Latency- and Hazard-Free Volume Memory Architecture for Direct Volume Rendering

M. de Boer, A. Gröpl, J. Hesser, R. Männer
Lehrstuhl für Informatik V, Universität Mannheim, D-68131 Mannheim, Germany
e-mail: boer@mp-sun1.informatik.uni-mannheim.de

Abstract

The computational power required for direct volume rendering like ray-casting or volume ray-tracing can be provided by high-speed rendering architectures. However the increasing processor speed makes a performance bottleneck obvious -- the volume memory. This paper describes a volume memory architecture that achieves at least a tenfold speed-up in read-out rate with moderate additional hardware. It has been simulated successfully. A multi-level cache system is used with software prefetching and latency hiding. Pre- and postcaches additionally speed up the read-out rate so that a 512^3 data set stored in a single memory module can be rendered at 3.125 Hz.

Introduction

Visualization of three-dimensional data sets can be performed by a class of algorithms known as direct volume rendering [1]. These algorithms do not require intermediate data structures to represent the three-dimensional information. They simulate the interaction between light and virtual matter, the latter represented by the data set. Typically the ray-casting approach is used where the interaction of light is restricted to viewing rays only. Algorithms that consider higher order rays are called ray-tracing. Normally ray-tracing operates on well defined surfaces of objects. Algorithms that directly work on volume data, i.e. on every voxel of the data set, are called below volume ray-tracing algorithms. Examples are the rendering algorithm realized on the VIRIM real-time renderer [2] and the Heidelberg Raytracer [4].

Lacroute and Levoy [5] have achieved 10 frames per second for a ray-caster using a 256^3 data set on a 16 processor SGI Challenge. To achieve this speed they used parallel light, precomputation of gradients, space-leaping (suppression of rendering on empty regions), early ray-termination, and shear warp. Perspective views, and on-line classification of voxels according to density and gradient magnitude, reduce the frame rate by a factor of 6-10.

Faster frame rates can only be achieved by using problem adapted hardware architectures since inter-processor communication limits the speed on conventional super-computers. We consider three architectures [3].

For Cube-4 [6] a massively parallel rendering and memory architecture has been proposed for high-speed rendering of large data sets. Cube-4 is assumed to process a full line of viewing rays in parallel so that a full line can be read in one step

from a skewed memory. This cubic memory is read out by random access. Using synchronous DRAM devices a read-out speed of 30 MHz per memory bank is assumed. The maximal system consisting of 1024 memory banks and 1024 processor nodes (32 MHz) will be able to render 1024^3 data sets at 30 Hz. Cube-4 operates only on parallel rays that are arranged at fixed distances. Therefore neither arbitrary walk throughs are possible, nor algorithms using filtering like volume ray-tracing or 3D data processing, nor segmentation (region oriented segmentation does not seem to be supported by the memory architecture).

For Vogue [7] it is proposed to map the ray-cast algorithm directly into parallel pipelined hardware. 4 VLSI devices are required for integrating the functions. Vogue uses 8 memory banks from which 8 neighboring voxels forming a sub-cube can be read simultaneously. Each bank is 8-fold interleaved, page mode is used, and a one-entry cache speeds up repeated accesses to the same memory location. FIFOs decouple the memory system from the remaining hardware to allow usage of the page mode of normal DRAM devices independently. Compared to random access the speed-up amounts to a factor of 2-3. One processing node will achieve 2.5 Hz for 256^3 data sets. Compared to Cube-4, Vogue additionally implements perspective view which allows arbitrary data walk-throughs.

VIRIM [2] is the first real-time direct volume rendering system which is already in practical use. It uses in principle the same memory architecture like Vogue but with 4 times fewer memory devices and provides a much higher flexibility. Due to its free programmability ray-casting and volume ray-tracing, data processing, and segmentation can be implemented (ray-casting and volume ray-tracing are already implemented; region growing is currently in the implementation phase).

In contrast to Cube-4, Vogue, and VIRIM, the DIV^2A system [9] will use space-leaping and early ray-termination to speed up rendering by a factor of up to 20. It has a similar flexibility like VIRIM and is supposed to achieve a 20 Hz frame rate on 256^3 data sets using a 16 processor system. The memory architecture consists of 8 memory banks realized in SRAM; but only 50% of the memory can be used since the data set must be stored twice in an interleaved way.

