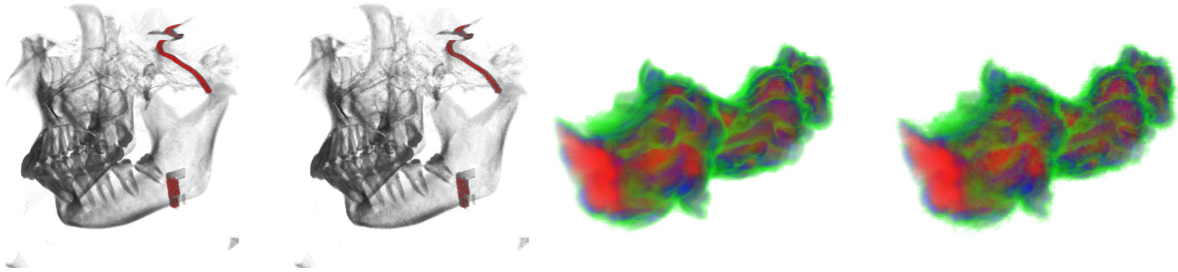
Compression Domain Volume Rendering

Jens Schneider* and Rüdiger Westermann†
Computer Graphics and Visualization Group, Technical University Munich



Results overview: First, a volumetric scalar data set of size 256^3 *requiring 16 MB is shown. Second, the hierarchically encoded data set (0.78 MB) is directly rendered using programmable graphics hardware. Third, one time step (*256^3*) of a 1.4 GB shock wave simulation is shown. Fourth, the same time step is directly rendered out of a compressed sequence of 70 MB. Rendering the data sets to a* 512^2 *viewport runs at 11 and 24 fps, respectively, on an ATI 9700.*

Abstract

A survey of graphics developers on the issue of texture mapping hardware for volume rendering would most likely find that the vast majority of them view limited texture memory as one of the most serious drawbacks of an otherwise fine technology. In this paper, we propose a compression scheme for static and time-varying volumetric data sets based on vector quantization that allows us to circumvent this limitation.

We describe a hierarchical quantization scheme that is based on a multiresolution covariance analysis of the original field. This allows for the efficient encoding of large-scale data sets, yet providing a mechanism to exploit temporal coherence in non-stationary fields. We show, that decoding and rendering the compressed data stream can be done on the graphics chip using programmable hardware. In this way, data transfer between the CPU and the graphics processing unit (GPU) can be minimized thus enabling flexible and memory efficient real-time rendering options. We demonstrate the effectiveness of our approach by demonstrating interactive renditions of Gigabyte data sets at reasonable fidelity on commodity graphics hardware.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Color, shading, shadowing and texture; I.3.8 [Computer Graphics]: Applications

Keywords: Volume Rendering, Vector Quantization, Texture Compression, Graphics Hardware

1 Introduction

Over the last decade many articles have extolled the virtues of hardware accelerated texture mapping for interactive volume rendering. Often this mechanism was positioned as the latest cure for the software crisis in scientific visualization - the inability to develop volume rendering algorithms that are fast enough to be used in real-time environments, yet powerful enough to provide realistic simulation of volumetric effects. Despite all the great benefits that were introduced by the most recent texture mapping accelerators, however, one important issue has received little attention throughout the ongoing discussion in the visualization community: volumetric texture compression.

As the demand for high-resolution three-dimensional texture maps in visualization applications like medical imaging or computational fluid dynamics is continuously increasing, there is also an increasing need for effective texture compression schemes. Besides enabling the optimal exploitation of limited texture memory, texture compression has the potential to significantly reduce the load on the transmission channel between the CPU and the GPU. Therefore, however, methods need to be developed to directly decode the data stream on the graphics chip at run-time, yet still enabling interactive frame rates.

Apart from the aforementioned requirements, one additional requirement becomes important once the compression scheme has to be applied to large-scale data sets: encoding optimization. As we aim at the compression of large volumetric data and sequences, the

*schneider@glhint.de
†westerma@informatik.tu-muenchen.de

encoding step can easily consume hours to generate an appropriately compressed data stream. Thus, to provide a tool that has the potential to be used in real-world scenarios, it is necessary to specifically address performance issues.

