Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computers & Graphics 28 (2004) 719–729

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

A refined data addressing and processing scheme to accelerate volume raycasting

Sören Grimm*, Stefan Bruckner, Armin Kanitsar, Eduard Gröller

Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11, A-1040 Vienna, Austria

Abstract

Most volume rendering systems based on CPU volume raycasting still suffer from inefficient CPU utilization and high memory usage. To target these issues we present a new technique for efficient data addressing. Furthermore, we introduce a new processing scheme for volume raycasting which exploits thread-level parallelism—a technology now supported by commodity computer architectures.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Volume raycasting; Bricking; Parallel computing; Thread-level parallelism

1. Introduction

Three main volume rendering approaches can be distinguished. Two of them are hardware based; the first one utilizes commodity graphics-cards; the second one utilizes special purpose hardware, e.g. VolumePro and Vizard; the third is CPU based [1–6]. Purely hardware based solutions provide real-time performance and high quality; however they are limited in their functionalities: Basic visualization systems are supported by hardware volume rendering solutions; consequently they are the mostly applied approach in practice. Advanced visualization systems provide preprocessing features such as filtering, segmentation, morphological operations, etc, if such operations are not supported by the hardware, they have to be performed on the CPU and data must be transferred back to the hardware. This transfer is very time consuming, thus no interactive feed-back is possible. In contrast to that, in a purely CPU based solution this transfer is unnecessary and therefore CPU based solutions are still commonly used in such systems. Another advantage of such a solution is the high flexibility. Many high-level optimization techniques have been developed to achieve high performance CPU solutions. Most of these techniques have one assumption in common: only parts of the data have to be visualized. This assumption is still valid, but the data delivered by new higher-resolution acquisition devices increases rapidly. This introduces a new main issue: enormous amount of data must be handled. Previous volume raycasting approaches, like Knittel [7] or Mora [6] achieved impressive performance by using a spread memory layout. The main drawback of these approaches is the enormous memory usage. In both systems, the usage is approximately four-times the data size, plus storage for gradients if shading is used. This memory consumption is quite a limitation, considering that the maximum virtual address space is about 3 GB on commodity computer systems. The main focus of our research was to address this issue in order to present a new approach using significantly less memory. In contrast to other methods in our approach, all computations are done on the fly. To accelerate this on-the-fly computation, refined data addressing techniques for a bricked volume layout are presented. Accordingly a data processing scheme is presented, which exploits common and new hardware technologies like thread-level parallelism. Such a technology enables more efficient CPU utilization and therefore provides significant speedup.

*Corresponding author. Fax: +43-1-58801-18672.
E-mail address: grimm@cg.tuwien.ac.at (S. Grimm).

The distribution of ideas of this paper is as follows: Section 2 surveys related work; Section 3 describes the volume raycasting system and its used new acceleration techniques; Section 3.1 provides a brief introduction to caches; Section 3.2 describes the used volume memory layout; Section 3.3 our new data addressing schemes; Section 3.4 thread-level parallelism; Section 3.5 the work-flow of the whole raycasting system; Section 3.6 the system is analyzed and results are presented. In Section 4 the system is discussed. Finally the conclusions of this research are presented in Section 5.

2. Related work

A prominent volume rendering approach which achieves high performance by using cache coherency is the Shear-Warp Factorization algorithm [8]. Cache coherency is achieved by re-sampling slice-wise and keeping the data in memory for each major viewing axis. The main drawbacks are the low quality and the three-fold memory usage. In contrast to this Knittel [7] achieved very high cache coherency by introducing a spread memory layout for fast access. He virtually locked all needed address look-up tables and color look-up tables into the cache. This leads to a rather high cache coherency and therefore high CPU utilization; memory usage is however increased by a factor of four. This memory storage requirement is way too high. Considering that the maximum virtual address space of today's mainstream workstations is three Gigabyte. Therefore the maximum data-set size is limited to 3 Gigabyte/4 = 768 MB. This seems to be an adequate size. However, in most of the visualization systems the examination of multiple data sets is required. Furthermore, additional volumes or data-structures have to be kept in memory to support various operations like segmentation, filtering, and others. Mora et al. [6] also based their approach on this spread-memory layout; he used an octree to obtain even more performance for first-hit volume raycasting. The enormous memory usage of both systems is a considerable limitation on state-of-the-art commodity computer systems; moreover these approaches are more limited by memory bandwidth than by CPU performance. This is not favorably, since the evolution of computer systems showed that CPU performance increases faster than memory bandwidth does.

Law and Yagel [9] proposed a parallel raycasting algorithm for a massively parallel processing system: they proved that appropriate data subdivision and distribution to the available caches lead to high cache coherency. The scheme can be adapted to commodity single- and multi-processors. The use of this scheme, leads to high cache coherency of all caches; however high CPU utilization is not inherent. Thread-level parallelism and advanced data addressing schemes turn out to be a solution to this utilization issue. Throughout the research process the basic data processing scheme was extended, in order to significantly increase CPU utilization, accelerating the raycasting process.

3. Cache coherent raycasting algorithm

There are two main requirements to achieve high CPU utilization: First, execution units have to operate at full capacity; Second a high cache hit rate is desirable, which implies that no cache thrashing occurs. The first condition can be fulfilled with thread-level parallelism, see Section 3.4. For the second condition we will define working sets so that they follow two known principles of locality, *temporal locality*—an item referenced now will be referenced again in the near future, and *spatial locality*—an item referenced now also causes its neighbors to be referenced.

3.1. Cache

The cache hierarchy of a x86-based system is shown in Table 1. Going up the cache hierarchy towards the CPU, caches get smaller and faster. In general if the CPU issues a load of a piece of data the request is propagated down the cache hierarchy until the requested data is found. It is very time consuming if the data is only found in a slow cache. This is due to the propagation itself as well as to the back propagation of data through all the caches.

Since the focus is on speed, frequent access to the slower caches has to be avoided. Accessing the slower caches, like hard disk and main memory, only once would be optimal. This is straightforwardly achieved for the hard disk level, as we assume that there is enough main memory to load all the data at once. To achieve this for the main memory is much more sophisticated.

Caching is important if data and instructions are not only used once but used repeatedly with temporal and spatial locality. In other words, working-sets are needed which fit into the caches and are used frequently.