All architectures have to cope with the limited data rate achievable with conventional memory devices. The upper limit of currently suggested memory architectures is 750 MBytes/s per memory.

In the following a new memory architecture is presented that achieves a 10-fold speed-up over the conventional architectures. It relies on efficient prefetching, latency hiding, and multi-level caching. Using commercial DRAM devices this architecture allows rendering of 512^3 data sets at >3 Hz per memory module.

The VIRIM System

VIRIM is the first available real-time direct volume special purpose rendering hardware[1]. A prototype is in daily use since June 1995. VIRIM will soon be commercially available. Its architecture is modular and scalable. Each module consists of two processors.

A geometry processor that resamples an image data set and estimates the gradients in the image space. Image data set voxels with gradients are transferred to the second processor, the ray-cast processor which performs shading and compo-

[1] The RealityEngine of SGI allows to use 3D texture mapping to implement direct volume rendering. However gradient estimation and Phong shading are not supported and must be precalculated.

siting. The ray-cast processor is assembled of freely programmable digital signal processors. It can therefore be used for any computation as long as it requires only loose coupling between processors.

The lead user is the Clinic for Head Surgery of the University of Heidelberg, Germany, which uses VIRIM for operation planning and control. Fig. 1 shows the system.

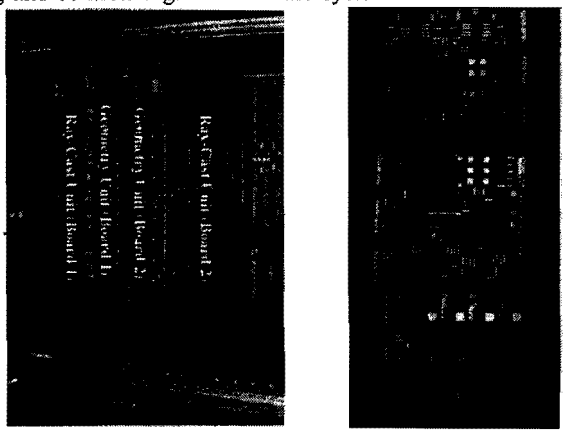


Fig. 1: VIRIM system. Left: full system. Right, top: geometry processor. bottom: ray-cast processor.

New Memory Architecture

The new memory architecture is based on the architecture used in the VIRIM system [2][3]. It uses image space parallelism to achieve real-time rendering rates. First the rendering approach is described followed by the basic architectural idea. A suggestion for implementation shows how this idea can be realized. The performance of an optimized version of this architecture is given.

Parallel Rendering Approach

The new memory architecture supports two volume oriented rendering approaches, ray-casting and volume ray-tracing. Ray-casting starts by casting a ray into the virtual scene from each image pixel [5]. Along each ray, sample points—that are image data set voxels— are generated by interpolation of their neighboring voxels in the object data set. On these sample points gradients are estimated using local difference filters. Gradients and local density are used for calculating the reflected light using Phong shading as reflection model. After shading the contributions of each sample point on the considered ray are composited to the final projection by using the *over*-operator [1].

In contrast volume ray-tracing additionally considers the absorption of incident light. This way shadows can be generated which are required for some applications [11].

Both algorithms, ray-casting and volume ray-tracing, have been implemented in the VIRIM system and are realizable using the new memory architecture as well.

For volume ray-tracing image data set voxels have to be processed in a predetermined order which can be parallelized only in the image data set. Therefore image space parallelism is used like for VIRIM. This rendering parallelism is realized as follows:

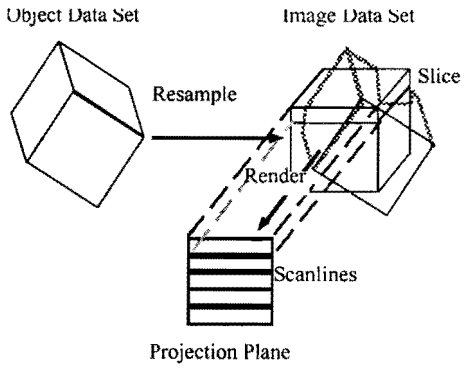


Fig. 2: Image space parallelism: First the object data set is resampled in an image data set that is parallel to the main viewing direction. After this transformation each image data slice is processed and produces one final scanline of the projection.

