To appear in the Visualization 2002 conference proceedings

Interactive Rendering of Large Volume Data Sets

Stefan Guthe Michael Wand Julius Gonser Wolfgang Straßer

WSI/GRIS, University of Tübingen

Abstract

We present a new algorithm for rendering very large volume data sets at interactive framerates on standard PC hardware. The algorithm accepts scalar data sampled on a regular grid as input. The input data is converted into a compressed hierarchical wavelet representation in a preprocessing step. During rendering, the wavelet representation is decompressed on-the-fly and rendered using hardware texture mapping. The level of detail used for rendering is adapted to the local frequency spectrum of the data and its position relative to the viewer. Using a prototype implementation of the algorithm we were able to perform an interactive walkthrough of large data sets such as the visible human on a single of-the-shelf PC.

Categories and Subject Descriptors: E.4 [Coding and Information Theory]: Data Compaction and Compression I.3.1 [Computer Graphics]: Picture and Image Generation – Graphics processors; I.3.3 [Computer Graphics]: Picture and Image Generation – Viewing algorithms

Keywords: Compression Algorithms, Level of Detail Algorithms, Scientific Visualization, Volume Rendering, Wavelets

1 INTRODUCTION

Many areas in medicine, computational physics and various other disciplines have to deal with large volumetric data sets that demand for an adequate visualization. An important visualization technique for the exploration of volumetric data sets is direct volume rendering: Each point in space is assigned a density for the emission and absorption of light and the volume renderer computes the light reaching the eye along viewing rays. The rendering can be implemented efficiently using texture mapping hardware: the volume is discretized into textured slices that are blended over each other using alpha blending [6].

Due to the enormous advances in graphics hardware, it is nowadays possible to perform this rendering technique in real-time on cheap of-the-shelf PCs [11, 23, 24]. However, the size of the data sets that can be processed is still very limited. A realtime rendering of large data sets (more that 256^3 voxel) is currently infeasible unless massive parallel hardware is used [3].

Most conventional hardware-texturing based approaches to volume rendering are brute-force methods, requiring a rendering time linear in the size of the data set. The rendering costs can be reduced dramatically by using a multi-resolution hierarchy. In this case, the rendering algorithm performs a *projective classification* to adapt the rendering resolution to the distance to the viewer, as proposed by LaMar et al. [21]. We will show formally in this paper that the rendering time for this technique is indeed $O(\log n)$ for a data set consisting of n^3 voxels. However, two problems still remain that prevent us from handling very large data sets: The first problem is the enormous size. The well known visible human data set [33] consist e.g. of 6.5 GB ($2048 \times 1216 \times 1877$ voxel, 12 bit), i.e. it is even larger than the address space of a conventional PC. Thus, the data must be stored out-of-core and swapped into main memory on demand. This leads to considerable bandwidth and latency problems. The second problem is the size of the voxel data that remains after projective classification: Although the size is $O(\log n)$, the constants in the "O-notation" are still much too high. The number of voxels exceeds by far the texture memory as well as the alpha-blending capacities of a commodity graphics board.

Our novel algorithm uses a hierarchical wavelet representation to tackle these problems: The volume is stored as a hierarchy of wavelet coefficients. Only the levels of detail necessary for display are decompressed and sent to the texturing hardware. The use of a wavelet representation allows us to compress the data by a ratio of typically 30:1 without noticeable artifacts in the image. This way, even very large data sets can be stored in main memory. The visible human data set can e.g. be stored in 222MB (instead of 6.5GB). During rendering, the wavelet representation allows us to analyze the local frequency spectrum in the data set and to adapt the rendering resolution to it. This way, we can reduce the size of the voxel set to be rendered considerably with minimal loss of image quality.

Using these techniques, we are able to render walkthroughs of large data sets in real time on a conventional PC. We will demonstrate an interactive walkthrough of the visible human data set at a resolution of 256^2 pixel, 10 frames per second and good image quality. To our knowledge, our algorithm is the first that achieves these framerates for data sets of this size on a single of-the-shelf PC.

As an alternative, one could think of using texture compression supported by the graphics hardware. However, as shown by Meissner et. al. [23] this severely reduces the image quality and is therefore unusable. Also other compression approaches that allow direct rendering of the compressed data, like vector quantization [25,26], discreet cosine transformation [38] and fractal compression [9], do not perform as well as a wavelet based compression scheme.

The remainder of the paper is structured as follows: In the next section, we will briefly review related work. Then, we will describe the hierarchical wavelet representation in Section 3. In Section 4, we will describe the rendering algorithm and caching strategies. Results are discussed in Section 5 and the paper concludes in Section 6 with some ideas for future work. The appendix contains a formal analysis of the running time of the rendering algorithm.

2 RELATED WORK

Visualization of large volume data sets is a classical problem in computer graphics. In this section, we will give a brief overview of related work in the area of volume visualization algorithms, multi-resolution methods and wavelet-based techniques.

Volume Visualization: The most efficient software-based technique for direct volume rendering is the shear-warp factorization by Lacroute et al. [20]. The technique can be adapted to exploit 2D-texturing hardware [29], achieving interactive frame rates. The usage of 3D texture mapping [1] allows for more flexibility and can provide a higher image quality. Recent visualization algorithms provide advanced shading techniques such as lighting [24], shadows [4], high quality post-classification using a pre-integration technique [11], gradient magnitude modulation [34] or

higher dimensional transfer functions [19]. Our algorithm uses a pre-integration approach combining lighting and gradient magnitude modulation, as described in [23].

Multi-resolution rendering: Multi-resolution volume rendering algorithms use a spatial hierarchy to adapt the resolution to the projection onto the screen: An octree or a similar spatial data structure is built for the data set. Each node of the octree contains a representation of the volume within its bounding box at a specific resolution. During rendering, nodes from the hierarchy are selected such that their resolution matches the display resolution. The technique was first proposed by Chamberlain et al. [8] in the context of rendering of surface models. They prove a logarithmic running time if surface fragments are distributed uniformly in space. We will derive a similar result for the volumetric case.

