Giga-Voxel Rendering from Compressed Data on a Display Wall

R. Parys, G. Knittel
WSI/GRIS, Tuebingen University
72076 Tuebingen, Germany
[parys | knittel]@gris.uni-tuebingen.de

ABSTRACT

We present a parallel system capable of rendering multi-gigabyte data sets on a multi-megapixel display wall at interactive rates. The system is based on Residual Vector Quantization which allows us to render extremely large data sets out of the graphics memory. At 0.75 bits per voxel, such large data sets can even be kept on a consumer-level graphics card. As an example we compress the whole full color "Visible Human Female" data set, approximately 21GByte in size, down to 700MByte. Taking advantage of the fixed code length and the extremely simple decompression scheme of RVQ, all decompression is done on the GPU at very high rates. For each frame the data set is decompressed into small subvolumes which are rendered front to back. Classification and shading can be moved into the decompression step, speeding up the rendering pass.

Keywords
Distributed and Parallel Graphics, Volume Rendering, GPU Programming

1. INTRODUCTION

Since quite some time volume rendering has made it from a purely academic research area into a well-established computer graphics application with high economic relevance. Still, comprehensive platform support as in the case of computer games, as an example, is largely missing. Nevertheless, visualization requirements steadily increase: data sets are getting larger, rendering algorithms are getting more complex, and display resolutions are increasing as well. In this work we present a fairly extreme example: volume rendering of a very large data set (about 7.5G voxels) on a high-resolution display wall (65,536,000 pixels). This display system (see Fig. 3 at the end of the paper) consists of 16 LCDs with a resolution of 2560×1600 pixels each. Each display is driven by a PC, being equipped with a Dual-Core 2.4GHz Intel CPU, 4GByte of main memory and a 8800GT graphics card from NVidia. The latter in turn has a video memory of 1GByte capacity. The PCs are connected via GBit Ethernet. As a side note, each display is connected to its PC via a dual-link DVI cable no shorter than 20 meters.

While the PC-cluster would otherwise represent a decent computing platform, given the task at hand it is somewhat underpowered. Thus, we need to apply efficient optimizations. Also, the workload should be placed where the strongest computing resources are, and notorious bottlenecks such as the network should be avoided as much as possible, even if this causes some amount of redundant computation. With respect to computational power and memory bandwidth there is an easy choice: GPUs are approaching and surpassing the teraflops-mark, and peak memory bandwidth on consumer-level cards approaches 150GByte/s. These numbers are unavailable anywhere else in a typical workstation. From these consideration we derived the design choices of our rendering system, which will be described in detail in the following sections.

2. RELATED WORK

Other work related to our project falls into the areas of parallel volume rendering, data set compression, and GPU-based volume rendering.

2.1 Data Set Compression

A study on lossless compression for volume data is presented in [6]. The authors report a maximum of about 50% reduction in size for selected data sets, however, the work was targeted at reducing storage space rather than increasing rendering speed. The use of vector quantization for volume rendering was first proposed in [18]. The presented rendering system operates directly on the compressed data. An improved version can be found in [22], however, the method provides only nearest-neighbor interpolation and is thus limited in rendering functionality. In other early work the authors used Block Truncation Coding in a space-filling way to reduce memory bandwidth

[12]. Wavelet-based encoding, however, has received the most attention in recent years [9],[17],[20],[21]. A hierarchical wavelet representation of large data sets is used in [7]. The authors claim to achieve a compression rate of 30:1 without noticeable artifacts in the image. A quality measure is derived from the wavelet representation during rendering to minimize the number of voxels to process. Interactive rendering speeds for large data sets can be achieved on standard PCs. We didn't follow this approach since decompression is most practically done on the CPU, and sending pixel data over the bus to the graphics card can severely limit performance (see Table 1).

2.2 GPU-based Volume Rendering

The use of graphics hardware for volumetric data processing dates back to [1], [3] and [4]. Originally, 3D texturing hardware was used for volume rendering. Screen-aligned slices were swept through the 3D texture and blended in back-to-front order. Thus, a volume data set was treated as a light-emitting, translucent material. Later improvements included gradient shading [15], multi-dimensional transfer functions [11], pre-integrated transfer functions [5], and the processing of pre-segmented data sets [8]. Where it is possible and useful we try to integrate these techniques into our framework. Some features have lower priority, though. For example, data sets of the size considered here are rarely pre-segmented due to the large effort it takes. Thus, support for this feature is postponed to later versions.