In this paper, the emphasis is on proposing a novel approach to volumetric texture compression that satisfies the mentioned requirements. The key to our approach is an improved implementation of vector quantization. A vector quantizer essentially maps any input n-dimensional vector to a single index that references a codebook. The respective codebook entry or codeword carries a vector of equal dimensionality that is used to reproduce the original one. Usually, no such codebook is known a priori. Hence, encoding the input set means finding a partitioning of the input set that calculates the codebook.

Our proposed scheme can effectively be applied to multi-dimensional data, and it can thus serve as a basis for region encoding of static and time-varying scalar or vector fields. Our choice of technique was mainly driven by the requirement to perform the decoding by means of the functionality available on consumer class graphics accelerators. Alternative compression techniques for volumetric data, for instance RLE encoding, IFS [Fisher 1995; Saupe et al. 1996], or compression schemes based on wavelet and Laplacian hierarchies [Muraki 1993; Westermann 1994; Ghavamnia and Yang 1995; Gross et al. 1995; Ihm and Park 1998; Bajaj et al. 2001; Nguyen and Saupe 2001], although in some cases they result in even better compression ratios, do not allow for simultaneous decoding and rendering of the compressed data on the GPU in general.

At the core of the quantization step the original data is split into a multiresolution representation. A covariance analysis at each resolution level followed by a relaxation step to minimize residual distortions allows us to compute sets of representative values at varying resolution. Temporal coherence between successive time steps in non-stationary fields is exploited in various ways to speed up the quantization process.

The data set is finally encoded into a RGB index texture, each color component being an index into a 2D dependent texture that holds the codebook generated at a particular resolution level. To access the compressed data stream and to decode the data we employ the Pixel Shader API, a specific set of instructions and capabilities in DirectX9-level hardware that can also be accessed from within the OpenGL API via the GL_ARB_fragment_program extension [OpenGL ARB n. d.]. Both the index texture and the dependent textures necessary to look up codewords from the codebook can be accessed on a per-fragment basis.

Although the decoding of compressed data on the GPU slows down rendering performance, above all it enables the rendering of volumetric data sets that would not have fit into texture memory otherwise. Even if the compressed data stream does not fit into texture memory, e.g. long sequences of large volumetric data sets are rendered, we achieve a considerable performance gain due to the minimization of bus transfer. One drawback of our rendering approach, however, is the fact that only nearest neighbor interpolation can be employed during rendering. This is a direct implication of vector quantization in general, and it can only be overcome by means of a considerably more expensive decoding scheme.

To further improve rendering performance, we effectively employ the early z-test that allows us to discard fragments before they enter the fragment program. Based on the observation that in the current implementation the loss in rendering performance is due to the complex shader program used to decode the data, we discard the execution of the shader program for those fragments that would most likely decode empty space.

The reminder of this paper is organized as follows. In Chapter 2 we review related work. The basic description of the hierarchical decomposition scheme is subject of Chapter 3. In Chapter 4 we describe GPU-based decoding and rendering of the compressed data stream. An improved implementation of vector quantization as well as performance issues and the compression of time-varying data sets is discussed in Chapter 5. We conclude the paper with a detailed discussion, and we show further results of our approach applied to real data sets.

2 Previous Work

Over the last decades the effective use of textures for realistic image synthesis has been demonstrated in various computer graphics applications. In particular, graphics chip manufacturers have spent considerable effort on the design of dedicated hardware to achieve real-time 2D and 3D texture mapping. As a result, hardware assisted texture mapping has now positioned itself as one of the fundamental drawing operations to achieve interactivity in applications ranging from shading and lighting simulation to advanced volume rendering techniques.

On the other hand, with the increasing attraction of texture maps limited texture memory becomes a major concern, and one is faced the problem to extend texture volume without increasing local texture memory. Particularly in volume rendering via 2D or 3D textures [Cabral et al. 1994; Westermann and Ertl 1998; Rezk-Salama et al. 2000; Engel et al. 2001; Kniss et al. 2001; Kniss et al. 2002], limited texture memory has become an important issue that often prohibits the rendering of large-scale data sets at the maximum possible frame rates.

