2.4 some modifications to the state machine introduced by extensions are depicted:

- The `NV_register_combiners` extension replaces part of the rasterization stage with a more flexible – yet still predefined – set of operations carried at the fragment level by so-called *combiners*. The operands are a set of *registers* which can be filled with incoming color data, texture data or user-specified constants. The outputs can be the primary and secondary colors, textures or spare registers to be used as input for the next combiner. A related extension, `NV_texture_shader`, exposes twenty-one "shader" operations. Depending on the operation, the mapping from texture coordinates to an RGBA color may depend on the given texture unit's state or the results of previous operations. This enables *dependent texture lookups*, that is, using the result of a texture fetch operation as input for a subsequent texture lookup.

- The `ARB_fragment_program` extension replaces part of the rasterization stage with a fully programmable unit. For this purpose, an assembler-like language is used. It provides vector operations (SIMD instructions operating on 4-component data), scalar operations, texture-fetching operations and an operation to kill fragments. The maximum program length is implementation-dependent, and the specification provides the minimum number of instruction of different types that have to be supported by any implementation. The programs can output RGBA and depth information.

2.1.2 Data representation

Volumetric datasets associate a value with each point on a 3D mesh, that is, a *scalar field* is defined on the volume. *Multi-modal datasets* contain data obtained using different acquisition techniques, and therefore associated with different meshes in the general case. In *time-dependent datasets*, all the points in the volume are defined for multiple points in time.

The most general classification for meshes is in two groups: *structured* and *unstructured*. For structured meshes the connectivity is given implicitly, while for unstructured ones it needs to be given explicitly. Structured meshes can be further classified into *uniform*, *rectilinear* and *curvilinear* meshes. Uniform meshes have cells of uniform shape and size. For the case of volume visualization, it is customary to use uniform meshes with orthogonal axes, that is, with cubic cells. Rectilinear meshes are more general since the sizes of the cells might depend on their relative position with respect to some arbitrary point, that is the coordinates (x,y,z) of the vertices are a function of their indices (i,j,k) in the mesh, like $(x(i),y(j),z(k))$. For curvilinear meshes the position of the vertices in the mesh is not restricted in the same way.

Since unstructured meshes have no predefined connectivity, this information needs to be stored along with the values, therefore requiring larger data structures. Their advantage is a greater flexibility and adaptability. Unstructured meshes can be used to discretize domains of any shape and topology, and their resolution can be adapted locally. The shape of the cells in unstructured meshes is not necessarily constrained, but one of the most important kinds are the *simplical meshes*, i.e., meshes where cells are simplices (e.g. triangles in two dimensions, and tetrahedra in three dimensions).

2.2 Volume visualization

Visualization of volumetric data is a challenging task since it is necessary to deal with occlusions in order to deliver effective visualizations. One way of achieving this goal is the extraction of a subset of the volume in order to minimize occlusions. The most simple example is the use of *slicing planes*, that is, intersecting the volume with a plane and rendering only the intersected voxels. Another possibility is to extract and render only one isosurface from the volume. The disadvantage of these methods is that a large part of the volume is eliminated and the user is not given enough contextual information to understand the displayed image.

Direct volume rendering is a different approach to this problem: a continuous trade-off between occlusion and data visibility is made, by assigning opacities to all data points. Points with high opacities are more prominent in the final image but they occlude other points. Points with low opacities are less prominent but at the same time less occluding. This provides the user with enough context information to identify features on the image more easily. Another important feature of the algorithm is that its performance is usually independent of the viewing parameters, i.e. it is possible to deliver visualizations at constant or almost constant refresh rates.

There are many volume rendering algorithms, but they can be classified into two large groups: image-order and object-order algorithms. Image-order algorithms build the final image on a pixel by pixel basis, by casting rays through the volume and computing the contribution of the data along these rays. One of the earliest of this sort of algorithm was presented by Levoy [Lev88]. Object-order algorithms on the other hand process the data on a sample by sample basis, and research has focused on efficient ways of computing the contribution of each of them to many image pixels at once.

There are three main approaches to volume rendering: software-based rendering; hardware-accelerated rendering using dedicated hardware; hardware-acceleration using commodity hardware.

2.2.1 Software-based volume rendering

Early research on volume rendering focused on how to achieve interactive framerates without visible image quality degradation. For this purpose massively parallel architectures were researched and developed. For example, Elvins [Elv92] presented an implementation running on a 64-processor nCube machine. This implementation required 94 seconds to render a single frame of a $200 \times 180 \times 91$ dataset. Interprocessor communication represents the largest bottleneck in this implementation. Elvins discussed several kinds of optimizations to reduce communication requirements: tracking bounding boxes in the image plane, splitting the compositing task across two processors and grouping slices in a single processor.

Wittenbrink and Harrington [WH94] presented a scalable architecture running on the Proteus system (based on the i860 processor). This achieved rates of 2 seconds per frame running on 32 processors for a 256^3 dataset and a 256×256 image. Their implementation uses an explicit partitioning of the input volume, a permutation warp of the data and a communication pattern that avoids saturating the underlying communication network.

Wilhelms et al [WGTG96] presented a software multi-resolution renderer that works on irregular grids. By working on a hierarchical representation of the data, a potentially visible set is determined. The authors had a purely software-based renderer as a goal due to the wide availability of general purpose processors and the unavailable of dedicated graphics hardware. The authors report a speed up of 3.25 when the algorithm runs on 4 processors.

Palmer et al [PTT97] presented a ray casting algorithm to take advantage of the deep memory hierarchy on the Power Challenge Array. By exploiting the memory access characteristics of this proprietary platform, from the processor cache to the interconnection network, they implemented a logical global address space for volume blocks with caching. They demonstrated this renderer using a 7.1 GB dataset rendered using 64 processors. They achieved refresh rates of 3-4 seconds per frame.

2.2.2 Hardware-acceleration using dedicated hardware

Ray et al [RPSC99] provide a survey of five special purpose volume rendering architectures: VOGUE, VIRIM, Array Based Ray Casting (VIZAR), EM-Cube, and VIZARD II.

The VOGUE architecture [Kni95] was developed at the University of Tübingen, Germany. One rendering engine provides high-quality, volume-rendered images with multiple light sources using four custom VLSI chips. The main goals of VOGUE are flexibility and compactness. It operates in three different modes that offer a trade-off between performance and image quality. Multiple boards can be connected to one another in order to increase interactivity.

The VIRIM architecture [GPR+95] has been developed and assembled at the University of Mannheim. It aims at delivering high image quality. It consists of a geometry unit and a ray casting unit. The geometry unit is responsible for interpolation and gradient calculation; the ray casting unit is responsible for implementing the actual ray casting algorithm. It is possible to use multiple modules in parallel, but the dataset needs to be replicated among them.

VIZAR [Dog95], developed at the University of New South Wales, is an object order ray casting architecture. This architecture consists of two parallel pipelined arrays used to rotate the dataset and to cast rays. These rotation arrays are connected between n memory modules and 1.5 n rendering pipelines, where n is the resolution of the dataset. In the second array, intersections with voxels are determined by using nearest neighbor or zero order interpolation. Each rendering pipeline performs shading and composition for a given scanline. In addition, the system is composed of a double-buffered input memory (it is possible to display one dataset while loading another), memory swapping array, and a framebuffer.

The VIZARD II [MKS98] architecture, developed at the University of Tübingen, aims towards interactive ray casting on desktop computers [33]. It is designed to interface to a standard PC system using the PCI bus. The dataset is stored in four interleaved memory banks along with a precomputed gradient index, segmentation index, and gradient magnitude for each voxel. The combination of precomputed gradients, caching, and early ray termination reduces the bandwidth requirements of the memory system.

EM-Cube is a commercial version of the Cube-4 architecture originally developed at the State University of New York at Stony Brook. The EM-Cube architecture led to the VolumePro board [PHK+99]. The last generation of the VolumePro boards have 2-4 GB of on board texture memory in a dual board configuration, and support 8-, 16- and 32-bit voxels. It has hardware for gradient estimation using central differences and classification via a 36-bit lookup table (24 bits for RGB, 12 bit for A). It performs Phong illumination using a precomputed reflectance map for the diffuse and specular terms.

2.2.3 Hardware-accelerated using commodity hardware

A cornerstone in volume visualization is the introduction by Cabral et al [CCF94], in 1994, of an object-order technique which exploits *texture-mapping hardware*. This technique is the basis for many hardware-accelerated volume rendering algorithms which are being successfully used today. The basic idea is to render a number of volume slicing planes (*slices* for short) one after the other in a consistent fashion: starting with the plane that is furthest away from the viewer and proceeding towards the closest one, or the other way around. Upon rasterization of these slices, the resulting fragments are shaded using a texture or group of textures which are derived from the original volumetric data and then they are *composited* in a buffer with the data which results of applying the same procedure to previous slices. For this purpose either 2D or 3D textures can be used. In the first case three stacks of 2D textures are used, and the slicing of the volume is done in an *object aligned* fashion, while in the second case a single 3D texture is required and the slicing is performed in a *viewport aligned* fashion (figure 2.6).

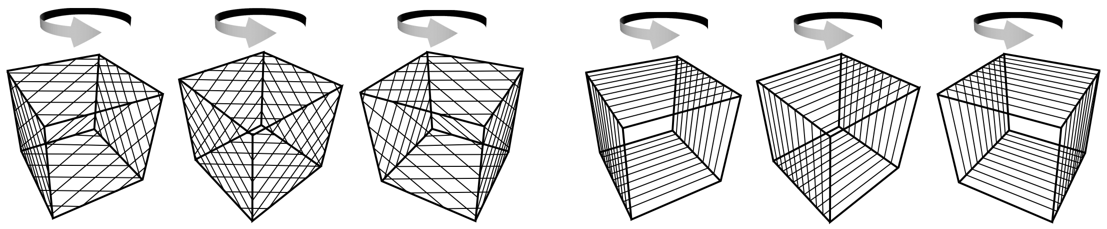


Figure 2.6: Alignment of texture slices for 3D texturing on the left, and 2D texturing on the right (image from Rezk-Salama et al [RSEB$^+$00]).

2.3 Physical basis of the algorithm

The physical model behind the volume rendering algorithm is based on the absorption and emission of radiation that happens inside a cloud of hot gas. Referring to figure 2.3, the change in power dP that happens when a beam of radiation spanning a solid angle $d\Omega$ passes through an infinitesimal length ds of material with area dA is given by:

$$\begin{aligned} dP &= emission - absorption \\ &= dI\, dA\, d\Omega \\ &= J(dA\, ds)d\Omega - K(I\, dA\, d\Omega)ds, \end{aligned} \tag{2.1}$$

where $dI\, dA\, d\Omega$ is lost power, $dA\, ds$ is the volume of material through which the radiation passes, $I\, dA\, d\Omega$ is the *incident* power, I is the intensity of the radiation in $\mathrm{erg}/(\mathrm{cm}^2 \cdot \mathrm{sterradian} \cdot \mathrm{s})$, J is the emissivity in $\mathrm{erg}/(\mathrm{cm}^3 \cdot \mathrm{sterradian} \cdot \mathrm{s})$ and K is the extinction coefficient in cm^{-1} which accounts for absorption and scattering effects. Dividing (2.1) by $dA\, d\Omega\, ds$ one obtains:

$$\frac{dI}{ds} = J - KI. \tag{2.2}$$

This differential equation is known as the *radiative transfer equation*. It can not be solved without specifying the nature of J and K. A detailed discussion of its properties and solutions under different conditions can be found in e.g. [GBR79]. In the rest of this section only the aspects relevant to volume rendering will be discussed.

Before considering the formal solution of this equation, there are two special cases of interest:

Emission only: For the case of $K = 0$ there is no extinction, radiation is only emitted and (2.2) has the solution:

$$I(s) = I(s_0) + \int_{s_0}^{s} J\, ds' \tag{2.3}$$

Absorption only: In the case of $J = 0$, (2.2) has the solution:

$$I(s) = I(s_0)e^{-\int_{s_0}^{s} K\, ds'} \tag{2.4}$$

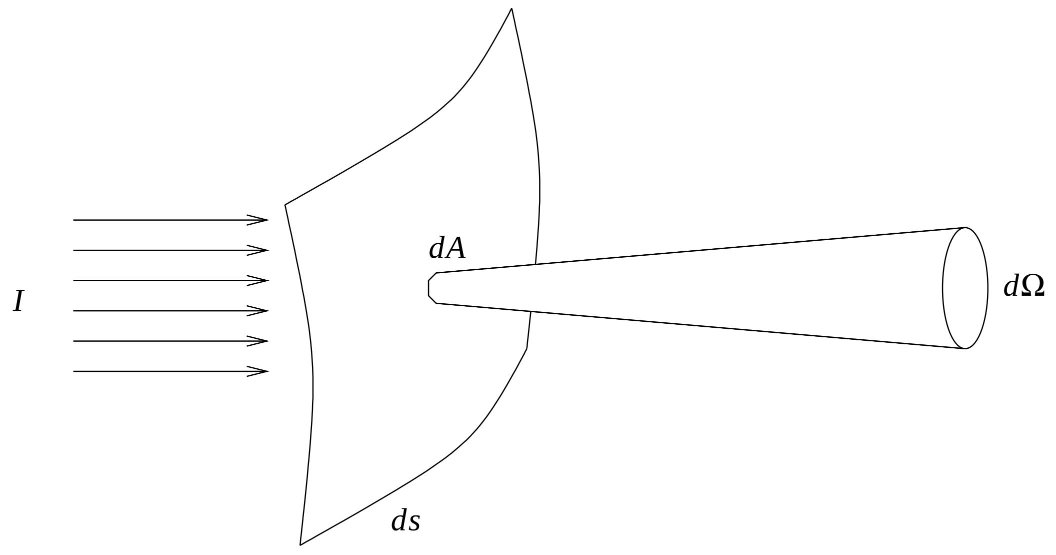


Figure 2.7: Physical configuration for the general radiative transfer problem

The quantity τ defined by $d\tau \equiv K ds$ is commonly called *optical depth*. It is a measure of how opaque a medium is to radiation passing through it. A medium is *opaque* (or *optically thick*) if $\tau > 1$ when integrated along a typical path. A medium is *transparent* (or *optically thin*) if $\tau < 1$. Considering a slab of material limited by two parallel planes, the optical depth is normally measured as the normal distance to the surface, so that ds is replaced by dz and $\tau = \tau(z)$. The point s_0 is arbitrary; it sets the zero point for the optical depth. Introducing this definition into (2.2), it can be rewritten as:

$$\frac{dI}{d\tau} = S - I \tag{2.5}$$

where $S \equiv \frac{J}{K}$ is called the source function. (2.5) can be integrated to obtain:

$$I(\tau) = I(0)e^{-\tau} + \int_0^\tau S(\tau')e^{-(\tau-\tau')}d\tau'. \tag{2.6}$$

The first term on the right-hand side describes the intensity emerging from the absorbing medium corrected for absorption. The second is the integrated source, diminished by absorption.

Considering an ensemble of particles with a density distribution $\rho(s)$, the extinction can be written as $\kappa_0\rho(s)$ and the emissivity can be written as $q_0\rho(s)$. With these conditions, considering a thin sheet of material, the last equation can be written as:

$$\begin{aligned} I(s) &= I(s_0)e^{-\tau(s_0,s)} + \int_{s_0}^{s} q_0\rho(s')e^{-\int_{s'}^{s}\kappa_0\rho(s'')ds''}ds' \\ &\approx I(s_0)e^{-\tau(s_0,s)} + \frac{q_0}{\kappa_0}\left(1 - e^{-\tau(s_0,s)}\right) \\ &= I(s_0)(1-\alpha(s_0,s)) + \frac{q_0}{\kappa_0}\alpha(s_0,s), \end{aligned} \tag{2.7}$$

where $\alpha(s_0,s) \equiv 1 - e^{-\tau(s_0,s)}$ is the *opacity* and the approximation is valid for regions where q_0 and κ_0 are approximately constant. This equation says that the intensity at s can be computed from the intensity at s_0, the background intensity, attenuated by a factor $1-\alpha(s_0,s)$ plus the local contributions attenuated by $\alpha(s_0,s)$. In computer graphics terms, this is a linear interpolation between the values $\frac{q_0}{\kappa_0}$ and $I(s_0)$.

Equation (2.7) provides an iterative method for computing the intensity contribution to a single image element. Looking at the definition of $\alpha(s_0,s)$, it is possible to derive an expression for computing the final opacity in an iterative fashion as well:

$$\begin{aligned}\alpha(s_0,s) &= 1 - e^{-\tau(s_0,s)} \\ &= 1 - e^{-\tau(s_0,s')}e^{-\tau(s',s)} \\ &= 1 - (1-\alpha(s_0,s'))(1-\alpha(s',s)) \\ &= \alpha(s',s) + (1-\alpha(s',s))\alpha(s_0,s') \end{aligned} \tag{2.8}$$

This form can be interpreted as there being two contributing elements, the first spanning from s_0 to s' and the second from s' to s. The equation then says that the opacity at the foreground is computed from the background opacity $\alpha(s_0,s')$ attenuated by a factor $1-\alpha(s',s)$ plus the local contribution $\alpha(s',s)$.

2.4 The transfer function

For visualization of a continuous scalar field expression (2.7) alone is not useful since the emission and absorption coefficients are not specified. For this reason, scalar values are *mapped* to physical quantities describing the absorption and emission characteristics at that point. This mapping is called *classification* and is performed using *transfer functions* which assign a *color emission* and *opacity* value to each point in the data. In the most simple form, transfer functions use just the scalar values for this mapping, but it is also possible to use the local derivatives or depth information for this purpose. The specification and generation, in automatic or semi-automatic fashion, of these functions are part of the set of unsolved problems in direct volume visualization.

Since the scalar field is not available in a continuous representation, but only in a discrete form, it is necessary to interpolate the available data at the sampled points. In the presence of a transfer function, it is necessary to decide if this interpolation step is done *before* or *after* the application of the transfer function. The former case is called *pre-classification* and the later *post-classification*. Given N sample points with values t_i, weight factors w_i and a transfer function $F(s)$, the value v produced by pre-classification is:

$$v = \sum_i^N w_i F(t_i) \tag{2.9}$$

and post-classification gives:

$$v = F\left(\sum_i^N w_i t_i\right) \tag{2.10}$$

If the function $F(s)$ is linear, these expressions are equivalent. The second expression is “correct” in the sense that values in the underlying continuous scalar field are being interpolated and then classified.

2.5 Hardware-accelerated volume rendering

In order to compute the contributions across large volumes using (2.7), that expression is applied in iterative fashion: the last expression is discretized and implemented by drawing the pixels in front-to-back or back-to-front order, and compositing the slices in the framebuffer. The process is accelerated by rendering all the rays “simultaneously” for each slice of the volume, thus creating the standard (implicitly parallel) texture-based volume rendering approach [CCF94, WE98]. This is achieved by rendering large polygons to the framebuffer, each of them textured with a slice of the volume data.

In OpenGL terms, the compositing operation is expressed as

$$C'_d = k_i C_i + k_d C_d \tag{2.11}$$

where C'_d is the new value to be stored in the destination buffer, C_s and C_d are the colors of the incoming and destination fragments respectively, k_i and k_d are their associated weighting factors. These factors can be constant or they can be derived from the fragment data itself. For the case at hand, these factors are α_i and $1-\alpha_i$ in correspondence with (2.7). This topic will be further discussed in chapter 4.

There are two major factors that limit performance using this technique: the speed of the pixel pipeline and the amount of available texture memory:

- For each frame several large polygons are rendered one after the other, with a *different part* of a texture applied to each of them. This means the rasterization unit needs to access the whole area of memory dedicated to store these textures in a frequent fashion. For individual pixels, this can be accelerated using caching and prefetching, but once the rendering of a particular polygon is done, already accessed parts of the texture memory will not be needed again until the next frame. This means that in this case the speed of the pixel pipeline is limited by the available memory bandwidth on the graphics card.

- The speed of the pipeline is only a soft limit, as the rendering time scales linearly with the data set and image sizes. On the other hand, the available texture memory imposes a hard limit: as soon as the data set does not fit completely in it, the texture allocator starts to swap textures to system memory, nullifying the advantage of the higher available memory bandwidth of the graphics card. Its impact on the attainable framerate is much more noticeable than that of the limited pixel pipeline speed.

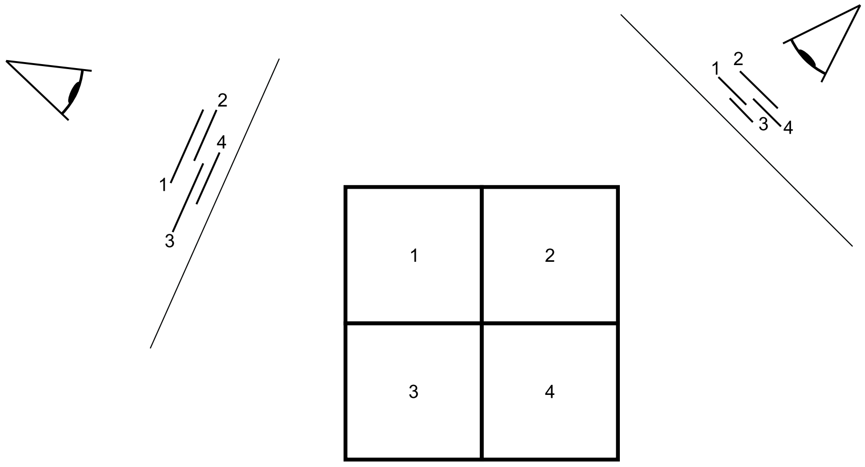


Figure 2.8: Sorting of volume bricks.

Since these limits depend on the size of the data set being rendered, both of them can be overcome using multiple renderers in parallel, where each of them works on a subset of the data. It can be seen from (2.6) that this process is explicitly parallelizable: the integral on the right-hand side can be broken up into several segments that can be added together at a later stage. As in the serial case these segments have to be kept in a consistent order while they are being blended.