2.3 Parallel Volume Rendering

Basically, parallel rendering can be done in two ways: object-space partitioning, and screen-space partitioning (see Section 3.2.6). To a large degree, the performance of systems using object-space partitioning is limited by the alpha blending of the intermediate images. Solutions are proposed in [14], [23] and [24]. In [24], it is described that for alpha-blending the CPU is used instead of the much better suited GPU. This highlights the difficulties of handling such large data streams in the network.

Rendering to a display wall of about 63M pixels is described in [16]. Here, isosurfaces are rendered from a total of 470M triangles. An example is shown with a rendering time of 15 seconds, demonstrating the challenges presented by these display systems.

3. THE GIGA-VOXEL SYSTEM

As previously mentioned, we prioritize the graphics card for all computations, even if this means a certain amount of redundant processing. A few benchmark figures might further motivate this choice. On a Dell XPS700 workstation, equipped with an Intel Core 2 Duo CPU at 2.13GHz and an NVidia GTX280 (optionally an 8800GT), we obtained the following results (measured with the SiSoft Sandra benchmark suite and "bandwidthTest" from the NVidia CUDA SDK [19]):

Test	Bandwidth
CPU ↔ Cache	98,520
CPU ↔ Memory, 16MB Blocks	2,100
CPU ↔ Graphics Card (PCIe 1.x)	1,500
GPU ↔ Video Memory GTX280	110,028
GPU ↔ Video Memory 8800GT	43,357

Table 1: Actual Bandwidth Measurements [MB/s]

Interestingly, the tools report the internal CPU cache bandwidth to be lower than the bandwidth to the external video memory on the GTX280. Thus the design target was set to keep all necessary data locally in video memory, and to use the GPU for all compute-intensive tasks. As a side effect, this frees the CPU to do supporting activities such as tissue simulations. In a cluster environment this means that the data must be replicated, and that pixel traffic must be kept to a minimum. Clearly, given the size of typical volume data sets, the former can only be achieved using data set compression. The compression scheme, however, has to fulfil contradicting requirements: it must provide a high compression rate at still high image quality, and the decompression must be simple and extremely fast. We found *Residual Vector Quantization* (RVQ) to be an interesting candidate for this purpose.

Thus, rendering a data set using the Giga-Voxel System is a two-stage process: first the data set needs to be compressed in an offline step, and then it can be loaded on the graphics cards and rendered. We'll start the description of the process chain with the compression step.

3.1 Residual Vector Quantization

Residual Vector Quantization has first been described in [10]. An excellent survey of RVQ and related techniques can be found in [2]. RVQ is an extension to standard vector quantization (VQ). In VQ, a (large) set of vectors is replaced by a (small) set of representative vectors (here called *codevectors*), while trying to minimize overall error. Often, clustering methods are used to find a proper set of codevectors (collectively called a *codebook*). A frequently used method is *k-means* [13]. Starting from an initial set of random codevectors (*seeds*), each vector is assigned to its nearest codevector, thereby forming clusters. Once finished, the codevectors are moved to the center of their respective cluster, and then clustering is started anew. This process is repeated until the system reaches a stable state. Each vector will now be replaced by the index of its codevector in the codebook. Decompression merely consists of a table look-up.

If the set of vectors is too large, or unknown at the time of codebook construction, a subset of vectors (*training set*) can be used to build the codebook. Any further vector is then replaced by the index of the best-matching codevector within the existing codebook.

For RVQ, the set of difference vectors is constructed, i.e., vector - codevector. This set of vectors, equal in number and dimension to the original vectors, is now subjected to yet another VQ, giving a second set of indices and codvectors. Each original vector is now replaced by two indices, and the corresponding codevectors are simply added to give the decompressed version of the vector. This process can be repeated for the desired number of levels.

There are a number of quantities which affect the performance of RVQ. The number of codevectors per codebook and the number of codebooks define the length of the index set (the *codelength*) in bits. The number of dimensions of the original vectors and the width of each component define the compression rate relative to a given codelength.

From experiments with a large number of images we have found that larger codebooks should be favored over a high number of levels, since the image quality in terms of PSNR is higher for a given codelength. Clearly, however, there is a practical limit in codebook size, both with respect to memory consumption and compression time. Compared to an on-chip decompressor, we can take advantage of the much larger but still fast video memory. A number of tests have shown that a high image quality can be achieved using four levels, with a codebook of 4k codevectors on each level. This gives a codelength of 48 bits.

Since our test data set is the “Visible Human Female” in the full-color version, a pixel is a 24-bit RGB quantity. A vector is formed by a 4×4×4 pixel cube in the image stack, and so the vector dimension is 192. The 64 pixels in a cube are compressed into 48 bits, giving a compression rate of 32:1, or 0.75bpp.