3.2. Bricked volume layout

The workflow of a standard volume raycasting algorithm on a linearly stored volume (commonly

Table 1
Cache hierarchy

Level	Access	Typical size
Register	1–3 ns	1 KB
Level-1 cache	2–8 ns	8–128 KB
Level-2 cache	5–12 ns	0.5–8 MB
Main memory	10–60 ns	256 MB–8 GB
Hard disk	8–12 ms	100–200 GB

xyz-storage order) is as follows. For every pixel of the image plane a ray is shot through the volume and the volume data is re-sampled along this ray. At every re-sample position re-sampling, gradient computation, shading and compositing is done. From a performance point of view this work flow is very inefficient:

- The closer the neighboring rays are to each other, the higher the probability is that they partially process the same data. Given the fact that rays are shot one after the other, the same data *has to be read several times from main memory*, because the cache is not large enough to hold the processed data of a single ray.
- Different viewing directions cause a different amount of cache-line requests to load the necessary data from main memory which leads to a *varying frame-rate.*

These are the two main reasons, which lead to a bad CPU utilization.

Our main research focus is to support transparent as well as first-hit views with high real-time performance, without increasing the memory usage dramatically. It is a known fact, that bricking itself is one way to achieve high cache coherency, without increasing memory usage. The concept of bricking supposes the decomposition of data into small fixed-sized data bricks. The brick data in our case is stored linearly in common *xyz*-order. The bricks itself are linearly stored in common *xyz*-order. However, accessing data in a bricked volume layout is very costly (see Parker et al. [10]). In contrast to their proposed two-level subdivision hierarchy, we choose a one-level subdivision of the volume data. This is due to the fact, that every additional level introduces costs for addressing the data. For this one-level subdivision layout we developed a very efficient addressing scheme (see Section 3.3).

In our system, we are able to support different brick-sizes, as long as each brick-dimension is a power of two. Especially, if we set the brick-size to the actual volume-dimensions we have a common raycaster which operates on a simple linear volume layout. This enables us to make a meaningful comparison between a raycaster which operates on simple linear volume layout and a raycaster which operates on a bricked volume layout with optimal brick size. To underline the effect of bricking we bench-marked different brick sizes. Fig. 1 shows the actual speedup achieved by brick-wise raycasting. For testing we specified a translucent transfer-function, such that the impact of all high level optimizations, like early ray termination, brick skipping, zero-opacity skipping, etc was overridden. In other words, the final image was the result of brute-force raycasting of the whole data. We tested data sizes up to 512 MB. The size of the dataset had no influence on the actual optimal performance gains. The exemplary tested data shown in Fig. 1 was a $256 \times 256 \times 256$ typical

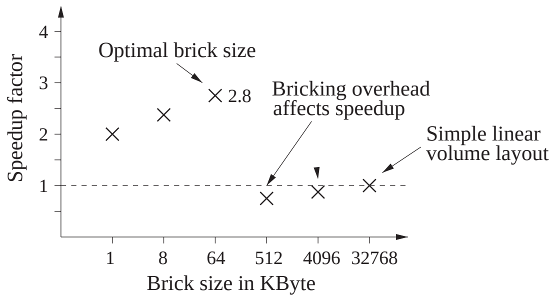


Fig. 1. Brick-based raycasting speedup compared to common raycasting on linear volume. Test system specification: Pentium 4 Xeon 512KB Level-2 cache.

medical dataset, with 16 bit per voxel. Furthermore, we did a worst-case comparison with respect to the viewing direction. In case of small bricks the worst case is similar to the best case. In contrast to that, using large bricks shows enormous performance decreases depending on the viewing direction. This is the well-known fact of view-dependent performance of a raycaster operating on a simple linear volume layout. The constant performance behavior of small bricks is one of the main advantages of a bricked volume layout. There is nearly no view dependent performance variation anymore.

Going from left to right in the chart shown in Fig. 1, first we have a speedup of about 2.0 with a brick-size of 1 KB. Increasing the brick-size up to 64 KB also increases the speedup. This is due to more efficient use of the cache. The chart shows an optimum at a brick size of 64 KB ($32 \times 32 \times 32$) with a speedup of about 2.8. This number shows the optimal trade-off between the needed cache space for ray-structures, sample data, and the color look-up-table. Further increase leads to performance decreases due to exceeding the cache capacity and bricking overhead. This performance drop-off is reduced, once the brick subdivisions gets close to no subdivision. In other words, with a brick size of the same size as the volume itself, the ray-structures have no influence on the performance. Since, in this case there is only one brick and therefore only one list of rays to process. This is exactly the same rendering context of a common raycaster for a simple linear volume layout.

3.3. *Efficient addressing*

The evolution of CPU design shows that the length of CPU pipelines grows progressively. This is very efficient as long as conditional branches do not initiate pipeline flushes. Once a long instruction pipeline is flushed there is a significant delay until it is refilled. Most of the present systems use branch prediction. The CPU normally assumes that if-branches will always be executed. It starts processing the if-branch before actually checking the outcome of the if-clause. If the if-clause returns false, the else-branch has to be executed. This means that the CPU flushes the pipeline and refills it with the else-branch. This is very time consuming.

Using a bricked volume layout one will encounter this problem. The addressing of data in a bricked volume layout is more costly than in a linear volume layout. To address one data element, one has to address the brick itself and the element within the brick. In contrast to this addressing scheme, a linear volume can be seen as one large brick. To address one sample it is enough to compute just one offset. In algorithms like volume raycasting, which need to address a certain neighborhood of data in each processing step, the computation of two offsets instead of one has quite some performance impact. To avoid this performance penalty, one can construct an if-else statement. The if-clause consists of checking, if the needed data elements can be addressed within one brick. If the outcome is true, the data elements can be addressed as fast as in a linear volume. If the outcome is false, the costly address calculations have to be done. On the one hand this simplifies address calculation, but on the other hand the involved if-else statement incurs pipeline flushes. In the following we take a look at this problem.

