The ULTRAVIS System

Gunter Knittel
Client and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2000-100
July 27th, 2000*

E-mail: knittel@hpl.hp.com

computer graphics, volume rendering, raycasting

This paper describes architecture and implementation of the ULTRAVIS system, a pure software solution for versatile and fast volume rendering. It provides perspective raycasting, tri-linear interpolation, on-the-fly classification using look-up tables, gradient shading (both diffuse and specular reflection), four light sources, and alpha blending. For high frame rates, early ray termination and empty space skipping are implemented. Furthermore, subsampling during motion is provided. The system accepts raw data sets of 8-bit voxels as well as pre-segmented data sets containing up to 16 different materials. For gradient shading, the gradients are precomputed and included in 32-bit voxels. Additionally, the system supports volume animation, i.e., the display of a sequence of data sets.

The system was specifically designed for Pentium III CPUs, and makes extensive use of MMX and Streaming SIMD instructions. It is a multi-threaded application and thus takes advantage of multi - processor platforms. Time-critical portions of the code have been hand-optimized in assembler. As a result, the system can achieve interactive to real-time performance.

ULTRAVIS runs on the Windows NT 4.0 operating system on standard PCs.

* Internal Accession Date Only Approved for External Publication
© Copyright Hewlett-Packard Company 2000

The ULTRAVIS System

Gunter Knittel
Hewlett-Packard Laboratories, Visual Computing Department, knittel@hpl.hp.com

ABSTRACT

This paper describes architecture and implementation of the ULTRAVIS system, a pure software solution for versatile and fast volume rendering. It provides perspective raycasting, tri-linear interpolation, on-the-fly classification using look-up tables, gradient shading (both diffuse and specular reflection), four light sources, and alpha blending. For high frame rates, early ray termination and empty space skipping are implemented. Furthermore, subsampling during motion is provided. The system accepts raw data sets of 8-bit voxels as well as pre-segmented data sets containing up to 16 different materials. For gradient shading, the gradients are precomputed and included in 32-bit voxels. Additionally, the system supports volume animation, i.e., the display of a sequence of data sets.

The system was specifically designed for Pentium III CPUs, and makes extensive use of MMX and Streaming SIMD instructions. It is a multi-threaded application and thus takes advantage of multi-processor platforms. Time-critical portions of the code have been hand-optimized in assembler. As a result, the system can achieve interactive to real-time performance.

ULTRAVIS runs on the Windows NT 4.0 operating system on standard PCs.

CCS Categories and Subject Descriptors: I.3.4 [**Computer Graphics**]: Graphics Utilities - *graphics packages;* I.3.3 [**Computer Graphics**]: Picture/Image Generation - *display algorithms*
Additional Keywords: volume rendering, raycasting

1 INTRODUCTION

It is commonly understood that real-time volume rendering requires special-purpose hardware [13], multi-processor servers [10], [16] or, with some restrictions, 3D texturing hardware [1], [5], [6], [8], [17]. On the other hand, the performance of commodity CPUs is increasing at a tremendous speed. Furthermore, specialized multi-media hardware extensions (e.g., MMX) can be used for many basic volume rendering operations such as tri-linear interpolations. Also, memory costs have decreased so much that all but the largest data sets can be placed into main memory for easy access. Thus, the use of a high-end PC for software-based volume rendering is intriguing.

However, one major obstacle towards high performance remains: the limited memory bandwidth, even more so because volume rendering requires three-dimensional access to the data set, and frequent access to tables. Thus, the use of the on-chip caches decides on the achievable performance.

There have been few attempts to achieve high-speed volume rendering in software on a single workstation or PC. Probably the most prominent one is the *Shear-Warp Factorization* algorithm [9]. In this method, a projection plane is defined which is perpendicular to the largest component of the view vector. The slices of voxels parallel to this plane are sheared according to the observer position. Then, a parallel projection is performed slice by slice in front-to-back order. The resulting distorted image is then corrected (warped) and displayed. Voxels are accessed in scan-line order, giving a good spatial coherence. Furthermore, the voxels are run-length encoded in all three dimensions, as well as the pixels in a scan line. Thus, runs of empty voxels or opaque pixels can be skipped, reducing memory traffic and processing time. The method achieved about 1frame/s for 256^3 data sets on a typical workstation of that time. However, the method requires extensive pre-processing, and is only fast for parallel projections.

In [7], the grayvalues in a 3×2×2 block are reduced to only two values such that mean and variance in the block are preserved. Each voxel is assigned one bit selecting the corresponding value. In case of 8-bit quantities, the data for one block fits into a 32-bit word. All blocks of the volume are compressed redundantly such that all 8 voxels needed for tri-linear interpolation are available after one single memory access. Again, interactive operation in the order of 1frame per second was achieved. However, the method uses lossy compression of the data set, which is unacceptable for many applications.

The data set is rendered into a set of layered depth images in [2], which are blended using 2D texturing hardware. Approximations of new views can quickly be generated by reusing some of the more distant images, provided the new viewpoint is sufficiently close to the previous one. Also, adaptive resolution can be used for the different images, which reduces the total number of samples. The method is used for interactive *volume navigation*, i.e., a viewing frustum of limited depth is placed inside the data set. The authors achieve about 4-6 frames/s using a 300MHz Pentium II CPU, with a resolution of 160×160 pixels and about 120k voxels in the view frustum.

The approach presented here aims at avoiding restrictions or image quality compromises of these kinds. Furthermore, it should be a pure software system (with the exception of hardware-supported bit block transfers) and still achieve interactive operation. An overview of the ULTRAVIS architecture is given in section 2. Sections 3 and 4 detail the implementation and the performance of the system, respectively.

2 THE ULTRAVIS ARCHITECTURE

The block diagram of the different software modules is shown in Figure 1. The ULTRAVIS system was designed as a client/server application, allowing a thin client to connect to a powerful storage/rendering server or server farm. The primary target platform, however, is a single PC. Then, client and server are running on the same machine.

The ULTRAVIS system currently supports four voxel types: 8-bit values ($V_{7..0}$) called *V*, 8-bit values plus identifiers for up to 16 different materials (16 bits, $I_{3..0}V_{7..0}$ with 4 MSBs unused) called *IV*, 8-bit values and gradient components ($Gz_{7..0}Gy_{7..0}Gx_{7..0}V_{7..0}$), called *GV* and the latter including material identifiers ($I_3Gz_{7..1}I_2Gy_{7..1}I_1Gx_{7..1}I_0V_{7..1}$), called *IGV*. Maximum data set dimensions are 256×256×256.