If a volume dataset does not fit into the available texture memory, it is still possible to render it breaking the dataset into smaller chunks commonly called *bricks*, and then swapping textures in and out the texture memory. When doing this it is necessary to consider two details: first, the bricks need to be rendered in an order that is consistent with the rendering order of the non-bricked dataset, i.e. front-to-back or back-to-front. It suffices to sort the distances from the midpoint of the brick to the observer (figure 2.8). The second is the need for the bricks to share information at the inner borders in order to be able to perform a correct interpolation.

2.6 State of the art in hardware-accelerated volume rendering

The method introduced by Cabral was expanded by Westermann and Ertl [WE98] in order to store density values and the corresponding gradients in texture memory and exploit OpenGL extensions for unshaded volume rendering, shaded isosurface rendering, and application of clipping geometry.

Rezk-Salama et al [RSEB$^+$00] presented a technique that significantly improved both performance and image quality of the 2D-texture based approach by exploiting the multi-texturing capabilities of consumer-grade graphics hardware. By exploiting the functionality exposed by NVIDIA's `NV_register_combiners` [Kil02] and `NV_texture_shader` [Kil03] extensions, they also introduced methods that implement fast shaded isosurfaces, better interpolation and volume shading.

Röttger et al [RKE00] introduced the idea of *pre-integrated classification*, which addresses the problem of visual artifacts introduced by the use of non-linear high-frequency transfer functions. The idea is to avoid high sampling frequencies by reconstructing a piecewise linear, continuous scalar function along the viewing ray, and evaluating the volume rendering integral between each pair of successive samples of the scalar field by table lookups. This method was further expanded by Engel et al [EKE01], providing high-quality images even for low-resolution volume data by exploiting multi-texturing, advanced texture fetch and per-fragment shading operations available on current programmable consumer graphics hardware.

Kniss et al [KKH01] presented an implementation that uses multi-dimensional (2D and 3D) transfer functions based on scalar values along with their first and second derivatives in order to extract specific material boundaries and convey surface details. Kniss et al [KPI$^+$03] later applied these ideas to multivariate data by restricting the type of transfer functions to those based on a sum of gaussians. By doing this, the contributions along the ray segment can be evaluated directly in hardware instead of having to store the transfer functions in large lookup tables. Kniss et al [KPHE02] use a simplified radiation scattering model which captures volumetric light attenuation effects to produce volumetric shadows and the qualitative appearance of translucency.

In order to tackle the problem of rendering volumetric datasets that do not fit into the texture memory of the graphics adapter, a number of authors have proposed compression and level-of-detail techniques. LaMar et al [LHJ99] presented a multiresolution technique for interactive texture-based volume visualization of large data sets. This method uses an adaptive scheme that renders the volume in a region-of-interest at a high resolution and the volume away from this region at progressively lower resolutions. The algorithm is based on the segmentation of texture space into an octree, where the leaves of the tree define the original data and the internal nodes define lower-resolution versions. Weiler et al [WWH$^+$00] present an adaptive approach to volume rendering via 3D textures at arbitrary levels of detail. This work is similar to that of LaMar et al, with the difference that authors developed a bricking technique that guarantees consistent interpolation between different resolution levels. The authors pay special attention to the fixing of rendering artifacts that are introduced by non-corrected opacities at level transitions. Guthe et al [GWGS02] presented a wavelet-based compression approach. They generate a hierarchical wavelet representation in a preprocessing step, which is decompressed on the fly during rendering. The level of detail used for rendering is adapted to the local frequency spectrum of the data and its position relative to the viewer.

All these efforts have something in common: they strive for better visual quality and they exploit the capabilities of the graphics hardware that was available at the time, sometimes working around the limits imposed by it. Even if the original works make use of OpenGL extensions available only on hardware from specific vendors, the same ideas can be implemented today in a more portable fashion by using the `ARB_fragment_program` [Lip03] extension.

Chapter 3

Parallel visualization

3.1 State of the art in parallel rendering with PC clusters

Before discussing the parallelization of the volume rendering algorithm, it is necessary to introduce some common concepts in the field of parallel rendering, as well as present a review of some of the current research areas and developments.

Molnar et al [MCEF94] see the problem of parallely rendering images as a *sorting problem*, and have developed a classification for parallel rendering algorithms based on the stage on the rendering pipeline at which sorting is performed. "The essence of the rendering task is to calculate the effect of each primitive on each pixel", and sorting in this case means redistributing data between processing units, in other words, redistributing the responsibility for primitives and pixels. Starting from the assumption that geometry processing and rasterization rates are high enough to be able to perform them in parallel, thus defining a *fully parallel renderer*, three sorting strategies are defined: sort-first, sort-middle and sort-last.

Sort-first: the image is divided into disjoint regions, and each processing element is responsible for rendering one of those regions. Geometrical primitives are partially transformed (e.g., by transforming only their bounding boxes) and redistributed to the processing elements according to their on-screen location.

Sort-middle: primitives are fully transformed, lighted and readied for rasterization. The geometry processors get arbitrary sets of primitives assigned to them whilst the rasterizers get only those primitives which fall into their respective screen region.

Sort-last: rasterizers are assigned arbitrary sets of primitives, independently of their on-screen location, and the result is passed on to *compositors* for visibility resolution.

There have been several approaches to parallel graphics over the last years. IRIS Performer [RH94] is based around a scene graph and is capable of using multiple renderers on a single machine in order to load-balance the rendering task. It presents several "pipes" to the application and requires the programmer to manage the parallelization explicitly. More recently, the OpenSG library uses a data replication and synchronization approach in order to achieve the same effect on a network of workstations (NOW) [VBRR02], with focus on rendering to large displays.

A second approach that has been used with success is building accelerators with a large degree of internal parallelism. This is the case of SGI's RealityEngine and its successors [Ake93]. The RealityEngine has multiple geometry and rasterization units which work in parallel. Here the programmer does not have to be aware of the parallelization in the graphics hardware. Since it operates in immediate mode, it is limited by the program's ability to provide data at the interface's full speed. This is notoriously also the case for modern PC graphics hardware. Hardware manufacturers, facing the demand of computer games for a higher geometrical complexity and a higher level of detail in the texture data, are building PC accelerators with an increasing number of geometry and fragment processing units, and just like with the RealityEngine architecture, programmers are confronted to the problem of sending data to the hardware at the maximum speed allowed by the interface.

More recently Eldridge at al have presented a proprietary system called Pomegranate [EIH00], which is based on point-to-point communication and focuses on scalability at the input interface: it allows multiple processes to submit commands to the graphics hardware simultaneously. It has very high bandwidth requirements, which preclude its use in the context of a NOW.

Another approach that has been explored by different authors in recent years is employing specialized compositing hardware along with a linear pipeline of commodity renderers. Examples of such systems are PixelFlow [MEP92], Sepia 2 [HM99] and Lighting-2 [SEP$^+$01]. Here a programmable compositor takes images from the application and another compositor, performs a single predefined operation on all the pixels and sends the result to the next compositor down the chain. In this fashion, a final composited image is generated and sent to the display. Modern implementations such as Sepia and Lighting-2 take the 24-bit RGB output from the graphics card's DVI interface and interpret it in an application-specific manner. For example, in order to be able to perform *depth compositing*, the application can set up a viewport which is twice as high as the final image. On the top half, the color data is sent to the compositing hardware, while the bottom half contains the depth information. All these systems are proprietary and not available on the market. Just recently, a product called "DVG" which is based around this idea is being commercialized by the company ORAD [Ora03].

Humphreys et al [HEB$^+$01] presented a system called WireGL. This is a sort-first system, built on top of the OpenGL API, which tries to be transparent for the application, and focuses on large tiled displays. WireGL's usage of multiple renderers is not intended to speed-up the application, only to allow it to render larger images. Samanta et al focus on scalable rendering rates by using a cost-based model for load-balancing across nodes in a cluster. Their algorithm is a hybrid sort-first sort-last one, which is able to produce images for a single display. Later, Humphreys et al [HHN$^+$02] reworked the WireGL architecture to produce Chromium, a *stream-based* processor for the OpenGL API. Chromium is based around the idea of *stream filters*. By rearranging these filters it can operate in either sort-first or sort-last fashion.

3.2 Parallel volume rendering on clusters of PCs

Figure 3.1 depicts the general pipeline used to parallelize the rendering process and figure 3.2 depicts its integration into the cluster environment. Since the aimed dataset size is 512 MB and larger, a *static partitioning* in object space has been opted for. This alleviates the need for constantly re-uploading the volumetric data to the graphics card. This kind of partitioning implies a *sort-last* algorithm. As the first step of the process, a *rendering configuration* is created using the number of nodes as the input parameter. Given n_r rendering nodes, the volume is partitioned in $x \times y \times z$ bricks, where $xyz = n_r$ and each rendering node is assigned one of these bricks. The factorization is selected in a way that matches the dimensions of the dataset to be rendered as close as possible, that is, if the dataset is larger along one of the axes, the corresponding factor is selected to be larger (figure 3.3). This avoids the problem of having bricks which are excessively large in one dimension, that is, the variations on projected size of the brick because of a change in the viewing direction are kept to a minimum. Using this data partitioning, care is taken for neighboring nodes to receive bricks which overlap by one voxel on the corresponding borders. Once the data has been transfered, the rendering loop is entered. For each frame, a projection and modeling matrix is recomputed and transfered as required. The modeling matrix for each node is corrected so that the brick is rendered at the correct screen position. Any other required rendering parameter (e.g. a transfer function) is transferred as well. As soon as each node receives the rendering parameters, it starts to render its brick to the framebuffer using a preselected volume rendering algorithm as described in the previous chapter, and taking care of performing the composition in the way described in chapter 4. Once a node has finished rendering its own brick, it enters the compositing stage of the process.

3.3 Parallel compositing algorithm

Assuming partial RGBα image data computed in the fashion described in chapter 4 is available, compositing takes place in the following way: given N different nodes, each of them having one out of N images to be composited, each image is split into N subimages. Each node sends $N-1$ of these images to its neighbors and receives $N-1$ from them. Image composition is then performed in the CPU and the result is sent to a previously defined node. If t is the cost of transmitting one image over the network and c the cost of compositing two full images together, this algorithm has a total cost of $(2t+c)(N-1)/N$ (assuming a full-duplex network). This algorithm requires a total of N^2-1 messages of size $\frac{A}{N}$, where A is the total image area in pixels. This is the “direct send” algorithm by Neumann [Neu93]. Ma et al [MPHK94] compare this algorithm with their “binary swap” algorithm. The binary swap algorithm has a higher communication cost ($\leq 2.43N^{1/3}A$), but its advantage is that it does not require sending messages to every other processor as might be the case with the direct send algorithm in its original formulation by Neumann. Instead of that, a total of $N(\log_2 N+1)-1$ messages are sent. When comparing binary swap and direct send in a PC cluster environment with a Myrinet interconnect, it is necessary to remember that the sending and receiving operations can overlap and that resource contention can be prevented by implementing the data exchange in a manner similar to:

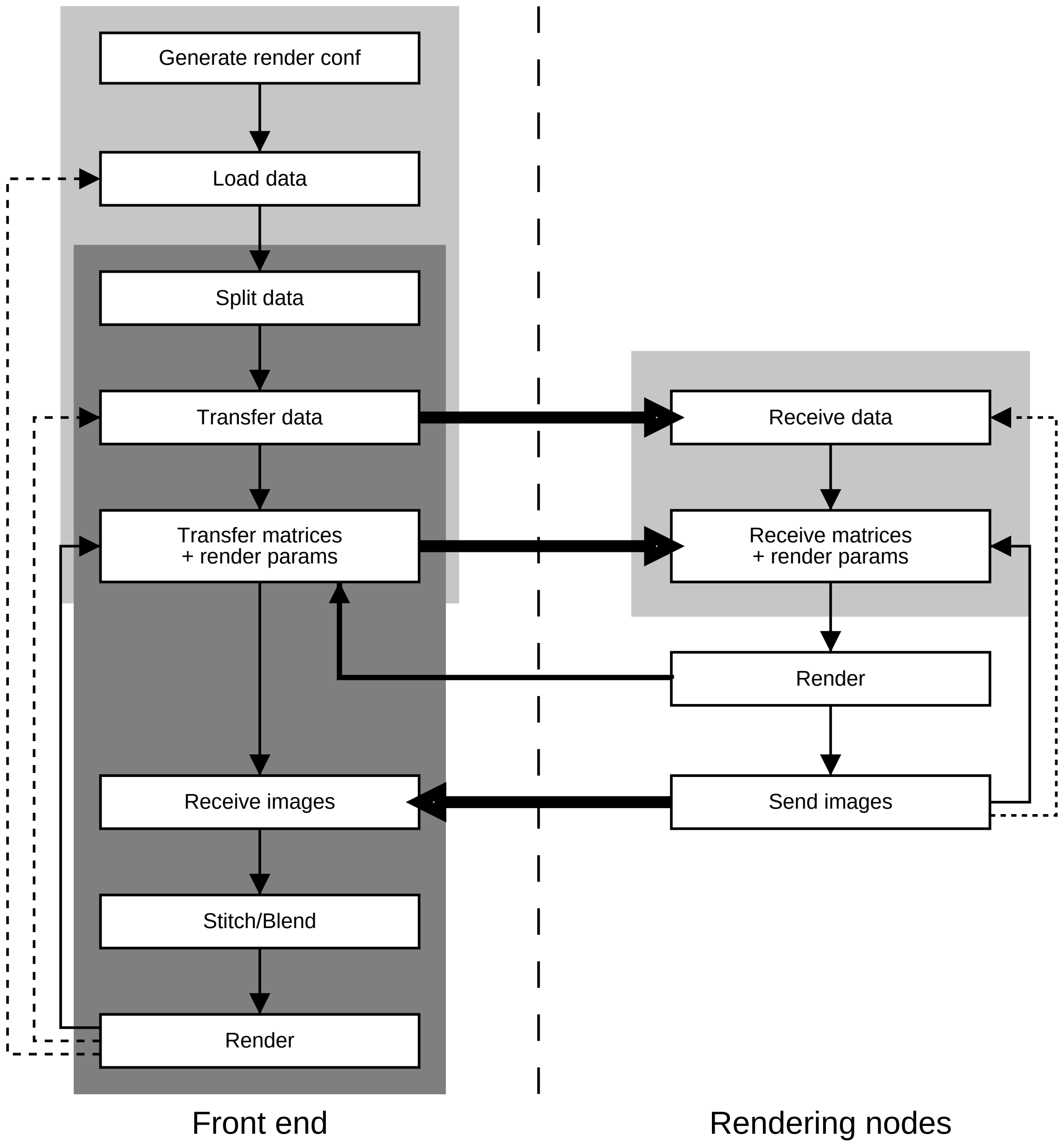


Figure 3.1: General program flow for the controlling and rendering processes.

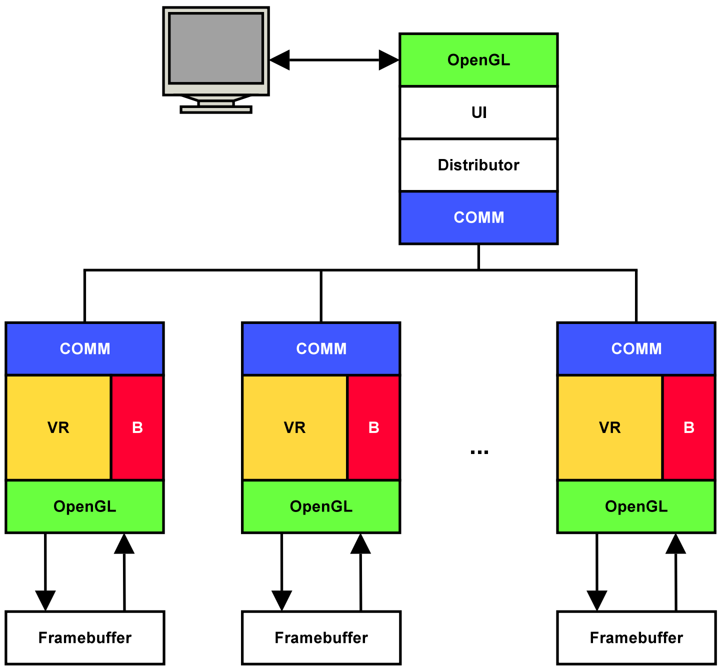


Figure 3.2: Integration of the volume renderers into a PC cluster environment. “COMM” is the inter-process communication subroutine; “VR” is the volume rendering subroutine; “B” is the compositing code.

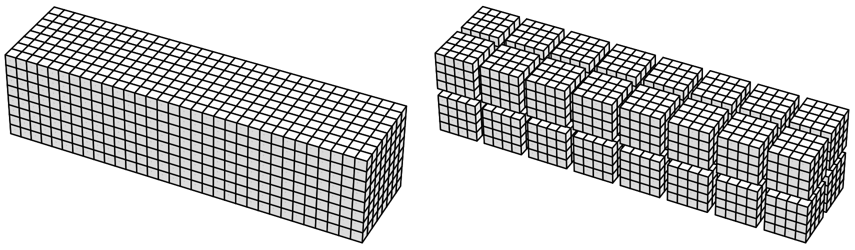


Figure 3.3: Left: a volume dataset where one of the directions is significantly larger than the other two. Right: the corresponding partitioning ($\mathbf{2} \times \mathbf{2} \times \mathbf{8}$) using 32 nodes.

```
for (i=1; i < size; ++i)
{
    // fb contains the framebuffer data
    // cb is the compositing buffer
    dst = rank ^ i;
    MPI_Isend(fb[dst], size, MPI_UNSIGNED,
              dst, DATA, comm, &(req[0]));
    MPI_Irecv(cb[dst], size, MPI_UNSIGNED,
              dst, DATA, comm, &(req[1]));
    MPI_Waitall(2, req, status);
}
```

Figure 3.4 shows the total time required for sending the image information over the network, compositing the data and sending the result to a single node as a function of the image size A for the 16 node case when using the direct send and the binary swap algorithms. It can be seen that the direct send algorithm is slightly faster than binary swap. The advantage of the direct send algorithm is that its implementation is straightforward.

Figure 3.5 shows the total time required for compositing N images using N nodes with the direct send and binary swap algorithms. It can be seen that the direct send algorithm scales slightly better than the binary swap algorithm.

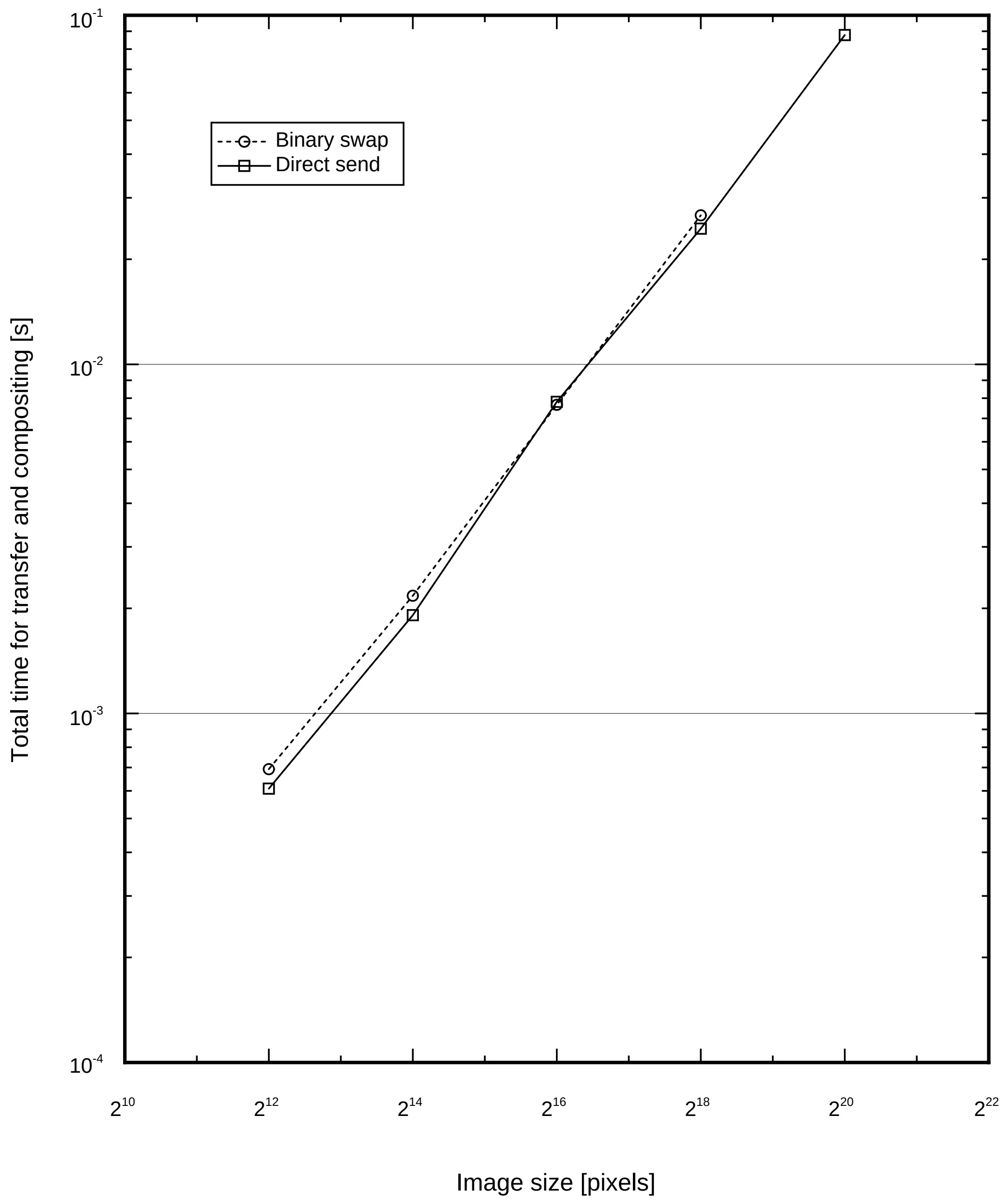


Figure 3.4: Total compositing time as a function of image size using 16 processing elements.