For raycasting, one can distinguish two major neighborhood access patterns. One is for re-sampling. The other one is for gradient computation. The latter will be solved generally for a 26-connected neighborhood access pattern. For the re-sampling computation the eight surrounding samples are needed. The necessary address computations in a linear volume layout are:

```
$\mathrm{SampleOffset}_{i,j,k}$          $\rightarrow \mathrm{i + j \cdot D_x + k \cdot D_x \cdot D_y}$
$\mathrm{SampleOffset}_{i+1,j,k}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1$
$\mathrm{SampleOffset}_{i,j+1,k}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{D}_x$
$\mathrm{SampleOffset}_{i+1,j+1,k}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{D}_x$
$\mathrm{SampleOffset}_{i,j,k+1}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{D}_x \cdot \mathrm{D}_y$
$\mathrm{SampleOffset}_{i+1,j,k+1}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{D}_x \cdot \mathrm{D}_y$
$\mathrm{SampleOffset}_{i,j+1,k+1}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{D}_x + \mathrm{D}_x \cdot \mathrm{D}_y$
$\mathrm{SampleOffset}_{i+1,j+1,k+1}$    $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{D}_x + \mathrm{D}_x \cdot \mathrm{D}_y$
```

Thereby $\mathrm{D}_{\{x,y,z\}}$ define the volume dimensions and i,j,k the integer parts of the current re-sample position in 3D. This addressing scheme is very efficient. Once the lower left sample is determined the other needed samples can be accessed just by adding an offset. In contrast to the linear volume addressing, the brick volume addressing is:

```
**if**((i' $< \mathrm{BD}_x - 1$) **and** (j' $< \mathrm{BD}_y - 1$) **and** (k' $< \mathrm{BD}_z - 1$))
{
$\mathrm{SampleOffset}_{i,j,k}$          $\rightarrow \mathrm{i' + j' \cdot BD_x + k' \cdot BD_x \cdot BD_y}$
$\mathrm{SampleOffset}_{i+1,j,k}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1$
$\mathrm{SampleOffset}_{i,j+1,k}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{BD}_x$
$\mathrm{SampleOffset}_{i+1,j+1,k}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{BD}_x$
$\mathrm{SampleOffset}_{i,j,k+1}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{BD}_x \cdot \mathrm{BD}_y$
$\mathrm{SampleOffset}_{i+1,j,k+1}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{BD}_x \cdot \mathrm{BD}_y$
$\mathrm{SampleOffset}_{i,j+1,k+1}$      $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{BD}_x + \mathrm{BD}_x \cdot \mathrm{BD}_y$
$\mathrm{SampleOffset}_{i+1,j+1,k+1}$    $\rightarrow \mathrm{SampleOffset}_{i,j,k} + 1 + \mathrm{BD}_x + \mathrm{BD}_x \cdot \mathrm{BD}_y$
}
**else**
{
$\mathrm{SampleOffset}_{i,j,k}$          $\rightarrow \mathrm{i' + j' \cdot BD_x + k' \cdot BD_x \cdot BD_y}$
$\mathrm{SampleOffset}_{i+1,j,k}$        $\rightarrow$ **ComputeOffset**$(\mathrm{i+1, j, k})$
$\mathrm{SampleOffset}_{i,j+1,k}$        $\rightarrow$ **ComputeOffset**$(\mathrm{i, j+1, k})$
$\mathrm{SampleOffset}_{i+1,j+1,k}$      $\rightarrow$ **ComputeOffset**$(\mathrm{i+1, j+1, k})$
$\mathrm{SampleOffset}_{i,j,k+1}$        $\rightarrow$ **ComputeOffset**$(\mathrm{i, j, k+1})$
$\mathrm{SampleOffset}_{i+1,j,k+1}$      $\rightarrow$ **ComputeOffset**$(\mathrm{i+1, j, k+1})$
$\mathrm{SampleOffset}_{i,j+1,k+1}$      $\rightarrow$ **ComputeOffset**$(\mathrm{i, j+1, k+1})$
$\mathrm{SampleOffset}_{i+1,j+1,k+1}$    $\rightarrow$ **ComputeOffset**$(\mathrm{i+1, j+1, k+1})$
}
**ComputeOffset**(i,j,k)                 $\rightarrow \mathrm{BlkOffset}_{i,j,k} \cdot (\mathrm{BD}_x \cdot \mathrm{BD}_y \cdot \mathrm{BD}_z) +$
                                         $\mathrm{OffsetWithinBlk}_{i,j,k}$
$\mathrm{BlkOffset}_{i,j,k}$             $\rightarrow (\mathrm{i'' + j'' \cdot BVD_x + k'' \cdot (BVD_x \cdot BVD_y)})$
$\mathrm{OffsetWithinBlk}_{i,j,k}$       $\rightarrow (\mathrm{i' + j' \cdot BD_x + k' \cdot (BD_x \cdot BD_y)})$
```

Thereby $\mathrm{BD}_{\{x,y,z\}}$ define the brick dimensions, $\mathrm{D}_{\{x,y,z\}}$ define the volume dimensions. $\mathrm{BVD}_{\{x,y,z\}}$ denote the brick volume dimensions defined by $\mathrm{BVD}_{\{x,y,z\}} = (\mathrm{D}_{\{x,y,z\}}/\mathrm{BD}_{\{x,y,z\}})$, i, j, and k are the integer parts of the current re-sample 3D-position, i', j', k' are defined by $i' = (i \bmod \mathrm{BD}_x)$, $j' = (j \bmod \mathrm{BD}_y)$, and $k' = (k \bmod \mathrm{BD}_z)$, and i'', j'', k'' are defined by $i'' = (i \operatorname{div} \mathrm{BD}_x)$, $j'' = (j \operatorname{div} \mathrm{BD}_y)$, and $k'' = (k \operatorname{div} \mathrm{BD}_z)$.

To avoid the costly if-else statement and the expensive address computations, one can create a look-up table to address all the needed samples.

The straightforward approach would be to create a look-up table for each possible sample position in a brick. Since our optimal brick size is 32^3, this would mean that we would need 32^3 different look-up tables to address the neighboring samples. In the re-sampling case, 7 neighbors need to be addressed; accordingly the size of the look-up tables would be $32^3 * 7 * 4$ Bytes = 896 KB (4 Bytes per offset). In the gradient computation case we need to address even more neighbors: 26 neighbors need to be addressed, which leads to a size of $32^3 * 26 * 4$ Bytes = 3.25 MByte (4 Bytes per offset) in total. Such a huge size for a look-up table is not preferable, due to the limited size of cache. However the addressing of such a look-up table would be straightforward, because the indexes in the look-up table would be the corresponding offsets of the current sample position, assuming the offset is given relative to the brick memory address.