First an image data set is resampled from the object data set. The image data set is oriented such that one of its main coordi-

nate directions is parallel to the viewing direction. For each voxel of the image data set its position in the object data set is determined. Its eight neighboring voxels in the object data set are read-out from the volume memory and are interpolated e.g. by trilinear interpolation. Gradients can be computed in parallel to the interpolation process by using the same 8 object data set voxels like in the Vogue system [7]. After resampling and gradient estimation, absorption, shading, and compositing are processed.

Image space parallelism allows to process each image data slice independently from others provided that the necessary data for resampling and gradient estimation are available for each rendering processor.

Basic Architectural Idea

In this paper we concentrate on the memory architecture for two reasons. First, memory read-out rate is currently the bottleneck that limits the speed achievable by the rendering hardware. Second, processor performance doubles approximately every 18 months but the memory performance cannot catch up with that speed and thus the gap between memory and processor performance widens. As typical example, Knittel reports processing rates of approximately 80 MHz for one ASIC in the Vogue system while the system fails to deliver the required data rate by a factor of 2-4 [7]. In this paper we show that memory bandwidth will not be a limiting factor for real-time rendering for the next few years, if e.g. the proposed memory architecture is used.

For direct volume rendering algorithms the high data rate is required for reading the data from the volume memory before interpolation of the image data set voxel value. Here 8 voxels in a 2×2×2 sub-cube are to be read from the object data set. Parallel read-out is possible as has been demonstrated [2][7] by an appropriate distribution of the data over 8 different memory units (see Fig. 3). This 8-fold sub-division is used in this new architecture as well.

The open question is how to achieve a maximum read-throughput for each of these 8 identical units. Since a volume data set is very large —several tens to several hundreds of megabytes are common— slow dynamic memories (DRAMs) have to be taken instead of fast static memories (SRAMs) for cost reasons.

Recently new DRAM interfaces are available that promise at least one to two orders higher read-out throughput than conventional DRAM. One of the fastest devices is the RDRAM (Rambus DRAM) which is chosen for the new memory architecture, although the architecture can also be realized with other modern DRAM interfaces. The new architecture is designed to obtain nearly the maximal bandwidth of the RDRAM and to combine it with the non-sequential and non-regular access requirements for direct volume rendering caused by random viewing directions and by allowing arbitrary data walk throughs.

Rambus achieve the high bandwidth by using multiple (internal) banks in a interleaved mode. The effective sustained bandwidth for 32-byte random access transfers is 480 MB/s [10]

In the following we discuss the design for one of 8 identical memory units.

In order to achieve the high throughput by the RDRAM the data must be read sequentially. However this contradicts the possibility to randomly access the data during rendering, e.g. by

changing the viewing direction arbitrarily. In order to solve this problem a cache architecture is used. The cache stores small non-intersecting sub-cubes of the object data set that are required to resample one or more image data slices. The sub-cube data voxels are stored on the RDRAM in a sequential address space such that they can be read at maximum bandwidth. The sub-cube voxels stored in the cache can be accessed randomly, since the cache is realized in SRAM. Temporal and spatial coherency required during rendering guarantee efficient use of the cache. Temporal coherence occurs since subsequent resampling points of a ray often require the same or nearby object data set voxels for interpolation. Spatial coherency occurs for neighboring rays that share object data set voxels for interpolation.

