Parallel Volume Rendering Using PC Graphics Hardware

Marcelo Magallón   Matthias Hopf   Thomas Ertl

Visualization and Interactive Systems Group, IfI, University of Stuttgart
Breitwiesenstr. 20-22, D-70656 Stuttgart, Germany
(magallon, hopf, ertl)@informatik.uni-stuttgart.de

Abstract

This paper describes an architecture that enables the use of commodity off the shelf graphics hardware along with high speed network devices for distributed volume rendering. Several PCs drive a number of graphic accelerator boards using an OpenGL interface. The frame buffers of the cards are read back and blended together for final presentation on a single PC working as front end. We explain how the attainable frame rates are limited by the transfer speeds over the network as well as the overhead implied by having to blend several images together limits . An initial implementation using four graphic cards achieves frame rates similar to those of high performance visualization systems.

1 Introduction

1.1 Motivation

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 [1]. 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.

In recent years the market for high end PC graphics accelerator boards has grown explosively, mostly driven by the computer gaming and entertainment industry. The capabilities and performance of this kind of graphics hardware have increased exponentially (and faster than Moore's law), bringing it to a position where it is a viable option for research and design of new graphics algorithms and applications, or adaptation of existing algorithms in order to exploit new capabilities. As others have already assessed [8], interactive volume rendering has become an invaluable technique to visualize 3D scalar data for a variety of applications in engineering, science and medicine. By exploiting the capabilities of current PC graphics hardware, direct volume rendering rates of about 30 frames per second have been achieved for $128 \times 128 \times 64$ datasets and even higher rates for isosurface extraction. These frame rates drop below 5 for a 256^3 dataset, the limiting factor being the available on-board memory for texture storage as well as the limited memory bandwidth. It has already been shown that this effect can be counteracted by using multiple rasterizers on shared memory architectures (e.g. Li et al [6]). 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.

In this paper we explore an approach to take advantage of the rasterization speed of COTS graphics hardware. We are interested in the specific case of volume rendering of *large datasets* (256^3 and larger) by means of off-screen remote rendering. Our initial approach uses a *sort-last* strategy (after the classification developed by Molnar et al [7]) and is better suited for *object based partitioning* of the scene to be rendered, but is also applicable in the case of *image based partitioning*.

1.2 Previous work

The idea of hardware accelerated parallel rendering using standard PCs and graphics hardware has already been considered by others:

- Samanta et al [9] implemented a hybrid sort-first/sort-last algorithm focusing on parallel *polygon* rendering. They implement both screen and object based partitioning, which helps to reduce the communication overhead implied by a pure sort-last approach.

- Humphreys et al [3, 4] have developed a drop-in OpenGL DLL replacement which uses a sort-first technique and focuses on *scalable displays* and leans towards an explicit parallelization for the OpenGL API.

In contrast to Samanta et al, we are (initially) focusing on *volume rendering* and not *polygon rendering*. In contrast to Humphreys et al, we pursue visualization of large datasets *on a single display*. The goal is to keep constant frame rates as a function of dataset size without resorting to a simplification of the dataset.

This paper is organized as follows: first we provide an overview of the different factors that limit the rendering speed when using PC class hardware. Next, we explain our partitioning strategy as well as the blending algorithms we have evaluated. Last, we report our results using the system on a small scale cluster.

2 Volume Rendering

In direct visualization of volumetric data, each pixel is computed by tracing rays through the volume, evaluating the volume rendering integral

$$I(s) = I_0\, e^{-\tau(s_0,s)} + \int_{s_0}^{s} q\, e^{-\tau(s',s)}\, ds' \,, \qquad (1)$$

which represents the intensity on a ray parametrized by s which enters the volume at s_0. In the emission-absorption model, q represents the emissive color per unit lenght and τ the optical depth of the absorbing material. Both quantities are derived from the original scalar data values by means of transfer functions. The first term represents the light coming from the background attenuated by the entire volume, and the second term is the contribution of each light emitting element in the volume multiplied by the transparency between s_0 and s. This can be discretisized and implemented by drawing the pixels in front-to-back or back-to-front order, using the standard over operator for blending. The process is accelerated by rendering all the rays simultaneously, thus creating the standard (implicitly parallel) texture based volume rendering approach [2, 10].