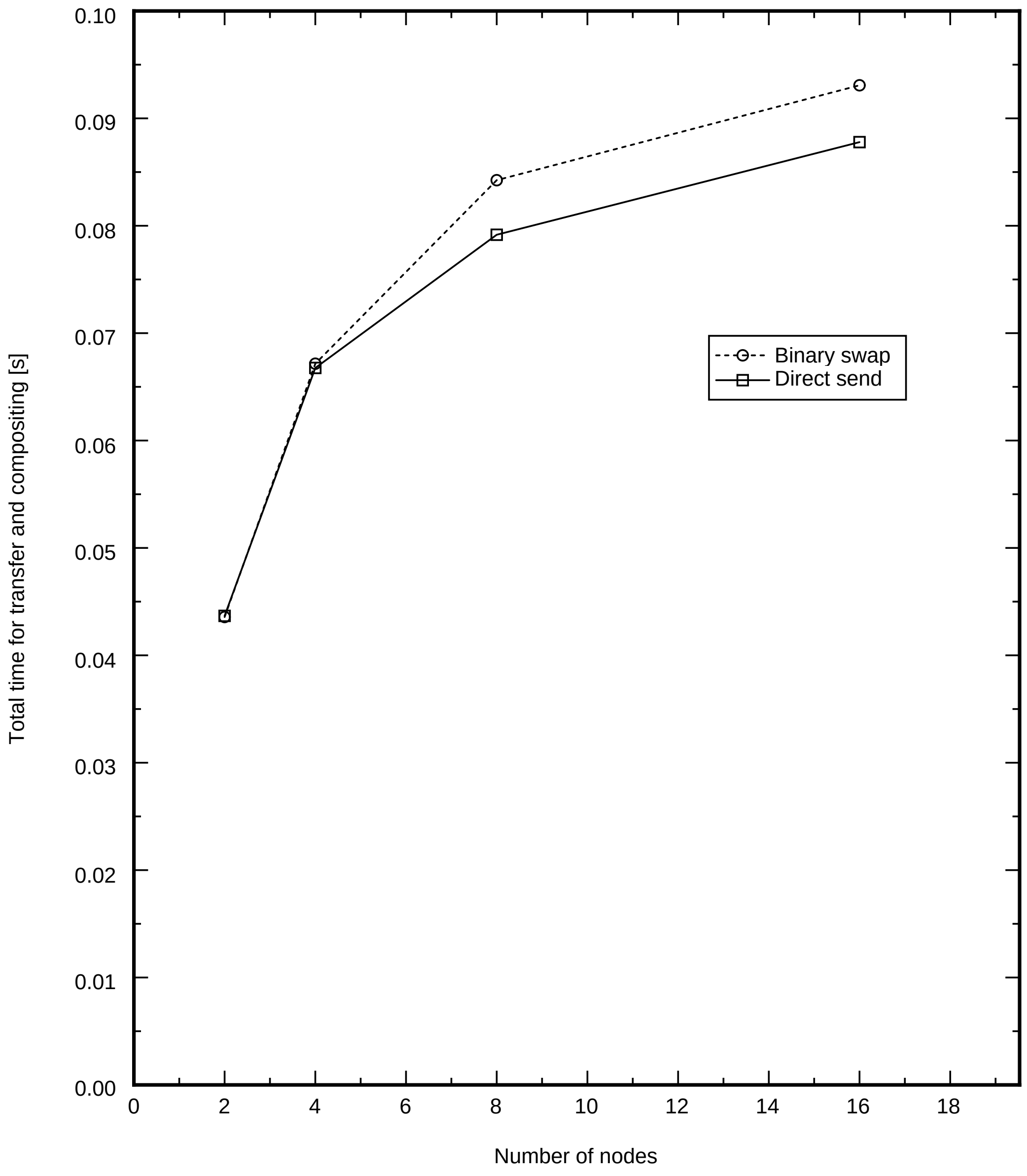


Figure 3.5: Total compositing time as a function of the number of nodes participating in the composition

Chapter 4

Image compositing and volume rendering

4.1 General image compositing

Foley et al define “image compositing” [FvDFH96] as “combining images to create new images”. The canonical work in this area is the paper “Compositing Digital Images” by Porter and Duff [PD84]. In this work, the authors note that image compositing is a good way to produce images in general, since it is easy to do and it helps saving time, since the regeneration of costly imaginery can be avoided or reduced. In the paper, the authors define an algebra for image compositing. Given two images A and B, the result of compositing them can in general be expressed as:

$$A \textbf{ op } B \equiv w_A A + w_B B \tag{4.1}$$

where w_A and w_B are per-pixel weighting factors assigned to A and B. Porter and Duff defined operators such as A **over** B, A **in** B and A **held out by** B. In this context, a so-called α-value is associated with each RGB triplet that defines the colors of the pixels in the image resulting in an RGBα image. From here on, it will be assumed that the values of each component lie in the range $[0,1]$ as per the OPENGL specification ([SA03], §2.14). This α-value can be understood as a measurement of the coverage of each pixel in the image. For example, considering a red pixel, RGB triplet (1; 0; 0), with a coverage of 50 percent, its α value would be 0.5. A 50 percent coverage means than only half of the pixel’s “surface” actually has a red color, which can be approximated by pixel fully covered by the color (0.5; 0; 0) (figure 4.1). That is, the RGBα value (R; G; B; α) represents the RGB value (αR; αG; αB).

Image composition of this sort has been supported in hardware since the early days of computer graphics, even if at first it was limited to *chroma keying*, which is the process of assigning α values to images based on the chromaticity of the pixels. Pixels with a preselected chromaticity value are handled as fully translucent ($\alpha = 0$) while the rest are fully opaque ($\alpha = 1$). For this application, if A is the image in the foreground and B the one in the background, it is enough to set the weighting factors to $w_A = \alpha_A$ and $w_B = 1 - \alpha_A$. With these assignments, it is possible to superimpose images over arbitrary backgrounds, placing for example a given subject on a location of choice. The only requisite is to capture the foreground subject in a location which background results in a well defined chromaticity value which does not show up in the subject

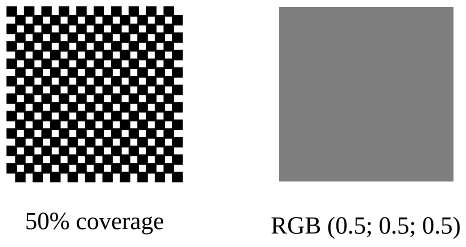


Figure 4.1: Opacity values interpreted as "coverage" information for a grey value of 0.5.

itself. This sort of compositing is used widely in the television and film industry.

Modern graphics accelerators support a wide range of composition operations. The OPENGL API exposes this functionality as a set of *blending equations* and *blending functions*. The OPENGL specification defines blending ([SA03], §4.1.8) as:

> Blending combines the incoming *source* fragment's R, G, B and A values with the *destination* R, G, B, and A values stored in the framebuffer at the fragment's (x_w, y_w) location.
>
> Source and destination values are combined according to the *blend equation*, quadruplets of source and destination weighting factors are determined by the *blend functions*, and a constant *blend color* to obtain a new set of R, G, B, and A values, [...] The resulting four values are sent to the next operation.

In OPENGL terms, the blending equation corresponding to (4.1) is denoted by `FUNC_ADD`, and is reexpressed as:

$$C = C_s S + C_d D \tag{4.2}$$

Here C_s and C_d are the source and destination colors and S and D are quadruplets of weighting factors determined by the blend functions. Some of the available blending functions are listed in table 4.1.

4.2 The compositing problem in volume rendering

For volume rendering applications, the compositing operation for two fragments f_i and f_d with color components C_i and C_d and alpha components α_i and α_d is given by:

$$\begin{aligned} C_i \circledast C_d &\equiv \alpha_i C_i + (1-\alpha_i) C_d \\ \alpha_i \circledast \alpha_d &\equiv \alpha_i + (1-\alpha_i)\alpha_d \end{aligned} \tag{4.3}$$

$C_i \circledast C_d$ means "apply C_i on top of C_d". This expression is congruent with (2.7) and with (2.8). As it will be discussed later, it is not possible to implement this expression with some implementations of OPENGL. Usually this is not a problem, since back-to-front compositing in hardware

Function	RGB blend factors	Alpha blend factor
`ZERO`	$(0,0,0)$	0
`ONE`	$(1,1,1)$	1
`SRC_COLOR`	(R_s,G_s,B_s)	A_s
`ONE_MINUS_SRC_COLOR`	$(1-R_s,1-G_s,1-B_s)$	$1-A_s$
`DST_COLOR`	(R_d,G_d,B_d)	A_d
`ONE_MINUS_DST_COLOR`	$(1-R_d,1-G_d,1-B_d)$	$1-A_d$
`SRC_ALPHA`	(A_s,A_s,A_s)	A_s
`ONE_MINUS_SRC_ALPHA`	$(1-A_s,1-A_s,1-A_s)$	$1-A_s$
`DST_ALPHA`	(A_d,A_d,A_d)	A_d
`ONE_MINUS_DST_ALPHA`	$(1-A_d,1-A_d,1-A_d)$	$1-A_d$

Table 4.1: Some of the blending functions available in OPENGL

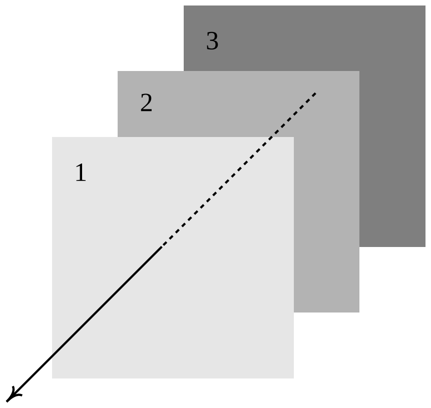


Figure 4.2: Three slices to be composited together in associative fashion

accelerated volume rendering does not require the final opacity values to be stored in the frame-buffer.

The operation defined by (4.3) is evidently non-commutative. As stated before, the parallelization of the volume rendering algorithm is based on the fact that the integral along the ray can be split into multiple integrals along segments of the ray. That means that the compositing operation needs to be associative. Is the operation as quoted above associative? Considering three slices, as in figure 4.2:

$$(C_1 \circledast C_2) \circledast C_3 = (\alpha_1 \circledast \alpha_2)(C_1 \circledast C_2) + (1-(\alpha_1 \circledast \alpha_2))C_3 \tag{4.4}$$

Expanding (4.4) using the definition (4.3), the following expression is obtained:

$$\begin{aligned}(C_1 \circledast C_2) \circledast C_3 = {} & \alpha_1^2\alpha_2(-C_1+C_2) \\ & + \alpha_1^2(C_1-C_2) + \alpha_1\alpha_2(C_1-2C_2+C_3) \\ & + \alpha_1(C_2-C_3) + \alpha_2(C_2-C_3) + C_3\end{aligned} \tag{4.5}$$

Now, performing the compositing operation in a different order:

$$\begin{aligned} C_1 \circledast (C_2 \circledast C_3) &= \alpha_1 C_1 + (1-\alpha_1)(C_2 \circledast C_3) \\ &= \alpha_1\alpha_2(C_3 - C_2) + \alpha_1(C_1 + C_2 - C_3) - \alpha_2 C_3 + C_3 \end{aligned} \tag{4.6}$$

Comparing (4.5) and (4.6) these equations are different, therefore the operation defined by (4.3) is *not* associative.

Considering the alternative definition:

$$\begin{aligned} C_i \oplus C_d &\equiv C_i + (1-\alpha_i)C_d \\ \alpha_i \oplus \alpha_d &\equiv \alpha_i + (1-\alpha_i)\alpha_d \end{aligned} \tag{4.7}$$

Performing the composition of 1 and 2 and then applying the result over 3:

$$\begin{aligned} (C_1 \oplus C_2) \oplus C_3 &= (C_1 \oplus C_2) + (1 - (\alpha_1 \oplus \alpha_2))C_3 \\ &= C_1 + (1-\alpha_1)C_2 + (1 - (\alpha_1 + (1-\alpha_1)\alpha_2))C_3 \\ &= C_1 + C_2 + C_3 - \alpha_1 C_2 - \alpha_1 C_3 - \alpha_2 C_3 + \alpha_1\alpha_2 C_3 \end{aligned} \tag{4.8}$$

On the other hand, performing the composition of 2 over 3, and then adding 1 on top:

$$\begin{aligned} C_1 \oplus (C_2 \oplus C_3) &= C_1 + (1-\alpha_1)(C_2 \oplus C_3) \\ &= C_1 + (1-\alpha_1)(C_2 + (1-\alpha_2)C_3) \\ &= C_1 + C_2 + C_3 - \alpha_1 C_2 - \alpha_1 C_3 - \alpha_2 C_3 + \alpha_1\alpha_2 C_3 \end{aligned} \tag{4.9}$$

Comparing (4.8) and (4.9) these equations are identical, therefore the operation defined in this way is associative. The operation defined by (4.7) is what Porter and Duff call the *over operator*. For completeness' sake, it is worth nothing that this operation has an identity element on the right:

$$A \oplus i_r = A + (1-\alpha)i_r = A \Leftrightarrow i_r = 0 \tag{4.10}$$

and on the left:

$$i_l \oplus A = i_l + (1-\alpha_{i_l})A = A \Leftrightarrow i_l = 0 \tag{4.11}$$

That is, there is a single identity element: 0. Furthermore, assuming the addition of color components is saturated, this operation is also closed. Thus the set of RGBα quadruplets $(r;g;b;\alpha)\ \forall\ r,g,b,\alpha \in [0,1]$ together with this operation form a mononoid. It fails to be a group because there does not exist an inverse element for each member of the set.

4.3 Pre-multiplication

In order to be able to render volume elements in parallel, the definition (4.7) needs to be used for the compositing operation instead of (4.3). Comparing both definitions, the difference is the factor α_i in front of the term C_i. In order to recover (4.3) for the composition of individual slices, it suffices to change the color component of each voxel from C_v to $\alpha_v C_v$, that is, the α-component of the voxel data needs to be *pre-multiplied* into the RGB components. To see this, consider again figure 4.2. Without loss of generality, considering a black background and applying slice 3 over it gives:

$$C_3 \circledast 0 = \alpha_3 C_3 \tag{4.12}$$

Then 2 over this:

$$C_2 \circledast C_3 = \alpha_2 C_2 + (1-\alpha_2)\alpha_3 C_3 \tag{4.13}$$

Then 1 over this:

$$C_1 \circledast (C_2 \circledast C_3) = \alpha_1 C_1 + (1-\alpha_1)(\alpha_2 C_2 + (1-\alpha_2)\alpha_3 C_3) \tag{4.14}$$

Making the transformation $\alpha C \rightarrow C'$; $\alpha \rightarrow \alpha'$ on the right-hand side:

$$= C'_1 + (1-\alpha'_1)(C'_2 + (1-\alpha'_2)C'_3) \tag{4.15}$$

which is exactly $C'_1 \oplus C'_2 \oplus C'_3$.

The composition of α-pre-multiplied images has been discussed by Blinn in detail [Bli94]. Besides the associative property already discussed, the technique is attractive because of its elegance: while the equation (4.3) is asymmetric between the color and α components, equation (4.7) is symmetric. This makes many algorithms easier to implement and eliminates some of the bookkeeping associated with the non-pre-multiplied version.

4.4 Pre-multiplication with OpenGL

There are several ways in which the pre-multiplication effect can be achieved in the context of OPENGL:

- Perform the pre-multiplication in the data itself.
- Use the `GL_EXT_blend_func_separate` OPENGL extension.
- Use the `GL_NV_register_combiners` OPENGL extension, and perform the pre-multiplication before the blending stage.
- Use the `GL_ARB_fragment_program` OPENGL extension, and perform the pre-multiplication before the blending stage, like in the previous case.

4.4.1 Pre-multiplying in the transfer function

Before discussing the other options, it is convenient to look in more detail at the possibility of performing the pre-multiplication during the application of the transfer function, that is, multiplying the α values defined in the transfer function with the corresponding color values.

Considering the case of tri-linear interpolation for texture fetches, the fetched value v is given by:

$$v = \sum_{i,j,k=0,1} \Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)t_{ijk} \tag{4.16}$$
$$\Lambda(x;i) \equiv (1-i)(1-x)+ix$$

where α, β, γ are the interpolation terms for the lookup, and t_{ijk} are the scalar values at texels ijk selected as samples, as described in [SA03], §3.8.8. For the rest of this section the ranges for the summation indices will be omitted for clarity of presentation, they always run between 0 and 1.

For pre-classification with a transfer function $F^c(x)$, this becomes:

$$v^c_{\text{pre}} = \sum_{i,j,k} \Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)F^c(t_{ijk}) \tag{4.17}$$

For post-classification on the other hand:

$$v^c_{\text{post}} = \sum_{l} \Lambda(\alpha';l)F^c_l(v) \tag{4.18}$$

Where F_l denotes the l-th sample in the transfer function. The additional linear interpolation comes from the fact that post-classification is implemented as a texture lookup. If the α multiplication gets done in the transfer function, defining $\tilde{F}^c(x) \equiv F^\alpha(x)F^c(x)$, these equations become:

$$\tilde{v}^c_{\text{pre}} = \sum_{i,j,k} \Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)\tilde{F}^c(t_{ijk}) \tag{4.19}$$

and

$$\tilde{v}^c_{\text{post}} = \sum_{l} \Lambda(\alpha';l)\tilde{F}^c_l(v) \tag{4.20}$$

On the other hand, if the multiplication is done at the blending stage, one obtains:

$$v^\alpha_{\text{pre}}v^c_{\text{pre}} = \sum_{i,j,k,l,m,n} \Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)\Lambda(\alpha;l)\Lambda(\beta;m)\Lambda(\gamma;n)F^\alpha(t_{ijk})F^c(t_{lmn}) \tag{4.21}$$

and

$$v^\alpha_{\text{post}}v^c_{\text{post}} = \sum_{l,m} \Lambda(\alpha';l)\Lambda(\alpha';m)F^\alpha_l(v)F^c_m(v) \tag{4.22}$$

In order for these equations to be pairwise equivalent (4.19 and 4.21; 4.20 and 4.22), the terms $F^\alpha(t_{ijk})F^c(t_{lmn})$ and $F^\alpha_l(v)F^c_m(v)$ need to reduce to $\tilde{F}^c(t_{ijk})$ and $\tilde{F}^c_l(v)$ respectively. This means that $\Lambda(\alpha;i)\Lambda(\alpha;j)$ needs to be 0 for $i \neq j$ and $\Lambda(\alpha;i)$ for $i=j$, which is in general not the case.

Furthermore, it is necessary to point out what happens in the case of pre-classification at the blending stage:

$$\begin{aligned} C'_{\text{dst}} &= C_{\text{src}} + (1-\alpha_{\text{src}})C_{\text{dst}} \\ &= \tilde{v}^c_{\text{pre}} + (1-\tilde{v}^\alpha_{\text{pre}})C_{\text{dst}} \\ &= \sum_{i,j,k} \Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)F^\alpha(t_{ijk})F^c(t_{ijk}) \\ &\quad + \left(1 - \sum_{i,j,k}\Lambda(\alpha;i)\Lambda(\beta;j)\Lambda(\gamma;k)F^\alpha(t_{ijk})\right)C_{\text{dst}} \end{aligned} \tag{4.23}$$

This means the opacity value that is being used in the first term is *not* the same opacity value used in the second term. The same happens with post-classification schemes, which give:

$$\begin{aligned} C'_{\text{dst}} &= C_{\text{src}} + (1-\alpha_{\text{src}})C_{\text{dst}} \\ &= \tilde{v}_{\text{post}} + (1-\tilde{v}^\alpha_{\text{post}})C_{\text{dst}} \\ &= \sum_l \Lambda(\alpha';l)F^\alpha_l(v)F^c_l(v) + \left(1-\sum_l \Lambda(\alpha';l)F^\alpha_l(v)\right)C_{\text{dst}} \end{aligned} \tag{4.24}$$

By performing the multiplication after the application of the transfer function, it is assured that the opacity value being used is *consistent* with the assumption that the underlying scalar field is continuous, that is, the opacity is derived from the interpolated scalar value.

4.4.2 Pre-multiplying in the data itself

This method follows the suggestion of Blinn and other authors of working in α-pre-multiplied space and is the most straightforward. The pre-multiplication is a preprocessing step and does not have to be constantly repeated. The disadvantage is that the result has to be stored as RGBα quadruples in the graphics card, therefore increasing the required texture memory. It also has a lower precision, as Wittenbrink et al [WMG98] argue. The method precludes the use of *post-classification* of the scalar field, since the transfer function needs to be applied before the interpolation takes place. It also requires that the texture data be uploaded anew every time the transfer function is modified.