We developed a more efficient approach. We differentiate the possible sample positions by the locations of the needed neighboring samples. The first sample location (i,j,k) is defined by the integer parts of the current re-sample position. The access pattern of adjacent samples during re-sampling is defined by accessing samples to the right, top, and back. The samples of a brick can be subdivided into subsets. For the largest subset the seven adjacent samples of a sample (i,j,k) lie within the same brick. The other subsets are defined by samples (i,j,k) on the border of the current brick. The adjacent samples lie partially or completely within neighboring bricks. These other subsets are defined by the needed neighbor bricks to access all seven adjacent samples. The 2D case is illustrated in Fig. 2(a). Only samples to the right and to the top are needed, thus there are just four cases. Basically if the sample (i,j) lies on one or two of the brick faces (top-, and right-face), neighboring bricks are needed. This can be mapped straightforwardly to the 3D case, by also taking into account the back-face. The eight occurring cases are shown in Table 2.

As mentioned before the bricks itself are stored in *xyz*-order, therefore the necessary offsets for the eight neighboring samples can be pre-computed and stored in a look-up table. Furthermore, the look-up table is for each brick the same. The look-up table contains $8 \times 7 = 56$ offsets. We have eight cases, and for each sample (i,j,k) we need the offsets to its seven adjacent samples. The seven neighbors are accessed relative to the sample (i,j,k). Since each offset consists of four bytes the table size is 224 bytes. This is an improvement of a factor of 4 compared to the straightforward solution.

By compressing the look-up tables in this way the index computations for the look-up table access become more difficult. It can be achieved efficiently if the brick dimensions are a power of two, and a power of two apart. The second constraint can be removed by introducing a simple shift operation to virtually hold the constraint. To exemplify the algorithm we assume that the brick dimensions are $32 \times 16 \times 8$. The input of the look-up table addressing function is the sample position (i,j,k). As first step the brick offset part from i, j, and k is extracted by adding the corresponding

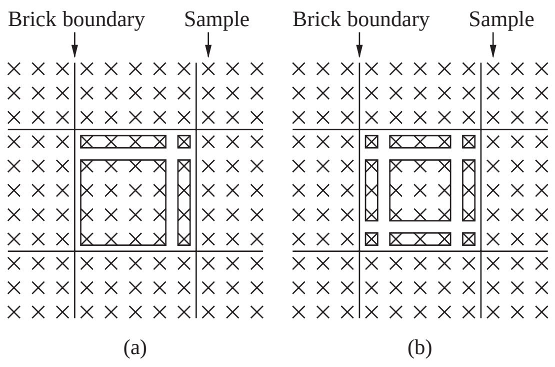


Fig. 2. Sample position (i,j,k) is defined by the integer parts of the re-sample position. The sample positions (i,j,k) of a brick are subdivided into subsets. The membership depends on the location of the adjacent samples. They are either in the same brick or in one of the neighboring bricks. (a) Re-sampling: four areas, because only samples to the right and to the top are accessed. (b) Gradient computation: nine subsets, because samples in every direction are accessed.

Table 2
The eight neighbor brick constellations

Case	$i \in$	$j \in$	$k \in$
0	$\{0, \ldots, BD_x - 2\}$	$\{0, \ldots, BD_y - 2\}$	$\{0, \ldots, BD_z - 2\}$
1	$\{0, \ldots, BD_x - 2\}$	$\{0, \ldots, BD_y - 2\}$	$BD_z - 1$
2	$\{0, \ldots, BD_x - 2\}$	$BD_y - 1$	$\{0, \ldots, BD_z - 2\}$
3	$\{0, \ldots, BD_x - 2\}$	$BD_y - 1$	$BD_z - 1$
4	$BD_x - 1$	$\{0, \ldots, BD_y - 2\}$	$\{0, \ldots, BD_z - 2\}$
5	$BD_x - 1$	$\{0, \ldots, BD_y - 2\}$	$BD_z - 1$
6	$BD_x - 1$	$BD_y - 1$	$\{0, \ldots, BD_z - 2\}$
7	$BD_x - 1$	$BD_y - 1$	$BD_z - 1$

$BD_{\{x,y,z\}} - 1$. The result can be seen in Table 3 second column. The next step is to increase all by one. By this operation the current maximal possible value $BD_{\{x,y,z\}} - 1$ is moved to $BD_{\{x,y,z\}}$. All the other possible values stay within the range $[1, BD_{\{x,y,z\}} - 1]$. Then a conjunction with the resulting value and the complement of $BD_{\{x,y,z\}} - 1$ is performed. After this operation the input values are mapped to $\{0, BD_{\{x,y,z\}}\}$, as shown in Table 3, column four. The last and final step is to add the three values and divide the result by the minimum of the three brick-dimensions $BD_{\{x,y,z\}}$, which maps the result into the range $[0, 7]$. This last division can be exchanged by a shift operation. The final algorithm for a $32 \times 16 \times 8$ brick is:

```
$\mathrm{SampleOffset}_{i,j,k}$              $\rightarrow$ **ComputeOffset**(i,j,k)
Index                                       $\rightarrow$ ((((i&0×1F) + 1)&0xE0) +
                                            $\rightarrow$ (((j&0×0F) + 1)&0xF0) +
                                            $\rightarrow$ (((k&0×07) + 1)&0xF8)) $\gg$ 3
$\mathrm{SampleOffset}_{i+1,j,k}$            $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][0]}$
$\mathrm{SampleOffset}_{i,j+1,k}$            $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][1]}$
$\mathrm{SampleOffset}_{i+1,j+1,k}$          $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][2]}$
$\mathrm{SampleOffset}_{i,j,k+1}$            $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][3]}$
$\mathrm{SampleOffset}_{i+1,j,k+1}$          $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][4]}$
$\mathrm{SampleOffset}_{i,j+1,k+1}$          $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][5]}$
$\mathrm{SampleOffset}_{i+1,j+1,k+1}$        $\rightarrow \mathrm{SampleOffset}_{i,j,k} + \mathrm{Lut[Index][6]}$
```

The *ComputeOffset* step can be simplified, to only the offset calculation within one brick; this is possible as the processing is done brick-wise; therefore the brick-offset remains constant, while processing one brick.