With the focus on the application to typical 2D textures as they arise in computer games, like stones, bricks and walls, chip manufacturers have already proposed hardware supported decoding of compressed texture maps. Based on compression technology provided by S3 Inc., these activities have lead to the S3 texture compression standard [OpenGL ARB n. d.], which is now part of the DirectX and OpenGL APIs. In the S3 standard, 4x4 texel values are grouped together and represented by two average color samples and 4 linear interpolation factors, each of which is addressed via 2 bit per texel. Although the S3 texture compression scheme yields decent fidelity when applied to the aforementioned textures, from the numerical point of view it generates insufficient results due to the linear fit that is performed even in non-homogeneous regions. In addition, due to the maximum compression ratio of 8:1, it is not appropriate for the application to volumetric textures or even time-varying data sets in general.

Customized compression schemes for 3D textures, on the other hand, have the potential to produce considerably better compression ratios. For texture based volume rendering, Guthe et al. [Guthe et al. 2002] employed wavelet transforms to significantly compress separate texture tiles. Although resulting in high fidelity, every tile has to be decoded before it can be transferred and rendered on the GPU.

If an application demands for the rendering of time-varying data sets, data compression becomes an even more important requirement. Hierarchical data structures and difference encoding schemes have been proposed to detect spatial and temporal coherence in consecutive time steps, and to use this information to accelerated the rendering process [Westermann 1995; Shen and Johnson 1994; Shen et al. 1999]. In general, however, none of these techniques has been shown to be able to decode the data on the graphics chip.

First approaches in which volumetric data decoding and rendering was performed on the graphics chip were proposed in [Lum et al. 2001] for time-varying sequences, and in [Kraus and Ertl 2002; Li and Kaufman 2002] for static data sets. The former approach is based on a transform coding of the original signal. DCT-coefficients are quantized and encoded into hardware assisted color tables, which can be reloaded interactively to generate animations at interactive rates. In the latter techniques, relevant areas of the

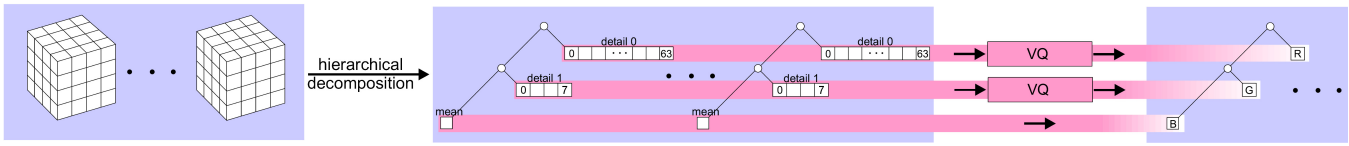


Figure 1: *The hierarchical decomposition and quantization of volumetric scalar data sets is illustrated. Blocks are first split into multiple frequency bands, which are quantized separately. This generates three index values per block, which are used to reference the computed codebooks.*

original texture are packed into a texture atlas of reduced size. Decoding is finally done by employing programmable graphics hardware to reconstruct the appropriate information for each rendered fragment.

A quite challenging alternative to the proposed compression schemes is vector quantization, which essentially enables the automatic computation of a reduced set of representative values used to approximate the original samples at reasonable fidelity.

In computer graphics many different applications have already benefit from vector quantization. Particularly in imaging applications, where true color images have to be displayed on devices exhibiting limited color resolution, the quantization of color samples has been used frequently to find an appropriate mapping [Heckbert 1982; Gervauz and Purgathofer 1990; Orchard and Bouman 1991; Buhmann et al. 1998]. Vector quantization has also been used in volume rendering applications for compression purposes by treating continuous data blocks as multi-dimensional vectors to be encoded [Ning and Hesselink 1992], and for the quantization of grey-scale gradients to be used further on in 3D texture based volume rendering to simulate local illumination [Van Gelder and Kim 1996]. The compression of light fields based on vector quantization was considered in [Levoy and Hanrahan 1996; Heidrich et al. 1999]. In the latter approach, Heidrich et al. demonstrated the efficient use of texture color tables for the encoding and hardware assisted decoding of vector components. Tarini et al. [Tarini et al. 2000] suggested to use vector quantization for the interactive simulation of local lighting effects on surfaces. Two-dimensional normal maps were quantized using a customized quantizer such that the lighting computation only had to be performed for each of the generated table entries.

