Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

Efficient Visualization of Large Medical Image Datasets on Standard PC Hardware

V. Pekar[1], D. Hempel[1], G. Kiefer[2], M. Busch[2], and J. Weese[2]

Philips Research Laboratories Hamburg[1] and Aachen[2], Germany
vladimir.pekar@philips.com

Abstract

Fast and accurate algorithms for medical image processing and visualization are becoming increasingly important due to routine acquisition and processing of rapidly growing amounts of data in clinical practice. At the same time, standard computer hardware is becoming sufficiently powerful to be used in applications which previously required expensive and inflexible special-purpose hardware. We present an efficient volume rendering approach using the example of maximum intensity projection (MIP), which is an important clinical tool. The method systematically exploits the properties of general-purpose hardware such as hierarchical cache memories and superscalar processing. In order to optimize the cache efficiency, the dataset is processed in blocks which fit into the processor cache. The innermost ray casting loop is transformed such that the arithmetic operations and memory accesses can be processed in parallel on current general-purpose processors. Combined with other optimization strategies, such as vectorization and block-wise ray skipping, this approach yields near-interactive frame rates for large clinical datasets using a standard dual-processor PC. Data compression and simplification methods have intentionally not been used in order to demonstrate the achievable performance without any quality reductions. Some of the presented ideas can be applied to other computationally intensive image processing tasks.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation; J.3 [Life and Medical Sciences]: Medical information systems.

1. Introduction

In recent years the quantity of acquired clinical image data has been growing very rapidly, in particular due to new imaging technologies, such as multi-array CT or 3D ultrasound. Data processing time becomes a critical factor in clinical routine, and there is a strong demand for fast image processing algorithms. One important application is volume data visualization, where the ability to interact with the system in real time is crucial for its acceptance by the clinicians. In this case, a system should render 5-10 frames per second, corresponding roughly to threshold timing of hand-eye coordination.

One way to achieve the required rendering performance is the use of dedicated hardware, e.g. VolumePro boards [1]. For many applications, however, there is a strong interest in fast volume rendering on standard PC hardware, since it is cheaper and more flexible to be adapted to specific demands [4]. Unfortunately, interactive rendering is still at the upper performance limit of standard hardware even for moderately sized datasets of 32-256 MB. Consumer graphics cards can be used to accelerate volume rendering using 2D or 3D textures [12]. However, their memory is typically too small to store the complete dataset, and the data transfer to the card may become a serious bottleneck.

The purpose of this paper is to develop and evaluate a software-based rendering approach. In order to be applicable in clinical practice, there must not be any restrictions on the size of the dataset. We also do not consider simplification techniques like (adaptive) downsampling, data quantization or lossy compression, which may speed up volume rendering considerably at the expense of (minor) quality reductions.

An important criterion for the performance assessment of visualization algorithms is their memory access behavior. One can distinguish between *image ordered* methods, such

as ray casting, and *object ordered* methods, such as splatting. Shear-warp factorization [5] is a fast hybrid technique, where the volume is mapped to a sheared image adapted to the object space. Mora et al. [8] propose an object ordered method which mimics the output behavior of ray casting at the expense of a larger total number of memory accesses and a slightly reduced computational accuracy.

The approach presented in this paper is based on ray casting, since it provides the best image quality among the techniques listed above. We chose as an application the parallel maximum intensity projection (MIP), however, most of the techniques described below are also applicable to generalized volume rendering with perspective projection.

State-of-the-art general-purpose processors provide super-scalar processing capabilities, deep pipelining and out-of-order execution [3]. The speedup achievable by these techniques is determined by control, data, and structural dependencies among software instructions. Similarly to some prior work [4, 11], we use software pipelining and vectorization in order to eliminate such dependencies and improve the utilization of the CPU.