Truly interactive volume visualization of large data sets using PC graphics hardware is currently only possible by means of a 2D texture stack, as Rezk-Salama et al [8] have presented. In this approach polygons aligned to the volume are drawn and blended back-to-front into the frame buffer. The polygons are textured with a set of 2D textures which represent the volumetric data. This approximates an implicit shear-warp decomposition of the viewing matrix as proposed by Lacroute and Levoy [5].

As this technique works well only as long as adjacent polygons have a large overlapping region, the stack of textures has to be switched when the angle of the viewing direction with respect to the polygon stack direction exceeds 45 degrees (Figure 1). Because of this the texture stack has to be replicated three times, once for each axis of the local coordinate system.

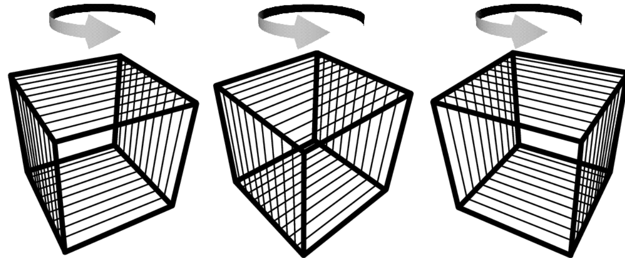

Figure 1. Volume rendering using volume aligned polygons

There are two major factors that limit performance using this technique: the speed of the pixel pipeline and the amount of available texture memory. On the one hand, for each frame several large polygons are rendered one after the other, with a different 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 regular 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 dataset 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 higher available memory bandwidth of the graphics card. The impact on the attainable frame rate is much more noticeable than that of the limited pixel pipeline speed. It is necessary to note that not all three texture stacks have to fit into graphics memory at the same time, since a small lag when switching the texture stack every once in a while is not as detrimental as a low frame rate.

Since these limits depend on the size of the dataset being rendered, both of them can be overcomed using multiple renderers in parallel, where each of them works on a subset of the data. It can be seen from (1) that this process is explicitly parallelizable: the integral on the r.h.s. 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.

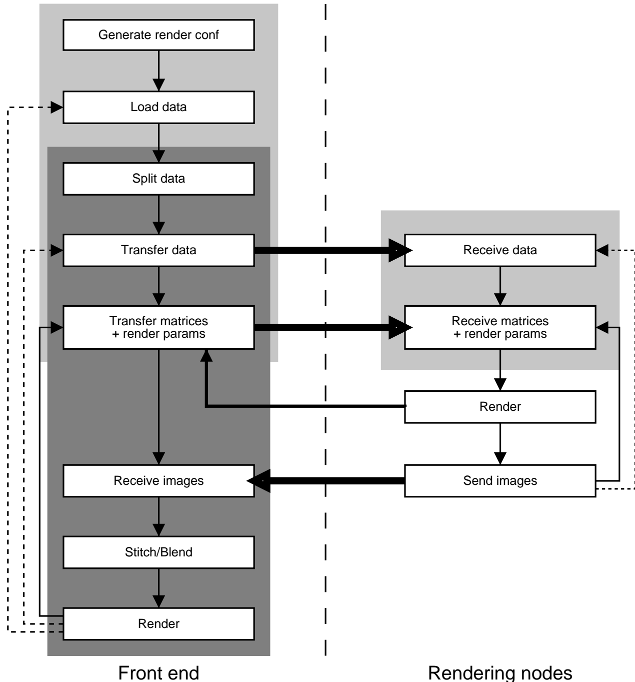

Figure 2. General program flow for the controlling and rendering processes.