In this paper we will focus on the memory data structure and the rendering operation.

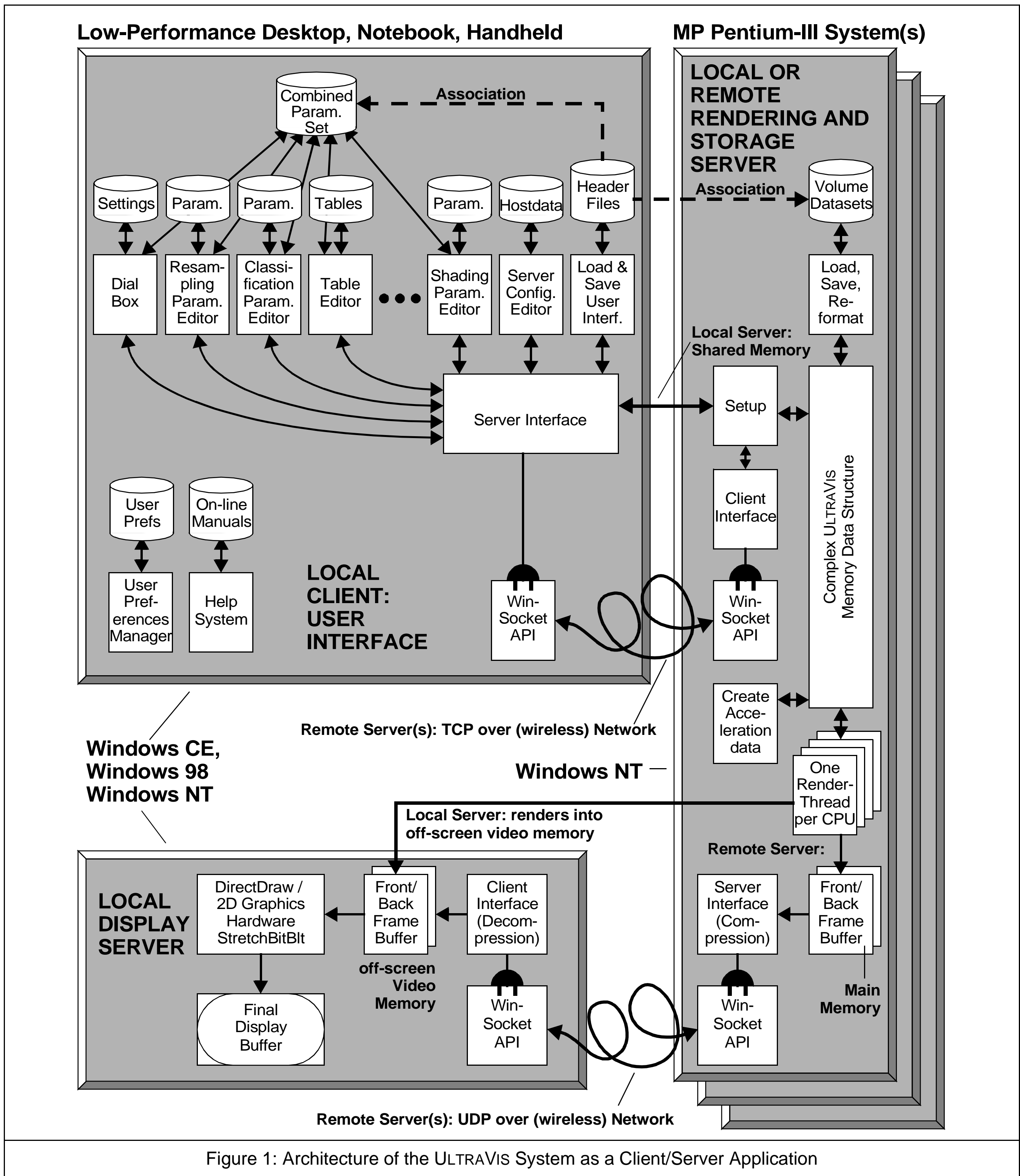

Figure 1: Architecture of the ULTRAVIS System as a Client/Server Application

3 IMPLEMENTATION

The ULTRAVIS system is a collection of well-known and new techniques for fast raycasting. Most of its performance comes from the unique cache optimizations and the use of the SIMD-extensions as described in the following sections.

3.1 SIMD-Extensions of Pentium III CPUs

We can only give a very short description of this technology. Excellent introductions can be found in [12] and [15].

SIMD extensions exist for both integer (MMX) and floating-point (SSE) data types. MMX provides eight 64-bit registers (MM0-MM7), which can hold one 64-bit operand, two 32-bit, four 16-bit or

eight 8-bit operands each. An MMX-instruction is applied to all operands in one or two MMX-registers. Most MMX-instructions execute in one clock (except multiply).

SSE provides a set of eight 128-bit registers (XMM0-XMM7), which can hold four single-precision floating-point operands each. Again, an SSE instruction is applied to all four floating-point operands or operand pairs. As an example, an ADD has a latency of 4 clocks with a throughput of 1 every 2 clocks [4].

3.2 Cache Optimizations and Spread Memory Layout

As stated earlier, the limited memory bandwidth of a PC is the main problem to solve. In our implementation, memory accesses occur due to the following reasons:

- Access to the data set itself. For each raypoint, 8 voxels of 1, 2 or 4 bytes must be read for tri-linear interpolation.
- One access to a color/opacity table per raypoint.
- Additional accesses to rendering parameters such as thresholds, shading coefficients and more per raypoint.

The means to alleviate this problem are the CPU caches, of which we primarily consider the L1 cache. In case of a Pentium III CPU, the L1 data cache is a four-way associative cache with a total capacity of 16KByte. There is a separate instruction cache of the same size [3], [4].

We'll start the discussion with the tables and parameters. Ideally, these data items would be placed into a fast on-chip RAM, under full software control, as it can be done on many DSPs (*Digital Signal Processors*). In the absence of such feature on the Pentium III CPU, we try to mimic a RAM using the L1 cache. To this end, a way must be found to *lock* a data item into the cache once it has been read. This is done using a "spread memory layout" as explained below.

First, let's consider a direct mapped cache as shown in Figure 2. Seen from the cache, the memory is organized as a set of consecutive *pages*, equal in size to the cache. The cache memory itself is organized in *lines* (32 bytes for the Pentium III). Data is transferred to and from the cache in units of complete lines.