Unfortunately, the rapid increase of CPU clock rates does not always lead to a corresponding performance increase of image processing algorithms. One significant reason for this is the limited speed of memory access. In order to achieve maximum computational performance, the caching strategy of modern CPUs should be carefully respected [11]. In the field of numerical algorithms, the transformation of multi-dimensional data structures into locally bounded blocks is a well known technique aimed at improving the cache usage [2, 7]. Volume decomposition into small sub-blocks is also a useful method for ray tracing and ray casting algorithms which dramatically improves the memory access performance [10, 13]. Analogously to these approaches, the strategy chosen here is to process the dataset block-wise.

The proposed techniques enable near-interactive rendering performance for real clinical datasets on a dual-processor PC. The methodology is also applicable in other image reconstruction and image processing algorithms (e.g. back-projection or Hough transformations).

In Section 2, the block-based ray casting approach is introduced which aims at optimizing the cache efficiency and forms the basis for some of the other optimizations. Section 3 describes how the super-scalar processing can be utilized. Multi-processor parallelization is addressed as well. Further algorithmic optimizations are described in Section 4. In Section 5, experimental results are presented.

2. Block-Based Ray Casting

The principal disadvantage of conventional ray casting with regard to computational efficiency is that the dataset is not processed in storage order. This can result in a considerable amount of cache misses depending on the viewing direction, and the rendering speed can consequently be severely degraded. In order to circumvent this problem, data locality must be improved.

Ideally, a cache line should be processed completely before being replaced by the next one. One way to optimize the cache usage is to process the volume in compact units (blocks). However, block-wise processing alone is not always sufficient due to the fact that several memory addresses may compete for the same cache line. Therefore, a cache-conscious storage scheme should also be used, where the data is stored in contiguous memory blocks fitting entirely into the cache. In order to load a minimum set of cache lines per block, the data should be stored with a small overlap depending on the radius of the interpolation kernel.

For block-wise data processing in a ray casting algorithm, pixels in the target image plane affected by each block must be determined. For example, pre-computed discrete projections were used to determine voxel footprints [9]. We use a different approach, and our block-based ray casting algorithm can be formulated in the following way:

1. Split dataset into blocks fitting into the cache.
2. Project block vertices onto the image plane.
3. For each pixel in the bounding box of the block projection send a ray back into the volume.
4. If the ray intersects the block, cast the ray within the volume and compute the corresponding pixel value.

For MIP, blocks can be processed in arbitrary order and the final pixel value corresponds to the maximum data value along the ray cast through all contributing blocks.

An important issue in the algorithm is the optimal block size. Today's PC and workstation processors have a hierarchical memory system with two or three levels of cache. Typical cache sizes are 8-64 KB for a level-1 cache and 256-1024 KB for a level-2 cache. For optimal cache performance, a data block should be small enough to fit into the level-1 cache which provides the fastest way of delivering data to the CPU. On the other hand, if the total number of blocks becomes large, the computational overhead is not compensated by the speed of data delivery, and the rendering performance decreases considerably.

The dependence of the rendering performance on the block size is shown in Fig. 1. The tests have been performed with a synthetic 256^3 dataset on a 1 GHz Pentium III PC for the viewing directions which correspond to the "best" and "worst" memory/cache interaction. It can be seen that blocks which fit into the level-2 cache (256 K) provide the best performance, whereas for small blocks which fit into the level-1 cache (16 KB) the overhead dominates over the additional speedup in memory access. In our experiments, the optimal block size was between 40^3 and 60^3. If the block does not completely reside in the cache, severe performance degradation may occur, depending on the view direction. If it fits into

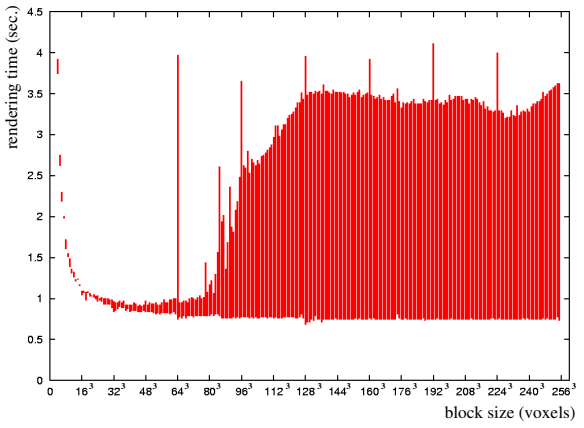

Figure 1: *Minimum and maximum rendering time in dependence on the block size.*

the cache, the dependency of rendering performance on the viewing direction is almost completely eliminated. Note that certain block sizes above 63^3 cause cache mapping conflicts, which result in characteristic peaks in the diagram.