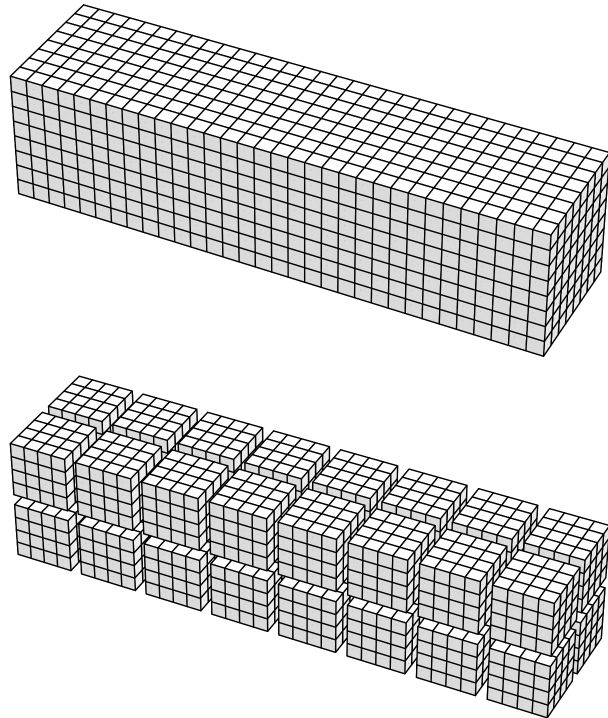

Figure 3. Above: a volume dataset where one of the directions is significantly larger than the other two. Below: the corresponding partitioning ($2 \times 2 \times 8$) using 32 nodes.

3 Parallelization

3.1 Partitioning strategy

Figure 2 depicts the general pipeline used to parallelize the rendering process. Since the aimed dataset size is 64 MB and larger (i.e. transfer times of 0.5s and longer for the MPI/Myrinet implementation being used), a *static partitioning* in object space has been opted for, which implies a *sort-last* algorithm. A *rendering configuration* is first created using the number of nodes as the input parameter. Given n_r rendering nodes, the volume is partitioned in $x \times y \times z$ ricks, 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). 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 subvolume to the frame buffer using a preselected volume rendering algorithm.

3.2 Blending

As soon as the subvolume has been rendered, the image is read from the frame buffer and transmitted to be blended with other subimages. We have evaluated two blending algorithms. The first one is theoretically optimal: given N different nodes, each of them with one out of N images to be blended, each image is split into N subimages which are sent to the $N-1$ neighboring nodes where they are blended and then sent to a previously defined node. If t is the cost of transmitting one image over the network and b the cost of blending two images together, this algorithm has a total cost of $(2t+b)(N-1)/N$ (for a full-duplex network). The drawback with this algorithm is that it requires a total of $N(N-1)$ transmissions over the network. For the second algorithm the N nodes are split into two groups, the first one receives the data and the second one sends it, that is, $N/2$ images are blended pairwise with another $N/2$ images.

The resulting set is split again and the process is repeated until all the images have converged into one node. This has a total cost of $(t+b)\log(N)$ and there are a total of $N-1$ network connections to be established. The reference solution for this problem is for $N-1$ nodes to send their images to one node, where they are blended. This has a total cost of $(t+b)(N-1)$ with $N-1$ required connections. It can be immediately seen that for any t and b and any N larger than 2 the two presented algorithms are faster than the reference. In general the second algorithm has a higher cost than the first one.

Before blending the images, they are depth sorted (either back to front or front to back) using the bounding box of the data subsets they represent. The implicit parallelization of the graphics hardware is not exploited because of the overhead incurred by writing to *and* reading from the color or depth buffer of the card. On current graphics hardware, this overhead is not to be neglected, as Table 1 shows. The large differences between the times for reading from and writing to the depth buffer are with certainty due to non-optimized paths in the card's drivers.

In the case of the second algorithm, the *last* blending step *could be* done directly in hardware. This is advantageous because the delay caused by having to wait for both images is compensated by being able to start the blending early.