A similar technique was applied to volume rendering by LaMar et al. [21]: The octree nodes store volume blocks resampled to a fixed resolution that are rendered using 3d-texturing hardware. Weiler et al. [35] propose an extension to the algorithm to avoid discontinuity artifacts between different levels of detail. These techniques can handle volume data sets that do not fit completely into the texture memory of the graphics hardware. However, the data must still fit into main memory. Therefore, large data sets like the visible human cannot be processed. Our algorithm improves on this by using a more efficient wavelet representation that allows storing data sets that are larger by one or two orders of magnitude. Additionally, we use a refined error criterion for the selection of octree nodes. It automatically adapts to the local smoothness of the data set, as proposed by Boada et al. [5] for the case of orthographic projection.

Wavelet Based Techniques: Wavelet-based encoding has become a standard technique for 2d-image compression [31]. The technique has been applied to the compression of volume data by several authors. Nguyen et al. [27] propose a blockwise compression scheme: The volume is divided into a regular grid of blocks which are compressed independently. Guthe et al. [16] propose a higher order wavelet compression scheme with extensions for encoding animated data sets. The method does not allow for access to parts of the volume without decompression of the whole data set. In our paper, we use the same basic techniques for encoding the volume data set as in the two aforementioned papers. However, our data structure provides a multi-resolution hierarchy with fast access to each node in the hierarchy.

To render large data sets using wavelet-based representations, two directions have been followed up to now: Firstly, several raycasting techniques were proposed that operate on a wavelet representation [17, 18, 30, 36]. However, raycasting of large data sets is not possible at interactive frame rates unless massive parallel hardware is used [3].

A second technique renders "x-ray" images directly from the wavelet representation by adding splats corresponding to the basis functions [15]. Unfortunately, it is not possible to extend this elegant technique to conventional volume rendering with emission and absorption effects.

3 WAVELET HIERARCHY

The first step of our algorithm is to convert the volume data, which is given as a three-dimensional array of integers with fixed precision (usually 8-16 bits), into a compressed wavelet representation during preprocessing. This representation is much more compact and allows for an efficient extraction of different levels of detail of the data set, since the wavelet transformation is equivalent to applying a series of lowpass and highpass filters to the original data. To be able to decompress parts of the data set efficiently, we apply a blockwise wavelet compression strategy.

3.1 Blockwise Hierarchical Compression of Volume Data

Firstly, we divide the data sets into cubic blocks of $(2k)^3$ voxels (in practice, $k=16$ is a good choice). Then, we apply the wavelet filters to each block. This results in a lowpass filtered block of k^3 voxels and $(2k)^3 - k^3$ wavelet coefficients representing different high frequency components that are no longer present in the lowpass filtered block (see Figure 1 and Figure 2). We carry on this scheme hierarchically: We group a cube of 8 adjacent lowpass filtered blocks to again obtain a block of $(2k)^3$ voxels. Then we can apply the filtering algorithm to this block recursively until only a single block is left. The result of this procedure is an octree (see Figure 2): Each node of the octree describes a volume of k^3 voxels and contains a set of high frequency coefficients that allow for the reconstruction of the child nodes from the current node. The resolution of a child node is twice as high (in each dimension) as that of a parent node. The lowpass filter of the specific wavelets we use assures that the downsampled data in the inner nodes does not show relevant aliasing artifacts.

3.1.1 Wavelet Basis

As basis functions, symmetric biorthogonal spline wavelets [10] are a good choice, as they lead to good compression results (they are also used in the JPEG 2000 standard). We use the tensor product construction (non standard decomposition, [31]) to obtain a three-dimensional basis of these functions. This means that the three-dimensional filtering is performed by applying the one dimensional filter in all three dimensions successively. We implemented the filtering using the integer wavelet transformation algorithm by Calderbank et al. [7] based on lifting steps. It provides some performance benefits: Firstly, all calculations can be performed using 16 bit integer arithmetic [32], saving memory and bandwidth in comparison to the floating point algorithm. The operations can be implemented efficiently using SIMD instructions like MMX. We use the Intel C++ compiler that applies some of these optimizations automatically. Secondly, the algorithm needs only about half the number of operations of the normal wavelet transformation algorithm.

For the examples in our paper, we use a linearly interpolating spline wavelet. This wavelet basis already allows for a very good compression ratio but it has still a small filter support (5/3 for the lowpass/highpass filter). A small support is desirable as the running time of the (de-)compression algorithm is linear in the number of non-zero entries in the (reconstruction) filter matrix. However, the strongest argument for choosing this wavelet is the property that an increase of the resolution with zero wavelet coefficients leads to a linear interpolation of the low resolution function, which is consistent with the interpolation performed by the texturing hardware used for rendering. This results in fewer popping artifacts when the resolution changes.

Our compression algorithm is block based. As the support of the filter is several voxels, we need a special treatment at the borders of the blocks. We use symmetric extension [10]: The original data is just mirrored at the border. This allows for a reconstruction without storing additional wavelet coefficients for values outside the block because our basis functions are symmetric.

As known from image compression literature, a blockwise compression can lead to discontinuity artifacts at the borders between different blocks. However, such artifacts only become visible at high compression ratio. In our work, we are interested in near-lossless compression because we do not want to introduce relevant compression artifacts into the renderings.

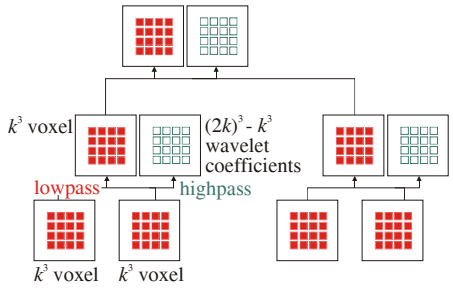


Figure 1: Construction of the wavelet tree.