The most characteristic feature of a direct mapped cache is that a line in memory at offset *n* can only go into the cache line at offset *n*. Thus, if a program only accesses the gray memory lines in Figure 2, frequent cache line replacements (or *thrashing*) will occur.

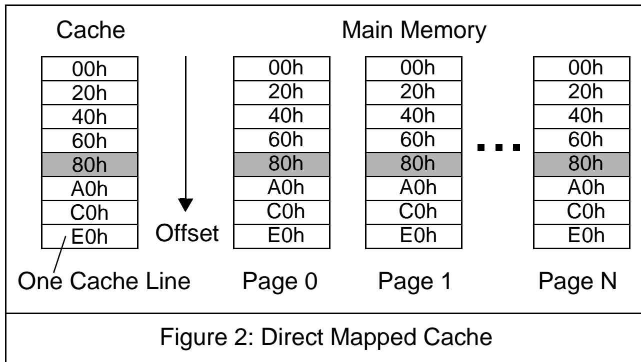

Figure 2: Direct Mapped Cache

N-way associative caches alleviate this disadvantage. Conceptually, a 4-way associative cache can be thought of as a collection of 4 direct mapped caches, as shown in Figure 3.

Then, each memory line has four places to go, and a program can access up to four lines at the same offset before a replacement occurs.

Now let's consider how the volume data set is placed into memory. We allocate memory space for four times the size of the data set, and store the data set such that only the first quarter of each page is used, such as shown in Figure 4a. As a consequence, voxels can only be cached in the first quarter of each cache block. Put differently,

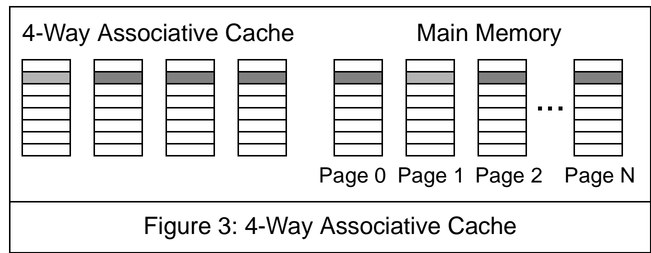

Figure 3: 4-Way Associative Cache

accesses to voxels can never cause replacement of data which happens to be in the other parts of the cache. This can be exploited for frequently accessed tables by placing them into the remaining parts of the first four memory pages, as shown in Figure 4b.

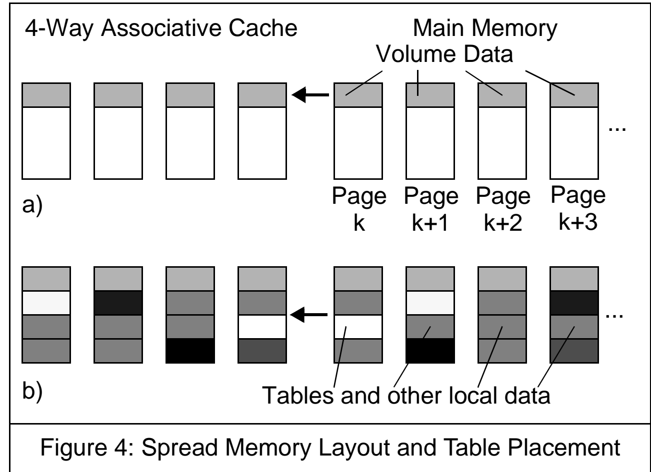

Figure 4: Spread Memory Layout and Table Placement

In this way, frequently accessed data items are virtually locked in the cache for fast access. Up to 12KByte are available for this method on a Pentium III CPU.

However, there are two disadvantages: the cache capacity for the volume data set has essentially been reduced to one fourth, and the required main memory size has been increased fourfold.

Let's first consider the reduced cache size. For a good cache coherence, we use the well-known technique of *tile casting* as shown in Figure 5. The idea is, that voxels fetched for a given raypoint can potentially be reused for many new raypoints in its 3D neighborhood.

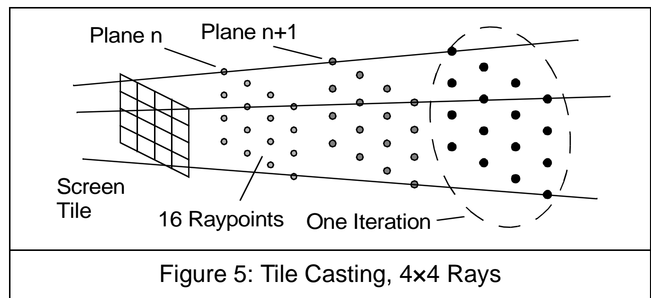

Figure 5: Tile Casting, 4×4 Rays

Still, storing the voxels in memory in a naïve way (for example, *offset* $= Z_{7..0} Y_{7..0} X_{7..0}$) can produce thrashing since the observer position is arbitrary. To avoid this, we use a cubic-interleaved address function, i.e.,

$$(Z_{7\ldots0}, Y_{7\ldots0}, X_{7\ldots0}) \rightarrow Z_7Y_7X_7Z_6Y_6X_6Z_5Y_5X_5\ldots \\ \ldots Z_4Y_4X_4Z_3Y_3 00 X_3Z_2Y_2X_2Z_1Y_1X_1Z_0Y_0X_0 \quad (1)$$

(In (1), we assumed 8-bit voxels. The two zero bits realize the spread memory layout.) Then, any arbitrarily located cubic region of dimension n occupies exactly $n \times n \times n$ different cache locations (n being a power of two, and $n^3 \leq$ cache size). Thus, as long as the bounding cube of the 16 raypoints fits into the cache, all voxels needed for their processing can be cached without mutual replacement. If the raypoint spacing is approximately the grid spacing, an 8×8×8 region can always hold the 4×4 raypoints. In case of 8-bit (32-bit) voxels, this requires as little as 512Byte (2KByte) cache capacity. Thus, even the algorithmically reduced cache capacity is sufficient to achieve a high hit ratio. Another advantage is that cache line fills always load a certain three-dimensional region.