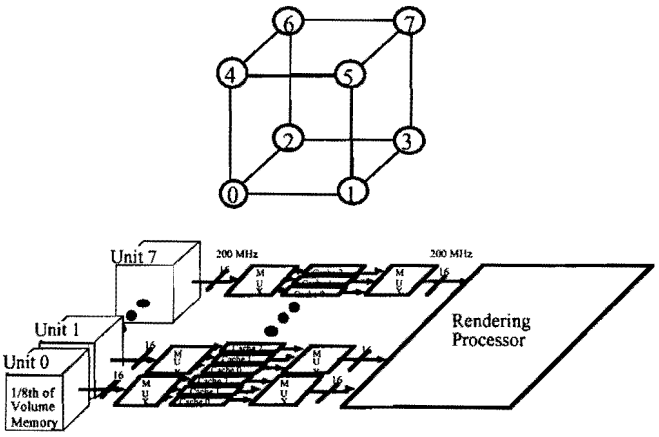


Fig. 3: New memory architecture. Top: Assignment of voxels of a 2×2×2 neighborhood to memory units. Bottom: The volume data is stored in an interleaved manner on 8 memory units. Each memory unit contains two or more caches. The rendering processor reads data directly from the caches of all memory units.

The second problem is the high latency for cache misses when the RDRAM must deliver the requested sub-cube. A prefetch strategy solves this problem. The prefetch strategy is software based and determines when which sub-cube is required for rendering before the sub-cube is actually required. Thus no cache miss occurs. The prefetch algorithm is a modified polygon filling algorithm (see Foley and van Dam [1]).

Third, since the rendering processor requires an uninterrupted access to the cache memory, a non-blocking cache is required. Non-blocking means that during read-out of the cache it can be loaded with new sub-cubes. Non-blocking is achieved by using multiple devices where one can be selected for read access and simultaneously a different one for write access.

Implementation of the Architecture

Resampling Order

The image data set —which is a cube— is generated by resampling slices parallel to the projection plane in front-to-back order (see Fig. 4). These slices are called x-z-slices in the following.

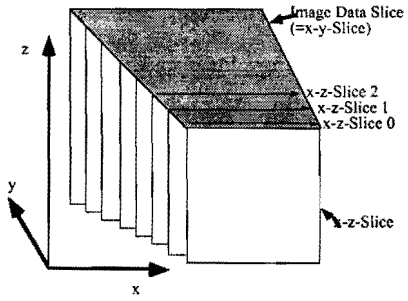


Fig. 4: The image data set consists of a stack of x-z slices that are perpendicular to the main viewing direction and parallel to the projection plane. These slices are resampled beginning by slice 0, i.e., in front-to-back order.

In the following it will be important that all these slices are parallel and have equal distance to each other. Due to the perspective projection however their size increases with the dis-

tance to the viewer so that the stack of x-z slices in the image data set cuts out a pyramid from the object data set[2].

Prefetching Sub-Cubes

For the proposed memory architecture latencies that come from cache misses and from the overhead of reading the first data from the RDRAM are hidden by prefetching data before it is required in the rendering process. The prefetch algorithm must precalculate when which sub-cube is required but it should not consume much computational power compared to the rendering algorithm. The precomputation runs in parallel to the rendering algorithm some slices ahead so that the latencies for reading sub-cubes from RDRAM and storing them into the caches are much shorter than the remaining time until these sub-cubes are read from the cache.

The algorithm determines which sub-cubes of the object data set are intersected by the currently considered x-z slice of the image data set. This intersection calculation is a generalized line drawing algorithm for planes in 3D space ([1], pp. 92-99.) where the basic building blocks are sub-cubes instead of voxels.

Sub-Cube Assignment to Caches

The data required for resampling each scanline is read out from the volume memory in the form of sub-cubes that are buffered in the caches. First sub-cubes required for one slice are loaded into the considered cache bank. The cache bank size is supposed to be large enough to store all these sub-cubes. If the cache bank is not filled yet, the sub-cubes required for the subsequent slices are loaded into the same bank until it is filled. The remaining sub-cubes of the last considered slice are loaded into the next cache bank. By this mechanism sub-cubes required for one slice can be distributed over two cache banks.

A software flag prevents sub-cubes that are already stored in the cached to be written twice thus saving bus bandwidth. The filling of the cache banks is done in a round-robin order, i.e., cache bank 0 is filled before cache bank 1, cache bank 2, and again cache bank 0 etc. The cache banks are filled continuously from its lowest address space irrespective of the previous contents.