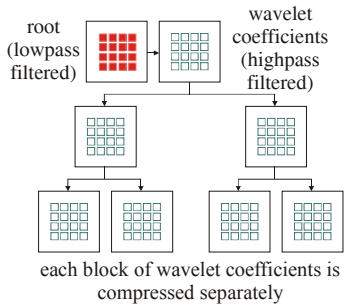


Figure 2: The compressed wavelet tree.

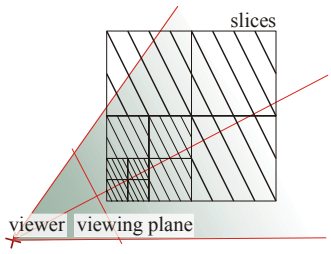


Figure 3: Multi-resolution rendering with view-plane aligned slices.

3.1.2 Compression

The compression consists of two steps: Firstly, wavelet coefficients of low importance are discarded and secondly, the wavelet coefficients must be encoded in a compact bit stream.

We reduce the number of wavelet coefficients to be stored by defining a threshold below which all coefficients are mapped to zero. Setting the threshold to zero leads to lossless compression: Due to the integer wavelet transform, there is no quantization error [7]. The fully lossless setting already permits compression ratios of up to 4:1 for typical data sets.

After choosing the relevant wavelet coefficients, they must be encoded efficiently. Codebook based approaches such as LZW (Lemple Ziv Welsh) or LZH (Lemple Ziv Huffman) can not be applied for this task since the codebook itself is larger than the data contained in a single node of our hierarchy most of the time. Progressive and embedded encoding schemes [13,14,22] on the other hand need some of the data of their parent nodes during decompression. To circumvent this, we use entropy coding with a suitable encoding model:

The coefficients are first mapped to positive values: Odd values represent positive coefficients ($c \rightarrow c \times 2\text{-}1$) while even values representing negative coefficients ($c \rightarrow c \times (\text{-}2)$). For compression of these values, two different algorithms have been implemented. Arithmetic coding, using the same model as Guthe and Straßer [16], is the best choice for maximum compression at a lossless or nearly lossless setting.

Run-length encoding combined with a fixed Huffman encoder on the other hand results in a very fast decompression, about ten times faster than arithmetic coding. The fixed model for the Huffman coder is defined as follows. A run of zeros is marked by a leading 0 bit. The following 7 bits store the number of consecutive zeros. This results in 1 to 128 zeros encoded in a single byte. Any other coefficient is converted into a positive value and stored by using n 1 bits, with n being the minimum number of bits needed to represent the coefficient. After a 0 bit the coefficient is stored using n-1 bits without the first bit.

The compression ratio for run-length Huffman coding at a lossless setting is lower (in practice about 10-15%) than for arithmetic coding. For a very lossy setting, the run-length Huffman coder is sometimes even able to outperform the arithmetic coder in terms of compression ratio since the adaptive model of the arithmetic coder is optimized for a large number of non-zero coefficients. To obtain higher compression ratios, sub-trees of the hierarchy containing only zero coefficients are completely stripped away. This stripping has more influence on the compression ratio than the coding of the blocks itself. For the compression setting used in the example walkthroughs, the increase of compression ratio is about 15% for the run-length Huffman coder and about 3% for arithmetic coding. This gain increases dramatically if the compression becomes more lossy.

4 RENDERING

From the perspective of the rendering algorithm, we have now a representation of the volume data in form of a multi-resolution octree: The root node in the tree contains a very rough approximation of the data set and the resolution can be increased by a factor of 2 (in each dimension simultaneously) by going downwards the hierarchy to a child node. Our task is to extract the information relevant for a certain point of view. This is done in two steps: Firstly, we perform a *projective classification* step to adjust the resolution of the data set to the screen resolution (Section 4.1). Secondly, we incorporate a consideration of the approximation error into the classification algorithm to further reduce the amount of data to be processed in each frame (Section 4.2). After extracting a suitable level of detail from the wavelet tree, we render the volume data using hardware texture mapping (Section 4.3). Rendering of walkthrough animations can be accelerated substantially by applying a suitable caching scheme (Section 4.4).

4.1 Projective Classification

Firstly, we need to extract nodes from the octree so that the resolution of these nodes matches the display resolution. Nodes outside the view frustum should be excluded from rendering. The task can be done using a straightforward algorithm originally proposed by Chamberlain et al. [8]: We traverse the hierarchy recursively, starting from the root node. For each node, we test whether it is located completely outside the view frustum. In this case, we stop the traversal, ignoring the current node. Otherwise, we determine the spacing between the voxel grid and project it to the screen. If it is equal to or below the screen resolution, we pass the node to the renderer. Otherwise, if the voxel resolution is still too coarse, we subdivide the node and apply the algorithm recursively to all 8 children.

This technique was already applied to volume data by LaMar et al. [21]. We will proof in the appendix that the technique reduces the rendering time for an n^3 voxel grid from $\Theta(n^3)$ to $\Theta(\log n)$. However, the analysis also shows that the constants hidden in the "O-notation" are very high. For a close-up of a volume with a depth of 2048 voxels, we still obtain more than 230 million voxels after projective classification (see appendix for details). This is about 4 times more than the texture memory of a typical contemporary graphics board (230MB versus 64MB). Therefore, we need a refined classification criterion for a further reduction.

4.2 View-dependent Priority Schedule

In most data sets, only a few regions contain high frequency details (e.g. due to sharp borders). Most regions can be sampled at a low sampling rate without sacrificing detail resolution. We utilize this observation to reduce the amount of voxels that has to

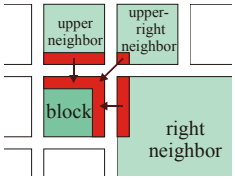


Figure 4: Copying data from neighbors for the 3d-texture blocks.