For the components of a codevector we use a higher precision to account for rounding errors. The RGB-fields of one pixel are packed into one 32-bit word in 11-11-10 format. Thus, a codevector consists of 64×4 = 256 Bytes. A codebook occupies 1MByte accordingly, for a total of 4MBytes for all levels.

3.1.1 *Compressing the Visible Human Female*

Originally, these images have a resolution of 2048×1216 pixels [25]. There are a total of 5189 images. The cadaver was submerged in a blue gel, which we have set to a black “empty space”. However, there is too much empty space around the data, so we have cropped the images to a final resolution of 1608×896 pixels. This gives an input data set size of 20.9GByte.

Compression time for a data set of this size would be too long, so we have selected a training set equivalent in size to 300 images. Construction of the four codebooks took roughly 21 hours. However, the code was running on CPUs (actually on an eight-core machine). Since this is not the focus of this work, we haven’t yet implemented a parallel cluster version, nor a GPU version. Since codebook construction mainly consists of nearest-neighbor searches which can easily be parallelized, there is reason to assume that compression speed can be improved significantly.

Compressing a slice of 1608×896×4 using the existing set of codebooks takes roughly 30 seconds. Thus, this second step adds about 10 hours to the overall compression time.

We give the image quality in terms of PSNR. All 4×4×4 cubes which are completely background have not been included into the PSNR computation. The overall PSNR is about 27dB. An example of an original image versus the decompressed image is shown in Fig. 1b and c. Both images form a stack of four 64×64 pixel cut-outs of the same image portion. Fig. 1a shows a codebook on level 0.

The result of the compression step is an array of 402×224×1297 = 116,792,256 index sets of 48 bits each, for total size of 700,753,536 Bytes. Thus, the entire compressed data set along with the codebooks fits on a graphics card with 1GByte of video memory. In this work we only consider the case that the compressed dataset fits entirely into the video memory. Otherwise swapping from main memory or even hard disk would be required, which, however, would also benefit from the high compression rate.

3.2 Rendering

In general, rendering is done by repeatedly decompressing subvolumes of the compressed data set into an intermediate 3D-texture in video memory, rendering this 3D-texture using a raycaster, and blending the resulting images. The raycaster we use is supplied with the SDK from NVidia. Classification using a 3D lookup-table, and optionally gradient extraction and shading, are integrated into the decompression step in order to not slow down the raycaster. Early-ray-termination is included on a per-ray basis in this raycaster, we added early-exit on a per-subvolume basis using occlusion culling. Empty-space-skipping is applied to subvolumes after classification, i.e., whenever the visible contribution of a subvolume according to the actual transfer function is below a user-supplied threshold. Multi-resolution rendering can also be integrated in an elegant way.

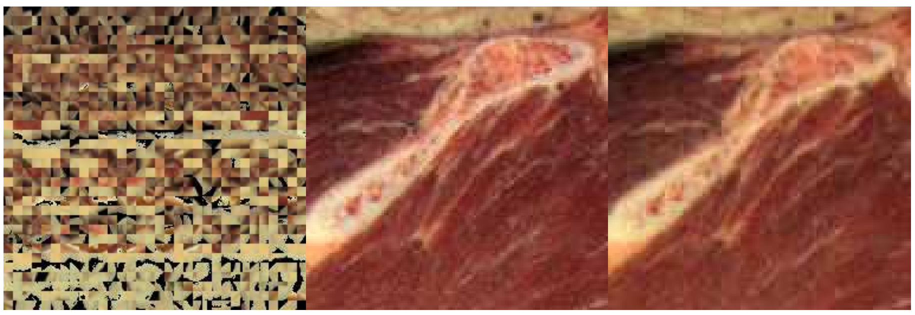


Fig. 1:a.) Codebook example on Level 0. b.) Original image. c.) Decompressed image.

We will now discuss the individual steps in more detail. A diagram depicting the overall flow is shown in Fig. 2.

3.2.1 Decompression

We have implemented the decompressor as a CUDA kernel, taking advantage of advanced features of the NVidia GPUs. Most notably, we make heavy use of the on-chip shared memory buffer. Processing is as follows.

Decompression is done in units of 256 index sets (worth 16k voxels), which are loaded into the shared memory. Each index set consists of 6 Bytes, which are unpacked into four 16-bit indices again into shared memory. For each voxel to be generated, there is one thread in the kernel. Each thread reads the unpacked index set, and fetches from memory those parts of the codevectors which it needs for its voxel. After unpacking the codevector components (from 11-11-10-format, see Section 3.1), and accumulation, the RGB-components of the voxel are written into shared memory.