Compared to the if-else solution which has the costly computation of two offsets in the else branch, we get a speed up of about 30%. The benefit varies, depending on the brick dimensions. For a $32 \times 32 \times 32$ brick size the else-branch has to be executed in 10% of the cases and for a $16 \times 16 \times 16$ brick size in 18% of the cases. With larger brick-sizes the percentage of the else-branch executions is smaller and therefore also the benefit decreases. But the focus is on small brick-sizes anyway. For these sizes we reduced the overhead significantly. The other important benefit is, that it does not matter anymore where in the brick adjacent samples are

Table 3
Look-up table addressing for re-sampling. Thereby $\mathrm{BD}_{x,y,z} = 32, 16, 8$

Case	$i\&$ (BD_x-1)	$j\&$ (BD_y-1)	$k\&$ (BD_z-1)	$i+1$	$j+1$	$k+1$	$i\&$ $\sim(\mathrm{BD}_x-1)$	$j\&$ $\sim(\mathrm{BD}_y-1)$	$k\&$ $\sim(\mathrm{BD}_z-1)$	$(i+j+k)/$ $\mathrm{Min}(\mathrm{BD}_x,\mathrm{BD}_y,\mathrm{BD}_z)$
0	0–30	0–14	0–6	1–31	1–15	1–7	0	0	0	0
1	0–30	0–14	7	1–31	1–15	8	0	0	8	1
2	0–30	15	0–6	1–31	16	1–7	0	16	0	2
3	0–30	15	7	1–31	16	8	0	16	8	3
4	31	0–14	0–6	32	1–15	1–7	32	0	0	4
5	31	0–14	7	32	1–15	8	32	0	8	5
6	31	15	0–6	32	16	1–7	32	16	0	6
7	31	15	7	32	16	8	32	16	8	7

accessed. It is always performed with constant computational time.

A similar approach can be done for the gradient computation. We present a general solution for a 26-connected neighborhood. Here we can, analogous to the re-sample case, distinguish 27 cases. The 2D case is illustrated in Fig. 2(b). Depending on the position of sample (i, j, k) a brick is subdivided into 27 subsets. In contrast to the re-sample situation, additionally we have to handle sample positions on the bottom-, left-, and front faces.

The first step is to extract the brick offset, by adding $\mathrm{BD}_{\{x,y,z\}} - 1$ as shown in Table 4, second column. Then we subtract one, and with $\mathrm{BD}_{\{x,y,z\}} + \mathrm{BD}_{\{x,y,z\}} - 1$, to separate the case if one or more components are zero. In other words zero is mapped to $(2 \cdot \mathrm{BD}_{\{x,y,z\}} - 1)$ (Table 4, third column). All the other values stay within the range $\{0, \ldots, \mathrm{BD}_{\{x,y,z\}} - 2\}$. The other case which has to be separated is the case if one or more of the components are $\mathrm{BD}_{\{x,y,z\}} - 1$. This can be done by adding one, after the previous minus one operation is undone by a disjunction with 1, without loosing the separation of the zero case. The result can be seen in Table 4, third column. Now all the cases are mapped to $\{0, 1, 2\}$ to obtain a ternary-system. This is done by dividing the components by the corresponding brick-dimensions. These divisions can be exchanged by fast shift operations. The last and final step is then $9 \cdot i + 3 \cdot j + k$ to get unique values in the range of $[0, 26]$. The final look-up table index computation for a $32 \times 16 \times 8$ brick is:

```
i'     →((i & 0x1F) −1) & 0x3F
j'     →((j & 0x0F) −1) & 0x1F
k'     →((k & 0x07) −1) & 0x0F
i''    →((i' | 0x01)+1)≫ 5
j''    →((j' | 0x01)+1)≫ 4
k''    →((k' | 0x01)+1)≫ 3
Index  →(i''·9+j''·3+k'')
```

The benefit is a 40% speedup. The index computation is more costly compared to the re-sample lookup table indexing computation. However, the percentage where the else-branch has to be executed nearly doubled. Therefore the more costly index computation is compensated by the higher percentage of costly cases. What we did not mentioned so far is the size of the look-up table. It is 27 cases $\times 26$ offsets $\times 4$ byte per offset $=$ 2808 Bytes. This can be reduced by a factor of two due to symmetry reasons. Therefore we have a very small look-up table of 1404 Bytes. This is an improvement of approximately a factor of 2427 compared to the straightforward solution. Thus, the re-sample look-up table and the 26-connected neighborhood look-up table, fit into 2 KB.

We assumed that bricks are stored linearly. This simplified the explanation of our addressing scheme. However, storing the bricks at arbitrary locations in memory is also possible. It requires creating a specific look-up table for each brick. The base structure of the address look-up tables and their indexing remain the same; only the stored offsets change according to the memory locations of the adjacent neighboring bricks. This possibility enables the exploitation of different bricks arrangements, such as arrangements based on space filling curves, to improve the spatial locality. Storing an address look-up table for each brick requires small additional storage of $(1/65536) \times (2808 + 224) \times 100 \approx 4.63\%$ per brick. The brick size in our case is $32^3 \times 2$ byte $= 65536$ byte, the re-sample look-up table size is 224 byte, and the 26-neighbor address look-up table size is 2808 byte. The symmetry of the 26-neighbor address look-up tables cannot be exploited, due to the arbitrary brick arrangement requirement.

Another possible option to simplify the addressing is to inflate each brick by an additional border of samples. However, such a solution increases the overall memory usage considerably. Using a brick size of $32 \times 32 \times 32$ increases the total memory usage of the volume data by approximately 20%. This is an inefficient usage of memory resources and the storage redundancy reduces the effective memory bandwidth. In contrast to that our approach has a memory usage increase of 4.63% per brick if an arbitrary brick arrangement is permitted. No additional memory is required for a linear brick

Table 4
Look-up table addressing for 26-connected neighborhood. Thereby $\mathrm{BD}_{x,y,z} = 32, 16, 8$