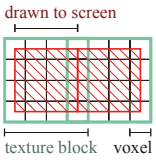


Figure 5: Texture interpolation, the blocks overlap each other by half a voxel.

be processed by the renderer: For each node in the wavelet tree, the L_2 error compared to the original data is measured during compression. During rendering we use this error as weight for the selection of nodes: Let $E(i)$ be the L_2 error of the normalized basis functions for the wavelets in the subtree below the node i. For leaf nodes, $E(i) := 0$.We can assign each node i a priority $P(i):=E(i)/z(i)$, with $z(i)$ being the minimum depth of a voxel in the node. Dividing by $z(i)$ accounts for the projection on the screen: The priority of nodes near the viewer should be higher than that of nodes far away. If $z(i) = 0$, we set the priority to *infinity*.

Using this priority function, we perform a generalized projective classification: We choose a maximum amount of voxel that we are able to process in the rendering stage. This is usually determined by the texture memory of the graphics board. We create a priority queue and insert the root node r of the hierarchy into the queue with priority $P(r)$. Then we successively fetch the node with the highest priority from the queue, decompress its high frequency wavelet coefficients and insert the child with the highest priority into the queue. A flag is set for the node to indicate that the child node has been added to the queue (all other children would still be drawn using the low resolution representation from the parent node). If all children are in the priority queue, the parent node is removed from the queue. Nodes with a projected voxel distance that is already equal to or below the screen resolution are not subdivided. The algorithm stops if the maximum amount of voxels for the nodes in the queue is reached.

4.3 Rendering of Blocks

Up to now, we have chosen a set of tree nodes, each containing k^3 voxels (on a regular grid) that provide a suitable approximation to the original volume for the current view point. To render these voxels, we use hardware texture mapping: We draw all blocks in back-to-front order. The order can be established easily by enforcing a back-to-front traversal order of the octree. For each block, a 3d-texture is created and loaded onto the graphics hardware. We place viewplane aligned slices into the block (see Figure 3) and render these slices in back-to-front order. Alpha blending delivers the volume integrals along viewing rays for all pixel on the screen.

For each block, we have to apply a classification function that assigns RGBα values to the scalars in a user defined way. To obtain a high rendering quality, especially in areas close to the viewpoint where the original data set is undersampled, we apply pre-integrated rendering [11]: We consider two adjacent slices in a block (called a *slab*) and determine the scalars at the position where the viewing ray enters and leaves the slab. For all 256^2 possible combinations of entry and exit values, the volume integral is precomputed numerically. The scalar values between entry and exit point are interpolated linearly. The precomputed lookup table is stored as a 256^2 RGBα texture on the graphics hardware. As hierarchy blocks of different resolution also have a different slice spacing, we compute such a preintegration lookup table for each possible slice spacing. During rendering, two adjacent slices are accessed by the texturing hardware. The scalar values at the entry and exit position are read from the texture using bilinear interpolation. The two values are interpreted as two dimensional texture coordinates that are used to fetch the preintegrated RGBα-value from the precomputed preintegration texture using dependent texture lookups [2, 28].

The trilinear interpolation performed by the texturing hardware needs special attention: The hardware is not able to interpolate across the borders of the octree blocks. This can lead to objectionable artifacts that reveal the underlying block structure. Our solution to this problem is straightforward: For each block to be rendered, we also fetch its 7 neighbors with the next higher x-, y- and z-coordinates from the octree (Figure 4). If these nodes are not present in the rendering set, the corresponding node is also decompressed and cached, but the neighbor's neighbors are of course not reconstructed. This lookup is not very expensive as a neighbor search in an octree can be done in expected time of O(1). We enlarge the block to be rendered by one voxel in x-, y- and z-direction and store the neighboring values there[1]. Using the additional voxels, we can perform a continuous linear interpolation (Figure 5). The texture memory necessary for rendering is increased by this technique because adjacent blocks overlap each other by one voxel. The overhead is $k^3 - (k-1)^3$ for k^3 voxels. For 16^3 voxel blocks we obtain an overhead of 21% and for 32^3 voxel blocks, the overhead is 10%.

For the examples in our paper, we also implemented some additional shading techniques like gradient magnitude modulation and classification of material properties. Details on the implementation of theses techniques using texturing hardware can be found in Meissner et al. [23]. The gradient information necessary is computed on-the-fly after the decompression of the volume data using a three dimensional Sobel operator.

4.4 Caching

Although our wavelet decompression algorithm already achieves a very high performance, we would not be able to perform an interactive walkthrough if we decompressed the wavelet representation for each frame from the scratch. It is not possible to perform a decompression and texture upload at a similar speed as the 3d-texturing is done by the graphics card on current hardware architectures. Fortunately, this is not necessary either, as we may anticipate reusing most of the decompressed data for subsequent frames. Therefore, we use three cache areas to store blocks for reuse:

Firstly, we cache decompressed volume blocks from the octree. To obtain a node in the octree, we must access its parent node, decompress the high frequency coefficients stored in the node and apply the reconstruction filter to obtain all 8 child nodes. The nodes consist of blocks of k^3 16 bit integers. The decompressed wavelet coefficients are not cached as these are only needed once to obtain the child nodes which are already cached. Caching is done according to an LRU-scheme. To maximize the performance of our algorithm, the user defines a fixed amount of cache memory. If we run short of memory, we always delete that decompressed leaf node in tree that was not accessed by the renderer for the longest time.

Secondly, we have to create 3d-textures from the cache. The texture contains the scalar values and optionally the corresponding gradient field for advanced shading effects[2]. Using again an LRU scheme, we fetch the most recently used subset of decom-

[1] As textures must have extents of a power of two, we must use blocks of size 2^n-1 in the wavelet tree (e.g.15^3 voxel).

[2] The gradients are stored as 8 bit RGB values and the scalars are stored in the alpha channel of the RGBα texture. The shading is done using pixel shaders similar to the approach of Meissner et al. [23].

pressed blocks and convert them into OpenGL texture objects. Gradient maps are computed at this point, if necessary.