These quantities are also used to access a 3D lookup-table which contains opacity (α) values. The a-value is again written into the shared memory, which completes the voxel generation. The system keeps track of the visible contribution of all voxels in a subvolume (color components multiplied by alpha), if the contribution of a subvolume to the final image is too low after classification, the subvolume is excluded from rendering (empty-space-skipping on the subvolume level).

When a certain number of threads have finished their voxel, the contents of the shared memory are written to video memory, that is, to the intermediate 3D texture.

By means of this process order we can make sure that memory transfers are mostly large bursts, and so bandwidth is high. Decompression performance is 1.86G voxels/s on the GTX280, and 0.60G voxels/s on the 8800GT.

Partitioning the volume into subvolumes always causes problems at the subvolume boundaries. During raycasting, the reconstruction filter (tri-linear interpolation) is missing voxels from the neighboring subvolume, during gradient extraction (see Section 3.2.2), the kernel hits the same problem. Most often, this problem is solved by using overlapping subvolumes, and we adopt this method. Each subvolume is extended in x-, y- and z-direction by two layers of 4×4×4 voxels. Net subvolume size was chosen to be 128^3, and so final subvolume size is 136×136×136. The added overhead is about 20%. Decompression performance for different levels of detail is summarized in Table 2.

3.2.2 Gradient Extraction and Shading

Once a given subvolume has been decompressed, the system can optionally perform gradient shading. In this work, we derive the gradient from the opacity, since steep changes in opacity represent the surfaces of regions of interest.

GPU	Level	Voxels/s	Subvolumes/s
8800GT	0	0.60G	254
GTX280	0	1.86G	776
8800GT	1	0.20G	716
GTX280	1	0.72G	2463
8800GT	2	0.03G	841
GTX280	2	0.06G	1667

Table 2: Decompression Performance

For smooth surfaces, we use a variant of a 3×3×3 Sobel filter (see Fig. 2). Two problems need to be addressed, however:

- high computation costs due to the large kernel,
- a certain amount of noise still in the image.

We solve both problems by using downsampled versions of the subvolume for gradient estimation (see also section 3.2.5, Multi-Resolution Rendering). The system generates two additional levels of detail: a 68^3, and a 34^3 subvolume. The gradients are computed only on the lowest-resolution grid, again using a CUDA-kernel. Performance is given in Table 3.

GPU	Gradients/s	Subvolumes/s
8800GT	17.8M	570
GTX280	63.9M	2045

Table 3: Gradient Estimation Performance

Gradient extraction and shading are done at the voxel positions, the contributions from specular reflection are added to the just decompressed RGB-quantities. Thus, the operation of the raycaster is not at all affected by the shading operation, and is therefore not slowed down. On the other hand, decompression speed does not suffer too much because of the still regular memory access pattern.

For gradient extraction the system can use a Central Difference (CD) operator, or a Gradient shading is again implemented as a CUDA-kernel. As before, each thread processes one voxel. Each thread reads a certain subset of the required voxel neighborhood, so that by the end of this step a large block of voxels resides in shared memory.

To speed up shading we assume light sources at infinity, and a constant viewing direction throughout the volume (only for the shading, not for the raycasting). Thus, the halfway vectors are all constant, and don't need to be computed for each voxel. It is true that the placement of the highlights will be incorrect, however, such artifacts are rarely disturbing. Exponentiation is done by a look-up in a precomputed table.

Gradient shading speed for CD and one light source is summarized in Table 4.

GPU	Level	Voxels/s	Subvolumes/s
8800GT	0	0.227G	111
GTX280	0	0.411G	201
8800GT	1	0.309G	966
GTX280	1	0.546G	1712
8800GT	2	0.271G	7426
GTX280	2	0.482G	13158