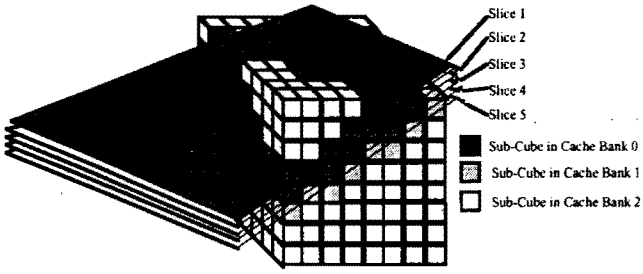


Fig. 5: Example of assigning sub-cubes to caches. The dark sub-cubes are required to resample image data slice 1 and are stored in cache bank 0. The information of this cache can be used for resampling image data slice 2 as well. Sub-cubes required for resampling slice three that have not yet stored in the cache bank 0 are stored in cache bank 1. To fill the cache bank some sub-cubes that are required for slice 4 only can be stored in this cache bank before switching to bank 2.

By choosing sufficiently large cache banks it is guaranteed that all sub-cubes required to resample an image data slice is read from at most two cache banks. When n is the edge-length of the sub-cubes the size is given by $512^2 \cdot 4n$ (details, see Appendix I).

Realization of Non-Blocking Caches

Non-blocking caches are realized by using individual cache banks and to allow to read from one bank while writing to another. Fig. 6 gives an example: While two cache banks are readable by the rendering processor one cache bank is writable from the RDRAM side. The cache banks are used in turn as

[2] In order to evade aliasing artifacts the maximal grid distance of rear slices must be below the grid distance of the object data set.

shown in Fig. 6. Assume for example, e.g., that cache banks 0 and 1 are used for read transfers and cache bank 2 is filled with sub-cubes from the RDRAM. When data from cache 2 is required, the status of each cache bank changes. Cache banks 2 and 0 are now readable while cache bank 1 is writable. In the next turn cache banks 1 and 2 are readable and cache bank 0 is writable.

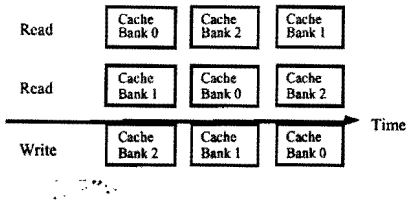


Fig 6: Three cycles of the filling scheme for the cache architecture.

Performance Estimation and Optimization of the Architecture

Optimal Cache Size

The memory system consists of two SRAM-based parts, the cache and an address-mapper memory that maps object data set absolute addresses into cache absolute addresses.

The size of the cache is determined by the sub-cube size; which is critical for the architectural efficiency as well. The larger the sub-cubes the higher the performance obtained by the RDRAM since the latencies per voxel are reduced. However the total size of the cache increases proportionally to the sub-cube size n since each cache has to store at least $4\times512^2\times n$ voxels for 512^3 data sets.

The address-mapper depends on the sub-cube size as well. To understand the relationship between address-mapper size and sub-cube size the implementation of the former has to be given (see also Fig. 7). The least significant bits of the object data set absolute address are interpreted as sub-cube relative addresses. The most significant bits define the sub-cube address in the data set. This sub-cube address is mapped by the mapping table onto an address with 1) an identifier that signifies which cache is to be accessed (and selects it) and 2) the memory address of the sub-cube in the respective cache.

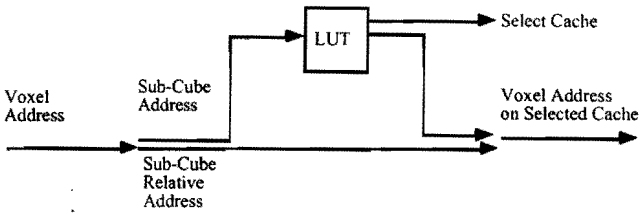


Fig. 7: Computation of the cache address of the voxel from its absolute address in the data set.

The cache absolute sub-cube address and the sub-cube relative address are then combined to address the selected cache.

Each time a new sub-cube is written into the cache memory the mapping table is updated accordingly.

The mapping table has $512^3/n^3$ entries, i.e., as many entries as there are sub-cubes in the volume memory.

Fig. 8 shows the total demand in SRAM for all memory units together as a function of n for cache and memory mapping table. It is assumed that for all 8 memory units only one mapping table is required since the sub-cube addresses are identical. In Fig. 8 SRAM size vs. sub-cube size is plotted. The graph has a minimum for n=4; which is a good choice in order to minimize the SRAM size of the whole system.