3. Super-Scalar Processing

Modern processors typically provide facilities for instruction-level parallelism, i.e. several instructions are processed in parallel using multiple independent execution units. The instruction scheduling can be done statically by the compiler or dynamically by the processor hardware. In either case, the parallelization and reordering of instructions is limited by dependencies, which can be classified as data dependencies (one operation needs the result of the other), structural dependencies (e.g. two multiplications cannot be parallelized if only one multiplier is present), or control dependencies (e.g. instructions located after a conditional branch cannot be executed before the branch condition is evaluated) [3]. Being aware of this, we adopted two measures in order to optimize the degree of instruction-level parallelism (Sections 3.1-3.2). Furthermore, the block-based rendering approach has been generalized for multi-processor parallelization (Section 3.3).

3.1. Eliminating Intra-Loop Dependencies

The most time-critical part of a ray casting algorithm is its innermost loop over `N` samples, which has the general form:

```
for i = 1 to N
    CalcAddress (i);
    LoadVoxels (i);
    Interpolate (i);
end for;
```

Each of the main blocks *CalcAddress*, *LoadVoxels*, and *Interpolate* may consist of approx. 20-40 machine instructions. In terms of hardware resources, *LoadVoxels* can be executed in parallel to *Interpolate* and *CalcAddress*, as *LoadVoxels* mainly needs the memory port but not the ALU, while the other two blocks contain ALU instructions, but no memory accesses. However, the implementation above does not allow this due to data dependencies: each step needs the result of the previous one. These data dependencies can be eliminated by a technique known as software pipelining [6]. In the following equivalent code, operations from different loop iterations have been rearranged to form a new loop body:

```
CalcAddress (1);
LoadVoxels (1);
CalcAddress (2);
for i = 2 to N-1
    Interpolate (i-1);
    LoadVoxels (i);
    CalcAddress (i+1);
end for;
Interpolate (N-1);
LoadVoxels (N);
Interpolate (N);
```

Now, the memory accesses of *LoadVoxels* can be executed in parallel to the arithmetic operations of *CalcAddress* and *Interpolate*. On PC processors like the Pentium 4, the parallelization is scheduled internally by the processor [14], and experiments have confirmed that the rearranged loop runs considerably faster than the original version, even though the number of operations is exactly the same in both cases.

Besides this, control dependencies are minimized by replacing conditional branches by conditional move operations as far as possible. Conditional branches that can not be eliminated are transformed such that they become more predictable by the branch prediction unit of the processor.

3.2. Vectorization

Most PC and workstation processors provide *Single Instruction – Multiple Data* (SIMD) instruction set extensions, which commonly provide vectorized integer and floating-point operations such as multiplications, additions as well as other primitive ALU and data transfer operations. Depending on the processor type, they differ in the vector width, the number of registers, and some details of the instruction set. Nevertheless, they can all be used in a similar way to speed up ray casting.

In order to make use of vector processing, the ray casting loop has been unrolled n times (where n is the vector size) and rearranged such that the computations for n subsequent sample points are performed in parallel. For example, the vector processing unit of a Pentium 4 processor provides 8 registers of 64 bit and 8 register of 128 bit. This is sufficient to unroll the inner loop 4 times and thus to process 4 samples in parallel. All intensity values are processed as 16 bit values. A reduction to 8 bit values would allow a higher

degree of vectorization but may be insufficient for medical applications.