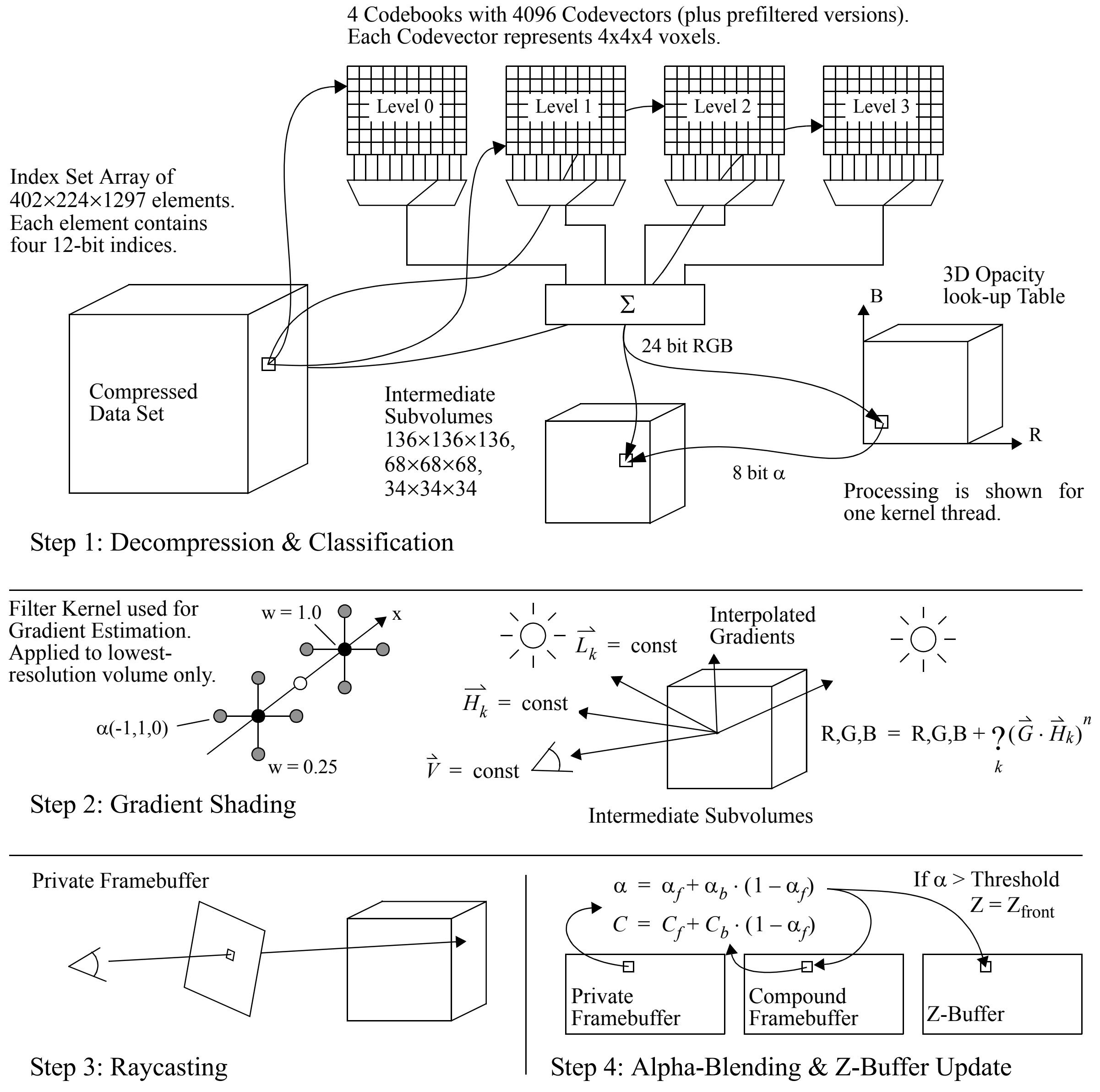
Table 4: Gradient Shading Performance (CD)

3.2.3 *The Raycaster*

As previously mentioned we use the raycaster in the SDK from NVidia. Since it is not the focus of this work, no attempt was made to optimize this code. On the GTX280 graphics card, the rendering time of this raycaster was measured to be an average of 3.97ms per subvolume (early-ray-termination disabled), and so to be about 4 times slower than the pure decompression. Thus, rendering time is largely dominated by raycasting; the time spent in the recurring decompression can be tolerated fairly well.

3.2.4 *Blending and Occlusion Culling*

The subvolumes are rendered in front-to-back order, according to their Manhattan Distance to the viewer. The result of the rendering of one subvolume is a private frame buffer of RGBα-values. This buffer is α-blended with the compound frame buffer, which in the end contains the final image.



4 Codebooks with 4096 Codevectors (plus prefiltered versions).
Each Codevector represents 4x4x4 voxels.
Level 0
Level 1
Level 2
Level 3
Index Set Array of 402×224×1297 elements. Each element contains four 12-bit indices.
Σ
3D Opacity look-up Table
B
24 bit RGB
R
Compressed Data Set
Intermediate Subvolumes 136×136×136, 68×68×68, 34×34×34
8 bit α
Processing is shown for one kernel thread.
Step 1: Decompression & Classification
Filter Kernel used for Gradient Estimation. Applied to lowest-resolution volume only.
w = 1.0
x
α(-1,1,0)
w = 0.25
$\vec{L}_k$ = const
$\vec{H}_k$ = const
$\vec{V}$ = const
Interpolated Gradients
R,G,B = R,G,B + $\sum_k (\vec{G} \cdot \vec{H}_k)^n$
Intermediate Subvolumes
Step 2: Gradient Shading
Private Framebuffer
Step 3: Raycasting
$\alpha = \alpha_f + \alpha_b \cdot (1 - \alpha_f)$
$C = C_f + C_b \cdot (1 - \alpha_f)$
If α > Threshold
$Z = Z_{front}$
Private Framebuffer
Compound Framebuffer
Z-Buffer
Step 4: Alpha-Blending & Z-Buffer Update

Fig. 2: Process Flow Diagram.