4.4.3 GL_EXT_blend_func_separate

In standard OPENGL and up to version 1.3, it is only possible to specify a single blending function which is used for RGB- and α-components, that is, the same function *has* to be used for both factors. It is necessary to notice in table 4.1, that there is no single blend function that specifies the blending factors as $(A_s, A_s, A_s, 1)$. This OPENGL extension allows to specify separate blending functions for the RGB- and α-components. This option has the advantage that the pre-multiplication is performed after the fragment has been shaded (figures 2.4 and 2.5) and inside the dedicated blending unit on the graphics card, which is likely to be able to perform this

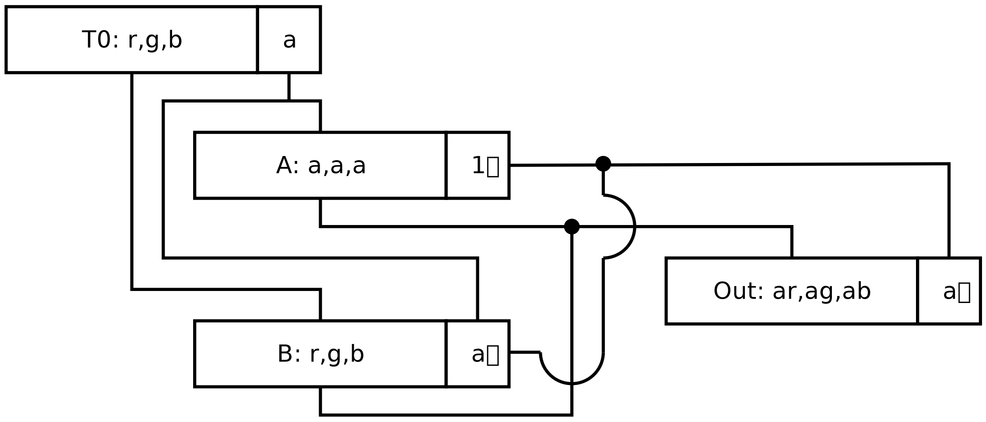


Figure 4.3: Register combiner setup for multiplying the alpha component into the color components before the blending stage

operation without loss of precission since it is a dedicated unit. The disadvantage is that this extension is rarely supported by consumer grade graphics hardware (for example, only recent versions of the Radeon chip from ATI support the functionality). This functionality has been incorporated into the core API starting with OpenGL 1.4 ([SA03], §G.9) so a wider support for it is likely to be seen in the future.

4.4.4 GL_NV_register_combiners

Here the idea is to multiply the α component of the voxel into its RGB values after performing per-fragment shading operations but just before passing the data to the blending stage (figure 2.4). The combiner arrangement in figure 4.3 is setup by the following code:

```
/* A[r,g,b] = T[0][a] */
glCombinerInputNV(
    GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
    GL_TEXTURE0, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

/* A[a] = (1-0)[a] */
glCombinerInputNV(
    GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA);

/* B[r,g,b] = T[0][r,g,b] */
glCombinerInputNV(
    GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
    GL_TEXTURE0, GL_UNSIGNED_IDENTITY_NV, GL_RGB);

/* B[a] = T[0][a] */
```

```
glCombinerInputNV(
    GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
    GL_TEXTURE0, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

/* T[0][r,g,b] = A[r,g,b] * B[r,g,b] == T[0][a] * T[0][r,g,b] */
glCombinerOutputNV(
    GL_COMBINER0_NV, GL_RGB, GL_TEXTURE0,
    GL_DISCARD_NV, GL_DISCARD_NV,
    GL_NONE, GL_NONE,
    GL_FALSE, GL_FALSE, GL_FALSE);

/* A[r,g,b] = (1, 1, 1) */
glFinalCombinerInputNV(
    GL_VARIABLE_A_NV, GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB);

/* B[r,g,b] = T[0][r,g,b] */
glFinalCombinerInputNV(
    GL_VARIABLE_B_NV, GL_TEXTURE0, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
```

The advantage of this option is that it is supported by a wide range of hardware which is commonly available. The disadvantage is that this extension is complex to work with, the operations have very limited precision (commonly 10-bits) and it is NVIDIA proprietary. No other company has shown interest in implementing this extension into their hardware, and given the current move towards flexible fragment programmability, it is unlikely that this will happen in the future. The differences in the final image introduced by this method are shown in figure 4.4.

4.4.5 GL_ARB_fragment_program

This is the same idea as before: multiply the α-value of each fragment into the color components just before passing to the blending stage. This option has the advantage that fragment operations are carried at a high precision on current implementations. The disadvantage is that it is limited to very recent hardware which has not been widely deployed, and imposes a performance hit when compared to the traditional method. It is very simple to implement:

```
!!ARBfp1.0
TEMP coord, color;
TEX coord, fragment.texcoord[0], texture[0], 3D;
TEX color, coord, texture[1], 2D;
MUL result.color, color, color.a;
MOV result.color.a, color.a;
END
```

The availability problem will likely disappear in the future, since this extension is officially sanctioned by the OpenGL Architecture Review Board and is expected to be incorporated into a future revision of the OpenGL standard.

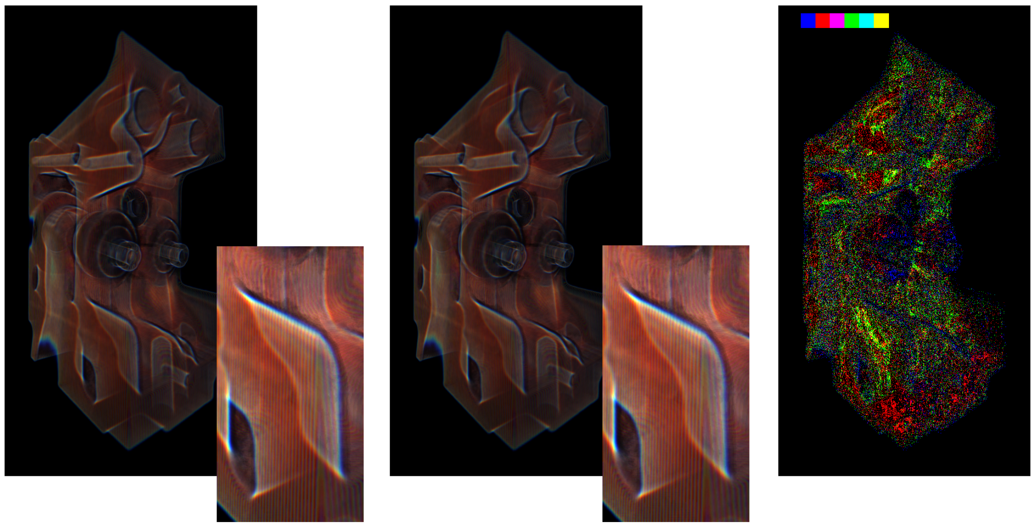


Figure 4.4: Comparison between the usual non-pre-multiplied blending operations (left) and the α-multiplication performed using register combiners (middle). On the right the difference between the two of them is presented using a paletted image. The insets correspond to the region of maximum difference. (See figure A.1 on the color plates section).

Chapter 5

Software inter-brick compositing

5.1 Motivation

It is technically possible to perform the compositing operation of the several subvolumes in the graphics hardware. To achieve this, the images have to be read *from* the framebuffer, saved in a temporary location and then written back *to* the framebuffer. If the temporary location is in main memory, as it must be the case when the renderers are installed on physically different computers, the easiest and most natural way of doing this is to use the functions `glReadPixels` and `glWritePixels`. This would exploit the implicit parallelization in the graphics hardware, but there is an overhead associated with this operation, and on current graphics hardware, this is not to be neglected, as table 5.1 shows. Compositing in hardware would amount to reading back the data in the framebuffer, which can and does happen in parallel, and then sequentially uploading the images to a single display adapter, and eventually reading back the data again. As an example, for a 16 node configuration, this would mean $19+16\cdot 16+19 \;=\; 294$ ms per frame (reading, writing 16 images, reading back), and this does not take into account the time it takes to collect the images from the different renderers. If all the compositors take part in this operation and ignoring any overhead, the time could be reduced to $19+16\cdot\frac{16}{16}+\frac{19}{16} \;=\; 36$ ms (reading, writing $\frac{1}{16}$ of an image 16 times, reading $\frac{1}{16}$ of an image).

The quoted numbers impose an upper limit of about 2-3 Hz on the refresh rate at which the application could run if performing the compositing in a sequential fashion, which is still too far away from any definition of *interactive rendering*. Performing the compositing in a parallel fashion puts an upper limit of 27 Hz on the refresh rate which is acceptable, but this is ignoring the transfer times and overhead.

As an alternative, instead of using the `glDrawPixels` function, it is possible to upload the data to a texture and then render a large polygon to the framebuffer to perform the compositing operation. The texture upload path is likely to be more optimized than the `glDrawPixels` one, since the former is an operation more commonly used by computer games, but this is not necessarily the case, as table 5.2 shows.

Because of this, an alternative CPU-based approach has been explored.

	color buffer			
	rgb	rgba	bgra	abgr
draw*	18	17	*16*	17
read*	21	20	*19*	20
draw*	49	33	22	31
read*	34	37	27	29
draw†	19	19	16	18
read†	22	22	21	22
draw‡	39	31	20	25
read‡	30	30	25	24

Table 5.1: Read and write times for color and depth buffers without blending or depth test enabled. The quoted number is time required to read or write a 1024×768 region (in ms). The GPU, chipset, CPU and OS kernel specifications follow: *GeForce Ti 4600/AMD-MP 760/Athlon MP 2000+/Linux 2.4.19; *GeForce/Intel 440BX/PIII 500 MHz/Linux 2.4.1; †GeForce2/VIA Apollo KT133/Athlon 900 MHz/Linux 2.4.1; ‡GeForce2/Intel 440BX/PIII 650 MHz/Linux 2.2.16.

Operation	Texture	DrawPixels
Clear	$\ll 1$	$\ll 1$
Texture upload	36	
Texture upload	36	
Render quad	$\ll 1$	
Render quad	$\ll 1$	
DrawPixels		16
DrawPixels		18
ReadPixels	42	41
Total	114	75

Table 5.2: Comparison between two alternative paths for performing image composition in hardware (the final images are identical). The path using the `glDrawPixels` function is 34% faster for this particular configuration. The images are 1024×1024 pixels in size and contain RGBA data. Both measurements were performed on the same configuration: GeForce3 Ti 500/VIA KT133/Athlon XP 1600+/Linux 2.6.0-test10. All measurements have a standard deviation of less than 2% of the indicated value and the operations have been performed in the sequence shown to avoid masking the effect of state changes and pipeline stalls. It is interesting to note that the second call to `glDrawPixels` is consistently slower than the first even if the same amount of data is being transfered. Along the same line, the `glReadPixels` operation in the texture path is consistently slower than the same operation in the drawpixels path, but the difference is not statistically significant.

5.2 Compositing images in software

Given two pixels A and B, when working with colors represented with 8 bits per channel, the operation

$$A_c + (1 - A_\alpha)B_c$$

translates to

$$A_c + \frac{(255 - A_\alpha)B_c}{255} \tag{5.1}$$

where the subscript c denotes the color component of the pixel. If x is a 16 bit value, the operation $\frac{x}{255}$ can be computed as:

$$\begin{aligned} \frac{x}{255} &= \frac{x}{256} \cdot \frac{256}{255} \\ &= \frac{x}{256} \cdot \frac{255+1}{255} \\ &= \frac{x + x/255}{256} \\ &= \frac{x + x/256 + x/(255 \cdot 256)}{256} \end{aligned} \tag{5.2}$$

The term $x/(255 \cdot 256)$ can be neglected since it is zero for any value $x \in [0, 65025]$ when performing integer division. To account for rounding up, 128 needs to be added to the numerator, thus:

$$\frac{x}{255} = \frac{x + (x + 128)/256 + 128}{256} \tag{5.3}$$

This expression only needs 16-bit operands to be computed, and yields correct results for any x in $[0, 65025]$ when compared to the same operation performed in floating point and then truncated to 8 bits. This is straightforward to implement as C code. As seen in figure 5.1 this introduces no visual artifacts.

5.3 Compositing images in assembly

Modern processors are able to carry out this sort of operations in a SIMD fashion. The following code exploits Intel's MMX extension to the 386 instruction set in order to perform alpha compositing of two images on the CPU. There is some room for improvement with regard to data prefetching and processor-specific optimizations (see for example [Adv02]). Nevertheless, this code composites in excess of 56 1024×1024 image pairs per second (117 megapixels per second, less than 18 ms per image pair) on an Athlon MP 2000+ CPU, which is equivalent to circa 28 CPU cycles per pixel in the final image. Returning to the example computation in section 5.1, this means $19 + 15 * 18 = 289$ ms for that case (read back, blend in software 16 images). It also compares favorably with respect to the numbers quoted in table 5.2. Under ideal conditions, this can scale linearly with the number of CPUs.

In the following listing, `%1` contains A, `%2` contains B and `%0` is the destination address. AT&T syntax is used for the operands.

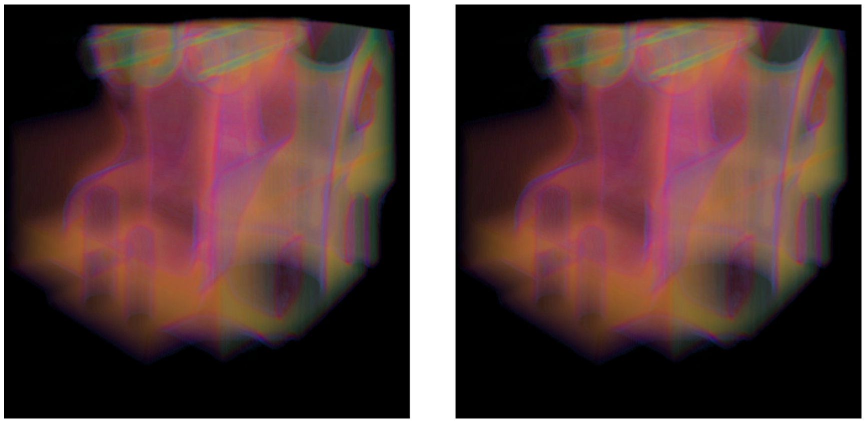


Figure 5.1: Comparison of images composited in hardware using the traditional single-brick path (left) and in software using multiple bricks (right). (See figure A in the color plates)

```
pxor           %mm2, %mm2   /* clear mm2              */

mov            $128, %eax
movd           %eax, %mm4
pshufw    $0, %mm4, %mm4   /* copy 128 to all words  */

movd           (%1), %mm0   /* copy a to mm0          */

movd           (%2), %mm3   /* copy b to mm3          */
punpcklbw      %mm2, %mm3   /* 16-bit expand b        */

pcmpeqb        %mm1, %mm1   /* fill mm1 with 1's      */
pxor           %mm0, %mm1   /* 1 - aalpha             */
punpcklbw      %mm2, %mm1   /* 16-bit expand 1-aa     */
pshufw    $0, %mm1, %mm1   /* copy 1-aa to all words */

pmullw         %mm1, %mm3   /* x = (1-aalpha)*b       */
paddusw        %mm4, %mm3   /* x += 128               */
movq           %mm3, %mm1   /* y = x                  */
psrlw            $8, %mm1   /* y /= 256               */
paddusw        %mm3, %mm1   /* y = y + x              */
psrlw            $8, %mm1   /* y /= 256               */

packuswb       %mm1, %mm1   /* pack result            */

paddusb        %mm1, %mm0   /* add a and (1-aalpha)b  */
movd           %mm0,  (%0)  /* copy result to memory  */
```

The AMD64 architecture extends Intel's x86 architecture to use a 64-bit address size and some register extensions, including the so called *128-bit media instructions*, which use 128-bit XMM registers and is a combination of the SSE and SSE2 instruction sets ([Adv03a], [Adv03b]). In order to take advantage of these the code presented previously needs to be modified as follows:

```
rex64 movd           (%1), %xmm0  /* x0 = aa'                */
rex64 movd           (%2), %xmm1  /* x1 = bb'                */

      mov     $0x00800080, %rax
      movd           %rax, %xmm2
      pshufd    $0, %xmm2, %xmm2  /* x2 = 00 80 x 8          */

      pxor          %xmm3, %xmm3  /* clear xmm3              */
      pxor          %xmm4, %xmm4  /* clear xmm4              */

      punpcklbw     %xmm3, %xmm0  /* put a pixels in place   */
      punpcklbw     %xmm3, %xmm1  /* put b pixels in place   */

      pcmpeqd       %xmm3, %xmm3  /* fill xmm3 with 1's      */
      punpcklbw     %xmm4, %xmm3  /* construct 16-bit 255    */
      pxor          %xmm0, %xmm3  /* xmm3 = 1 - alpha        */
      pshufhw   $0, %xmm3, %xmm3  /* put 1 - alpha on all    */
      pshuflw   $0, %xmm3, %xmm3  /* words                   */

      pmullw        %xmm3, %xmm1  /* x1 = (1-a)*b            */
      paddusw       %xmm2, %xmm1  /* x1 += 128               */
      movdqa        %xmm1, %xmm2  /* x2 = x1                 */
      psrlw            $8, %xmm2  /* x2 /= 256               */
      paddusw       %xmm2, %xmm1  /* x1 += x1/256            */
      psrlw            $8, %xmm1  /* x1 /= 256               */
      packuswb      %xmm1, %xmm1  /* saturate & pack result */
$
      packuswb      %xmm0, %xmm0  /* XXX: pack a again       */

      paddusb       %xmm1, %xmm0  /* x0 += x1                */
rex64 movd          %xmm0,   (%0) /* copy result to memory   */
```

This version is able to composite images at a rate of 115 image pairs per second when running on an Opteron system clocked at 1.8 GHz. This gives, for the example case $19+15*9 = 154$ ms. Figure 5.2 shows a comparison between both implementations running on different hardware.

If multiple CPUs are available on a single system, it is possible to perform the compositing operation in parallel across them. There is some overhead associated with thread management as well as memory-bandwidth contention. Nevertheless, a dual Athlon MP system running at 1.7 GHz composites 91 image pairs per second, and a dual Opteron system running at 1.8 GHz composites 193 image pairs per second giving 96 ms total time for the example case.

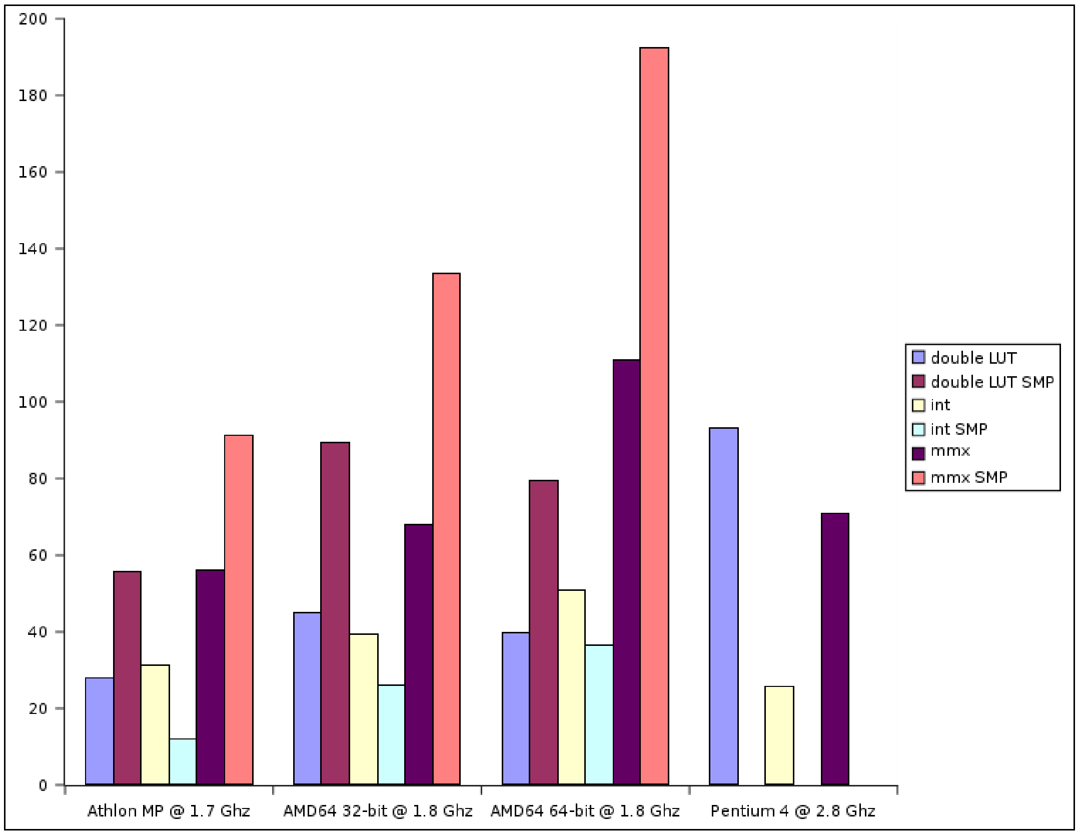


Figure 5.2: Performance comparison for software blending across different systems. The numbers represent image pairs per second, for images with size 1024×1024 pixels. The “double LUT” data is included for reference; this is a look-up table implementation that is filled with $256 \times 256 \times 256$ values computed using double precision floating point arithmetic. “int” is a C implementation of the algorithm as discussed in the text. “mmx” is either of the implementations quoted in the text. It is interesting to note that the LUT-based method is faster than the MMX code on the Pentium 4, which is probably due to the higher memory bandwidth available on that architecture.

5.4 Discussion

Since reading images from the framebuffer takes of the order of 20 ms, any software path which is able to composite images faster than that, that is, any path compositing more than 50 pairs per second, turns the process of obtaining the data from the framebuffer into the application's bottleneck. As shown in figure 5.2 this is already the case for CPUs which have been available on the market for almost two years. Recently available CPUs, like AMD's Opteron, are capable of rates well in excess of that. The bottleneck here is the connection between the host's main memory and the graphics card. Uploading data to the card is a very common operation and both drivers and hardware are heavily optimized for it. On the other hand, reading data back from the graphics card into the host's memory is not so common and vendors usually do not invest much time optimizing this path. Until recently, it was not usual to see a performance increase by just upgrading drivers. A very immature implementation of NVIDIA's video drivers for Linux achieved less than 60 MB/s for this operation, while a later release used on the same hardware reached over 120 MB/s. Nevertheless, it is not realistic to hope for significant improvements without moving from AGP connections to a different architecture, for example *PCI Express* parts, which should theoretically be able to achieve transfer rates *from* the graphics card, at a rate of 1 GB/s. Until such a change happens, in-CPU compositing should still be an attractive option.