4 Results and discussion

The system has been implemented on the *Kepler PC-Cluster*[1], which operates at the University of Tübingen. This cluster consists of 96 computing nodes communicating over a Myrinet 1.28 + 1.28 Gb/s interconnect. Each node has two Pentium III 650 MHz processors and 1 GB RAM. Additionally, there are two front-end machines, with similar specifications. For testing purposes, four GeForce2 GTS-based cards with 64 MB of dedicated memory are installed on the nodes, with an additional 32 similar cards waiting for their approval.

The blending of images has proven to be a bottleneck for large images (1024×1024 pixels). This problem is computational-bound and scales linearly with CPU and memory speed. Using a C implementation, blending four images (BGRA format, one unsigned byte per channel) of this size using floating point arithmetic takes 2208 ± 3ms on the PCs currently running in the cluster. If integer arithmetic is used instead, this time is reduced to 234 ± 2ms. This alone imposes an upper limit of $\approx$ 4fps on the whole process. Using a MMX implementation we can reduce this time to about 95.1 ± 0.1ms ($\approx$ 10fps). Using two CPUs per node, this is reduced even further, to 50.4 ± 0.1ms ($\approx$ 19fps). Given this figure, transferring the blending process to the graphics card is not feasible, since each node that receives images to be blended would have to download multiple images to the card, let the card perform the actual blending, and then upload the final composited image to transmit it again. For four images, such a process imposes an upper limit of $\approx$ 9fps, with the limiting factor being the available memory bandwidth and not the speed of the GPU. Currently available CPUs with clock speeds of 1.33 GHz would nearly halve this time (double the maximum frame rate). The increase in memory bandwidth required to reach and surpass these values is not likely to happen in the near future.

[1] http://kepler.sfb382-zdv.uni-tuebingen.de/

The second bottleneck is to be expected: since multiple images are being sent almost at the same time over the network resource contention becomes noticeable. Figure 4 shows the total transfer time and the attained transfer speed for N images between N nodes for each of the algorithms explained in section 3.2, labeled *opt* (for *optimal*) and *log* (for *logarithmic*). It can be seen that for four nodes and large images the *logarithmic* algorithm has slightly shorter transfer times than the optimal one. For four 1024×1024 BGRA images, the total time to transfer them from the original four nodes to the final one using the logarithmic algorithm is 69.6 ± 0.1ms and 80.3 ± 0.1ms for the optimal algorithm.

When combining these two results, transferring and blending four images from their original nodes to their final destination takes 105.3 ± 0.1ms using the logarithmic algorithm and 93.3 ± 0.1ms using the optimal one. This imposes an upper limit of $\approx$ 10fps for the whole rendering process. It is noteworthy that this limit is largely dominated by the network speed, but it can be raised a little by using faster CPUs for the blending part. Also noteworthy is the fact that the actual rendering time per frame is almost completely contained within the time required to transfer the data over the network, which allows for the implementation of a pipeline, where images are rendered on the card while previously rendered images are being blended.

Using this method, we can render a 256^3 dataset using a viewport of size 1024×1024 in $\approx$ 240ms per frame, which corresponds to $\approx$ 4.2fps. This result compares well with the one reported by Rezk-Salama et al [8], who have obtained 4.3fps using a 600×600 viewport. Using a similarly sized viewport we can achieve frame rates of $\approx$ 12fps, which compares favorably with respect to the frame rates reported by the same authors when using a high performance visualization system. At the moment only a rudimentary front-end has been implemented, which does not exploit the pipelining possibility mentioned before, which would provide a small performance increase.