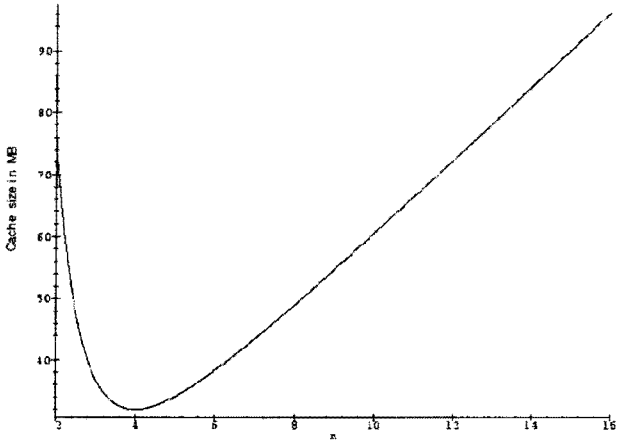


Fig 8: Total SRAM (cache and mapping table) required for an architecture with 3 cache banks per memory unit versus size n of the transferred cubes.

Further Cache Optimization by Fragmentation

The discussion above shows that the cache size is relatively large compared to the volume memory since it has to store those sub-cubes that are necessary to resample a x-z slice of the image data set. This way full spatial and temporal coherency of and within rays can be exploited when accessing object data set voxels for resampling. Each voxel is read only once from the volume memory and multiple references to the same voxel occur on the cache only.

By relaxing the amount of coherency however the size of the caches can be reduced by a factor of 4; which leads to a performance loss of only a ≈9%.

Performance

The cache architecture allows to resample image data set voxels at a rate close to 200 MHz assuming a sufficiently fast SRAM memory and a comparable pipeline processor and that processes the data at the same speed[3]. For a 512^3 data set $1.3 \cdot 10^8$ image data set voxels have to be generated for each frame. This architecture can thus support rendering systems that achieve frame rates of 1.6 Hz on 512^3 data sets using only one volume memory (8 modules with 3 banks each).

The contributions of each building block of the architecture, the cache principle, the prefetch, and the non-blocking cache design are listed below:

Without caching an approach like that taken for VIRIM would have to be used with the above mentioned performance (see Introduction) where the realistic difference between conventional DRAM and other, modern DRAM interfaces is at most a factor of 2-3.

Without prefetching cache misses occur. To handle a cache miss, the pipeline processor has to be stalled, i.e., all current states have to be stored. Then a package request is sent to the RDRAM that answers after 120 ns with the first data of the requested sub-cube of size 4^3. After at least 320 ns the sub-cube is stored in the cache. A realistic figure for this latency is therefore between 500 ns and 1 µs. Since the whole data set consists of approximately 2 million sub-cubes and since a cache miss occurs for each of these sub-cubes the total latency adds up to 1-2 seconds, i.e., the performance is reduced by a factor of 2.

Conventional caches require that the rendering processor must be stalled during the write to the caches. Since each voxel is accessed 8 times and has to be read (nearly) only once the performance loss is 1/8≈12.5% [4]. If the rendering processor is

[3] The implementation of 200 MHz devices on board level is difficult but multi-chip-module integration is realizable.

[4] Here it is assumed that a sufficiently large cache allows to harness the full spatial coherency. In general this is not the case however since nearby sub-cubes may lie far away in the linear address space of the cache and thus sub-cubes may be purged from the cache before the rendering processes accesses them a second time.

designed in a full pipeline design non-blocking caches do not require stall cycles and therefore reduce the hardware overhead of that processor.

Speedup by Post-Caches

During the simulation phase it turned out that the temporal coherency could be used in another memory hierarchy, a post-cache. It speeds-up the rendering rate by a factor of 2 by using only small additional hardware.

Each pipeline processor contains 8 such post-caches, one for each 16 bit wide channel to each memory unit. The post-cache works as follows (see Fig. 9): Assume the pipeline processor resamples point A. When sampling the next point B another 8 neighborhood is fetched which is disjunct to the previously fetched 8 voxels. This new neighborhood allows to resample the following point C as well. In total the access to the main cache is reduced by a factor of 2; which can be used to feed another rendering processor and thus speeds-up the rendering by a factor of 2. Using post-caches the rendering speed can therefore be raised to 3.2 Hz for 512^3 data sets.