However, the construction of the memory offset from the coordinates of a voxel is quite complex. Here we use a table-based conversion as reported in [14]. The bit patterns produced by the individual coordinates are independent from each other, and can therefore be looked up in three address translation tables (*ATT*) and ORed together. Actually, two tables are always the same except for a one-bit shift, see the Y- and Z-patterns in (1). Thus, the memory offset Ω of a voxel at X, Y, Z is given by

$$\Omega = \mathrm{ATT}_0(X_{7\ldots0}) \vee \mathrm{ATT}_1(Y_{7\ldots0}) \vee \mathrm{ATT}_1(Z_{7\ldots0}) \ll 1 \qquad (2)$$

The ATTs have 1KByte each and, as one might expect, are locked into the cache using the technique just described.

Next, intermediate results for the 16 rays must be stored in a tilebuffer. For each pixel, we store the coordinates and the plane number of the current raypoint, the vector to the next raypoint (all vector components in a 16.16 fixed-point format), and the accumulated colors (8.8) and translucency (0.16). The pixel entries are organized as a double-linked list for early ray termination (see section 3.8). 1KByte is allocated for the tilebuffer in the cache, as well as for all local variables and rendering parameters.

The next class of tables are the color look-up tables (CLUT). A CLUT is accessed by the interpolated 8-bit function value of a raypoint, yields a 32-bit RGBα-quadruple and has 1KByte. Each material has its own CLUT. Thus, the size of all CLUTs exceeds the remaining cache capacity. However, the CLUTs are stored such that they can only replace themselves in the cache. Furthermore, elements of up to four CLUTs can be kept in the cache. Thus, if the data set has four materials or less, this is again approximate to having a dedicated on-chip RAM.

Finally included in the spread memory layout is an acceleration data structure (*ADS*) for empty space skipping (see section 3.7). This gives the memory layout as shown in Figure 6.

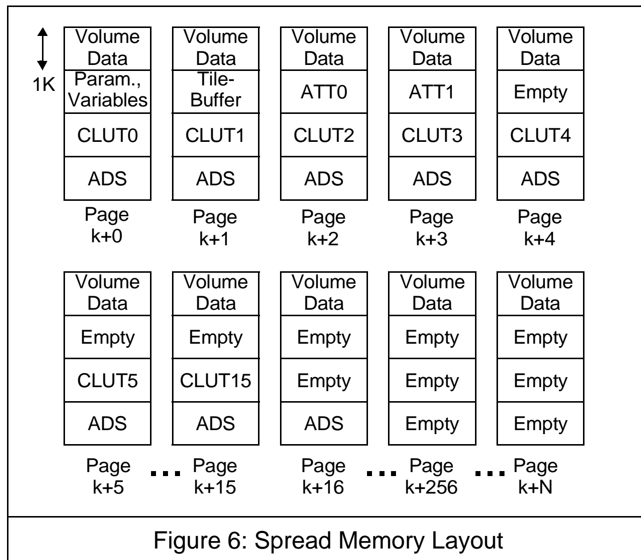

Figure 6: Spread Memory Layout

Although the addressing scheme and the memory layout seem to be quite convoluted, they are well worth the effort as outlined in section 4. A single cache miss can consume as much time as literally hundreds of CPU operations.

It should be noted, though, that the cache structure can only be maintained during the actual frame generation. If the system executes the user interface, or during API calls or task switches, this cache structure will be corrupted.

The disadvantage of the increased memory footprint is simply the price we pay for high performance. However, memory capacity is much more easily available than bandwidth.

3.3 Ray-Volume Intersection Tests

Conditional branches can severely reduce performance on today's deeply pipelined CPUs. For perspective raycasting from arbitrary viewpoints, the ray-volume intersection calculation is subject to these problems, even in the optimized form as given in [18]. However, SSE and a modification to [18] allow an optimized algorithm to be used, which removes all conditional branches. As shown in Figure 7, the observer can be in 27 sub-spaces: either inside the data set (not considered here) or outside with one, two or three faces visible.

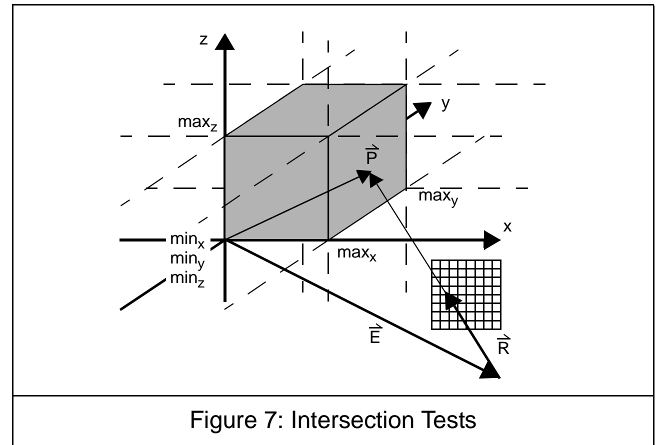

Figure 7: Intersection Tests

Given observer position $\vec{E}$ and a viewing ray $\vec{R}$, the intersection point $\vec{P}$ with any of the volume faces is given by

$$\vec{P} = \vec{E} + t \cdot \vec{R} \qquad (3)$$

One coordinate of $\vec{P}$ is always known, and so t is determined. In the case of the face $x = max_x$,

$$t = \frac{max_x - E_x}{R_x} \qquad (4)$$

A maximum of three different candidates exist, for example

$$t_1 = \frac{max_x - E_x}{R_x} = \frac{Q_x}{R_x} \qquad t_2 = \frac{min_y - E_y}{R_y} = \frac{Q_y}{R_y}$$
$$t_3 = \frac{max_z - E_z}{R_z} = \frac{Q_z}{R_z} \qquad (5)$$

The just defined vector $\vec{Q}$ is constant for a given viewpoint, and can therefore be precomputed as

$$Q_x = \begin{cases} min_x - E_x & \text{for} \quad E_x < min_x \\ max_x - E_x & \text{for} \quad E_x > max_x \\ 0 & \text{else} \end{cases} \qquad (6)$$

(Q_y and Q_z accordingly). After performing (5), the one possible candidate must be isolated. Since the bounding box of the data set is convex, t is given by

$$t = max(t_1, t_2, t_3) \tag{7}$$

Thus, by using (5)-(7), all 26 cases have been reduced to one. After performing (3), however, the resulting intersection point $\vec{P}$ must still be tested for being on the bounding volume of the data set. The following relations must be satisfied:

$$(min_x \le P_x \le max_x) \wedge (min_y \le P_y \le max_y) \wedge (min_z \le P_z \le max_z)$$