3.3. Multi-Processor Parallelization

A crucial property of the block-based ray casting algorithm is that rendering tasks for individual blocks are mutually independent, and the resulting image can be composed from the partial images by simple 2D operations. Hence, the algorithm can easily be parallelized on multi-processor systems with both shared and distributed memory. For MIP, the resulting image can be composed by a maximum operation from the intermediate images. This facilitates the parallelization on distributed memory systems, e.g. PC clusters, since only 2D images have to be transferred across the network.

We implemented a shared memory parallelization using threads for dual-processor PCs. A separate thread on each CPU picks up blocks from a queue and renders a separate image. The speedup factor by rendering on a dual-processor system instead of a single-processor one is typically 1.7–1.8.

4. Further Optimizations

4.1. Data Interpolation

Interpolation order is one of the most important issues in the rendering process, since it determines the quality of the rendered image. A commonly used method is to sample the volume at equidistant locations and to compute the respective sample values by tri-linear interpolation (see Fig. 2a). Linear interpolation offers a reasonable compromise between good image quality and relatively low computational costs.

Instead of equidistant sampling we use samples located at the voxel cell boundaries (Fig. 2b). Compared to equidistant sampling this technique allows to reduce computational effort considerably. As tri-linear interpolation is defined by the weighted sum of two bi-linearly interpolated values, the number of memory accesses and arithmetic operations for tri-linear interpolation is more than twice as high as for bi-linear interpolation. With the samples being located at cell boundaries, tri-linear interpolation is equivalent to bi-linear interpolation which can be computed much faster. In addition, cell-border sampling adapts the sampling rate to the data resolution while avoiding undersampling atrifacts which may occur in the case of equidistant sampling with a comparable rate.

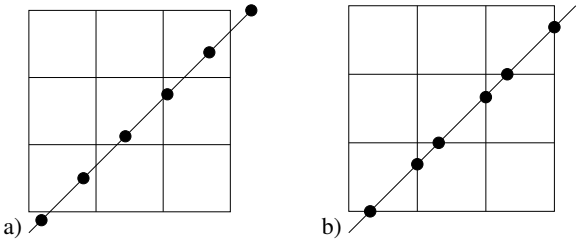

Figure 2: *Equidistant (a) and cell-border (b) sampling.*

Fig. 3 shows a comparison between cell-border sampling and standard equidistant sampling. For the latter case, $4\times$ oversampling was used. When observing fine vessel structures, no visible differences in the images can be seen.

4.2. Block-Wise Ray Skipping

The idea of block-wise ray skipping is to reduce the number of rays cast through individual blocks. A block does not contribute to a pixel value in the final MIP if its maximum data value is less than or equal to the current pixel value. Block-wise ray skipping can then be implemented as follows:

1. Maximum data values are computed for each block when loading the dataset from disk.
2. The block list is sorted in decreasing order according to the maximum block value.
3. During the rendering process, if the value of a pixel affected by the block is already greater than the maximum block value, the ray is not cast through this block.

Due to the processing order of the block list according to the maximum value, brighter structures are rendered first, and more rays can be skipped compared to the processing in an arbitrary order. This results in a speedup factor of 1.3–1.6, depending on the data and the block size.

5. Results

We tested our MIP algorithm on three medical datasets: a contrasted 512x512x376 thorax CT dataset (Fig. 4), a blood pool contrasted 512x512x120 MRA dataset (Fig. 3), and a smaller 256x256x256 CT dataset (Fig. 5). In all cases, we generated a sequence of images corresponding to a complete rotation of the dataset around the z-axis. Note that this does not include the view direction of the worst memory/cache interaction but all of the clinically used views. The image sizes are 512x512 pixels for the first two datasets and 256x256 pixels for the 256^3 dataset. For the experiments, we used a computer with two 2.2 GHz Pentium 4 Xeon processors. The ray casting loop (Sections 3.1-3.2) was implemented in assembly language. All computations on intensity values were performed at an accuracy of 16 bit. By reducing the accuracy to 8 bits, a considerable speedup can be achieved as only half of the memory traffic is required and the degree of SIMD vectorization can be increased by a factor of two. However, this may lead to visible quantization artifacts after possible post-processing steps which is not acceptable for some clinical applications.