Chapter 6

System architecture

6.1 Message-Passing Interface

The Message-Passing Interface (MPI) is the de-facto API for message-passing software used for developing high-performance portable parallel applications [GLS94]. It is defined by a standard [SOHL$^+$98] and implementations for specific hardware platforms are provided by the respective vendors. It is not tied to a particular operating system (implementations are available for several UNIX variants as well as the Microsoft Windows family of operating systems) nor hardware architectures (e.g., Cray's C90 and T3E, SGI's Origin and Onyx, IBM's SP1 and SP2, Intel's Paragon) nor shared memory models (besides the architectures already mentioned, multiple implementations are available for NOWs with several kinds of interconnects). Particularly notorious is the MPICH implementation, developed by Argonne National Laboratory and Mississippi State University [GLSD96]. This implementation is the basis for many of the implementations by the previously mentioned vendors, which has earned its reputation for good portability across platforms and its good message passing performance. These two reasons are the main motivation for choosing MPI as the communication layer on which the rest of the parallelization framework rests.

6.2 Network performance

MPICH uses a layered software architecture, as shown in figure 6.1. This architecture allows vendors to implement different low-level devices without having to worry about the high-level MPI implementation. For example, Myricom[1], producers of the Myrinet PC-interconnection hardware, distribute their own MPI implementation, MPICH-GM, which uses the GM Myrinet drivers for the low-level device communication. The *PC Cluster Consortium*[2] (and previously the “Real World Computing Partership”) provides the *SCore Cluster System Software*, “a high-performance parallel programming environment” for networks of workstations. SCore includes

[1] `http://www.myri.com/`
[2] `http://www.pccluster.org/`

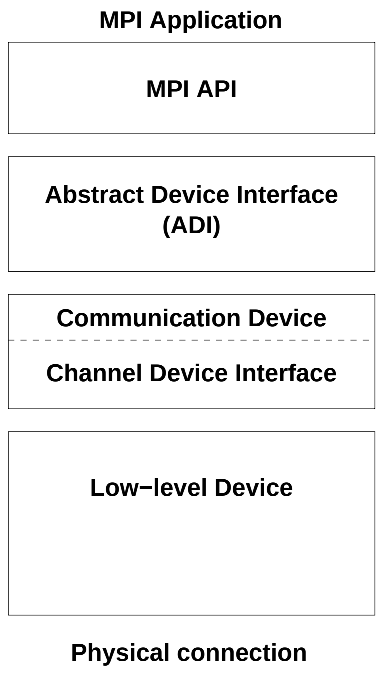


Figure 6.1: MPICH layered architecture.

low-level Myrinet drivers (PM) as well as an MPICH-based MPI implementation working on top of these drivers. An interesting aspect of SCore is that it makes several communication devices available to the applications. The shared memory device is automatically used for communication across tasks running on the same physical node. For the communication across tasks running on different physical nodes there are two devices available: `ch_score` (the default) and `ch_score2`. The later makes optimizations that are much more agressive, but early implementations of SCore had stability problems when using it. In recent versions of SCore the `ch_score` device has incorporated many of these optimizations and `ch_score2` has been discontinued. The effect of switching between devices can be seen in figure 6.2. For large messages, the optimized device can achieve data-transmission rates in excess of 150 MB/s.

MPICH uses different communication protocols according to the size of the message being transmitted across nodes. The boundaries between the regions where each protocol is used are determined based on the system's limitations (communication media, buffering space) and average-case optimization criteria. Generally these protocols are: short, eager (long messages) and rendezvous (very long messages). The effect of crossing the boundaries across protocols can be clearly seen in figure 6.3 as sharp discontinuities in the transfer speed. In this case the rendezvous protocol achieves circa 100 MB/s. The performance measurement was done by having the test program exchange messages of the given size across two nodes in a concurrent fashion (using `MPI_Sendrecv`), which reproduces the actual conditions under which the parallel rendering code works.

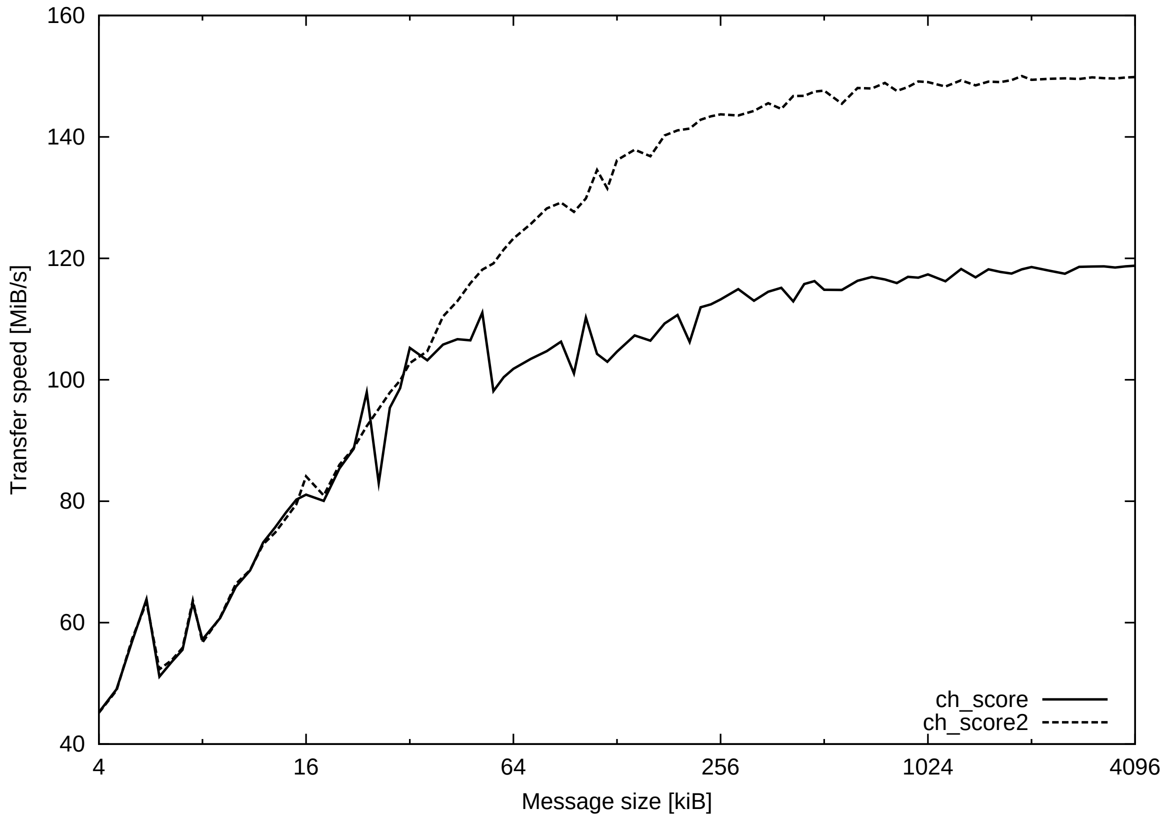


Figure 6.2: Measured network bandwidth as a function of the message size for two different devices. In both cases the same full duplex Myrinet network is used for the physical data transmission. In order to measure performance, the test program sends messages of the give size from one node to another and then back.

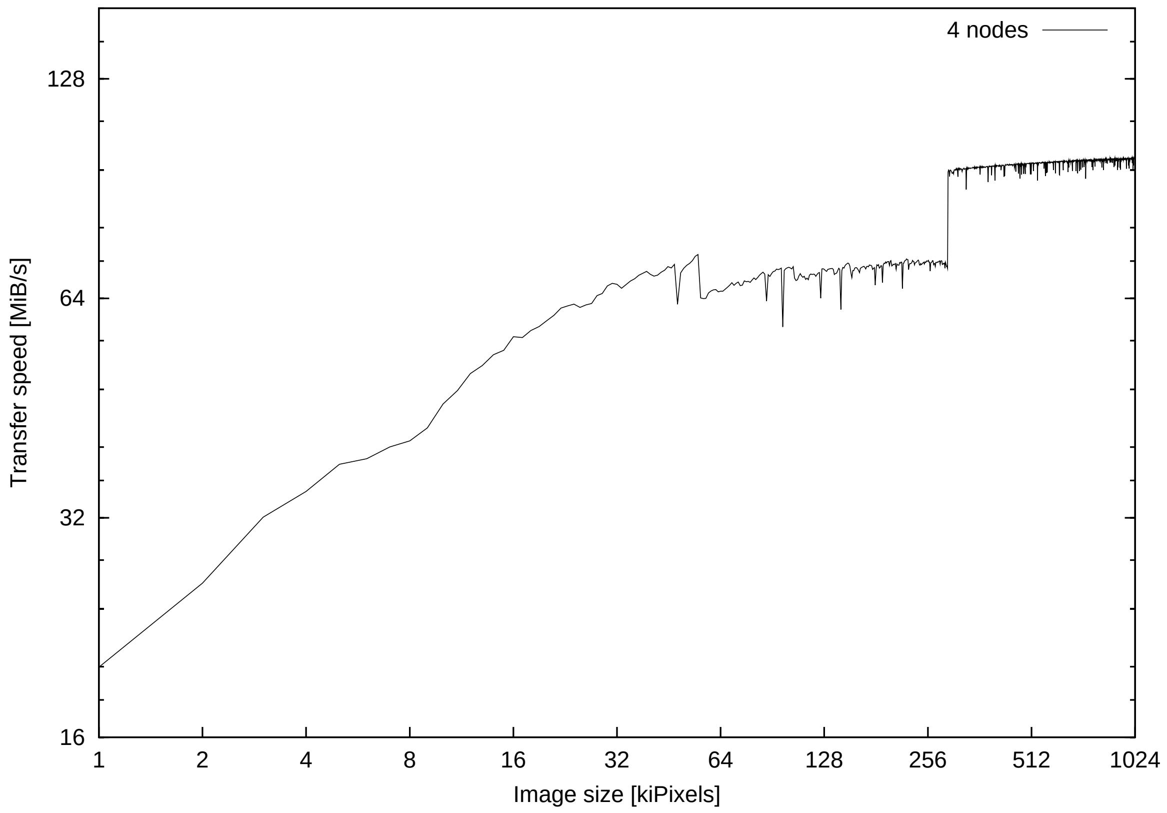


Figure 6.3: Transfer speed across four different nodes as a function of image (message) size. The measurement was done using four PCs interconnected via a full duplex Myrinet network.

6.3 Multithreading support

As mentioned before, the actual implementation of the system is more convoluted than desired, one of the reasons being the fact that the MPI implementation used at the main testing site does not allow the application to use multiple execution threads. It is common for MPICH-based implementations to lack support for threads in the program and SCore is no exception. This is due in part because the original MPICH implementation does not support it (since there is no platform-independent threading API, or in general a portable method for threaded programming) and in part because it is a difficult aspect to implement (the MPI standard leaves some room for implementation-specific behaviour). Protopopov and Skjellum [PS01] provide an in-depth discussion of the difficulties involved in the design and implementation of a multithreaded MPI architecture as well as the reasons for the deficiencies in the MPICH design.

With respect to multithreading support, the MPI standard (p. 8) has the following to say:

> MPI does not specify the excecution model for each process. A process can be sequential, or can be multi-threaded, with threads possibly excecuting concurrently. Care has been taken to make MPI "thread-safe", by avoiding the use of implicit state. The desired interaction of MPI with threads is that concurrent threads be all allowed to excecute MPI call, and calls be reentrant; a blocking MPI call blocks only the invoking thread, allowing the scheduling of another thread.

The MPI 2 standard expands this language to read ([HL97], p. 193):

> [...] This section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, it is not required that all MPI implementations fulfill all the requirements specified in this section.

A distinction is made between a *thread compliant implementation* and an application program using threads. This means that even if the MPI implementation is *not* thread-safe, an MPI program *could* use threads, as long as the communication is restricted to one single thread. This programming model is what the MPI 2 standard calls *funneled*:

> MPI_THREAD_FUNNELED The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are "funneled" to the main thread).

This would allow the program to overlap communication and computation in a simple way. Unfortunately, due to an implementation issue, SCore does not allow even this programming model. Since Myrinet hardware is capable of asynchronous communication and the drivers support it, the problem is somewhat alleviated, but the use of multiple CPUs per node to speed up some operations is precluded. This could be worked around by spawning N tasks per node (where N is the number of CPUs per node, 2 for the particular case of the testing facility), but this could introduce some unnecessary and measurable overhead because of the extra memory copy operations that would be needed. Yet this alone is no reason for disregarding this implementation alternative completely.

6.4 OpenGL integration

The "implementation issue" in SCore that prevents programs from using the "funneled" threading model also prevents the direct use of the OpenGL API in MPI programs running on the SCore environment. A design decision in SCore [Ish01] prevents programs from being dynamically linked against libraries. On one hand, SCore favors statically linked programs for performance reasons. It is generally agreed that programs making use of dynamically linked symbols run measurably (albeit not significantly) slower and have (in the general case) a larger memory footprint than their statically linked counterparts because of relocation overhead, but this is in the general case amortized by the time the function spends doing useful work. A good discussion about dynamic linking has been presented by Franz [Fra97]. On the other hand, in order to implement a single system image model SCore provides its own versions of some functions in the standard C library. These considerations taken together made the SCore developers ban the use of shared libraries in the SCore environment. As a result it is not possible to use either the POSIX threads library (`libpthread`) because it is usually tightly coupled to internal mechanisms of the C library, nor the OpenGL library (`libGL`) because it is usually available only as a dynamically shared object (in OpenGL's case, relocation overhead is insignificant compared to other factors which need to be taken into account into the design of a high-performance OpenGL implementation [KBH95]).

The solution employed to work around this problem is to split each MPI task into two processes: one of them in charge of the communication over the network and the other one in charge of the actual rendering operations. The two processes communicate with each other over two channels: synchronization messages (short) are sent over a socket or pipe and data messages (large) are exchanged over a shared memory area (standard System V functionality).

This implementation has one deficiency which can not be easily avoided, namely, the latency of the operations is determined by the latency of the slowest device. Assuming the implementation is fair in terms of polling devices, the average latency is equal to half the sum of the latencies of the devices being polled, which is on the order of magnitude of the largest latency in the group. The devices to be considered are the input devices in the system, the network devices and the shared memory link across processes. Polling locally attached input devices for event availability has a latency on the order of magnitude of 10^{-5} s and the same can be said of the shared memory link. The Myrinet network device has, theoretically, a latency on the order of 10^{-6} s. For the case of accessing the service from a remote location, in the manner described by Stegmaier at al [SME02], the latency of the input devices increases to the order of 10^{0} s $\sim 10^{-3}$ s.

6.5 Access to the hardware

One of the first issues to be resolved was granting access to the X server. Under the usual X11 model, access is granted to the user sitting at the console, that is, to the user directly interacting with the X server. Since the environment precludes the existence of "an user sitting at the console" at first access control was completely disabled. This means that any user logged on the machine can get access to the X server by pointing the X client to the local display. In a

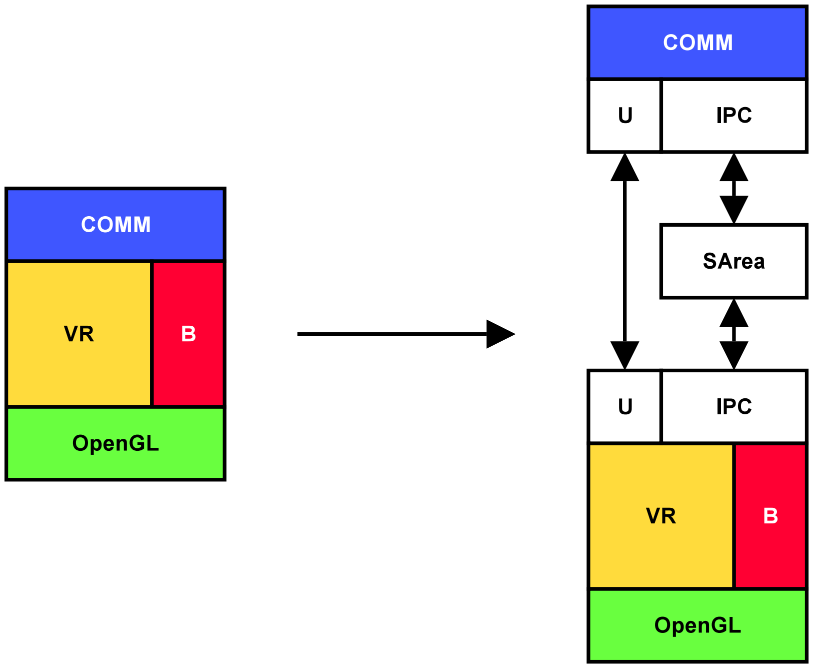


Figure 6.4: Architecture of the interprocess communication.

controlled environment, this might be a non-issue, but since there is no control over the users of the system, some form of access control was desired.

A custom PAM module was written for this purpose, which was attached to the login service. PAM stands for “Pluggable Authentication Modules” and is a suite of shared libraries that enable the local system administrator to choose how applications authenticate users. This is achieved without having to modify the actual applications. This particular PAM module grants a lock (and access along with it) if no other user is logged in at the moment. It uses the *X Security Extension* API to ask the X server for a “MIT magic cookie” and installs it on the user’s `.Xauthority` file. This enables the X clients to authenticate themselves with the X server. Once the user logs out, the cookie is invalidated in the X server and the lock is removed. By using this method exclusive resource allocation is ensured and other users are prevented from snooping the display. It is worth nothing that using the NVIDIA drivers for Linux, it’s possible to read the contents of the framebuffer by opening `/dev/nvidiaN` and mapping certain parts of it into the process’ address space.

Chapter 7

Application development with OpenGL

7.1 Motivation

Working with OpenGL can at times annoying. It is sometimes difficult to understand why the expected output does not show up or even worse, why nothing at all shows up on the display. Over the years, every OpenGL programmer develops a toolbox of macros and functions to deal with this situation. Even so, many programmers prefer OpenGL over the alternatives, perhaps for two basic reasons: 1. It is platform independent, and there are good implementations for a myriad of platforms; 2. The nature of its development process leaves plenty of room for hardware manufacturers to experiment with new ideas in a way that allows programmers to tinker with the hardware as soon as it becomes available thanks to the so called "OpenGL extension mechanism". There are nevertheless two issues that are a common cause of grief among programmers:

- Using extensions can be cumbersome and hard to maintain from a software development point of view.
- Debugging OpenGL applications can lead to time-consuming "`printf` debugging" cycle.

Two answers to these problems will be described in the rest of this section.

7.2 Debugging OpenGL applications

The IRIX operating system ships a utility called `ogldebug` which allows the programmer to place breakpoints at those places in the program where OpenGL calls are made. Reimplementing this utility can be done easily: a customized OpenGL dynamic library is installed on the system, and it is used instead of the native OpenGL library. In this customized version, the OpenGL API entrypoints are exported to the application, but instead of performing their usual tasks, they provide the desired tracing and breakpointing functionality. Taking for example the case of the `glBegin` entrypoint, its new implementation might look like this:

```
GLvoid glBegin(GLenum mode)
{
    trace(GLBEGIN, mode);
    break_if(GLBEGIN);
    return;
}
```

In this hypotetical implementation, `trace` performs all the necessary tasks for tracing the calls, including converting numeric constants back to their human-readable representations (the consequences of this conversion will be discussed later). The problem with this implementation is that the original functionality of the entrypoint has been lost. As explained in more detail in chapter 8, the addresses of the entrypoints in the original library can be recovered easily. With those at hand, the new implementation might look like:

```
GLvoid glBegin(GLenum mode)
{
    trace(GLBEGIN, mode);
    break_if(GLBEGIN, PRE_CALL);
    real_glBegin(mode);
    break_if(GLBEGIN, POST_CALL);
    return;
}
```

Regarding the maintainability of this idea, the OpenGL *Sample Implementation*[1] from SGI includes a machine-readable description of the whole OpenGL API. This description is used among other things to generate the header files for the C language and the dispatch tables for the entrypoints that any OpenGL implementation has to provide. This means this description is an ideal starting point for automatically generating the code required by a debugging facility similar to `ogldebug`.