A fast SSE implementation of this algorithm is outlined in Figure 8, which uses the SSE operations RCP (fast reciprocal, 2 cycle latency with a maximum absolute error of 1.5×2^{-12}), MUL, MAX (returns the maxima in four pairs), CMP (returns all “1” if true, else all “0” in destination register), AND (bitwise AND) and MOVMASK (special instruction which transfers the four sign bits into an integer register). Note that we perform 16 intersection tests in one loop (see Figure 5), and that most constants can be kept in the SSE registers during that operation for further speedup.[1]

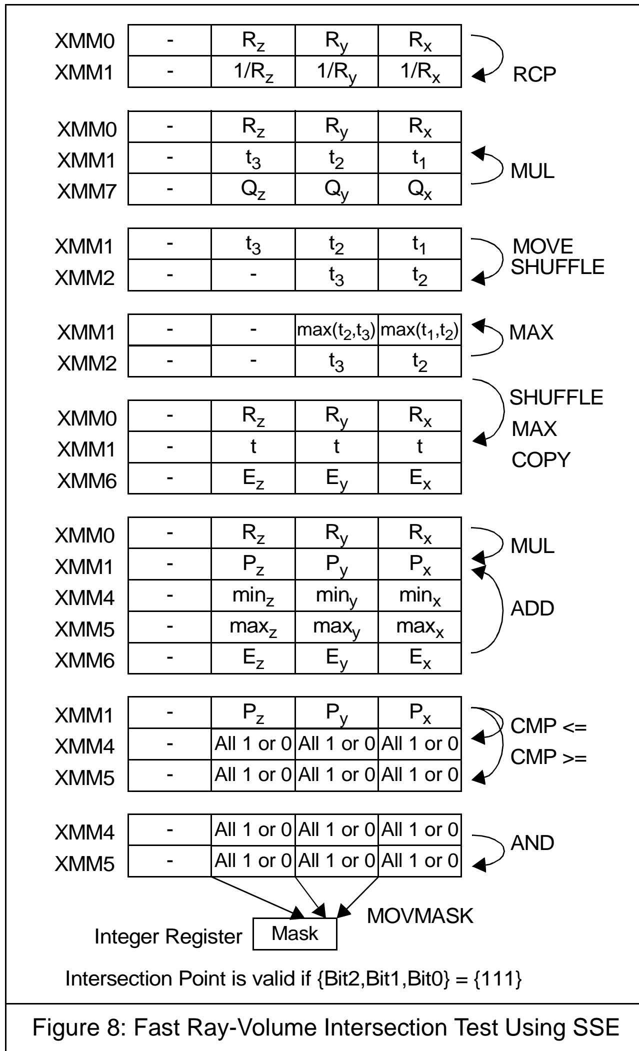

Figure 8: Fast Ray-Volume Intersection Test Using SSE

1 The problematic case $\pm\infty \times 0$ is forced to 0 by using a bitmask derived from $\vec{Q}$. Not shown for clarity.

3.4 Fetching the Voxels from Memory

It should be noted that only three accesses to the ATTs need to be done per raypoint for the tri-linear interpolation. This is because two neighboring table entries can be loaded into an MMX-register in one access, e.g., $\mathrm{ATT}_0(\lfloor x \rfloor)$ and $\mathrm{ATT}_0(\lceil x \rceil)$, x being the x-coordinate of the raypoint. From these six bit patterns, all eight voxel addresses are constructed by logical operations according to (2).

3.5 Tri-Linear Interpolation and Classification

If the data set contains material identifiers, the first step for each raypoint is to fetch the identifier of the nearest-neighbor voxel. This is the material i which is assigned to the raypoint. The user can disable materials individually, if that has been done for i, the raypoint is discarded. Otherwise the eight neighboring voxels are fetched. The material identifiers of all voxels are then compared to i. In case of a mismatch, the corresponding voxel is set to zero. This generates crisp material boundaries, especially if gradient shading is used.

For the tri-linear interpolation we can exploit the Multiply-and-Add MMX-instruction, which can be used to perform two linear interpolations. Thus, four such instructions are needed for one tri-linear interpolation. If gradients are present, the components are tri-linearly interpolated as well.

The resulting raypoint value V_P is checked against a user-supplied threshold and, if above, used as index into the proper CLUT, i.e., $C_{\lambda,P}, \alpha_P = \mathrm{CLUT}_i(V_P)$ with $\lambda = \{R,G,B\}$.

3.6 Gradient Shading

For data sets which include gradients, the following illumination model is evaluated for each (valid) raypoint:

$$I_{\lambda,P} = k_{a,i} \cdot C_{\lambda,P} + k_{d,i} \cdot C_{\lambda,P} \sum_{m=0}^{3} B_m \cdot \left\| \vec{G} \bullet \vec{L}_m \right\| + \tag{8}$$

$$k_{s,i} \sum_{m=0}^{3} B_m \cdot \max(n_i \cdot \left\| \vec{G} \bullet \vec{H}_m \right\| + a_i, 0)$$

In (8), $I_{\lambda,P}$ is the light intensity of a raypoint emitted towards the eye, k_a, k_d and k_s are the ambient, diffuse and specular reflection coefficients, respectively, i denotes the material identifier, $\vec{L}_m$ is the direction to the m-th light source with brightness B_m, and $\vec{H}_m = \frac{\vec{V} + \vec{L}_m}{\left\| \vec{V} + \vec{L}_m \right\|}$. Also, $k_{s,i} = 1 - k_{d,i}$.

The two remaining quantities n_i and a_i and the second and third term in (8) require more explanation.

These terms are computed in floating-point format using SSE. The interpolated gradient is transferred to the SSE-unit and normalized using the fast Square-Root-Reciprocal SSE-instruction (same performance as RCP, see column to the left).

We assume white light sources at infinity. The light directions are always relative to the main viewing direction, i.e., the direction from the eye to the center of the screen. One light direction is just that (similar to miner’s helmet), the other ones are 45° from the right, left and above. Using the absolute value of the dot products implements two-sided shading.