Case	$i\&$ $(\mathrm{BD}_x - 1)$	$j\&$ $(\mathrm{BD}_y - 1)$	$k\&$ $(\mathrm{BD}_z - 1)$	$i - 1\&$ $(\mathrm{BD}_x + \mathrm{BD}_x - 1)$	$j - 1\&$ $(\mathrm{BD}_y + \mathrm{BD}_y - 1)$	$k - 1\&$ $(\mathrm{BD}_z + \mathrm{BD}_z - 1)$	$i \mid 1 + 1$	$j \mid 1 + 1$	$k \mid 1 + 1$	i/BD_x	j/BD_y	k/BD_z	$9 \cdot i + 3 \cdot i + k$
0	1–30	1–14	1–6	0–29	0–13	0–5	2–30	2–14	2–6	0	0	0	0
1	1–30	1–14	7	0–29	0–13	6	2–30	2–14	8	0	0	1	1
2	1–30	1–14	0	0–29	0–13	15	2–30	2–14	16	0	0	2	2
3	1–30	15	1–6	0–29	14	0–5	2–30	16	2–6	0	1	0	3
4	1–30	15	7	0–29	14	6	2–30	16	8	0	1	1	4
5	1–30	15	0	0–29	14	15	2–30	16	16	0	1	2	5
6	1–30	0	1–6	0–29	31	0–5	2–30	32	2–6	0	2	0	6
7	1–30	0	7	0–29	31	6	2–30	32	8	0	2	1	7
8	1–30	0	0	0–29	31	15	2–30	32	16	0	2	2	8
9	31	1–14	1–6	30	0–13	0–5	32	2–14	2–6	1	0	0	9
10	31	1–14	7	30	0–13	6	32	2–14	8	1	0	1	10
11	31	1–14	0	30	0–13	15	32	2–14	16	1	0	2	11
12	31	15	1–6	30	14	0–5	32	16	2–6	1	1	0	12
13	31	15	7	30	14	6	32	16	8	1	1	1	13
14	31	15	0	30	14	15	32	16	16	1	1	2	14
15	31	0	1–6	30	31	0–5	32	32	2–6	1	2	0	15
16	31	0	7	30	31	6	32	32	8	1	2	1	16
17	31	0	0	30	31	15	32	32	16	1	2	2	17
18	0	1–14	1–6	63	0–13	0–5	64	2–14	2–6	2	0	0	18
19	0	1–14	7	63	0–13	6	64	2–14	8	2	0	1	19
20	0	1–14	0	63	0–13	15	64	2–14	16	2	0	2	20
21	0	15	1–6	63	14	0–5	64	16	2–6	2	1	0	21
22	0	15	7	63	14	6	64	16	8	2	1	1	22
23	0	15	0	63	14	15	64	16	16	2	1	2	23
24	0	0	1–6	63	31	0–5	64	32	2–6	2	2	0	24
25	0	0	7	63	31	6	64	32	8	2	2	1	25
26	0	0	0	63	31	15	64	32	16	2	2	2	26

arrangement, as all bricks share one global address look-up table. In conclusion we presented a very efficient approach to access neighboring samples within a brick based volume layout by using a small look-up table. Furthermore, we presented a refined index computation to access the look-up tables very efficiently.

3.4. Thread-level parallelism (hyper-threading technology)

So far CPU designers tried to improve the CPU performance mainly by increasing the clock-rate. Today's main processors perform at 3 GHz resulting in 0.33 ns per clock-cycle. Achieving higher rates becomes more and more difficult due to physical laws and manufacture costs. Other directions to increase CPU performance were explored. The Pentium CPU was the first to allow the parallel execution of several instructions per clock-cycle. However, this feature was insufficient, because normally there are not enough sequential instructions which can be performed in parallel. To overcome this issue an out-of-order execution unit was introduced. This unit reorders the instruction stream such that the CPU can execute more instructions in parallel. This concept is called instruction level parallelism. At first sight this is a very efficient solution, but studies have shown that in a typical application at most 2.5 instructions can be found to be executed simultaneously.

Thus, there are still unused execution resources on the CPU. To use them, they introduced hyper-threading technology for commodity CPUs to exploit thread-level parallelism. With this technology the CPU designers go one step further. Additionally to the instruction level parallelism, thread level parallelism (TLP) is introduced to identify even more instructions for parallel execution. Before, the out-of-order execution unit could choose from an instruction buffer of only one thread. Now, this buffer contains instructions of two threads which obviously increases the likelihood of finding data-independent instructions. This technology makes a single *physical* processor appear as two *logical* processors. It just duplicates the architectural state, while the physical execution resources and caches are shared (see Fig. 3). In other words the CPU is capable of holding two thread contexts at the same time. The two threads are executed simultaneously on the same execution units, using the same caches. If one thread stalls due to a cache miss, the other one uses the idle execution resources. More information on hyper-threading technology can be found in [11–13].

In Section 3.2 we presented a bricked memory volume layout with a highly optimized addressing scheme. This layout is now the base for our volume raycasting system. The main idea to do raycasting on a bricked volume layout is to have data working-sets which can be shared

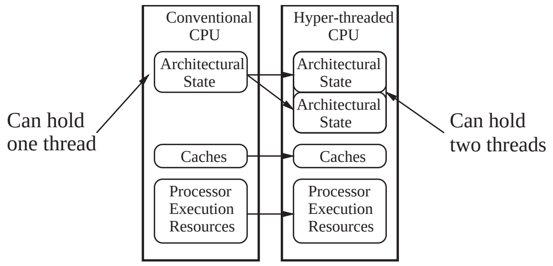


Fig. 3. Hyper-threading technology duplicates the architectural state of the physical processor, providing two *logical* processors.

between two hyper-threads. This is very important since hyper-threads share caches.

3.5. Volume raycasting algorithm work-flow utilizing TLP

To get optimal cache coherence, high CPU utilization, and the ability to do efficient threading we based our system on Law's and Yagel's [9] raycasting method. They proposed a parallel raycasting algorithm for a massively parallel processing system (Cray T3D Supercomputer). The data was subdivided into small units and then evenly distributed among the processors, such that optimal cache coherence was achieved.

This distribution scheme can still be used on current multi-processor systems; however it cannot be used straightforward for a hyper-threaded system. Therefore we extend their distribution scheme to support logical CPUs within one physical CPU. The main difference is, that logical CPUs within one physical CPU need to operate on the same data to be efficient.

The workflow of the algorithm is as follows: Our algorithm is based on the previous described bricked volume layout. The used optimal brick size is $32 \times 32 \times 32$, see Section 3.2. Each brick contains data-structures for high-level optimizations and a reference to a list of rays to process. The bricks themselves are stored in *xyz*-order. The workflow of the algorithm shown in Fig. 4.

Initially a list of bricks is created. It is sorted by the traversal order of the rays. Therefore each brick has to be processed only once. That this has to be done only once for the eight view octants, was shown by Law and Yagel [9]. Each brick has initially an empty list of rays. In the pre-processing phase all viewing rays are assigned to those bricks through which they enter the volume first. During volume raycasting, each of these bricks is processed until all the rays enter subsequent bricks. If a ray enters a subsequent brick, it is removed from the current brick and assigned to the subsequent one. Subsequent bricks, which now contain the rays, are processed in the same manner. By this mechanism the rays are completely carried through the volume as

```
Initialization:
1.) Create ordered list of bricks to process list.

Preprocessing:
2.) Add rays to bricks which are hit first.

Raycasting:
for (all bricks "b")
   while (brick "b" contains rays to process)
      for (all active rays "r" of brick "b")
         1.)  Re-sampling at position of ray r.
         2.)  Gradient computation at position of ray r.
         3.)  Shading at position of ray r.
         4.)  Compositing at position of ray r.
         5.)  Advance ray r.
         6.)  if(ray enters subsequent brick)
                 (i) Remove ray from current brick.
                 (ii) Assign ray to the subsequent brick.
```

Fig. 4. Brick-wise raycasting algorithm.