The actual implementation of this idea has been called `spyGLass` and is available for download from `http://spyglass.sf.net/`. The command line version provides only tracing facilities, for example:

```
$ spyglass ./spyglass_demo
glXChooseVisual(0x804a260, 0, attriblist);
glXCreateContext(0x804a260, 0x804be18, (nil), 1);
glXIsDirect(0x804a260, 0x804f790);
glXMakeCurrent(0x804a260, 23068673, 0x804f790);
glClearColor(0.5, 0.5, 0.5, 1);
glXMakeCurrent(0x804a260, 23068673, 0x804f790);
glViewport(0, 0, 400, 400);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

[1]`http://oss.sgi.com/projects/ogl-sample/`, visited 2/12/2004

```
glOrtho(-1, 1, -1, 1, -1, 1);
glXMakeCurrent(0x804a260, 23068673, 0x804f790);
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
glVertex2d(-0.5, -0.5);
glVertex2d(0.5, -0.5);
glVertex2d(0.5, 0.5);
glVertex2d(-0.5, 0.5);
glEnd();
glEnable(GL_NO_ERROR);
Error: GL_INVALID_ENUM [ 0x8048a63 0x400a706d 0x400a72c5 ... ]
glGetError();
glXSwapBuffers(0x804a260, 23068673);
glXMakeCurrent(0x804a260, 23068673, 0x804f790);
```

Notice how the program calls `glEnable` with `GL_NO_ERROR` as parameter and the debugger automatically reports the error. Looking at this more closely, it is necessary to notice that the program is making a call *equivalent* to `glEnable(GL_NO_ERROR)`. The enumerant `GL_NO_ERROR` has the value `0x0`, and so do the enumerants `GL_FALSE`, `GL_POINTS`, `GL_ZERO` and `GL_NONE`. This means that when converting the value `0x0` back to a human-readable representation, it is necessary to pick one out of at least five possibilities. `spyGLass` is not, by design, a source code debugger, therefore it is not possible to refer to the source code in order to find the actual line that is producing the error in question. For the large majority of enumerants this is not a problem in practice since they have different values and new values are assigned in such a way that they do not collide with existing ones. Only bitmasks need to be handled in a special way.

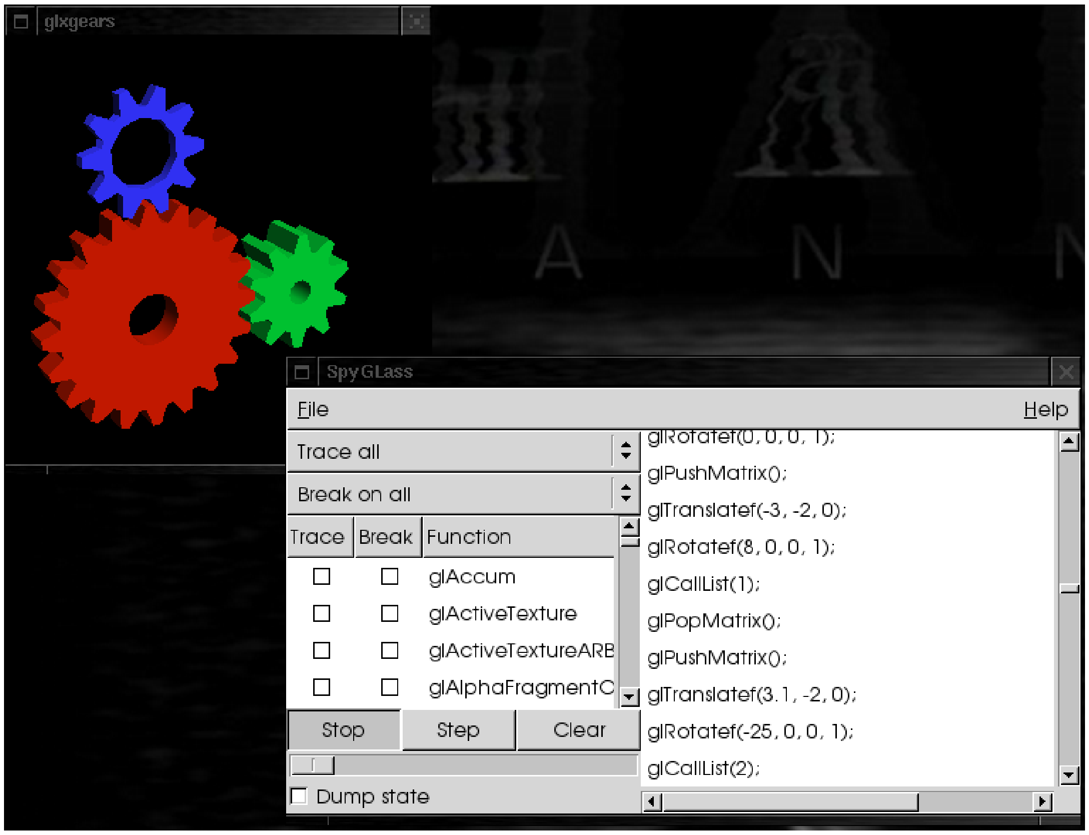
The information reported after the error are addresses which can be used as input for, e.g., the `addr2line(1)` program (part of the GNU binutils package) to obtain the location in the source code where the error is happening.

A graphical user interface is also available. This provides breakpointing and filtering facilities (see figure 7.1).

7.3 Managing OpenGL extensions

7.3.1 Motivation

The other problem mentioned before was managing OpenGL extensions. As the name implies, these are extensions to the core OpenGL functionality. Usually these are introduced by hardware vendors as *vendor specific extensions* (e.g. `GL_NV_vertex_program`). In some cases, if the developer base demonstrates enough interest, they are promoted to *multi-vendor* (denoted by the string `EXT` in their name, e.g. `GL_EXT_texture3D`) or *ARB-approved* status (e.g. `GL_ARB_vertex_program`). An extension specification consists of, among other information, a *name*, one or more *name strings*, and a list of *new procedures and functions* and *new*



```
glRotatef(0, 0, 0, 1);
glPushMatrix();
glTranslatef(-3, -2, 0);
glRotatef(8, 0, 0, 1);
glCallList(1);
glPopMatrix();
glPushMatrix();
glTranslatef(3.1, -2, 0);
glRotatef(-25, 0, 0, 1);
glCallList(2);
```

Figure 7.1: Graphical user interface for `spyGLass`.

tokens which are introduced by the extension. All this information is kept current in a central location called “the OPENGL extension registry”[2].

The problem is not actually coping with extensions, but the many misconceptions that have found their way into the common knowledge during the years. For example, at the OPENGL website[3] the following text is found:

> With the isExtensionSupported routine, you can check if the current OPENGL rendering context supports a given OPENGL extension. To make sure that the `EXT_bgra` extension is supported before using it, you can do the following:

```
    /* At context initialization. */
    int hasBGRA = isExtensionSupported("GL_EXT_bgra");

    /* When trying to use EXT_bgra extension. */
#ifdef GL_EXT_bgra
    if (hasBGRA) {
        glDrawPixels(width, height,
                     GL_BGRA_EXT, GL_UNSIGNED_BYTE,
                     pixels);
    } else
#endif
    {
        /* No EXT_bgra so bail (or implement
         * software workaround). */
        fprintf(stderr, "Needs EXT_bgra extension!\n");
        exit(1);
    }
```

> Notice that if the `EXT_bgra` extension is lacking at either run-time or compile-time, the code above will detect the lack of `EXT_bgra` support. Sure the code is a bit messy, but the code above works. *You can skip the compile-time check if you know what development environment you are using* and you do not expect to ever compile with a `<GL/gl.h>` that does not support the extensions that your application uses. But the run-time check really should be performed since who knows on what system your program ends up getting run on. *(emphasis by the author)*

The problem with the proposed idiom is exactly the one explained by the text following it: the extension has to be present at compile time in order to be able to use it. The original software developer might have control over this, but once the software starts to be distributed in source

[2]`http://oss.sgi.com/projects/ogl-sample/registry/`, visited 2/12/2004

[3]`http://opengl.org/developers/code/features/OGLextensions/OGLextensions.html`, visited 2/12/2004.

code form, that control is lost. There is nothing that guarantees or even requires that the extension definitions are present on a randomly selected development environment. With the code quoted above, this means that compiling the source on different development environments produces different programs.

The proposed idiom suffers from yet another problem, which is better illustrated by the following code fragment:

```
1 #ifdef GL_NV_register_combiners
2     if (hasRegisterCombiners)
3     {
4         glEnable(GL_REGISTER_COMBINERS_NV);
5         glCombinerParameteriNV(GL_NUM_GENERAL_COMBINERS_NV, 1);
6         // ...
7         glDisable(GL_REGISTER_COMBINERS_NV);
8     }
9 #endif
```

This code is conditionally compiled if the macro `GL_NV_register_combiners` is defined (line 1), which guards against the extension tokens and entry-point declarations not being available at compile time. It tests for the extension being available at runtime (line 2) and then proceeds to use the tokens and functions defined by the extension. The problem is that this code requires the extension entry-points (`glCombinerParameteriNV`) to be available to the program at startup time. This means that if the extension is not available at start up, the program might fail to start up at all. To avoid that problem, OPENGL defines a mechanism to retrieve pointers to entry-points at runtime. This mechanism is itself defined as an extension to the several windowing system dependent bindings (e.g., AGL, GLX and WGL). The reasons for this dependency are out of the scope of this discussion, but it suffices to say that it exists and hinders portability. Obviously it is possible for the programmer to have an abstraction layer on top of the different windowing systems in order to cope with this problem but this adds yet another piece of code which has to be actively maintained. Most of the maintainance work would be associated with the extension information itself. When using this mechanism, the previous listing might look like:

```
    struct opengl_extension_table glext;
    // ...
#ifdef GL_NV_register_combiners
    if (glext.hasRegisterCombiners)
    {
        glEnable(GL_REGISTER_COMBINERS_NV);
        glext.CombinerParameteriNV(GL_NUM_GENERAL_COMBINERS_NV, 1);
        // ...
        glDisable(GL_REGISTER_COMBINERS_NV);
    }
#endif
```

In this case, the `opengl_extension_table` structure would contain a list of flags indicating if a given extension is available or not, and pointers to the corresponding entry-points, if any. Something akin to:

```
struct opengl_extension_table
{
    int hasRegisterCombiners;

    PFNCOMBINERINPUTNV CombinerInputNV;
    PFNCOMBINEROUTPUTNV CombinerOutputNV;
    PFNCOMBINERPARAMETERFNV CombinerParameterfNV;
    PFNCOMBINERPARAMETERFVNV CombinerParameterfvNV;
    PFNCOMBINERPARAMETERINV CombinerParameteriNV;
    PFNCOMBINERPARAMETERIVNV CombinerParameterivNV;
    PFNFINALCOMBINERINPUTNV FinalCombinerInputNV;
    PFNGETCOMBINERINPUTPARAMETERFVNV GetCombinerInputParameterfvNV;
    PFNGETCOMBINERINPUTPARAMETERIVNV GetCombinerInputParameterivNV;
    PFNGETCOMBINEROUTPUTPARAMETERFVNV GetCombinerOutputParameterfvNV;
    PFNGETCOMBINEROUTPUTPARAMETERIVNV GetCombinerOutputParameterivNV;
    PFNGETFINALCOMBINERINPUTPARAMETERFVNV
        GetFinalCombinerInputParameterfvNV;
    PFNGETFINALCOMBINERINPUTPARAMETERIVNV
        GetFinalCombinerInputParameterivNV;
};
```

The “has” flags are initialized using e.g. the `isExtensionSupported` function found at the OPENGL website, and the function pointers are initialized by calls to the appropriate facility of the windowing system. Maintaining this sort of code for a large number of extensions by hand is error-prone and quickly becomes tedious.

7.3.2 GLEW: The OPENGL Extension Wrangler

Faced with these problems a solution was developed during this thesis. Milan Ikits developed independently a similar solution. These two efforts were later joined in a library called `GLEW`. The goals of this project are:

- To be simple to use
- To be portable
- To be simple to maintain
- To guarantee forwards binary-compatibility

Simplicity of use is achieved in two ways: on one hand, there is a single function that needs to be called in order to initialize the library (`glewInit()`). After this, all the functionality is readily available to the programmer. On the other hand, the API is such that the programmer only needs to read the desired extension's specification to be able to use the corresponding functionality via GLEW. The compile-time problem mentioned before does not exist anymore, since GLEW itself provides the tokens for all the extensions it knows about. The run-time problem does not exist either, since all the known entry-points are available to the program at link-time. After a successful initialization of the library, variables of the form `GLEW_`*`extension_name`* indicate the runtime availability of the extension (or lack thereof). This means the previous example can be simplified and rewritten as:

```
    glewInit();
    // ...
    if (GLEW_NV_register_combiners)
    {
        glEnable(GL_REGISTER_COMBINERS_NV);
        glCombinerParameteriNV(GL_NUM_GENERAL_COMBINERS_NV, 1);
        // ...
        glDisable(GL_REGISTER_COMBINERS_NV);
    }
```

Portability is achieved by abstracting the differences between systems in the library itself. The only system-dependency that is exposed to the programmer is the one he already has to be aware of: windowing system specific extensions. The library has been used sucessfully with several OpenGL implementations on different operating systems, including Microsoft Windows, several Linux distributions, Silicon Graphics' IRIX (both old and new generation systems) and Apple's MacOS X.

The effort required to keep the library up to date is minimal, since the bulk of the code is automatically generated from the extension specification files found at the OpenGL extension registry. This is done in a two step process: first the text of the specifications is parsed, normalized and stored in an intermediate database and then the normalized form is converted to C source code for distribution. This allows the developers to manually add extensions for which no specification is available in a suitable format. Using the information available from the OpenGL Sample Implementation mentioned before could have been an option, but observational evidence suggests this is not kept as up-to-date as the extension registry.

Because the dynamic nature of this library (the basic API changes little, but the list of supported extensions is always growing), guaranteeing forward binary compatibility was an important design consideration. This means that a binary linked against an older version of the library should work without problems with a newer version of the library. The two consequences of this are:

- No extensions will be removed, even if they are deprecated by other extensions or incorporated into the core of OpenGL.

- Entry-points and flags are exported as freestanding variables instead of being packed inside of a structure. This pollutes the global namespace, but is necessary to avoid (implicit) dependencies on the size of the structures in question.

Chapter 8

Remote access of visualization facilities

8.1 Motivation

During the development of this work, the need to be able to access remote visualization facilities in a generic fashion became obvious. The common generic approach for remote visualization employs the OpenGL remote rendering facilities. In this approach the application is run remotely while the local host does the actual rendering. This solution is application-independent and therefore very popular, but it does not benefit from special facilities or hardware features available at the remote host. For this reason, the second most popular solution is to embed enough functionality into the application for it to be able to shift some of the workload from the local system to the remote one (e.g., Bethel [Bet00], Engel [EHT$^+$00, ESE00, EE99], Ma [MC00], among others). Being application-specific, this solution can exploit the knowledge about application behavior to achieve a higher degree of interactivity or to better balance the workload between the client and the server. The disadvantage of the solution lies also there: being application-specific (or in the best case, toolkit-specific), applications need to be modified to take advantage of this. In the case of toolkit-specific solutions, it might be possible to come up with an implementation that does not require modifications at the application level, but special versions of the toolkit libraries are nevertheless needed.

In the general case, a generic solution will perform worse than specific solutions for the same problem, so why should one care about application independence? Rapid prototyping is one reason. Just being able to test existing applications under different operating conditions is enough reason to strive for application independence. Another reason is shifting applications from their original operating environment to new ones, probably unforeseen in the original design. A good example of this is being able to embed applications originally written to be run in a desktop or workstation environment to a web-services type of platform. The disadvantage of the generic solution is that, by nature, it has to be implemented at a very low level and has to work with low level primitives. The simplest most generic approach involves letting the application render to the framebuffer in the usual fashion and introduce some mechanism to *copy* the contents of the framebuffer to a remote location. This means pixels are the primitives that the solution will be working with, and the necessary modifications are to be introduced at the basic graphics API or windowing system level.

In this particular implementation of the idea the well known concept of *dynamic linking*[HO91] and certain characteristics of the *X Window System* are exploited to achieve the desired result. OpenGL rendering requests are redirected to the hardware where the program is *running* instead of being sent to the hardware where the program is being *displayed*. Once the rendering is done, the resulting image is read from the framebuffer and sent over the network to the display together with all other GUI elements. Since all the OpenGL rendering happens on the remote display, it is possible to use an existing OpenGL-based application without requiring the availability of OpenGL-capable hardware locally, while taking advantage of advanced graphics features as well as hardware acceleration by the *remote* hardware. By leveraging already existing remote-access applications this solution also makes it possible to access remote visualization resources from operating environments that do not provide implementations for the required protocols.

Some other solutions with varying degrees of genericity or flexibility are:

- Silicon Graphics, Inc. provides a commercial solution called OpenGL VizServer [Sil01], that enables lightweight clients such as SGI O_2 and PC workstations to access the rendering capabilities of SGI Onyx servers. Because of design decisions, other architectures cannot be used as servers for this application. The VizServer software relies on dynamically linked executables in order to be able to implement its functionality without modifying the target application.

- Ma and Camp [MC00] developed a solution for remote visualization of time-varying data over wide area networks. It involves a dedicated display daemon and display interface. The first receives data from a render process, compresses it and passes it to the second, which in turn decompresses the data and presents it to the user. By using a custom transport method, they are able to employ arbitrary compression techniques.

- Bethel [Bet00] presented Visapult, a prototype system developed at Lawrence Berkeley National Laboratory that combines minimized data transfers and workstation-accelerated rendering. Visapult also requires modifications of the application in order to make it "network aware" and relies to some extent on the existence of hardware acceleration on the local display.

- Engel and Ertl [EE99] developed a solution for remote collaborative volume visualization which exploits the characteristics of the application domain to reduce latency as well as required network bandwidth. Engel et al [EHT$^+$00] further developed this approach and implemented a hybrid rendering mechanism to obtain better framerates.

The rest of this chapter is an extended version of the ideas and discussion published in [SME02].

8.2 OpenGL Graphics with the X Window System

`GLX`[WL98] is an extension to the X protocol [Nye95] that allows clients to create a so-called "`GLX` context" which can be used to issue OpenGL calls that can be executed using either a

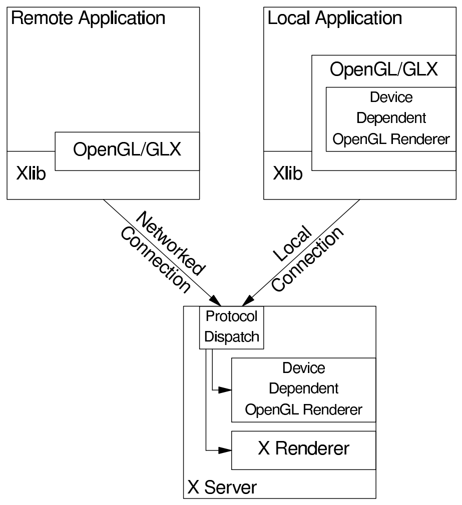


Figure 8.1: `GLX` architecture as presented by Kilgard [Kil96]

hardware-accelerated rendering engine or a software-based one. By sitting on top of X, network transparency is obtained for free. A `GLX` context can operate in either *direct* or *indirect* mode. In indirect mode, the client sends requests to the X server which propagates them to the hardware. In direct mode, the X server only functions as a marshal making sure that the OpenGL state of each client is kept consistent. Since the X protocol is bypassed in direct mode, OpenGL rendering can achieve the maximum performance of the hardware [KBH95]. Direct rendering implies that the process is running on the same machine as the X server and not over the network. In the case of remote rendering, only indirect mode is possible, if at all. Figure 8.1 illustrates both cases.

8.3 Dynamic Linking

In modern systems, programmers have the choice of *statically* or *dynamically* linking programs during compilation. In static linking, all the object references of a program are resolved during the link phase of the compilation. In contrast, with dynamic linking, the executable file contains references to, but not the actual code of the required library functions. In this case, symbol resolution is carried out during load-time. The dynamic linker is in charge of finding these references in the executable and *resolving* them using a list of *shared object names* (libraries) that the executable contains. One of the advantages of load-time linking is that it facilitates code reuse, it simplifies the task of fixing bugs and reduces the memory requirements imposed on systems, since code pages can be shared among unrelated processes. By its very nature, it also enables users to replace libraries with custom versions designed to modify the behavior of a program. The only requirement in this case is to keep the application binary interfaces unmodified. A typical dynamic linker implementation maps libraries into memory searching for the symbols the application requires, and for every required symbol it uses the first definition that it finds. This means the process of resolving symbols is dependent on the order the libraries are loaded into memory, and in general there is no way to ensure that any given order will be used. In order to allow for the selective overwriting of symbols without forcing the user to modify the application in any way, some operating systems offer a mechanism sometimes called *library preloading*[1]. The *user* can specify an arbitrary list of shared objects that are loaded into memory *before* the libraries required by the application. This means the dynamic linker will use the symbols defined by these "preloaded libraries" for resolving the symbols required by the application.

In addition to load-time linking, it is also possible to perform *runtime linking*, as first described by Ho and Olsson [HO91] and later explained in the Linux/GCC case by Lu [Lu95]. In this approach, additional objects can be opened at run-time and symbols can be selectively added to the running program. The most widely used interface for this purpose is `dlopen(3)`, available on systems such as IRIX, Linux, and Solaris, among others. This can be used in cooperation with the preloading mechanism mentioned before to wrap code around the original library functions: first the function is overridden using preloading, then its original code is recovered using dynamic linking and it is used by the customized version of the function to provide the original behavior if necessary. This sort of trickery is not that unusual: a good example is Joost Witteven's `fakeroot` package[2], which is widely used to provide a fake "root environment" where non-privileged users can manipulate filesystem attributes as if they were indeed the `root` user.

An implementation that takes advantage of facilities available on operating systems such as Linux and Solaris might look like this:

```
    for_each (symbol_name in symbols_to_be_loaded)
    {
        *(symbol_pointer) = dlsym(RTLD_NEXT, symbol_name);
        if (dlerror() != NULL)
            /* error handling */
    }
```

A more general implementation would be:

```
    void *handle;
    handle = dlopen(library_name, RTLD_NOW | RTLD_LOCAL);
    /* Error handling: handle should not be NULL */

    for_each (symbol_name in symbols_to_be_loaded)
    {
        *(symbol_pointer) = dlsym(handle, symbol_name);
        /* Error handling: the return value of dlerror()
           should not be NULL at this point. */
    }
```

[1]There is not a standardized term for this operation, but the term "preloading" is used at least on Linux's and Solaris' documentation. IRIX does not really have such a facility, but its dynamic linker allows the user to overwrite the complete list of libraries to be used by the application, achieving the same effect, even if it is not really "preloading" anything.

[2]`http://fakeroot.alioth.debian.org/`, visited 2/12/2004

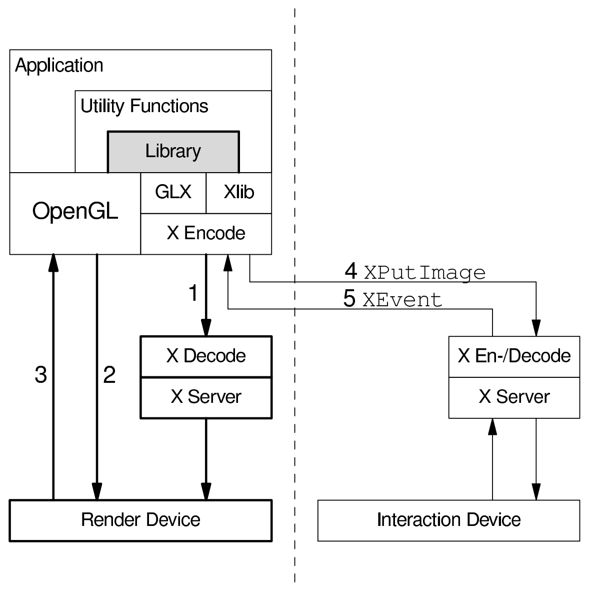


Figure 8.2: System architecture. (1) The application issues a GLX request which is sent to the render server. (2) The application issues OpenGL calls, which are handled by the render device. (3) The library reads the contents of the framebuffer and (4) sends it to the interaction server using an `XPutImage` request. (5) `XEvent`s are sent from the interaction server to the application.

Compare line 3 in the first listing with line 7 in the second. The parameter `RTLD_NEXT` is an extension which allows the program to obtain the address of the "next" symbol named `symbol_name` in the process' symbol table. On the other hand, the more generally applicable code addresses a specific library file via a previously obtained handle. The disadvantage of this is that the names of the relevant libraries have to be made available in the code, i.e. a system dependency has to be introduced.

8.4 Implementation

By taking advantage of the dynamic linking facilities explained above, it is possible to modify the behavior of any given program for the X Window System without changing its source code or that of the libraries it uses. In this implementation there are two X servers involved: one that supports `GLX` and a second one, which takes care of the user interaction, and which does not have to support `GLX`. In the following discussion, the first X server will be called *render server* and the second one the *interaction server*. *Display* will be used in the same way the X Window System defines it.

The application is started locally to the render server but its environment is configured to have it displayed on the interaction server. It is loaded in such a way that a custom version of every `GLX` function is used, whose job is to redirect `GLX` requests to the render server. Since the display is part of the `GLX` context's properties, OpenGL calls are automatically redirected to the render

`glXChooseVisual`	return a visual that matches the specified attributes
`glXCreateContext`	create a new `GLX` rendering context
`glXMakeCurrent`	attach a `GLX` context to a window or `GLX` pixmap
`glXSwapBuffers`	exchange front and back buffers

Table 8.1: Modified `GLX` functions

server. In a sense, a *new library* is inserted between the application and the system's OpenGL library, as depicted in figure 8.2. Once the application requests a buffer swap, the contents of the framebuffer is read and written to an `XImage` structure, which is sent to the interaction display via a `XPutImage` request. User interaction works transparently since events are transported between the interaction X server and the render server without modification.

For most `GLX` functions the only required change is redirecting the request from the interaction server to the render server. Only a few functions, listed in table 8.1, have to be treated in a special way. In this discussion, it is assumed that the client is using `GLX` 1.2. For later versions a similar implementation applies.

`glXChooseVisual` is used to select a visual[3] that matches the attribute set specified by the application. The application is neither required to use the visual returned by `glXChooseVisual`, nor is it prevented from calling it multiple times. The visual returned by `glXChooseVisual` has to be valid on the *interaction server*, since it will be used to create a widget there. The custom version of the function matches visuals across the two X servers and returns the best visual that is compatible with the given attribute set. In the general case this metric is not well defined: consider for example the case of a rendering server running at a color depth of 24 bits per pixel, and an interaction server running at 15 bpp. In this case it would be possible to present the application with a visual with color components that have 32 bits per pixel (RGBA), a 24-bit depth buffer and a 8-bit stencil buffer, which is impossible to obtain on a server running at 15 bpp. Depending on how the application is coded, it might or might not be able to cope with this. The other problem at this point is that the application is not required to actually *use* the visual returned by `glXChooseVisual`. It is also allowed to call the function multiple times or none at all (`glXChooseVisual` is ultimately a just convenience function). For this reasons, the only way to ensure the implementation will work with the largest number of applications is to make a per-application configuration possible. Those details are beyond the scope of this discussion.

When the library creates a drawable for rendering, it tries to use in-hardware off-screen preserved buffers (called *preserved PBuffers*) and falls back to regular on-screen windows if the former are not available. PBuffers are preferred because they are not obscured by other windows. The use of PBuffers has to be specified upon context creation using `glXCreateContext` but the size of the PBuffer is specified later with a call to `glXCreatePBuffer`. This later call might fail because of insufficient resources. Since the library cannot obtain the size of the drawable preferred by the application until `glXMakeCurrent` is called, a situation is possible where a PBuffer context is created but is later unusable. This situation can be avoided

[3]A visual in X11 is an structure that describes the display attributes used by a given window, e.g. the color depth and color masks

by noticing that the value returned by `glXCreateContext` is a pointer to an opaque data type. This allows the wrapper function to generate its own value when the application calls `glXCreateContext` and defer the actual creation of the context to the moment when the application calls `glXMakeCurrent`.

When the application calls `glXSwapBuffers` buffers are swapped, the rendered image is read from the framebuffer and transmitted to the interaction server as an `XImage`. The normal image transport method, `XPutImage`, incurs a high overhead because the data is read from the client's memory space and is copied to the X server memory space which hinders performance significantly. If the MIT Shared Memory Extension is available, `XShmPutImage` is used instead. The XShm extension cannot be used if shared memory is not available, as it is the case when the client and the server run on different hosts. The case of clients not using double-buffered visuals can be handled heuristically, for example by trapping calls to `glFlush` (since the client wishes to ensure the OpenGL buffer is emptied and the screen is updated), with a rate-limiting algorithm (clients that issue too many synchronization calls, at rates in excess of hundreds per second, have been found). This brings another issue to attention: the implementation has to be very careful about spawning threads, installing signal handlers and manipulating file descriptors and streams in general since applications have been coded against a well defined standard, and the kind of side effects of the modified function calls might introduce have not been considered into the design of these applications.

8.5 Discussion

Reading the rendered images from the framebuffer and sending them over the network are expensive operations and reduce the maximum achievable framerates of the applications that use the library. Reading the framebuffer can be considered an atomic operation that cannot be optimized, therefore optimizations have to occur at the image transmission level.

The core X protocol does not include any form of image compression. This has been implemented via an extension oriented towards low bandwidth environments called LBX [FK93]. By using LBX on a local area network reduced network traffic was observed but little performance gains were obtained. Another way of obtaining stream compression on top of X is VNC [RSFWH98]. VNC is a free multi-platform client-server application for displaying and interacting with remote desktops. The protocol underlying VNC, the *Remote Framebuffer Protocol* (RFB) is only capable of sending rectangular framebuffer updates to the client. VNC provides a variety of custom tailored compression algorithms to increase the efficiency of image transmission. The Unix variant of the VNC server is based upon a standard X server (using a software renderer) which has been extended to communicate with RFB clients. Figure 8.3 shows how VNC can be integrated into this solution. Using this alternative, significant improvements in the interactivity of the applications have been observed.

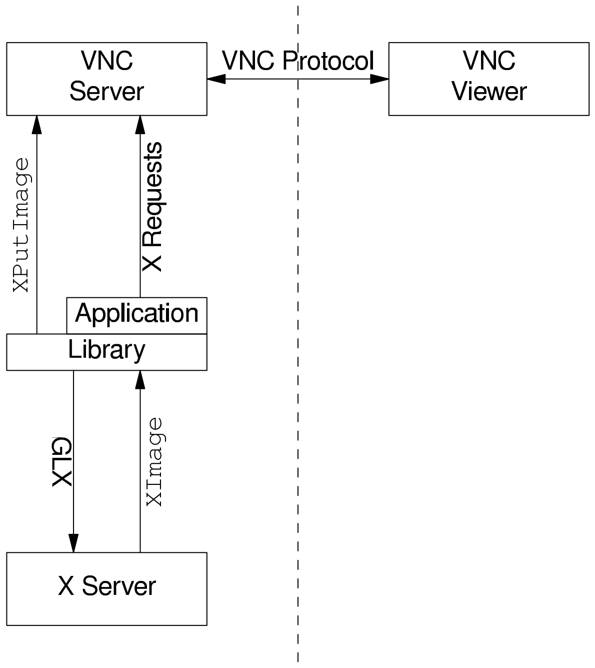


Figure 8.3: System architecture when VNC is included

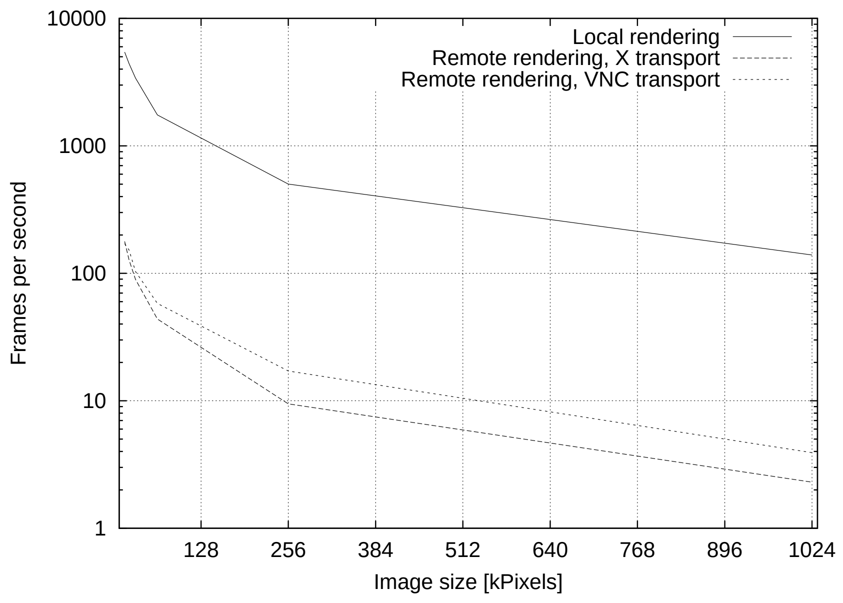
8.5.1 Results

The upper diagram in Figure 8.4 compares the library's performance with Brian Paul's GLX port of gears (`glxgears`) over a Fast Ethernet connection. In the test configuration, this application is mostly bounded by the speed at which the CPU can transfer commands to the graphics card. The results are shown as a function of window dimensions (all windows had the same width and height) for different transport mechanisms (X display redirection and VNC). The application's framerate in the local case is provided for comparison.

The lower diagram shows the results of the same measurements using a texture-based volume renderer [RSEB$^+$00] and a $256 \times 256 \times 128$ volume data set of an aneurysm. Again the frame rates are shown for different transport mechanisms and several window dimensions.

As can be seen framerates using our framework are quite acceptable for interface-bound applications (since they are going to be able to render at the maximum speed that the network connection can handle) but drop sharply when applied to applications that are bounded by the speed of the hardware renderer. Reading the framebuffer is a blocking operation: the graphics hardware can not start to work on the next frame until the reading operation is done, which means the application can not issue commands to render the next frame before the framebuffer has been duplicated in the host's main memory. It should be possible to mask the cost of sending the data to the display server by splitting this operation off to a second thread, but as mentioned before the implementation in the general case would not be trivial.

The system as described can be used with any OpenGL application, but for the specific case of volume rendering on PC clusters, there is an optimization opportunity. In figure 8.2,



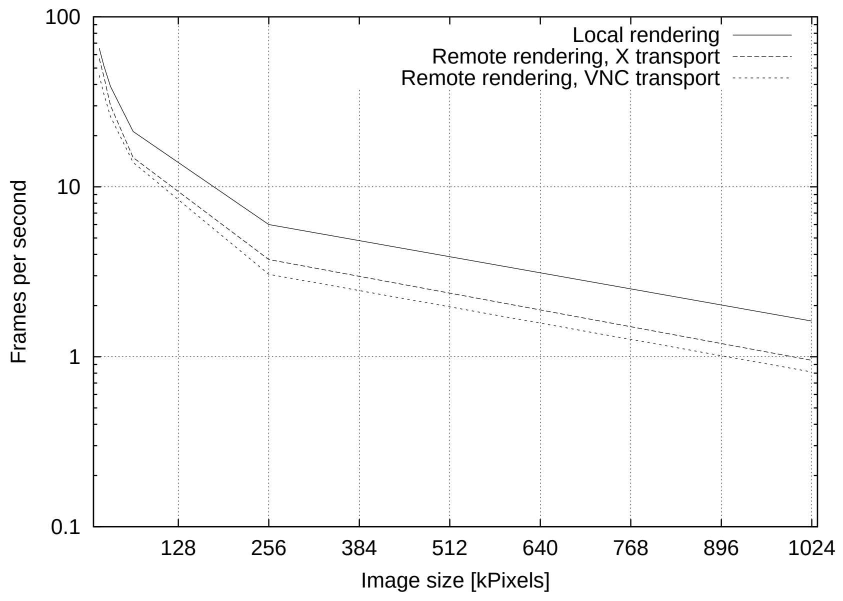


Figure 8.4: Measured framerates for "gears" (upper plot) and a volume renderer (lower one). The render server that was used for all measurements was equipped with a 900 MHz AMD Athlon CPU, 256 MB RAM and a GeForce 2 from NVIDIA. The interaction server was equipped with a 1.2 GHz AMD Athlon CPU, but since the interaction server was rarely operated at more than half capacity a much slower processor would also have sufficed. Both systems were running a Linux 2.4.10 Kernel and were connected using 100 Mbps Fast Ethernet.

steps 3 and 4 can be eliminated: there is no need to upload the final composited image to the framebuffer to read it back immediately. Looking at the problem more closely, it is not necessary to use OpenGL for the display at all. It suffices to create an `XWindow` and use `XPutImage` to upload the data to the X server. For this purpose, it is enough to follow the scheme previously described, and *preload* the call to `glDrawPixels` (and similar ones) to copy the data to a pixmap instead of uploading it to the framebuffer. Alternatively, a dedicated viewer application, based purely on Xlib, can be written, which allows for better event handling. Such an approach has been used during the development of this work with success.

Chapter 9

Results and future perspectives

9.1 Implementation environment

For the implementation of this system, an existing volume renderer was used as a starting point. It was modified only to conform to the requirements explained in chapter 4, namely calculating the opacity contributions according to equation (4.7) using one of the possibilities outlined in section 4.4. The test bed used for performance measurements was the already existing *Kepler cluster*. This cluster is installed at the University of Tübingen, Germany, and it is used mainly for astrophysical simulations. It consists of 128 SMP systems running Linux 2.4.19. 96 of the nodes have Intel Pentium III processors running at 650 MHz and 1 GB of main memory each. The other 32 nodes were added at a later point in time, and have AMD Athlon MP 2000+ processors and 2 GB of main memory each. The interconnection is a full-duplex Myrinet 1.2 Gb/s network. The high bandwidth, the fully switched nature and the low latency characteristics of the Myrinet technology was a driving point for using this cluster as a visualization platform. The SCore suite, discussed in chapter 6, provides the software and drivers for the communication layer.

When this work was started, four *NVIDIA GeForce 2* cards were installed on the cluster in order to assess the feasibility of the idea of using a PC cluster for distributed hardware-accelerated volume rendering. The first problem that was encountered was actually getting the graphics cards to work on the nodes. There were incompatibilities between the graphics cards, their drivers, the Myrinet cards and the chipset used on the nodes. Since the original installation, the situation has improved greatly, but the lesson learned was to stick to well known, widely available chipsets. According to experience, chipsets from Intel, like the 440BX and its later siblings, represent a good choice for this task, both in terms of stability, support and performance. The newer nodes using Athlon processors have an AMD-760 chipset and *NVIDIA GeForce Ti4600* graphics cards. After some initial trouble, these have shown to be stable. More recent chipsets from Intel, like the E7205 and later models have a better feature set, with support for newer standard interfaces like AGP 3.0; this chipset has been found to work reliably with *NVIDIA GeForce Ti4600* parts, too.

Most of the problems mentioned were solved by software reconfiguration or upgrades. This highlights one of the downsides of using commodity parts for this purpose, namely, the final system has not been put together by a single company, as it would be the case with e.g. an InfiniteReality 4, but it has been assembled “in place”. This means some of the parts are going to be used in environments or configurations that the vendor has not tested. For example, when installing a newly acquired set of Pentium 4 workstations (Intel E7205 chipset) with NVIDIA Ti4200 cards it was found that they could not operate in AGP mode. After some investigation the problem was traced to a bug in the AGP driver of the Linux 2.4 kernel. The bug was triggered only on systems with 4 GB of main memory, a configuration that had probably not been tested by the driver author because it was not available to him[1].

Another aspect that causes some grief is the general support for the Linux kernel by hardware vendors, and in particular by graphics hardware vendors. The lastest generation of graphics hardware is not supported by drivers for which source code is freely available. The reason behind this is that vendors are not willing to provide enough technical documentation about the programming interfaces for their hardware in an open fashion. ATI provides partial documentation for their hardware up to the second generation of the Radeon chip under a non-disclosure agreement that allows to freely release the resulting source, but this has met with limited success. Both NVIDIA and ATI do release binary-only drivers that support their latest generation hardware. NVIDIA has established, after some initial trouble, a good record of supporting the Linux kernel this way, while the response from ATI still generates mixed reactions among users. Even if this approach satisfies a large fraction of the user-base, it still takes away some of the benefits associated with the otherwise normal development strategy of the Linux kernel and the surrounding operating system. In particular, it makes it harder or impossible to fix even trivial bugs and excludes altogether the possibility of porting the drivers to new architectures unless the vendor itself has an interest on the task (in this regard NVIDIA does provide support for Linux on the Intel x86, Intel Itanium and AMD64 architectures, as well as FreeBSD on the Intel x86 architecture).

Nevertheless, the advantages outlined in chapter 1 (better performance, faster development cycle, easier to replace parts, easier upgrades) have been found to hold true. It is the opinion of the author that these advantages still weight out the disadvantages mentioned above.

9.2 Performance measurements

The visible “female dataset”, part of the “visible human project”, was used for benchmarking the application (figure A). The visible human project, carried out by the United States National Library of Medicine, has the goal of creating a complete, anatomically detailed, three-dimensional representations of the normal male and female human bodies. Acquisition of transverse CT, MR and cryosection images of representative male and female cadavers has been completed. The male was sectioned at one millimeter intervals, the female at one-third of a millimeter intervals. The dataset used consists of a volume $512 \times 512 \times 1734$ in size, with 8-bit density data at each voxel. This amounts to 443 MB of texture data. This has been split to 16 bricks, each with a

[1]Even if the nature and reasons why the bug existed in the fi rst place are out of the scope of this discussion, it is necessary to remark that is was possible to fi nd and fi x this bug because of the availability of source code for the AGP driver.

dimension of $256 \times 256 \times 512$ for a $2 \times 2 \times 4$ configuration. Each node is able to render its brick in a buffer with 1024×1024 pixels with a framerate of circa 10 Hz (100 ms per frame). At this resolution, depending on the on-screen orientation of the volume, instead of having multiple neighbouring voxels contribute to a single pixel, single voxels contribute to only a few pixels of the final image, thus avoiding masking features because of aliasing effects. Reading the data out of the framebuffer takes 19 ms. Compositing the 16 images takes circa 5 ms on average, with an additional overhead of 15 ms for transmitting the data over the network, and another 15 ms for collecting the composited data on a single node. On average the application updates the display with a rate of circa 6 Hz. In this case the bottleneck is not network but the rendering of the data itself. A single node can render this dataset with an update rate of circa 4 *seconds* per frame (0.25 Hz), giving a speed up of 24 when compared to the parallel case. The super-linear speedup comes from the fact that the serial application needs to update the textures 8 times for each frame that is rendered while the parallel version does not perform texture updates at all.