In contrast to the described applications of vector quantization, in our work the focus is on a somewhat different topic. In particular; the goal of our approach is two-fold: to demonstrate that real-time decoding and rendering of quantized contiguous texel regions in large volumetric data sets and sequences can be achieved by means of standard graphics hardware *and* to propose an enhanced vector quantization scheme that is extremely performant, yet resulting in high fidelity and in the ability to efficiently process multi-dimensional data.

3 Hierarchical Data Decomposition

Prior to the in-depth discussion of the proposed compression scheme, let us first outline the hierarchical setting our approach is based upon as well as the technique to render the hierarchically encoded data stream on programmable graphics hardware. Our implementation exploits the GL_ARB_fragment_program extension to the OpenGL API on the latest ATI technology, the ATI 9700. In particular, we exploit the possibility to perform arithmetic operations and dependent texture fetches on a per-fragment basis.

Starting with the original scalar field, the data is initially partitioned into disjoint blocks of size 4^3. Each block is decomposed into a multiresolution representation, which essentially splits the data into three different triadic frequency bands. Therefore, each block is down-sampled by a factor of two by averaging disjoint sets of 2^3 voxels each. The difference between the original data samples and the respective down-sampled value is stored in a 64-component vector. The same process is applied to the down-sampled version, producing one single value that represents the mean value of the entire block. The 2^3 difference values carrying the information that is lost when going from 2^3 mean values to the final one are stored in a 8-component vector. Finally, a 1-component vector stores the mean of the entire block. The basic procedure is illustrated in Figure 1.

In performing this task, the data is decomposed into three vectors of length 64, 8, and 1, respectively, which hierarchically encode the data samples in one block. This approach has two main advantages. Firstly, the detail or difference coefficients will most likely become small, or they are already zero in homogeneous or empty regions. By applying a simple thresholding, many of the difference vectors can be mapped to the zero vector. In this way, the number of codewords used to represent significant non-zero vectors is maximized. Moreover, the performance of the quantization process is considerably increased, because the relaxation process that assigns vector elements to quantization bins becomes a simple operation for zero vectors. Secondly, by quantizing each frequency band separately and by assembling the original signal from these contributions, we enhance fidelity by using combinations of entries stored in each band.

By means of the vector quantizer, which will be described below, appropriate mappings and codebooks containing 64- and 8-component codewords are computed for both high frequency bands. Let us therefore assume that the length of each codebook is 256, such that the respective index into the codebook can be stored as one 8 bit value. Let us also assume that mean values are stored in an 8 bit value. They can thus be used directly without the need to lookup the respective value in a codebook.

We thus end up with three 8 bit values per block: one value represents the mean of each block, while the other two values are indices into the respective codebooks representing the difference information. All three values are stored in one single RGB index texture, I, of size $(N_x/4)\text{x}(N_y/4)\text{x}(N_z/4)$. Here, N_x, N_y and N_z are the size of the original volume in every dimension.

The two codebooks are stored in two separate 2D textures C_1 and C_2 of size 256x64 and 256x8, respectively. They are indexed via (s,t) texture coordinates, the s coordinates being directly given by the G- and the B-component of I. To decode a particular block, its mean and the respective difference information from C_1 and C_2 have to be added. Both operations, the lookup of the difference information and the reconstruction of the final value can be done efficiently by means of per-fragment operations as described in the following.

4 Compression Domain Rendering

Starting with a 3D scalar data set, the hierarchical quantization scheme generates a RGB index set of 1/4 the original size in every dimension. This data set is converted to a 3D texture map, and it is rendered via hardware accelerated 3D texture mapping. In our

current implementation the texture is sliced either in back-to-front or front-to-back order via planes orthogonal to the viewing direction. Because each texel of the index texture is expanded to a 4^3 block during rendering, care has to be taken to perform the slicing with as many slices as necessary to render the original data.

To assemble each quantized block on a per-fragment basis, a shader program is issued that performs the following tasks:

- The respective texture sample is fetched from the index texture I.
- The difference vectors are indexed via the G- and B-components, and they are fetched from the dependent textures C_1 and C_2.
- The R-component of I (mean value) is added to the difference information.
- The final scalar value is mapped to color and opacity via a 1D dependent texture map (color table).
- Color and opacity is drawn to the framebuffer.