For the specular (third) term in (8), a number of simplifications are made. First, we assume a constant viewing vector $\vec{V}$ per tile (only for

the shading, not for the raycasting) as the direction from the center of the tile to the eye. Thus, the vectors $\vec{H}_m$ are also constant and can be precomputed prior to the rendering of a given tile. Second, the exponentiation is replaced by a multiplication, an add and a clamp. This is outlined in Figure 9. Essentially, the cosine is stretched by the multiplication and shifted such that $n \cdot \cos(0) + a = 1$, from which it follows that $a = 1 - n$. [1]

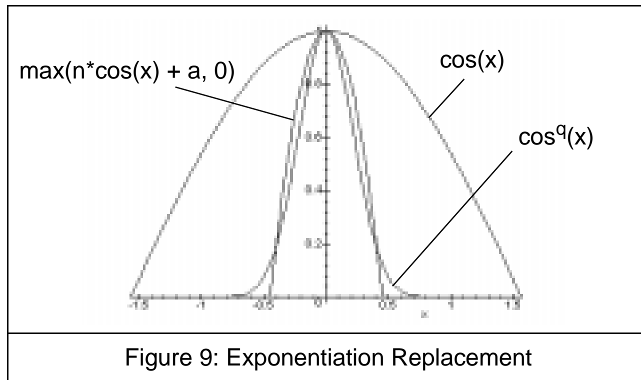

Figure 9: Exponentiation Replacement

This method shows a number of advantages. First, the function is smooth around the origin where it is required most, even for large *n*. This stands in contrast to table-based methods, which are always prone to produce aliasing for narrow highlights. Second, the largest deviation occurs for small values, and so the discontinuity may not even be noticeable. Also, the approximation gets better for large *n*. Finally, the method is computationally inexpensive since it avoids the exponentiation.
Depending on the rendering mode, many products of the specular term can be precomputed, in which case the computing requirements are in the same order as diffuse shading.
SSE can speed up the required computations significantly. An example is shown in Figure 10, which computes four dot products using only five SSE instructions and also shows why there are exactly four light sources.

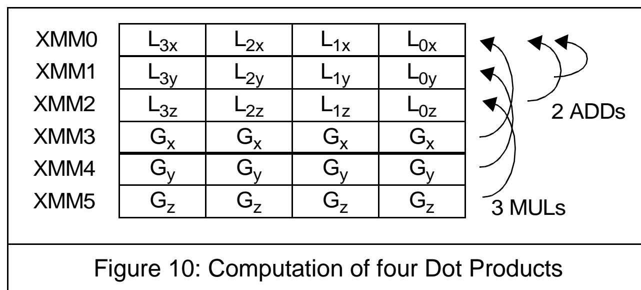

Figure 10: Computation of four Dot Products

Note that the user can set k_a, k_d and *n* for each material, and thus control the appearance (glossiness etc.) of each material individually. Also, the user can set a threshold for the gradient magnitude below which no shading is done. This can be used to highlight structures inside the volume data set.

3.7 Empty Space Skipping

For empty space skipping, we use a separate acceleration data structure (*ADS*). Other than any form of Distance Coding [19], we use just one bit to indicate whether or not a region of size 2×2×2 is empty. Note that for such a region 27 voxels are considered, and that the data set is divided into such regions in a space-filling manner.
A region is not empty if at least one voxel belongs to a material which is not currently disabled, and its value is above the user-supplied threshold for that material.
The bits are written into the ULTRAVIS memory structure such that one byte and one quadword (64 bits) describe a region of size 4×4×4 and 8×8×8, respectively.
Although the ADS itself is not hierarchical, the operations we perform on its elements are so (a technique called *hierarchy compression*). This is because we can load one quadword in a single access into an MMX-register and test it for being all zero in one single operation. If the test fails, we can test an individual byte in that quadword, and finally a single bit.
If a region is found empty, the plane of the next raypoint on the actual ray is computed using a technique similar to that described in section 3.3 (see also Figure 5). This plane number is written into the tile-buffer. Processing of all intermediate raypoints on that ray basically involves one read from the tile-buffer and one compare.
Additionally, if a region is found solid (not a single bit indicates “empty”), the plane number for which the next empty space test must be performed is computed, and also written into the tile-buffer. This reduces the overhead of empty space skipping.
The ADS is rebuilt each time after the user either switches a material on or off or adjusts the thresholds. In case of a sequence of data sets, each data set has its own ADS. Since the overhead of empty space skipping does not always pay off, the user can switch it on and off at any time.

3.8 Compositing and Early Ray Termination

Currently, the system only supports standard alpha-blending [11]. In case the accumulated translucency of the actual ray has fallen below a user-supplied threshold, its entry in the tile buffer is removed from the double-linked list. Thus, despite the tile-oriented processing, these rays do no longer consume processing time.

3.9 DirectDraw

Using DirectDraw, an off-screen double buffer of 256×256 pixels each (which is also the number of rays shot per frame) is allocated in the video memory of the graphics adapter, and made available to the rendering threads. Each completed tile is written from the on-chip cache via MMX registers directly to the video memory using non-temporal store instructions to avoid cache pollution [4]. After the frame is completed, it is copied to the visible frame buffer and at the same time magnified to the final image resolution of 512×512 pixels using 2D graphics hardware, again under control of DirectDraw.
In case of subsampling during motion, 128×128 rays are shot through the volume. The resulting 128×128 pixel images are again magnified to 512×512 screen pixels using fast 2D hardware. Thus, the speed-up is typically greater than 3.5.

3.10 Multi-Threading

For each CPU in the PC, the ULTRAVIS system creates one rendering thread. We use screen-space partitioning in units of 4×4 pixel tiles. For an even workload, the threads assign themselves tiles using a shared tile counter (dynamic self-scheduling).

3.11 Volume Animation

All data sets in a sequence must fit into main memory for maximum performance. The threads run continuously in wrap-around mode. Still, however, the user has complete control over the operation and can move the data set and adjust rendering parameters while the animation is running, due to the multi-threaded architecture.

1 It is of course not the task to find an n which best approximates the shape of a given $\cos^q(x)$. The user can simply adjust n until the results are satisfactory.

4 PERFORMANCE

The test machine is an HP Kayak XU PC with two 500MHz Pentium III CPUs, 1GByte of main memory and a graphics adapter using the TNT2 Ultra from NVidia. The test data sets are frequently used as benchmarks: engine and MRI-head (courtesy UNC Chapel Hill). The latter contains material identifiers for four tissue types. Both data sets have about 256×256×110 voxels, however, the actual bounding box can be smaller depending on the threshold values.
The *VTune*-tool from Intel, which monitors the CPU performance using the various event counters of the processor, was used to measure the following performance details.