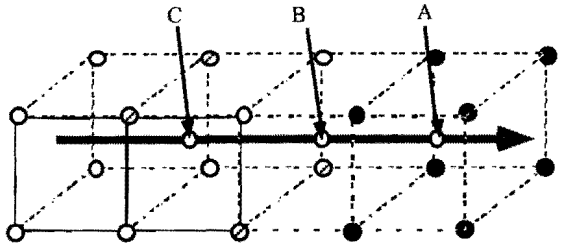


Fig. 9: When sampling A the voxels marked black are read from the memory units. For sampling B the hatched voxels are read. These hatched voxels allow to resample C without additional access to the memory units. A speed-up of about a factor 2 is achieved.

Outlook

The volume memory architecture described in this paper gives at least a speed-up of a factor of 10 compared to the best current approaches. Using this architecture, the volume memory is no longer the limiting factor for the next-generation direct volume rendering systems. Operating at 200 MHz a fully pipelined rendering processor of the pizza-box size could render 512^3 data sets in interactive time.

Acknowledgments

This work is supported by the Ministry of Education and Research, Germany under grant 01 IR 406 A8 and by the Landesforschungsschwerpunktprogramm of Baden-Württemberg under grant 7532.24-2-16.

References

[1] J.D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, Reading, MA, 2d. ed., 1990.

[2] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, H.-J. Baur. VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine. W. Straßer, 9th Eurographics Workshop on Graphics Hardware, Oslo, Norway, 1994, pp. 103-108.

[3] J. Hesser, R. Männer, G. Knittel, W. Straßer, H. Pfister, A. Kaufman. Three Special-purpose Architectures for Real-Time Volume Rendering. Eurographics '95, Maastricht, The Netherlands, 1995, pp. C-111—C-122.

[4] H.-P. Meinzer, K. Meetz, D. Scheppelmann, U. Engelmann, H.-J. Baur. The Heidelberg Ray Tracing Model. IEEE Comp. Graphics & Appl., Nov. '91, pp. 34.

[5] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform. Computer Graphics, Proc. of SIGGRAPH '94, Orlando, FL, 1994, pp. 451-457.

[6] H.-P. Pfister, A. Kaufman. Towards a Scalable Architecture for Real-Time Volume Rendering 10th Eurographics Workshop on Graphics Hardware, Maastricht, The Netherlands, 1995, pp. 123-130.

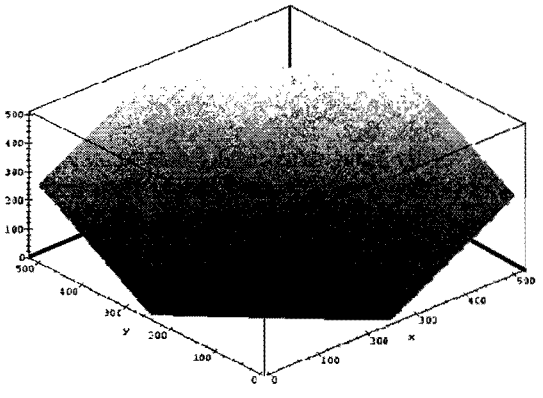
[7] G. Knittel, W. Straßer. A Compact Volume Rendering Accelerator. 1994 Symp. on Vol. Vis., ACP press., NY, 1994, pp. 67-74.

[8] F. Jones. A new year of fast dynamic RAMs. IEEE Spectrum, Oct. 1992, pp. 43-49.
[9] J. Lichtermann. Design of a Fast Voxel Processor for Parallel Volume Visualization. W. Straßer, 10th Eurographics Workshop on Graphics Hardware, Maastricht, The Netherlands, 1995, pp. 83-92.
[10] Improved Rambus DRAM Reduces Latency, Doubles Effective Bandwidth. Rambus Press Realease,Mountian View,CA. May 13,1996
[11] H. J. Wieringa. MEG, EEG and the integration with Magnetic Resonance Images. Ph.D. thesis, Univ. Twente, The Netherlands, 1993.