Thirdly, the texture objects must be uploaded to the texture memory of the graphics adapter before rendering. This is done automatically by the OpenGL driver, again using an LRU caching scheme. By setting corresponding memory restrictions (see Section 4.2), the renderer assures that we do not use more texture objects per frame as fit into a given amount of video memory, thus avoiding cache thrashing.

5 RESULTS

In this section, we discuss the results obtained with a prototype implementation of our algorithm. The algorithm was implemented in C++ using OpenGL with nVidia extensions for rendering. All benchmarks were performed on a 2Ghz Pentium 4 PC with 1GB of ram and an nVidia GeForce 4 Ti4600 graphics board with 128MB of local video memory. In the following, we start with a description of three example data sets that we use to evaluate our algorithm. Then, we discuss the influence of the compression efficiency on the running time and image quality. After that, we discuss the results for interactive examination of the three example data sets.

5.1 Example Data Sets

We use three different data sets for the evaluation of our algorithm. All three are too large to be visualized at interactive framerates using conventional brute-force rendering approaches.

The first data set is a computer tomography scan of a Christmas tree [37] at a resolution of $512 \times 512 \times 999$ voxel with 12 bits per voxel. The data set was acquired at the technical university of Vienna to provide a large benchmark scene for volume rendering algorithms. The other two data sets are the visible human male and female data sets [33]. Both are computer tomography scans of a male and a female human body. We use the variants of the data sets that are registered against the cryosection RGB images. The visible human male data set has a resolution of $2048 \times 1216 \times 1877$ voxel and the visible human male data set has a resolution of $2048 \times 1216 \times 1734$ voxel. The example renderings were made using gradient based lighting and a classification function with several semi-transparent iso-surfaces. The iso-surfaces correspond to high derivatives in the classification function. These settings are very sensitive to noise and other reconstruction errors in the volume data and thus allow a good evaluation of the errors introduce by our rendering technique.

5.2 Compression Efficiency

In the compression algorithm, we have the option to use different encoding algorithms for the wavelet coefficients. We have implemented two alternatives: arithmetic coding and run-length Huffman coding. The decompression speed heavily depends on the compression algorithm. Using arithmetic coding, we achieve a decompression speed of 4.5 MB/s, including the wavelet reconstruction. The run-length Huffman codec is able to decompress 50 MB/s (including the wavelet reconstruction). The compression ratio of the arithmetic coding is typically only about 10% to 15% higher than that of the run-length Huffman coding. Therefore, we use the run-length Huffman coding for all examples in our paper.

A second parameter of the compression algorithm is the threshold for removing small wavelet coefficients prior to encoding. If we keep all coefficients, we obtain a lossless compression scheme. Using lossless compression, we achieve a compression ratio of 3.9:1 (arithmethic coding) and 3.4:1 (RLE-Huffman coding) for the Christmas tree data set. The visible human data sets could not be compressed using the lossless settings because the compressed data and the caches would exceed the 2GB address space. For higher compression ratios, we must apply lossy compression: Figure 6 shows the dependency between compression ratio and reconstructed signal quality for the three different test data sets: We obtain a peak signal-to-noise ratio (PSNR) of 60 dB for a compression ratio[3] of about 12:1 (1 bit per voxel), while a PSNR of 50 allows a compression ratio of roughly 50:1 (0.25 bits per voxel). Figure 8 shows a visual comparison of the rendering results for the Christmas tree data set. The compression ratios obtained by our algorithm at a given PSNR are close to the results of Nguyen and Saupe [27]. These results show that it is possible to achieve good compression results although we use only linear interpolating wavelets and blockwise compression.

Another important parameter is the block size used for the construction of the wavelet hierarchy. If we use small blocks, we are able to classify the data according to local frequency spectra and projected size very accurately. However, we have high hierarchy traversal costs. If we use larger blocks, the traversal costs decrease but we must process more voxels for the same image quality because our classification is less accurate. Additionally, the block size must be a power of two (minus one, for neighboring voxels, see Section 4.3) due to OpenGL restrictions. In practice, 31^3 blocks are not adaptive enough and 7^3 blocks introduce too much overhead. 15^3 blocks are a good compromise. We use this block size in all examples in this paper.

5.3 Interactive Walkthroughs

We applied our algorithm to render an interactive walkthrough of the three test data sets. The results are shown in Figure 9 (see also the accompanying video for a real-time capture of the walkthroughs). The resolution of the output image is 256^2 pixel for all tests. The Christmas tree data set was compressed using lossless compression (3.4:1), the visible human data sets were compressed using lossy compression. (40:1 for the femal and 30:1 for the male data set). The preprocessing time was 1 hour for the Christmas tree and about 5 hours for each of the two visible human data sets. The preprocessing times are dominated by hard disk access (seek times). The CPU-utilization was only 6-7% during compression.

During the walkthrough, we can adjust the quality parameter to trade off image quality for rendering speed. The quality parameter is given as the maximum projected error value for the rendered hierarchy nodes. We use three different settings with high, medium and lower image quality. The high quality settings uses up to 2048 blocks and a maximum projective error of $^1/_{128}$ (an average error of $^1/_{128}$ of the peak signal per pixel for each block) and therefore shows only very little artifacts due to a reduced resolution. Nevertheless, we still obtain an average framerate of 3-4 frames per second during the walkthrough. The low quality settings uses only 512 blocks and a maximum projective error of $^1/_{32}$ thus permitting framerates of about 10 frames per second at an acceptable image quality. The medium quality setting is a good compromise with 1024 blocks and a maximum projective error of $^1/_{64}$: The image quality is still high at a rendering speed of about 7 frames per second. The rendering speed for the visible human male data set is lower than that of our other test data sets, because the data set contains more noise. Thus, a higher voxel resolution is necessary to obtain the same projected error as in the other example scenes.