We have shown that using commodity off the shelf graphics hardware in combination with high speed network devices by means of off-screen remote rendering makes the direct volume visualization of large datasets is not only pos-

	color buffer								depth buffer							
	rgb		rgba		bgra		abgr		float		int		uint		ushort	
draw*	16	49	24	33	35	22	25	31	14	56	0.60s	1.3s	11	71	0.60s	1.3s
read*	23	34	21	37	29	27	27	29	17	46	0.90s	0.87s	27	29	27	29
draw†	42	19	41	19	49	16	43	18	21	37	0.50s	1.6s	17	46	0.50s	1.6s
read†	35	22	35	22	37	21	36	22	23	34	1.2s	0.65s	37	21	41	19
draw‡	20	39	25	31	40	20	31	25	16	49	0.70s	1.1s	12	65	—	—
read‡	26	30	26	30	31	25	33	24	17	46	1.1s	0.71s	25	31	—	—

*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

Table 1. Read and write times for color and depth buffers without blending or depth test enabled. The number on the left is the transfer rate (in MPixels/s) and the number on the right is time required to read a 1024×768 region (in ms, unless otherwise stated), respectively. The GPU, chipset, CPU and OS kernel are indicated.

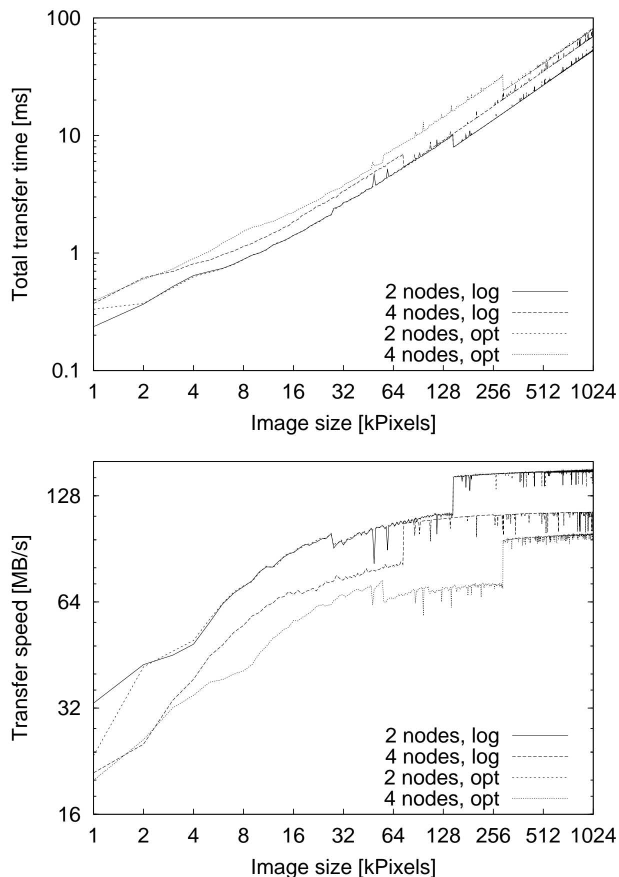

Figure 4. Transfer time (top) and transfer speed (bottom) as a function of image size using two different algorithms and different number of CPUs.

sible but also practical, as the performance is comparable to that of high end equipment at a fraction of the cost. In the future we want to attempt rendering really large datasets (several thousands of samples along one of the axes). This will require a different blending approach, as the current one scales almost linearly with the number of rendering nodes involved.

5 Acknowledgments

We would like to thank Dr. Peter Leinen from the University of Tübingen who made the implementation possible by helping us with the installation of graphics hardware on the new PC cluster ‘Kepler’. This work has been partially supported by the Deutsche Forschungsgemeinschaft, SFB 382.

References

[1] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. 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.

[2] 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.

[3] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. In *Proceedings of Supercomputing 2000*, 2000.

[4] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Computer Graphics (Proceedings of SIGGRAPH 01)*, 2001.

[5] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Computer Graphics Proceedings*, Annual Conference Series, pages 451–457, Los Angeles, California, July 1994. ACM SIGGRAPH, Addison-Wesley Publishing Company, Inc.

[6] P. Li, S. Whitman, R. Mendoza, and J. 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.

[7] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *Computer Graphics and Applications: Special Issue on Rendering*, 14(4):23–32, July 1994.

[8] 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.

[9] K. L. Rudrajit Samanta, Thomas Funkhouser and J. P. 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.

[10] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *Computer Graphics (SIGGRAPH '98)*, 32(4):169–179, 1998.