During blending, the system also updates a Z-buffer. Whenever the α-value of a pixel in the compound frame buffer exceeds a threshold, the corresponding entry in the Z-buffer is set to Z-front. This is then used to exclude subvolumes which are occluded by opaque structures in the data set from decompression and rendering. To this end, an occlusion query is submitted with the bounding box of the subvolume, which returns the number of visible pixels. Depending on a user-defined threshold, the subvolume is rendered or rejected.

Occlusion queries can be accelerated by submitting a batch of bounding boxes. In our system, all subvolumes with the same Manhattan Distance could be rendered in parallel and in any order, so they are queried in one batch. Subvolumes which are located at any of the visible faces of the entire volume will be rendered in any case and are excluded from occlusion query.

3.2.5 Multi-Resolution Rendering

To avoid subsampling of the data during raycasting, and to speed up rendering of distant subvolumes, the system can generate decompressed subvolumes at different resolutions. Here we can take advantage of the fact that a downsampled version of the subvolume can be generated from a *downsampled version of the codebooks*. Thus there is no need to keep a separate index set array for each level of detail in video memory, all we need is a small amount of extra memory for the downsampled codebooks.

The system supports decompressed subvolumes with 136, 68, and 34 voxels along each axis. By the use of normalized texture coordinates, the raycaster automatically performs proper voxel access and filtering. Only the opacity must be adjusted, we accomplish this by using a separate 3D look-up table for each resolution.

During rendering, the proper resolution of each subvolume is selected according to the raypoint spacing on neighboring rays, or, of course, according to user input.

3.2.6 Parallel Rendering on the Cluster

In general, the work can be partitioned in two ways: object-space partitioning (OSP), and screen-space partitioning (SSP). In OSP, a workpackage consists of a subvolume. This is rendered to a private frame buffer including the alpha-channel. Since a subvolume can project to any part of the global frame (spanning all displays), at least a subset of pixels need to be sent over the network for blending into the local compound frame buffer at the receiving node. Depending on how many subvolumes contribute to a given screen pixel, each final pixel may have caused multiple transfers over the network. Although sophisticated schemes have been developed to optimize this operation [14],[23], this pixel traffic still represents a severe bottleneck, especially over relatively slow GBit Ethernet. The upside is, though, that any subvolume is processed at most once.

In SSP, a workpackage consists of a rectangular region on the screen (a *tile*). A machine having been assigned a certain rectangle renders this tile to completion, and sends the final pixels to the destination screen. Since the viewpoint can be at an arbitrary location, each node needs a complete copy of the data. Most obviously we have selected this method, and use the RVQ-compression to fulfil this requirement.

In SSP, the view frustum of a given tile can intersect a number of subvolumes, which contribute only partially to the tile pixels. Such subvolumes need to be processed again for neighboring tiles, which introduces a certain overhead. Since the raycaster will not process rays redundantly, but decompression will only generate complete subvolumes, the overhead mainly consists of redundant decompression (plus redundant occlusion queries). Since the decompression is very fast, we opted for sacrificing GPU cycles in favor of reduced network traffic.

Tiles are assigned dynamically on demand. Thus, there is a scheduling thread in the system which hands out tiles to requesting nodes. As a further optimization, each requesting node first gets tiles from its own display.

4. PERFORMANCE

A photo of the display wall showing a rendering of the Visible Human Female is shown in Fig. 3. The alpha-threshold for occlusion culling was set to 0.95. Renderings like these rotating around the z-axis take an average of 3.9 seconds per frame. If using only downfiltered subvolumes, average rendering time decreases to 2.8 or 2.3 seconds per frame, for 68^3 and 34^3 subvolumes, respectively. Tests have shown that image completion time is reduced by roughly 50% if empty pixel packets are transferred, i.e., only synchronization messages are sent. This confirms our choice of rendering mode, since the network is already saturated with this minimal amount of pixel data. We have included lossless image compression before sending tile pixels (a LZW-variant), but coincidentally the reduced transmission time was exactly offset by compression and decompression times. Thus, reducing network overhead remains a research topic in this project.