A second dataset was used for performance measurements, the so-called "jet stream dataset" (figure A). This is a time-dependent dataset which comes from a turbulent flow simulation. It consists of 89 time steps, each of which corresponds to a volume of $256 \times 256 \times 256$ voxels. This was split in $128 \times 128 \times 64$ bricks (1 MB of texture data), for a $2 \times 2 \times 4$ configuration. In this case each node renders its brick in about 37 ms, and the other times are similar to those of the visible female dataset. The difference here is that textures are swapped in and out of texture memory as time goes by in the visualization. Each time step can be uploaded to the graphics card in 4 ms. When reproducing the animation at a rate of 1 time step per second, this means there is an overhead of 4 ms once each second. This gives a mean refresh rate of 8 Hz. A single node can render this dataset at a rate of circa 1 Hz.

Looking at table 9.1, it is easy to see why the speedup in the case of the jet stream dataset is lower than in the visible female case: the compositing step starts to play a more significant role: the rendering of the data amounts to only 31% of the total time, while compositing (reading back data and then sending it over the network, blending and collecting it) amounts to 69 % of the total time. Ignoring optimizations, the compositing stage runs in constant time of circa 80 ms, which gives an upper limit for the framerate of 12 Hz.

Transporting the images to the user has not been taken into account in this discussion. The most simple solution is connecting a display device to one of the nodes in the cluster, and work *in situ*. As explained in chapter 8, this poses practicability problems. Nevertheless, there are scenarios where this is desirable, for example, when using this kind of application to drive display walls. As discussed in the same chapter, it is possible to leverage commodity software such as VNC for this purpose and provide access over public networks using infrastructure such as the world wide web. In that case the refresh rates for user interaction are decoupled from the refresh rates the application can deliver.

9.3 Conclusions

During the development of this work, many technologies evolved driven by mass market dynamics: as the rasterization performance of graphics cards increases, it becomes more efficient

Operation	Visible female	Jet stream
Render	100	37
Readback	19	14
Transport	15	15
Blend	5	5
Collect	15	15
Total	154	86

Table 9.1: Time to render a single frame of the visible female and jet stream datasets. The differences in rendering time are due to the size of the bricks rendered on each node. The differences in the readback times are due to the fact that only drawn portions of the image are read. This data does not take into account the synchronization overhead, which adds circa 10 ms to the total times.

to allocate larger bricks as workload for each node on the cluster reducing the number of nodes that have to participate in the rendering of a single dataset in order to achieve a predetermined framerate. Second, as the on-board memory on the graphics cards increases, it becomes possible to render larger datasets using the same number of nodes. Third, as the available CPU power increases, the compositing bottleneck lessens. As new network technologies emerge and become more widespread, this bottleneck is further reduced. As discussed above, the network performance is at this moment one of the limiting factors in this approach, yet this is the component that most slowly evolves. The reason for that is clear: data warehouses are the largest customers of high-bandwidth networks. Because of the hardware volume managed at these installations and the high cost of downtime, they have a slow hardware upgrade cycle, which in turn makes the market less profitable when compared to general purpose processors and to a lesser extent to dedicated graphics processors. A second factor is that this customers have traditionally favored the use of proprietary hardware architectures, precisely because of the availability of better inter-processor technologies. As the price/performance gap between proprietary supercomputers and COTS clusters widens, the need for higher bandwidth networks for the later becomes more pressing. The effect of this is the appearance in recent years of technologies such as InfiniBand, which currently delivers point-to-point transfer rates of 10 Gb/s, and promises the availability of 30 Gb/s general purpose networks in the near future. These represent an eight-fold and 25-fold performance increases with respect to current 1.2+1.2 Gb/s Myrinet technologies, and still compare favorably with respect to dual 4+4 Gb/s Myrinet technologies. The development of these technologies is in part slowed down by the fact that the fastest buses widely available are not able to cope with much higher transfer speeds. For example, 64-bit 133 MHz PCI-X delivers 8.5 Gb/s of bandwidth. PCI-X 266 could extend this to 17 Gb/s, and PCI-X 533 could deliver another two-fold increase. A different approach is PCI Express, which uses a serial interface instead of the traditional parallel one. This enables the existence of high-performance point-to-point connections between devices, which in turn provides deterministic low-latency.

The topic of hardware-accelerated direct volume rendering techniques is vast and active. In this work only a basic introduction is provided in chapter 2, and the focus of the work lies at direct volume rendering in uniform meshes. The ideas presented later on are nevertheless not constrained in any way to any specific technique. For example, the results are extensible to *non-uniform* meshes by fitting the non-uniform mesh into a larger uniform mesh and performing a remeshing along the borders of the subvolumes. As explained in chapter 4 the only requirement is to satisfy the associativity of the operation by using the appropriate blending equation, which is in turn guaranteed by using α-premultiplication. In the same chapter several alternatives are presented in order to implement this operation with old and new generations of commodity graphics hardware. In chapter 5, it has been shown that the option of performing the compositing of partial results of the subvolumes using the graphics hardware for the task does not yield a performance gain on modern architectures. Even if the compositing operation itself can be executed faster on the graphics card, uploading and downloading the data is a slow process. In this case the bus between the CPU and the dedicated graphics memory represents a bottleneck. Again, newer technologies such as PCI Express might change this situation. In order for this path to be faster than a software-based one, the interconnection bus needs to offer a bandwidth in excess of 5.3 Gb/s. This is within the peak bandwidth offered by AGP 4x, but actual measurements do not support this statement. In order to achieve interactive refresh rates, the compositing of the partial results of the rendering process has been optimized using assembler code. As shown in chapter 5 this implementation delivers compositing rates that make the interactive visualization possible.

In this work solutions for another three different problems were developed:

- Remote accessing of visualization facilities. This has been implemented using a big-server thin-client model and is application independent. This allows to have minimal or no extra requirements on the client side (e.g. a VNC client or an X11 server) and access these facilities even from devices with comparatively small CPUs like PDAs.

- Debugging of OpenGL applications and management of OpenGL extensions. These have been developed in such a manner that it is easy to keep them up to date with current standards.

Appendix A

Color plates

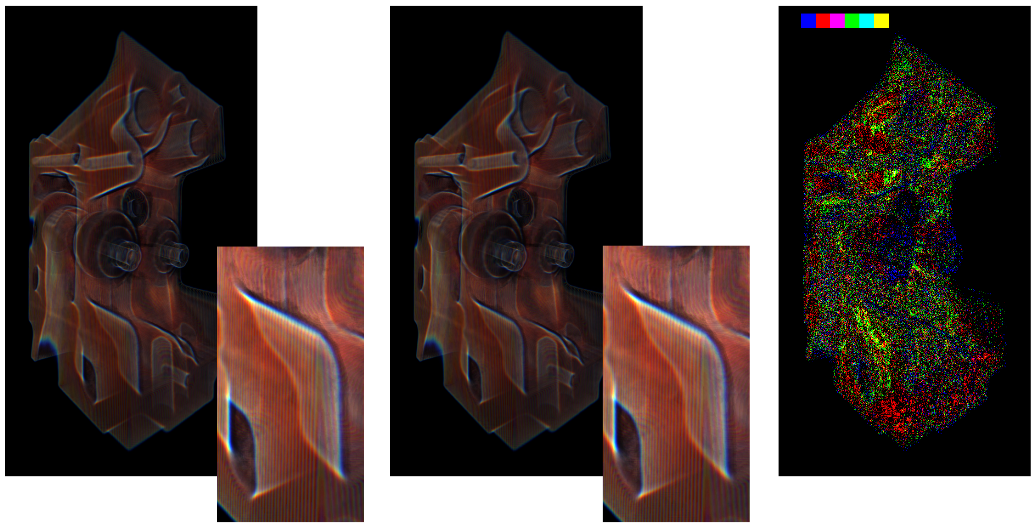


Figure A.1: Color version of figure 4.4 on page 60

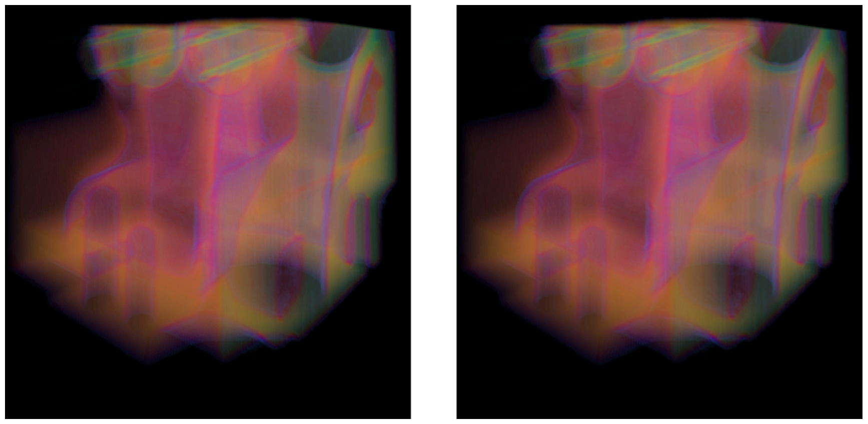


Figure A.2: Color version of figure 5.1 on page 64

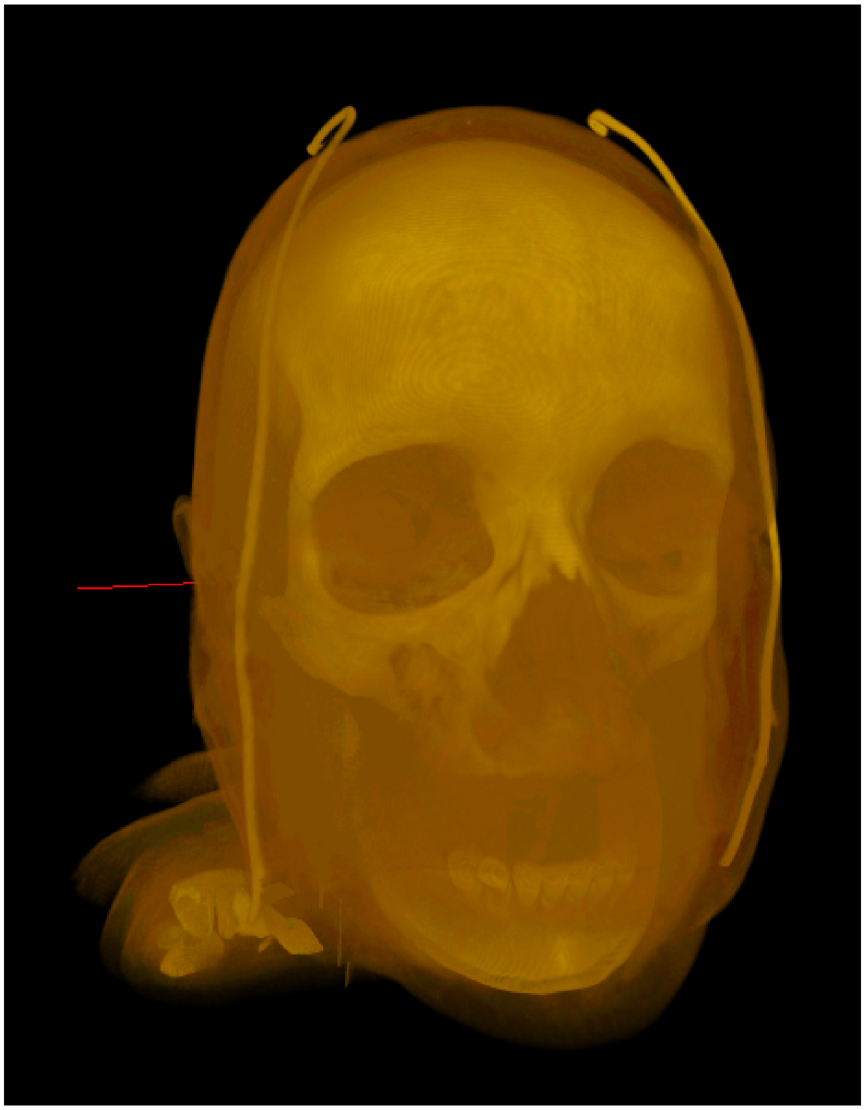


Figure A.3: Subset of the visible female dataset

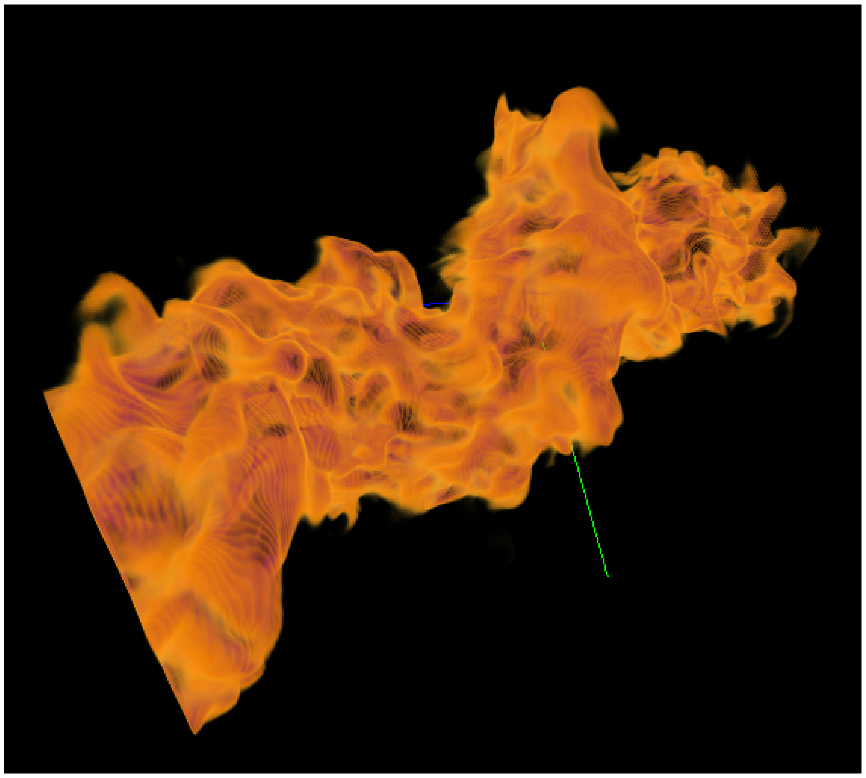


Figure A.4: Jet stream dataset

Bibliography

[Adv02] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. Publication 22007, revision K.

[Adv03a] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual volume 3: General purpose and system instructions*, September 2003. Publication 24594, revision 3.09.

[Adv03b] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual volume 4: 128-bit media instructions*, September 2003. Publication 26568, revision 3.05.

[Ake93] Kurt Akeley. RealityEngine graphics. In *Proceedings of SIGGRAPH 1993*, pages 109–116. ACM SIGGRAPH, August 1993.

[Bet00] Wes Bethel. Visualizaton dot com. *Computer Graphics and Applications*, 20(3):17–20, May 2000.

[Bli94] James F. Blinn. Jim blinn's corner: Compositing part 1: Theory. *IEEE Computer Graphics and Applications*, pages 83–87, September 1994.

[BSS$^+$95] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Us ing Texture Mapping Hardware. In *Symposium on Volume Visualization*, pages 91–98, October 1994.

[Coh03] Ariel Cohen. A performance analysis of 4X InfiniBand data transfer operations. In *International Parallel and Distributed Processing Symposium*, page 202b. IEEE, April 2003.

[Dog95] M.C. Doggett. An array based design for real–time volume rendering. In *10th Eurographics Workshop Graphics Hardware*, page 93101, August 1995.

[EE99] K. Engel and T. Ertl. Texture-based Volume Visualization for Multiple Users on the World Wide Web. In Gervautz, M. and Hildebrand, A. and Schmalstieg, D., editor, *Virtual Environments '99*, pages 115–124. Eurographics, Springer, 1999.

[EHT$^+$00] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications. In *Proceedings of the conference on Visualization '00*, pages 449–452. IEEE, 2000.

[EIH00] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A fully scalable graphics architecture. In *Proceedings of SIGGRAPH 2000*, pages 443–454. ACM SIGGRAPH, July 2000.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.

[Elv92] T. Todd Elvins. Volume rendering on a distributed memory parallel computer. In Arie E. Kaufman and Gregory M. Nielson, editors, *Proceedings of the conference on Visualization '92*, pages 93–98, 1992.

[ESE00] K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 167–177,291, May 2000.

[FK93] Jim Fulton and Chris Kent Kantarjiev. An update on low bandwidth X (LBX). *The X Resource*, (5):251–266, January 1993.

[FMS02] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-time halftoning: A primitive for non-photorealistic shading. In *13th Eurographics Workshop on Rendering '02*, 2002.

[Fra97] Michael Franz. Dynamic linking of software components. *IEEE Computer*, pages 74–81, March 1997.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and practice*. Addison-Wesley Systems Programming Series. Addison-Wesley Publishing Company, 2nd ed. in c. edition, 1996.

[GBR79] Alan P. Lightman George B. Rybicki. *Radiative Processes in Astrophysics*. Wiley-Interscience, New York, 1979.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[GLSD96] W. Gropp, E. Lusk, A. Skjellum, and N. Doss. MPICH: A high-performance, portable implementation for the mpi message-passing interface. *Parallel Computing*, 22:789–828, 1996.

[GPR$^+$95] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *9th Eurographics Hardware Workshop*, volume 19, pages 705–710, 1995.

[GWGS02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Proceedings of the conference on Visualization '02*, pages 53–59, October 2002.

[HEB$^+$01] Greg Humphreys, Matthew Eldridge, Ian Buck, Matthew Everett, Gordon Stoll, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. In *Proceedings of SIGGRAPH 2001*, pages 129–140. ACM SIGGRAPH, 2001.

[HHN$^+$02] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters. In *Proceedings of SIGGRAPH 2002*, pages 693–702. ACM SIGGRAPH, 2002.

[HL97] Steve Huss-Lederman. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi2-report.html, July 1997.

[HM99] A. Heirich and L. Moll. Scalable distributed visualization using off-the-shelf-components. In *IEEE Parallel Visualization and Graphics Symposium 1999*, pages 55–60, 1999.

[HO91] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software, Practice and Experience*, 21(4):375–390, 1991.

[Ish01] Yutaka Ishikawa. Personal communication, 2001.

[KBH95] Mark J. Kilgard, David Blythe, and Deanna Hohn. System support for OpenGL direct rendering. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 116–127. Canadian Human-Computer Communications Society, 1995.

[KHKS94] Arie Kaufman, Karl Heiy Höhne, Wolfgang Krügrer, and Peter Schröder. Research Issues in Volume Visualization. *Computer Graphics and Applications*, 14(2):63–67, March 1994.

[Kil96] Mark J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley, 1996.

[Kil02] Mark J. Kilgard. OpenGL extension 191: `NV_register_combiners` specification (version 1.5), June 2002. http://oss.sgi.com/projects/ogl-sample/registry/.

[Kil03] Mark J. Kilgard. OpenGL extension 230: `NV_texture_shader` specification, July 2003. http://oss.sgi.com/projects/ogl-sample/registry/.

[KKH01] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of the conference on Visualization '01*, pages 255–262. IEEE, October 2001.

[Kni95] G. Knittel. A scalable architecture for volume rendering. *Computer and Graphics*, 19(5):653–665, 1995.

[KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of the conference on Visualization '02*, pages 255–262. IEEE, October 2002.

[KPI$^+$03] Joe Kniss, Simon Premoze, Milan Ikits, Aaron Lefohn, Charles Hansen, and Emil Praun. Gaussian transfer functions for multi-field volume visualization. In *Proceedings of the conference on Visualization '03*, pages 497–504. IEEE, October 2003.

[Kra03] Martin Kraus. *Direct volume visualization of geometrically unpleasant meshes*. PhD thesis, Universität Stuttgart, 2003.

[Lev88] M. Levoy. Display of surfaces from volume data. *Computer Graphics and Applications*, 8(3):29–37, May 1988.

[LHJ99] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of the conference on Visualization '99*, pages 355–361, October 1999.

[Lip03] Benj Lipchak. OpenGL ARB extension 27: `ARB_fragment_progarm` specification (revision 26), August 2003. http://oss.sgi.com/projects/ogl-sample/registry/.

[Lu95] H. Lu. ELF: From the programmer's perspective, 1995. ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ ELF.doc.tar.gz.

[LWMT97] Peggy Li, Scott Whitman, Roberto Mendoza, and James Tsiao. ParVox – A Parallel Splatting Volume Rendering System for Distributed Visualization. In *Proceedings of 1997 Symposium on Parallel Rendering*, pages 7–14. ACM SIGGRAPH, IEEE Computer Sociery Press, 1997.

[MC00] Kwan-Liu Ma and David M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Supercomputing*, 2000.

[MCEF94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *Computer Graphics and Applications: Special Issue on Rendering*, 14(4):23–32, July 1994.

[MEP92] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. In *Proceedings of SIGGRAPH 1992*, pages 231–240. ACM SIGGRAPH, July 1992.

[MHE01] M. Magallon, M. Hopf, and T. Ertl. Parallel Volume Rendering using PC Graphics Hardware. In *Pacific Graphics Conference 2001*, 2001.

[MKS98] M. Meißner, U. Kanus, and W. Straßer. VIZARD II: A PCI-card for real-time volume rendering. In *SIGGRAPH/Eurographics Workshop Graphics Hardware*, pages 61–67, August 1998.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MPHK94] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, pages 59–68, July 1994.

[Neu93] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *IEEE/SIGGRAPH Parallel Rendering Symposium*, 1993.

[Nye95] Adrian Nye, editor. *Volume 0: X Protocol Reference Manual*. X Window System Series. O'Reilly & Associates, 4th edition, January 1995.

[Ora03] Orad. Dvg. Website, http://www.orad.tv/, March 2003.

[PD84] T. Porter and T. Duff. Compositing digital images. In *Proceedings of SIGGRAPH 1984*, pages 253–259. ACM SIGGRAPH, 1984.

[PHK$^+$99] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro realtime ray-casting system. In *Proceedings of SIGGRAPH 1999*, pages 251–260. ACM SIGGRAPH, 1999.

[PS01] Boris V. Protopopov and Anthony Skjellum. A multithreaded message passing interface (mpi) architecture: Performance and program issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, 2001.

[PTT97] M.E. Palmer, S. Taylor, and B. Totty. Exploiting deep parallel memory hierarchies for ray casting volume rendering. In *Parallel Rendering Symposium '97*, pages 15–22, October 1997.

[RH94] John Rolhf and James Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings of SIGGRAPH 1994*, pages 381–394. ACM SIGGRAPH, July 1994.

[RKE00] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the conference on Visualization '00*, pages 109–116, 2000.

[RPSC99] Harvey Ray, Hanspeter Pfister, Deborah Silver, and Todd A. Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[RSEB$^+$00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118,147. Addison-Wesley Publishing Company, Inc., 2000.

[RSFWH98] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[SA03] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 1.5), 2003. http://www.opengl.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. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, 2001.

[SFLS00] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 97–108. Addison-Wesley Publishing Company, Inc., 2000.

[Sil01] Silicon Graphics, Inc. Vizserver, November 2001. http://www.sgi.com/software/vizserver/.

[SME02] Simon Stegmaier, Marcelo Magallón, and Thomas Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '02*, 2002.

[SOHL$^+$98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1998.

[SW01] J. Schneider and R. Westermann. Towards real-time visual simulation of water surfaces. In *Vision, Modelling and Visualization '01*, 2001.

[VBRR02] Gerritt Voß, Johannes Behr, Dirk Reiners, and Martin Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *Parallel Graphics and Visualization 2002*, pages 33–38, September 2002.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH 1998*, volume 32, pages 169–179. ACM SIGGRAPH, July 1998.

[WGTG96] J. Wilhelms, A. V. Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Proceedings of the conference on Visualization '96*, pages 57–64, October 1996.

[WH94] Craig M. Wittenbrink and M. Harrington. A scalable MIMD volume rendering algorithm. In *Eighth International Parallel Processing Symposium*, pages 916–920, April 1994.

[WL98] Paula Womack and Jon Leech. OpenGL graphics with the X Window System, version 1.3, 1998. http://www.opengl.org/.

[WMG98] Craig Wittenbrink, Tom Malzbender, and Michael E. Goss. Opacity-weighted color interpolation for volume sampling. In *Proceedings of the 1998 Symposium on Volume Visualization*, pages 135–142. ACM, October 1998.

[WWH$^+$00] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3D textures. In *Volume Visualization and Graphics Symposium '99*, pages 7–13. IEEE, 2000.