Appendix I: Cache Size

The data to resample one entire image data slice should always fit into one cache bank. The worst case occurs for a slice given by the equation z = x+y-256. Here a maximal number of sub-cubes is required for resampling. Fig. 10 shows that for this case 4 layers of sub-cubes are required, i.e., the cache must store at least 512^2 4n voxels per bank. The estimation is described in detail below:

Two voxels are called independent from each other if one voxel is not in the 26-connected neighborhood of the second voxel. For a cache bank of the discussed size the voxels contained in the most recently filled cache bank and in the cache bank to be filled next are separated by the voxels stored in the currently filled cache bank (see Fig. 10).



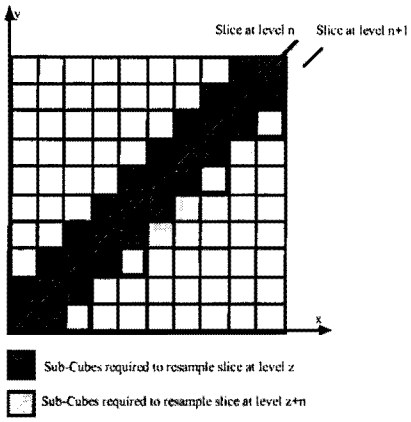


Fig. 10: Worst case situation for resampling an image data slice. Some sample points lie between 8 sub-cubes and thus require 8 of them for resampling (top: 3D-view, bottom: slice view). Those sample points of the same slice that differ by a z value of n require additionally sub-cubes that are dotted. No more than 4n of these sub-cubes are required. The right figure gives a 3D view of the worst-case slice.

Assume cache bank 0 and cache bank 1 have been filled and the next cache to be filled is bank 2. The sub-cubes in cache bank 0 and cache bank 2 are separated by at least one sub-cube stored in cache bank 1, because each cache bank contains 512^2 4n voxels. Thus any two sub-cubes in cache bank 0 and 2 respectively are independent in respect to a 26-connected neighborhood.

An exact analysis obtained by simulation of the worst case condition gives a tighter bound: For resampling the worst case slice it was assumed that 512^2 4n voxels were needed. A more detailed investigation gives 512^2 3n voxels for this case.

Appendix II: Cache Fragmentation

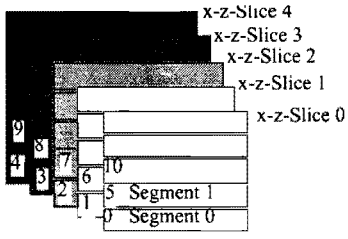


Fig. 11: Processing order denoted by 0-10 in which the segments and x-z slices are processed.

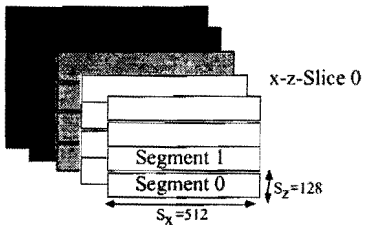


Fig. 12: A segment is a part of the x-z slice. In this example the x direction is parallel to the scanline and the z direction is perpendicular to both the main viewing direction and the scanline. 4 segments of 128 voxels z-thickness are required for an image data slice of 512×512.

Instead of resampling a whole x-z-slice it is resampled in equally large portions, called segments (see Figs. 11 and 12). Each segment consists of S_z lines parallel to the scanlines in the projection plane. The more segments are used the smaller is the required cache size. For $m = 512/S_z$ segments the cache size is given by $3n \times 512 \times S_z$, i.e., it is reduced by a factor m.

However the spatial coherency of data resampling two subsequent slices is destroyed if a new segment is processed since the required sub-cubes have been purged from the cache. These additional sub-cubes must be loaded again; which leads to the performance loss. The amount of data for each new segment is $3n \times 512 \times 512$ for 512^3 data sets, i.e., at most the amount of data to resample a slice. For m slices 3n $m \times 512 \times 512$ voxels are required, i.e., $3n/S_z$. For $S_z = 128$ the performance loss is ≈9% and the cache size is reduced by a factor of 4.