We investigated the impact of the described acceleration techniques on the rendering performance. Table 1 shows the minimum, average and maximum rendering times computed with the optimal block size as well as the average speedup

	CT dataset				*MRA dataset*				*Thorax dataset*			
	Min.	Avg.	Max.	Speedup	Min.	Avg.	Max.	Speedup	Min.	Avg.	Max.	Speedup
Simple ray casting	0.48	2.60	3.78		0.89	5.10	7.20		3.31	15.28	20.09	
+ memory opt.	0.52	0.73	0.84	3.56	1.02	1.46	1.66	3.49	3.63	4.57	5.09	3.34
+ super-scalar proc.	0.28	0.41	0.50	1.78	0.56	0.84	1.00	1.74	2.03	2.68	3.03	1.71
+ ray skipping	0.17	0.28	0.41	1.46	0.41	0.67	0.81	1.25	1.47	2.01	2.19	1.33
+ second CPU	0.09	0.16	0.22	1.75	0.22	0.38	0.44	1.76	0.83	1.15	1.34	1.75
total speedup				16.25				13.42				13.29

Table 1: *Rendering times (in sec.) per frame and speedup factors for individual optimizations.*

factors. With all optimizations enabled, we achieved an average frame rate of 6.3 frames per second with the 256^3 dataset. For the other datasets, the rendering time increases linearly with the number of voxels, showing that the approach is scalable for large datasets. The largest gain is achieved just by optimizing the memory access behavior via block-based processing.

6. Conclusions

We have presented an efficient concept for the visualization of large volumetric datasets by ray casting on standard PC or workstation hardware, using the example of MIP. Without any loss in image quality, an average frame rate of 6.3 has been achieved for a 256^3 dataset.

One of the goals of this work was to determine the achievable performance on standard hardware and to determine the most relevant bottlenecks. A timing analysis of the various optimization techniques has shown that block-based data processing leads to the largest performance gain. This indicates that ray casting is clearly memory-limited with current hardware. Further techniques support super-scalar processing and allow to perform arithmetic computations in parallel to the memory accesses by reducing data dependencies.

The presented algorithmic methodology is also applicable to other computationally intensive tasks in image reconstruction and processing, in which it is possible to alter the data processing order towards the processing of compact memory-aligned data blocks rather than scattered patterns.

Acknowledgments

We thank Dr. Toombs and Dr. Flamm of St. Luke's Episcopal Hospital, Houston, for providing the MRA dataset and Dr. Rogalla of Charité Hospital, Berlin, for providing the thorax CT data.

References

1. TeraRecon, Inc. http://www.terarecon.com
2. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. *Supercomputing '92*, pages 114–124.
3. J. Hennessy, D. Patterson. Computer Architecture: A Quantitative Approach. *Morgan Kaufmann Publishers*, 2nd edition, 1996.
4. G. Knittel. The UltraVis system. *VolVis '00*, pages 71–79.
5. P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics (Proc. of SIGGRAPH' 94)* 28:451–458.
6. M.S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN '88 Conf. on Programming Language Design and Implementation (PLDI)*, pages 318–328.
7. M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimization of blocked algorithms. *4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74.
8. B. Mora, J.-P. Jessel, R. Caubet. A New Object-Ordered Ray-Casting Algorithm. *IEEE Visualization 2002*, pages 203–210.
9. L. Mroz, H. Hauser, and E. Gröller. Interactive high-quality maximum intensity projection. *EUROGRAPHICS 2000*, pages 341–350.
10. S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, 1999.
11. I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *EUROGRAPHICS 2001*, pages 153–164.
12. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *SIGGRAPH '98*, pages 169–177.
13. C.-K. Yang and T. Chiueh. I/O conscious volume rendering. *VisSym'01*, pages 263–272.
14. *Intel Pentium 4 Processor Optimization Reference Manual.* http://developer.intel.com