5. CONCLUSIONS

We have presented a parallel volume graphics system for rendering very large data sets on a high-resolution display wall. It provides the following features:

- Compression of the data set down to 0.75bpp, thereby enabling data set replication on all nodes, as well as placement of large data sets entirely in fast video memory,
- fast and simple decompression, entirely handled by the GPU,
- on-the-fly classification and gradient shading,
- empty-space-skipping on a per-subvolume basis,
- occlusion culling on a per-subvolume basis,
- multi-resolution rendering,
- and parallel rendering with screen-space partitioning.

The system can render our compressed version of the Visible Human Female at interactive speed. Still fine details of the anatomy, such as thin blood vessels, are preserved.

Future work will be directed at improving the image quality, and at increasing the rendering speed. Improving the image quality is not a matter of better rendering in the first place, but of better compression. Thus we will try to increase the PSNR and alleviate the block artifacts in the decompressed image. As a first step the use of even larger codebooks will be investigated, which will most likely not affect rendering speed.

The latter can still be improved by further optimizing the CUDA kernels. It is not always intuitive which codes lead to a speed-up and why. Thus, kernel optimization often means time-consuming try-and-error. We are confident, however, that significant speed gains can still be achieved.

6. REFERENCES

[1] K. Akeley, "*RealityEngine Graphics*", Proc. ACM Siggraph 93 Conference, pp. 109-116

[2] C. F. Barnes, S. A. Rizvi, *"Advances in Residual Vector Quantization: A Review"*, IEEE Trans. on Image Processing, Vol. 5, No. 2, 1996

[3] B. Cabral, N. Cam, J. Foran, "*Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware*", Proc. ACM Symposium on Volume Visualization 1994, pp. 91-97

[4] U. Cullip, U. Neumann, "*Accelerating Volume Reconstruction with 3D Texture Hardware*", UNC Tech Report TR93-0027, 1993

[5] K. Engel, M. Kraus, T. Ertl, "*High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*", Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2001

[6] J. E. Fowler, R. Yagel, "*Lossless Compression of Volume Data*", Proc. ACM Symposium on Volume Visualization 1994, pp. 43-50

[7] S. Guthe, M. Wand, J. Gonser, W. Strasser, "*Interactive Rendering of Large Volume Data Sets*", Proc. IEEE Visualization Conference 2002, Boston, MA, pp. 53-60

[8] M. Hadwiger, C. Berger, H. Hauser, "*High-quality two-level volume rendering of segmented data sets on consumer graphics hardware*", Proc. IEEE Visualization Conference 2003, pp. 301-308

[9] I. Ihm, S. Park, "*Wavelet-based 3D compression scheme for very large volume data*", Proc. Graphics Interface 1998, pp. 107-116

[10] B. H. Juang, A. H. Gray, *"Multiple Stage Vector Quantization for Speech Coding"*, Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 1, Apr. 1982, pp. 597-600

[11] J. Kniss, G. Kindelmann, C. Hansen, "*Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets*", Proc. IEEE Visualization Conference 2001, pp. 255-262

[12] G. Knittel, *"High-Speed Volume Rendering Using Redundant Block Compression"*, Proc. IEEE Visualization '95 Conference, Atlanta, GA, October 29 - November 3, 1995, pages 176-183

[13] S. P. Lloyd, *"Least squares quantization in PCM"*, IEEE Trans. on Information Theory, Vol. 28, 1982, pages 129-137

[14] K.-L. Ma, J. S. Painter, C. D. Hansen, M. F. Krogh, "*Parallel volume rendering using binary-swap compositing*", IEEE Computer Graphics and Applications, Vol. 14, No. 4, 1994, pp. 59-68

[15] M. Meissner, U. Hoffmann, W. Strasser, "*Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering*", Proc. 10th IEEE Visualization Conference 1999 (VIS '99), p. 32

[16] K. Moreland, D. Thompson, "*From Cluster to Wall with VTK*", IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003, October 20-21, 2003, Seattle, Washington, USA, pp. 25-31

[17] K. Nguyen, D. Saupe, "*Rapid high quality compression of volume data for visualization*", Computer Graphics Forum 20, 13, 2001

[18] P. Ning, L. Hesselink, "*Vector Quantization for Volume Rendering*", Proc. ACM Workshop on Volume Visualization 1992, Boston, MA, pp. 69-74

[19] NVIDIA Corporation, "*CUDA Zone*", http://www.nvidia.com/object/cuda_home.html#

[20] F. Rodler, "*Wavelet based 3D compression with fast random access for very large volume data*", Proc. Pacific Graphics 1999, pp. 108-117

[21] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, W. Strasser, *"Smart Hardware-Accelerated Volume Rendering"*, Proc. EG/IEEE TCVG Symposium on Visualization 2003, pp. 231-301