The cache efficiency for our walkthrough settings is very high. During the high quality rendering of our test dataset, only 40-60 blocks have to be decompressed per frame and 20-30 textures have to be constructed on the average. If we deactivate the caching, i.e. perform wavelet decompression, gradient calculation, and transfer to graphics memory from the scratch for each frame,

[3] All compression ratio measurements are based on 12 bit datasets.

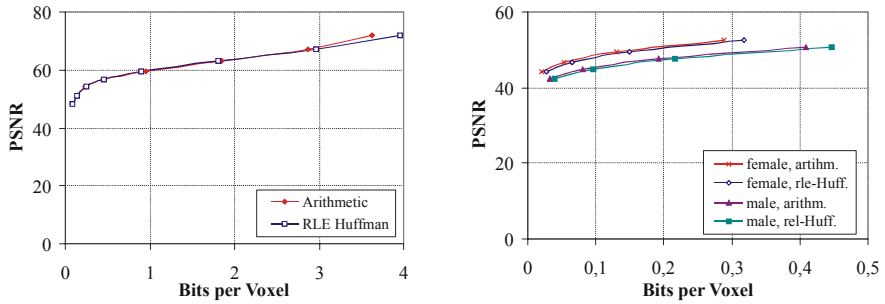


Figure 6: PSNR for Christmas tree dataset, and the visible human dataset. The compression of the male dataset is not as good as for the female dataset because of a higher noise in the ice surrounding the body.

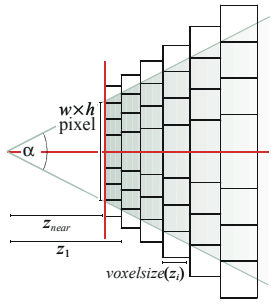


Figure 7: Analysis of the projective classification strategy (appendix).

we obtain an average framerate of 0.3 fps for all of our test scenes. This is also the limit framerate if we had no temporal coherence, i.e. a turn of 180 degrees or moving to a random position within the dataset. For this test, the renderer was configured for highest quality, i.e. to use exactly 2048 volume blocks. Thus, the framerate corresponds to a processing speed of 614 blocks per second or 10 MB of texture data per second.

To measure the exact timing of each part of the visualization is not an easy task in itself. This is due to the concurrent execution, i.e. decompression and gradient calculation are already executed while waiting for the last frame to complete rendering. For the example walkthrough animations about 6% of the time are spend for decompressing blocks and an additional 5% are spend for the gradient calculations. Transferring the textures onto the graphics board consumes another 1% of the time (part of this already runs in parallel), while the vast majority of the time with 88% is spend for the actual rendering, i.e. the processor is waiting for the graphics hardware.

The animation still shows some popping and discontinuity artifacts due to different resolutions in the rendered blocks. This is only a minor problem for high quality settings, but clearly visible for the low resolution settings. It should be quite straightforward to reduce these artifacts by employing mipmapping and techniques similar to [35]. This will be subject of future work.

6 CONCLUSIONS

We presented a rendering algorithm for the visualization of very large data sets. The algorithm uses a hierarchical wavelet representation to store very large data sets in main memory. The algorithm extracts the levels of detail necessary for the current view point on-the-fly. An error metric that minimized the loss of high frequency information in the projected image is used to determine a suitable level of detail. This technique allows interactive walkthroughs of large volume data sets like the visible human data set on a single commodity PC. To our knowledge, our algorithm is the first that achieves an interactive visualization of data set of this size on a single PC.

Our rendering algorithm scales provably good. Thus, we believe that data sets of even much larger size than the visible human data set can be processed. To overcome the storage problems if even the compressed data set does not fit into main memory any longer, we should generalize our caching technique to swapping to hard disk. We believe that the compressed representation will be useful in an out-of-core scenario, too, as it can significantly reduce the necessary bandwidth. A special problem of out-of-core rendering is latency due to hard disk seek times. To circumvent this problem, the data must be transferred in large blocks and stored in caches in main memory. A (at least lossless) compression scheme would be useful to reduce the corresponding memory overhead. Other future directions should include improved rendering techniques to minimize discontinuity artifacts between different resolutions [35] and a generalization to full RGBα volume data without classification, for example for rendering the cryosection visible human data, too. It would also be interesting to examine whether the wavelet coefficients in each block can be used more effectively to obtain a better adaptation of the rendering resolution to the local frequency spectrum.

ACKNOWLEDGEMENTS

Part of this work has been funded by the SFB grant 382 of the German Research Council (DFG).

APPENDIX: Analysis

How efficient is octree-based projective classification? To answer this question we first assume that we could discretize the volume in voxels of arbitrary size (see Figure 7). Parameters to the algorithm are a camera position and a constant vertical viewing angle of α. We also assume w.l.o.g.[4] that the original resolution of the voxel grid exactly matches the display resolution of $w \times h$ pixel at the near clipping plane z_{near}. To cover the whole volume, we add m layers of resampled cube-shaped voxels with side length $voxelsize(i)$, $i = 1..m$, so that the projected size of the larger voxel still matches the display resolution. Let z_i be the depth of voxel layer i. Then obviously $z_{i+1} = z_i + voxelsize(i)$ and

$$voxelsize(i) = \frac{\tan\alpha/2}{h/2} z_i = q \cdot z_i \text{ with } q := \frac{\tan\alpha/2}{h/2}.$$

This recurrence leads to $z_i = z_{near}(1+q)^i$. Let z_{far} be largest depth of a voxel in the volume. Then we can bound the number of layers of resampled voxels to:

$$m = \frac{\log\left(z_{far}/z_{near}\right)}{\log(q+1)}$$

Thus, the number of resampled voxels is $m \cdot w \cdot h$. Note that the ratio z_{near}/z_{far} is always bounded by the maximum diameter of the data set (measured in voxels). For a volume of n^3 voxels the diameter is at most $\sqrt{3}n \in O(n)$. Therefore, we obtain a total amount of $O(\log n)$ resampled voxels.