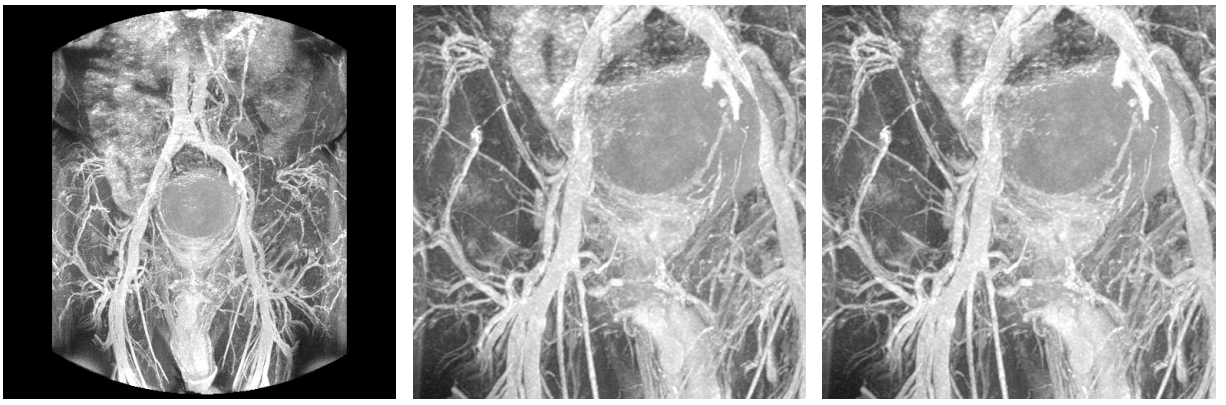

Figure 3: *Maximum intensity projection (MIP) of a contrast enhanced MR angiography dataset. Frontal view (left) and diagonal close-up views. Tri-linear interpolation with* $4 \times$ *oversampling (middle) and bi-linear interpolation at the voxels boundaries (right).*

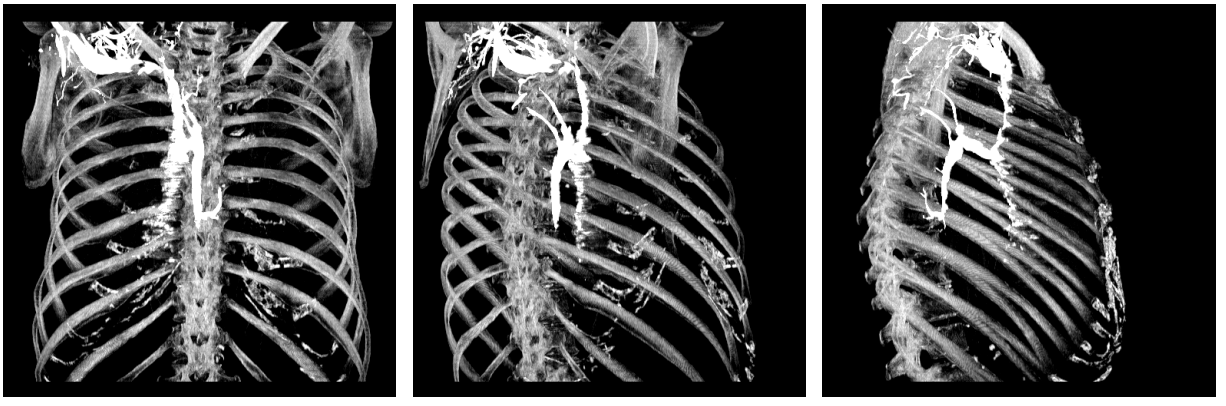

Figure 4: *Selected MIPs from the image sequence with a thorax dataset.*

Figure 5: *Selected MIPs from the image sequence with a CT dataset.*