4.1 Ray-Volume Intersection Test

The pure ray-volume intersection test as shown in Figure 8 (including all move, shuffle and logical instructions, but excluding type conversion of the results) was implemented using 26 assembler instructions. One test takes 30.82 clocks on average, and thus, the intersection tests for one tile are performed in approximately 1µs.

4.2 Tri-Linear Interpolation

Decomposed into seven linear interpolations and performed using the Multiply-and-Add MMX-instruction, the tri-linear interpolations were implemented using between 14 (*V*) and 30 (*IGV*) assembler instructions. One tri-linear interpolation takes between 12.6 and 19.5 clocks on average, giving a performance of 25M to 40M tri-linear interpolations per second per CPU. [1]

4.3 Diffuse Shading

For *GV* and *IGV*, the user can select between no shading, diffuse-only shading and combined diffuse and specular shading. Diffuse shading (including gradient normalization) was implemented using 33 (*GV)* or 35 (*IGV)* assembler instructions. Diffuse shading for one raypoint takes about 53 clocks or 106ns on average.

4.4 Diffuse and Specular Shading

For *GV*, combined diffuse and specular shading as defined in (8) was implemented in 54 assembler instructions and takes 72.5 clocks or 145ns on average. Thus, this simplified method of generating highlights increases the computational expenses of diffuse shading by only about 37% in this case.
For *IGV*, fewer terms can be precomputed since they depend on the material identifier of the raypoint. Thus, 70 instructions are needed, which take 102 clocks or 204ns on average per raypoint. Note that all performance figures include the processing of four light sources.

4.5 Cache Hit Rate

For the images in Figure 11, we measured the cache hit rates for accesses to the Address Translation Tables (*ATT*), to the volume data set and to the CLUTs. Misses to both the L1 and L2 caches have been counted. One L2 data cache miss always causes 32 bytes to be read from main memory [3]. The measurements have been done using a single-threaded version of the program. The results are summarized in Table 2.
In all cases, the images have been generated using tri-linear interpolation, empty space skipping and early ray termination with the translucency threshold set to 1/256. The raypoint distance was set to 0.75 grid units.

For Figures 11a-c, empty space was defined as the set of voxels with values below 30. Thresholds for the four different tissue types cerebrum, cerebellum, brain stem and all remaining tissue were {25, 21, 23, -} (Figure 11d), {25, 21, 23, 41} (Figure 11e) and {51, 16, 6, 38} (Figure 11f).

For the engine data set, each raypoint in non-empty space requires three accesses to the ATTs and eight accesses to the volume data set. The CLUT is only accessed for raypoints whose values after tri-linear interpolation exceed the threshold.

For the MRI-head, each raypoint in non-empty space first requires three accesses to the ATTs and one to the volume data set in order to determine the material identifier of the raypoint. If the material is not switched off, another three accesses to the ATTs and eight to the volume data set follow. Since no materials are switched off in Figures 11e and f, column 4 = 6×column 3 and column 6 = 9×column 3. This doesn't hold for Figure 11d, since raypoints in non-empty space can still be in disabled material. Again, the CLUTs are only accessed for raypoints exceeding the corresponding threshold.

As can be seen in Table 2, the mechanism to virtually lock data items in the cache works very well. Note that columns 5 and 11 show the total number of cache misses during frame generation for accesses to the ATTs and CLUTs, respectively. In many cases the numbers imply that table elements are read from memory only once.

Equally important is the very high cache hit rate for accesses to the volume data set in main memory, which can exceed 98% (column 8). These results demonstrate the efficiency of the spread memory layout and the cubic-interleaved address function, and give reason to hope that the performance of the ULTRAVIS system will scale well with the CPU clock frequency.

4.6 Frame Rates

The frame rates were measured using the high-resolution performance counter of the PC. The rendering parameters were the same as above, except that both CPUs were used. Subsampling during motion was disabled. Table 1 summarizes the ULTRAVIS performance on our test machine.

Fig.	Voxeltype	Frames/s	Typ. Range
11a	V	10	6 - 14
11b	V	2.2	1.5 - 2.5
11c	GV	8	6 - 11
11d	IV	6.2	4 - 7
11e	IV	1.7	1.5 - 2.5
11f	IGV	2.2	1.7 - 3

Table 1: Frame Rates (Examples)

5 CONCLUSIONS AND FUTURE WORK

While it is undisputed that special-purpose hardware accelerators will always be superior in performance, efficient use of advanced features of general-purpose CPUs can still result in a useful volume rendering system. Furthermore, substantial performance leaps can be anticipated for the CPUs and PC systems of the near future, from which the ULTRAVIS system will benefit automatically.

It is planned to include support for additional voxel types such as *RGB*, *RGB*α and *IRGB*α, as well as support for the mixed rendering of polygonal and volume data.

1 The ray-volume intersection test needs 20 arithmetic FP operations. Thus, a 500MHz Pentium III CPU achieves 1.19 clocks per instruction (CPI), 1.54 clocks per floating-point operation and 324MFLOPS in this part of the algorithm. In our implementation, a tri-linear interpolation accounts for 24 arithmetic integer operations. In case of *V*, one CPU achieves 0.9CPI and 0.53 clocks per arithmetic integer operation. Thus, a 500MHz CPU achieves 952MIOPS here.

1	2	3	4	5	6	7	8	9	10	11
Fig.	Type	Raypoints in non-empty Space	Accesses to ATTs	Cache Misses L1 / L2	Accesses to Voxels	Cache Misses L1 / L2	Hit Rates L1 / L1+L2 (%)	Bytes read from Volume Data Set (KB)	Accesses to CLUTs	Cache Misses L1 / L2
11a	V	378,156	1,134,468	58 / 58	3,025,248	32,169 / 17,501	98.9 / 99.4	547	270,712	29 / 29
11b	V	2,588,550	7,765,650	57 / 57	20,708,400	232,244 / 116,385	98.9 / 99.4	3,637	2,173,670	29 / 29
11c	GV	319,143	957,429	124 / 58	2,553,144	98,185 / 56,125	96.2 / 97.8	1,754	211,059	29 / 29
11d	IV	674,783[1]	3,538,812	90 / 90	4,713,351	61,980 / 49,952	98.7 / 98.9	1,561	497,921	48 / 48
11e	IV	2,416,976[2]	14,501,856	118 / 118	21,752,784	424,562 / 268,059	98.0 / 98.8	8,376	1,886,104	91 / 68
11f	IGV	1,064,492	6,386,952	239 / 119	9,580,428	363,976 / 184,496	96.2 / 98.1	5,766	627,973	61 / 60

Table 2: Cache Hit Rates

1 Raypoints in non-empty space, belonging to enabled material: 504,821
2 Measurement had to be terminated prior to image completion due to prohibitively long simulation times.