Up to now, our analysis still neglects the fact that we cannot access resampled voxels of arbitrary size but only octree nodes. This leads to two different kinds of overhead: Firstly, we are forced to use blocks of k^3 voxels (typically $k = 16$) of the same, fixed resolution. Secondly, we can choose the resolution in powers of 2 only (in each dimension). We consider the overhead due to the blocking first: Using some elementary trigonometry, we see that the number of voxels per unit length does not increase by more than a factor of

$$\rho_{max} = 1 + \sqrt{3}\frac{2k\tan\alpha/2}{h}$$

between the foremost and the most distant voxel in each block. The bound can be derived by considering blocks diagonal to the viewing direction and comparing the number of voxels per unit length. The voxel density

[4] There is no problem if the near clipping plane is closer to the viewer: As the discretization in voxel is never finer than the original resolution of the data set, there are always less than $w \cdot h \cdot \cot\alpha \in O(1)$ voxel in front of z_{near}.

per unit area is given by the density per unit length squared. Thus, the average factor of increase of voxels due to the blocking in blocks of k^3 voxels is given by:

$$overhead_{block} = \int_1^{\rho_{max}} x^2 dx / (\rho_{max} - 1) = \left(\frac{\rho_{max}^3}{3} - \frac{1}{3}\right) / (\rho_{max} - 1)$$

For typical block sizes k, this leads only to a small overhead (h = 256, α = 45°):

k	8	16	32	64	128
overhead	4,6%	9,2%	19,0%	40,2%	88,9%

However, the overhead is increased due to the fact that the resolution can be changed only in powers of two. This is easy to quantify: If we assume that we need all scales of resolution between 1^3 and 2^3 voxels with equal probability, we obtain an average oversampling factor of $\int_1^2 x^3 dx = 3.75$. This factor usually dominates the factor due to the blocking.

Example: For a resolution of 256^2 pixel, 90° vertical viewing angle, and a depth of 2048 voxels we obtain 858 layers containing 56 million resampled voxels. The approximation with an octree with blocks of 16^3 voxels increases the amount of voxels to at most 230 million voxels. A 2048^3 data set contains 8.6 billion voxels.

In conclusion, we see that projective classification using an octree leads to a running time logarithmic in the size of the input data. However, the constants hidden in the O-notation are fairly high. Thus, the algorithm scales very good but additional techniques are necessary to obtain interactive performance, as described in our paper.

References

[1] Akeley, K: RealityEngine graphics. In: *Siggraph 93 Conference Procedings,* 109–116, 1993.

[2] ATI. Developer Relations. http://www.ati.com/

[3] Bajaj, C., Ihm, I, Koo, G., Park, S.: Parallel Ray Casting of Visible Human on Distributed Memory Architectures. In: *Data Visualization '99*, 1999.

[4] Behrens, U., Ratering, R.: Adding shadows to a texture-based volume renderer. In: *IEEE Symposium on Volume Visualization*, IEEE, ACM SIGGRAPH, 39–46., 1998.

[5] Boada, I., Navazo, I., Scopigno, R.: Multiresolution volume visualization with a texture-based octree. In: *The Visual Computer*, 17(3), 185–197. Springer, 2001.

[6] Cabral, B., Cam, N., Foran, J.: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *Symposium on Volume Visualization*, 1994.

[7] Calderbank, R., Daubechies, I., Sweldens, W. Yeo, B.: Wavelet transforms that map integers to integers. Technical Reprort, Department of Mathematics, Princeton University, 1996.

[8] Chamberlain, B., DeRose, T., Lischinski, D., Salesing, D., Snyder, J. Fast rendering of complex environments using a spatial hierarchy. In: *Graphics Interface '96*, 132–141, 1996.

[9] Cochran, W.O., Hart, J.C., Flynn, P.J.: Fractal Volume Compression. In: *IEEE Transactions on Visualization and Computer Graphics*, 313-322, December 1996

[10] Daubechies, I.: *Ten Lectures on Wavelets*. CBMS-NSF Lecture Notes Nr. 61, SIAM, 1992.

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

[12] Ertl, T., Westermann, R., Grosso, R.: Multiresolution and hierarchical methods for the visualization of volume data. *Future Generation Computer Systems*, 15(1), 31–42, 1999.

[13] Fowler, J.E., Fox, D.N.: Embedded Wavelet-Based Coding of Three-Dimensional Oceanographic Images with Land Masses. In: *IEEE Transactions on Geoscience and Remote Sensing*, 284-290, February 2001

[14] Fowler, J.E., Fox, D.N.: Joint Embedded Coding of Data and Grid Using First-Generation Wavelet Transforms. In: *IEEE Data Compression Conference*, 432-442, 2002

[15] Gross, M.H., Lippert, L., Dittrich, R. Häring, S.: Two methods for wavelet-based volume rendering. In: *Computers and Graphics*, (21)2, 237-252, 1997.

[16] Guthe, S., Straßer, W.: Real-time decompression and visualization of animated volume data. In: *IEEE Visualization 2001,* 2001.

[17] Ihm, I., Park, S.: Wavelet-based 3D compression scheme for very large volume data. In: *Graphics Interface '98*, 107–116, 1998.

[18] Kim, T., Shin, Y.: An efficient wavelet-based compression method for volume rendering. In: *Pacific Graphics '99*, 147–157, 1999.

[19] Kniss, J., Kindelmann, G., Hansen, C.: Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In: *IEEE Visualization 2001*, 255–262, 2001.

[20] Lacroute, P., Levoy, M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In: *Computer Graphics*, 28 (Annual Conference Series), 451–458, 1994.

[21] LaMar, E.C., Hamann, B., Joy, K.I.: Multiresolution techniques for interactive texture-based volume visualization. In: *IEEE Visualization '99*, pages 355–362, 1999.