[22] J. Schneider, R. Westermann, "*Compression Domain Volume Rendering*", Proc. 14th IEEE Visualization Conference 2003 (VIS'03), p. 39

[23] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, J. Patchett, "*SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering*", IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003, October 20-21, 2003, Seattle, Washington, USA, pp. 33-40

[24] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, T. Ertl, "*Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters*", Proc. EG Symposium on Parallel Graphics and Visualization 2004, pp. 41–48

[25] United States National Library of Medicine, "*The Visible Human Project*", http://www.nlm.nih.gov/research/visible/getting_data.html

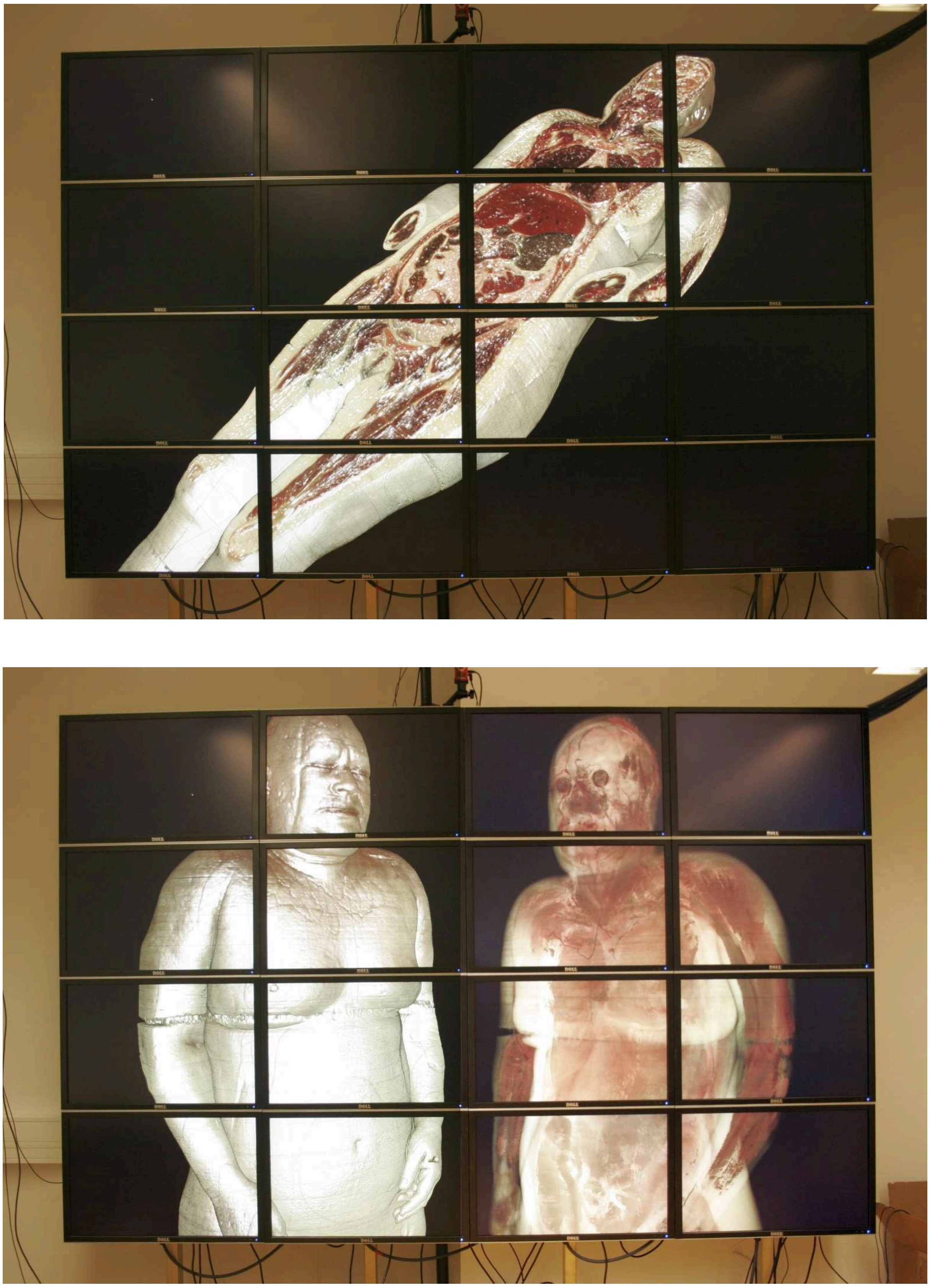


Fig. 3: The Display Wall.