6 REFERENCES

[1] **K. Akeley,** *"RealityEngine Graphics"*, Proceedings SIGGRAPH 93, pages 109-116

[2] **M. L. Brady, K. K. Jung, H. T. Nguyen, T. PQ Nguyen,** "*Interactive Volume Navigation*", IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 3, 1998, pages 243-256

[3] **Intel Corporation,** "*Intel Architecture Software Developer's Manual, Volume3: System Programming*", Order Number 243192, 1999, page 9-3

[4] **Intel Corporation,** "*Intel® Architecture Optimization Reference Manual*", Order Number 245127-001, 1999

[5] **T. J. Cullip, U. Neumann,** *"Accelerating Volume Reconstruction with 3D Texture Hardware"*, Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993

[6] **A. Van Gelder, K. Kim,** *"Direct Volume Rendering with Shading via Three-Dimensional Textures"*, Proceedings 1996 Symposium on Volume Visualization, pages 23-30

[7] **G. Knittel**, *"High-Speed Volume Rendering Using Redundant Block Compression"*, Proceedings IEEE Visualization '95 Conference, pages 176-183

[8] **G. Knittel,** *"TriangleCaster - Extensions to 3D-Texturing Units for Accelerated Volume Rendering"*, Proceedings EG/SIGGRAPH Workshop on Graphics Hardware '99, pages 25-34

[9] **P. Lacroute, M. Levoy,** *"Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation"*, Proceedings SIGGRAPH '94, pages 451-458

[10] **P. Lacroute,** "*Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization*", Proceedings IEEE/ACM '95 Parallel Rendering Symposium, pages 15-22

[11] **M. Levoy,** "*Display of Surfaces from Volume Data*", IEEE Computer Graphics & Applications, Vol. 8, No. 3, May 1988, pages 29-37

[12] **A. Peleg, U. Weiser,** "*MMX Technology Extension to the Intel Architecture*", IEEE Micro, Vol. 16, No. 4, August 1996, pages 42-50

[13] **H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, L. Seiler,** "*The VolumePro Real-Time Ray-Casting System*", Proceedings SIGGRAPH 99, pages 251-260

[14] **G. Sakas, M. Grimm, A. Savopoulos,** "*Optimized Maximum Intensity Projection (MIP)*", Proceedings 6th Eurographics Workshop on Rendering, Springer-Verlag 1995, pages 51-63

[15] **S. (Ticky) Thakkar, T. Huff,** "*Internet Streaming SIMD Extensions*", Computer, Vol. 32, No. 12, pages 26-34

[16] **M. Wan, A. Kaufman, S. Bryson,** "*High Performance Presence-Accelerated Ray Casting*", Proceedings IEEE Visualization '99 Conference, pages 379-386

[17] **R. Westermann, T. Ertl,** *"Efficiently Using Graphics Hardware in Volume Rendering Applications"*, Proceedings SIGGRAPH 1998, pages 169-177

[18] **A. Woo,** *"Fast Ray-Box Intersection"*, Graphics Gems, edited by A. S. Glassner, Academic Press, 1990, page 395

[19] **K. J. Zuiderveld, A. H. J. Koning, M. A. Viergever,** *"Acceleration of Ray-Casting Using 3D Distance Transforms*", Proceedings of Visualization in Biomedical Computing, 1992, pages 324-335

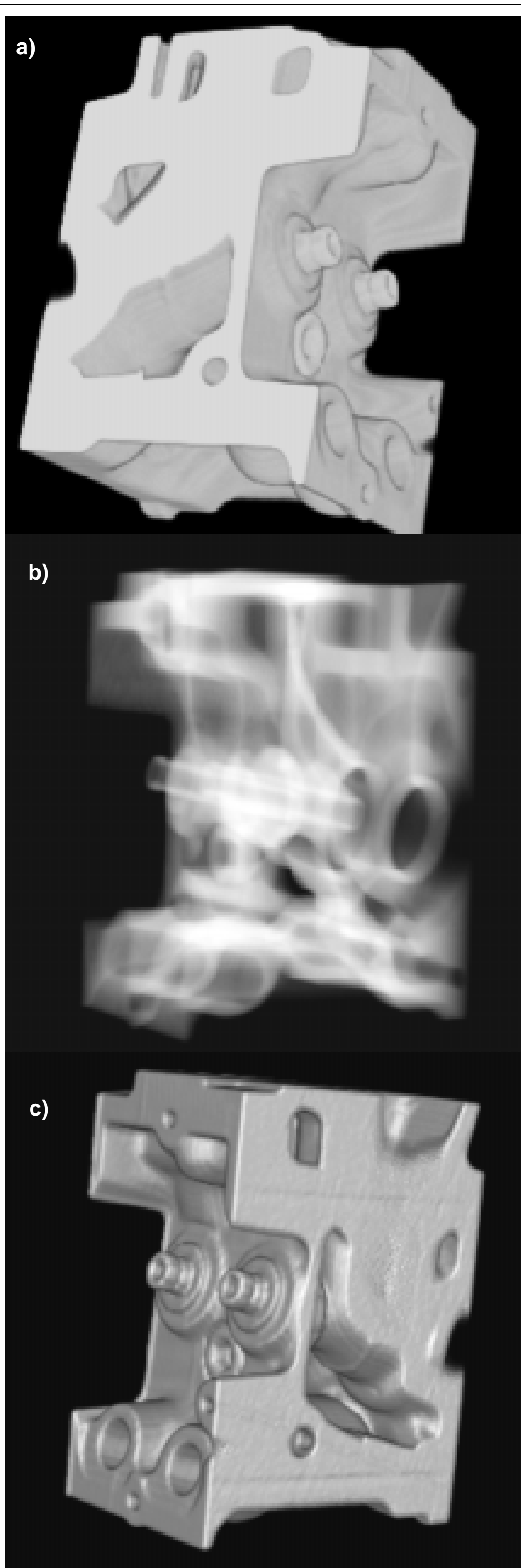

Figure 11: Sample Images, generated from 256^2 Rays