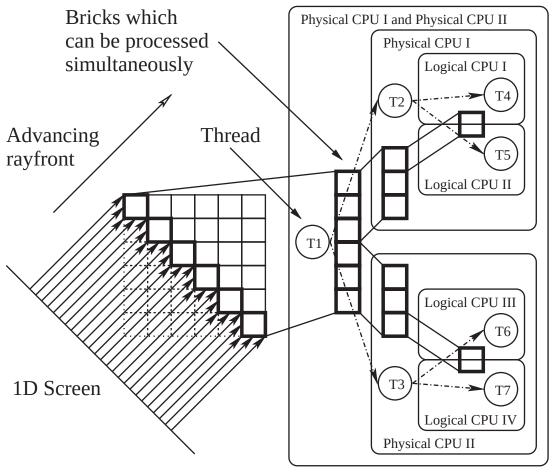


Fig. 5. Volume raycasting system exploiting thread-level parallelism speedup.

soon as all bricks are processed. These bricks basically define the data working-sets which were mentioned in Section 3.1.

The workflow of the volume raycasting system exploiting TLP on a system with two physical CPUs supporting hyper-threading technology, illustrated in Fig. 5.

In the beginning seven threads, T1, …, T7, are started. T1 is responsible for all the preprocessing. In particular it has to be assigned the rays to those bricks through which the rays enter the volume first. Then it has to choose the lists of bricks which can be processed simultaneously, with respect to the eight to distinguish viewing directions. Each list is subdivided evenly by T1 and sent to T2 and T3. After a list is sent, T1 sleeps until its slaves are finished. Then it sends the next list to process, and so on. T2 sends one brick after the other to T4 and T5. T3 sends one brick after the other to T6 and T7. After a brick is send, they sleep until their slaves are finished. Then they send the next brick to process, and so on. T4, T5, T6, and T7 perform the actual raycasting. T4 and T5 simultaneously process one brick, and T6 and T7 simultaneously process one brick. By this mechanism all bricks are processed in the correct order.

3.5.1. Implementation issues

No special parallel programming library was used, to keep complete control over the thread execution and synchronization flow. The threads are created once during startup, according to the number of physical and logical CPUs and synchronized by light-weighted events. Expensive thread creation is avoided and thread context switches are kept low. For optimal CPU utilization a CPU specific compiler was employed. Full optimization was enabled performing interprocedural optimizations and inlining across multiple source files.

3.6. Results

Test system specification: Dual Pentium Xeon 2.4 GHz, 512 KB level-2 cache, 8 KB level-1 data cache, 1 GB Rambus memory, and a GeForce IV graphics-card. The graphics card capabilities are only used to display the final image. Our system is able to force threads on specific physical and logical CPUs. By following this mechanism we forced it to run only on one physical CPU using both logical CPUs. Benchmarks were performed using several different data sets and transfer functions. Fig. 6 shows the results of an exemplary test run using a CTA scan of human head with enhanced venous system ($512 \times 512 \times 333$, 12 bit). Different transfer-functions were specified in order to achieve high work loads. The images (512×512) from left to right show renderings with progressively less translucent transfer function settings. Measured render timings are: (a) 10.1 s, (b) 5.8 s, (c) 1.7 s, and (d) 1.3 s. Non-translucent transfer functions lead to frame rates of up to 2.5 fps for this particular data set. All test runs consistently showed the same speedup factors.

The achieved thread-level parallelism speedup is shown in Fig. 7. Testing thread-level parallelism on only one CPU showed an average speedup of 30%. While changing the viewing direction, the speedup varies from 25% to 35%, due to different transfer patterns between the level-1 and the level-2 cache. Whether hyper-threading is enabled or disabled adding a second CPU approximately reduces the computational time by 50%. This shows that our thread-level parallelism scheme scales very well on multiprocessor machines.

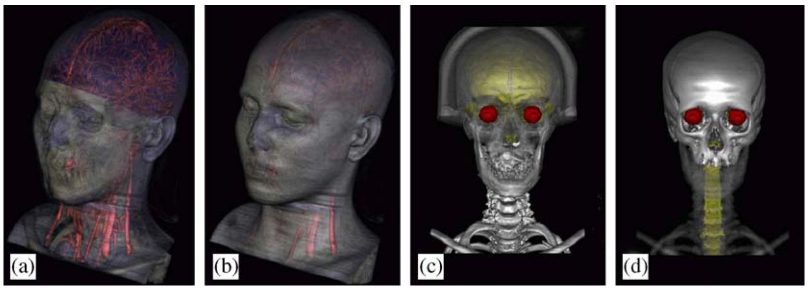
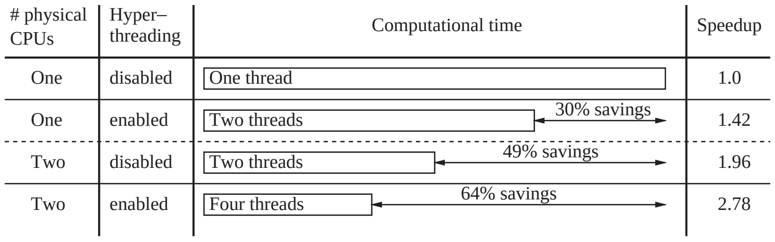


Fig. 6. CTA scan of human head with enhanced venous system.



# physical CPUs	Hyper–threading	Computational time	Speedup
One	disabled	One thread	1.0
One	enabled	Two threads — 30% savings	1.42
Two	disabled	Two threads — 49% savings	1.96
Two	enabled	Four threads — 64% savings	2.78

Fig. 7. Thread-level parallelism speedup.