[22] Machiraju, R., Zhu, Z., Fry, B., Moorhead, R.: Structure Significant Representation of Computational Field Simulation Datasets. In: *IEEE Transactions on Visualization and Computer Graphics*, April-June 1998

[23] Meißner, M., Guthe, S., Straßer, W.: Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In: *Graphics Interface 2002*, 209-218, 2001.

[24] Meißner, M., Hoffmann, U. Straßer, W.: Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering using OpenGL and Extensions. In: *IEEE Visualization '99*, 207–214, 1999.

[25] Ning, P., Hesselink, L.: Vector Quantization for Volume Rendering. In: *Workshop on VolVis '92*, 69-74, 1992

[26] Ning, P., Hesselink, L.: Fast Volume Rendering of Compressed Data. In: *IEEE Visualization '93*, 11-18, 1993

[27] Nguyen, K.G., Saupe, D.: Rapid High Quality Compression of Volume Data for Visualization. In: *Computer Graphics Forum*, 20(3), 2001.

[28] NVIDIA. Developer Relations. http://www.nvidia.com.

[29] Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., Ertl, T.: Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-Textures and Multi-Stage Rasterization. In: *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[30] Rodler, F.: Wavelet based 3D compression with fast random access for very large volume data. In: *Pacific Graphics '99*, 108–117, 1999.

[31] Stollnitz, E.J., DeRose, T.D., Salesin, D.H.: *Wavelets for Computer Graphics: Theory and Applications*, Morgan Kaufmann, 1996.

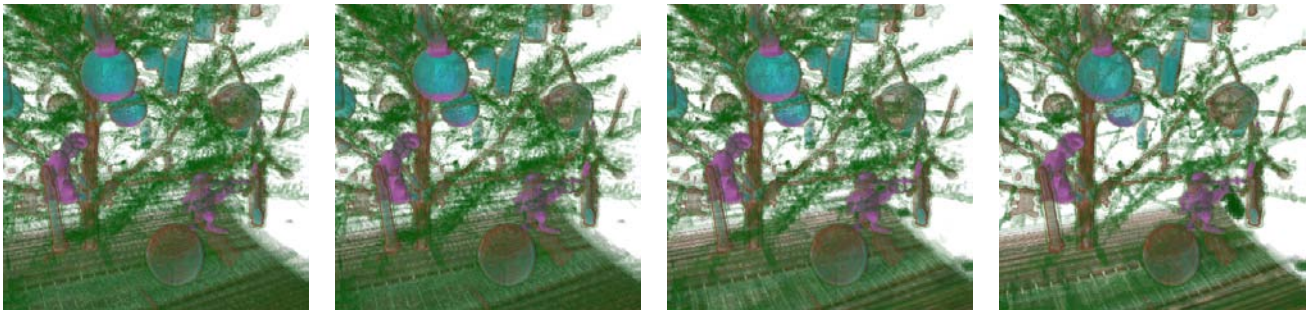
[32] Sweldens, W., Schröder, P.: Building your own wavelets at home. In: "Wavelets in Computer Graphics", *SIGGRAPH Course Notes*, 1996.

[33] The National Library of Medicine. The Visible Human Project. http://www.nlm.nih.gov/research/visible/visible_human.html.

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

[35] Weiler, M., Westermann, R., Hansen, C. Zimmerman, K., Ertl, T.: Level-of-detail volume rendering via 3d textures. In: *IEEE Volume Visualization and Graphics Symposium*, 2000.

[36] Westermann, R.: A multiresolution framework for volume rendering. In: *Symposium on Volume Visualization*, 51–58, 1994.



(a) lossless compression, compression ratio 3.4:1

(b) compression ratio 8.8:1, PSNR 63 dB

(c) compression ratio 66:1, PSNR 53 dB

(d) compression ratio 195:1, PSNR 48 dB

Figure 8: comparison of the image quality at different compression ratios (Christmas-tree data set)

(a) high quality, 3.1 fps

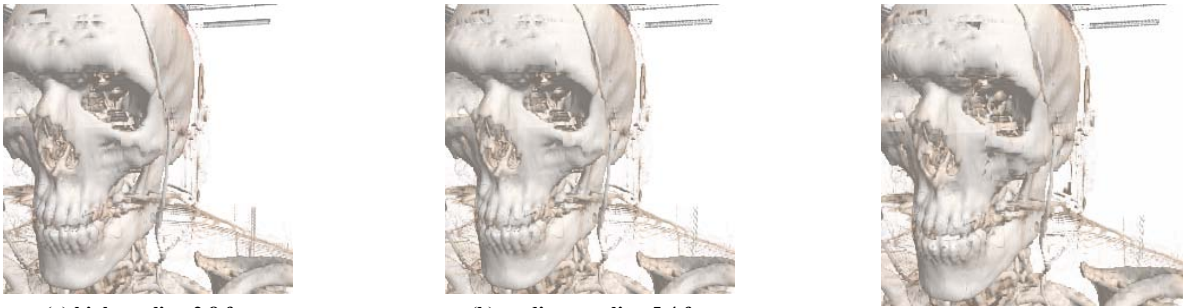
(b) medium quality, 7.6 fps

(c) low quality, 9.8 fps

(d) high quality, 3.3 fps

(e) medium quality, 6.6 fps

(f) low quality, 9.5 fps



(g) high quality, 2.8 fps

(h) medium quality, 5.4 fps

(i) low quality, 7.1 fps

Figure 9: Comparison of the image quality for interactive walkthroughs of the Christmas tree, visible human female and visible human male data set. The framerates were measured as averages for a camera path through the whole data set. The image resolution is 256 × 256 pixel.

[37] Christmas Tree Data Set. http://ringlotte.cg.tuwie.ac.at/datasets/XMasTree/XMaxTree.html

[38] Yeo, B-L., Liu, B.: Volume Rendering of DCT-Based Compressed 3D Scalar Data. In: *IEEE Transactions on Visualization and Computer Graphics*, 29-43, March 1995