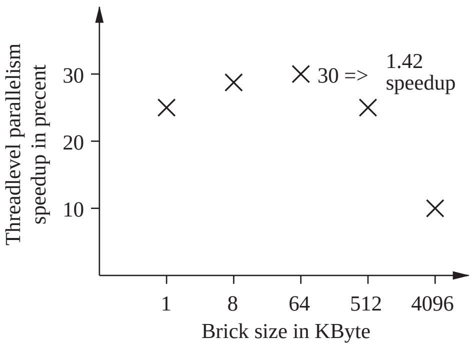


Fig. 8. Thread-level parallelism speedup for different brick sizes.

Also the hyper-threading benefit of approximately 30% is maintained if the second hyper-threaded CPU is enabled.

Fig. 8 shows the TLP speedup according to different brick sizes. The speedup significantly decreases with larger brick sizes. Once the level-2 cache size is exceeded, the two threads have to request data from main memory. Therefore the CPU execution units are less utilized. Very small brick sizes suffer from a different problem. The data fits almost into the level-1 cache. Therefore one thread can utilize the execution units more efficiently, and the second thread is idle during this time. But the overall disadvantage is the inefficient usage of the level-2 cache. The optimal speedup $1/((100-30)/100) \approx 1.42$ is achieved with 64 KB ($32 \times 32 \times 32$). This is also the optimal brick size for the bricked volume memory layout, see Section 3.2.

4. Discussion

Efficient use of hardware resources for basic graphics algorithms is very important. It is a known fact, that TLP can increase performance by a factor of 30% for very well parallelized algorithms. However, very well parallelized in this sense means that threads have to operate on coherent data. Our tests have shown that large brick-sizes lead to very low TLP performance benefits. This is due to the fact, that using large brick-sizes leads to a low cache hit rate and therefore to execution stalls because of expensive main memory requests. For example, by just splitting the image plane in half and assigning each half to a hyper-thread, you will encounter a performance decrease instead of increase. This is, because the two threads constantly request data from different memory locations. This leads to enormous cache thrashing, since the two threads share caches.

The bricking speedup is about 2.8. However, it is important to note that this speedup factor characterizes the improvement in traversal, re-sampling and gradient computation. These are the components of the system which are directly affected by the accelerated memory access. Other parts, such as compositing and shading do not benefit from the presented optimizations. In our system, however, these parts only play a minor role in overall performance. We use front-to-back compositing and phong shading with two light sources.

Our experiments have shown that with the optimal brick-size of $32 \times 32 \times 32$ a speedup factor of 2.8 is achieved. Enabling TLP results in an additional speedup of 1.42. The combined speedup is $2.8^*1.42 \approx 4.0$. High-level optimizations, such as empty space skipping or early ray termination, did not influence this speedup factor. Our efficient addressing scheme considerably reduces the cost of addressing in a bricked volume layout. Its influence on the overall performance gains depends on the filter support size used for re-sampling and gradient estimation as well as on the complexity of the remaining calculation such as shading and compositing.

5. Conclusion

We have presented a raycasting system utilizing TLP. We utilized a bricked volume layout in order to design a highly efficient threading scheme which maximizes the benefits of TLP. The high cache coherency inherently present in a bricked volume layout combined with the two refined addressing schemes significantly reduced the costs for re-sampling and gradient computation.

For efficient use of TLP we introduced a multi-threading scheme, such that two threads running on one physical CPU simultaneously process one data brick. Processing the same data brick simultaneously with both hyper-threads is essential for exploiting this technology. The results have proven that inefficient CPU utilization can be significantly reduced by using hyper-threading technology. The realization of the system showed that using this new technology is not straightforward. Systems have to be adapted to take advantage of this architecture. Most of today's used multi-threaded systems have to be redesigned. By just starting more threads one can encounter significant performance decrease instead of an increase. This is due to the fact, that hyper-threads share caches.

We achieved a significant speedup with our new addressing method in a bricked volume layout. The new addressing scheme can be used for any volume processing algorithm, which has to address adjacent samples. The results showed that conditional branches have quite some performance impact, due to the growing length of the CPU pipeline.

In conclusion, we have shown that advanced low-level optimizations lead to efficient CPU utilization and a significant speedup factor of 4.0.

Acknowledgements

The work presented in this publication has been funded by the AD_APT project (FFF-804544). AD_APT is supported by *Tiani Medgraph*, Vienna (http://www.tiani.com), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See http://www.cg.tuwien.ac.at/research/vis/adapt for further information on this project.

References

[1] Dachille F, Kreeger K, Chen B, Bitter I, Kaufmann A. High quality volume rendering using texture mapping hardware. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1998. p. 69–76.

[2] Pfister H, Hardenbergh J, Knittel J, Lauer H, Seiler L. The VolumePro real-time ray-casting system. In: SIGGRAPH, 1999. p. 251–60.

[3] Zuiderveld KJ, Koning AHJ, Viergever MA. Acceleration of ray-casting using 3d distance transforms. In: Proceeding of Visualization in Biomedical Computing, 1992. p. 324–35.

[4] Westermann R, Ertl T. Efficiently using graphics hardware in volume rendering applications. In: SIGGRAPH, 1998. p. 169–77.

[5] Meissner M, Grimm S, Strasser W, Packer J, Latimer D. Parallel volume rendering on a single-chip SIMD architecture. In: Symposium on Parallel and Large Data Visualization and Graphics, 2001. p. 107–13.

[6] Mora B, Jessel J, Caubet R. A new object order ray-casting algorithm. In: Proceedings of Visualization, 2002. p. 107–13.

[7] Knittel G. The Ultravis system. In: SIGGRAPH Volume Visualization and Graphics Symposium, 2000. p. 71–8.

[8] Lacroute P, Levoy M. Fast volume rendering using a shear-warp factorization of the viewing transformation. In: SIGGRAPH, 1994. p. 451–8.

[9] Law A, Yagel R. Multi-frame thrashless ray casting with advancing ray-front. In: Proceedings of Graphics Interfaces, 1996. p. 70–7.

[10] Parker S, Shirley P, Livnat Y, Hansen C, Sloan P. Interactive ray tracing for isosurface rendering. In: Proceedings of Visualization, 1998. p. 233–8.

[11] Intel Cooperation. IA-32 Intel Architecture Software Developer's Manual: Volume I, II, III. Intel, Order Number 245470-010, Order Number 245472-010, Order Number 245472-010, 2003.

[12] Intel Cooperation. Intel Pentium 4 and Intel Xeon Processor Optimization, Reference Manual. Intel, Order Number 248966-007, 2003.

[13] Marr DT, et al. Hyper-Threading Technology Architecture and Microarchitecture. Intel, www.intel.com, 2003.