Although by the G- and B-component of the index texture I, for every fragment the s texture coordinates to index the respective codewords in C_1 and C_2 are uniquely determined, the relative address (t texture coordinate) of each fragment within this codeword is not yet available. Therefore, we issue an address texture, A, in the shader program. This texture is of the same size as the encoded texel region, 4^3 in the current example, and it is mapped via nearest neighbor interpolation. It stores relative addresses to access particular components in one codeword, and it is used by each fragment to determine the missing s texture coordinate.

In the R-component of A, 64 different addresses are coded to index C_1. In the G-component, however, only 8 different addresses have to be coded to assign the codewords in C_2. Therefore, blocks of 2^3 adjacent texels get assigned the same address, thus yielding 8 different values necessary to access C_2. For sake of simplicity, the basic idea is illustrated in Figure 2 for the encoding of 2x2 regions.

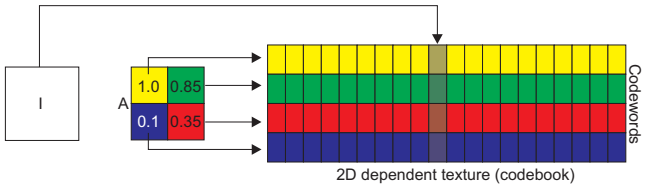


Figure 2: *Illustration of on-chip decoding for a 2x2 texel region. Texture values from two different texture sources are used to fetch a difference sample from the dependent 2D texture.*

Because the address texture only holds the addresses for one single block, an enlarged texture as large as the original data set with its content repeated across the domain has to be mapped. This approach, however, is not appropriate, because it does not allow for the saving of any memory at all. Consequently, we proceed in a different way. To access the index texture, for each slice that is rendered we issue texture coordinates ranging from 0 to $N_x/4$, $N_y/4$ and $N_z/4$, respectively, and we set the wrap parameter for texture coordinates to GL_REPEAT. Now, the address texture is periodically repeated over the domain thus yielding the correct addresses at every fragment. As a matter of fact, the G- and B-component of T together with the R- and G-component of A uniquely determine the addresses into the dependent textures C_1 and C_2, respectively.

4.1 Early Shader Termination

For the simultaneous decoding and rendering of the compressed data sets as proposed, we perceive a loss in performance of about a factor of 2-3 compared to the rendering of uncompressed data. Based on the observation that in the current implementation the loss in performance is due to the complex shader program, and in particular due to the many texture indirections, we discard the execution of the shader program for fragments that are most likely to decode empty space.

Therefore, we employ the early z-test, which is available on our target architecture. The early z-test discards a fragment and thus avoids execution of the pixel shader program, if depth values are not explicitly modified in the shader program and other per-fragment tests are disabled.

To exploit the early z-test we render each slicing polygon twice. In the first pass, a simple pixel shader program is issued. In the shader, index texture I is accessed and the mean value stored in the R-component is checked. If it is zero, then the fragment is rendered with color and opacity set to zero. In this way, it does not affect the color buffer, but the depth value in the depth buffer is set to the fragments depth. If the mean value is not zero, then the fragment is discarded by means of a kill instruction. This instruction does not allow one to gain performance, but it takes care that the fragment does neither affect the color nor the depth buffer. As a matter of fact, if a fragment gets discarded in the shader program, the depth value of the fragment that was rendered last is still valid in the depth buffer.

In the second pass, the slicing polygon is rendered again, but now the complex shader as described in the previous section is called. However, because the depth test is set to GL_GREATER, the early z-test only lets those fragments pass that have been discarded in the forgoing simple shader pass. All other fragments will be discarded before entering the complex shader program.

In this way, in empty regions only one texture fetch operation has to be performed to access the mean of each block. If this value is zero, then the difference information will be zero as well and the data sample does not have to be decoded. Particularly for the rendering of numerical simulation results, in which large empty regions occur quite frequently, the proposed acceleration technique results in a considerable speed-up.

5 Vector Quantization

Let us now briefly summarize the basic concept and the features provided by the improved vector quantizer that is used as a basis for volumetric texture compression and hardware accelerated texture decoding.

In its most general form a vector quantizer takes a n-dimensional vector as input and maps it to a single index that references a codebook. The respective codebook entry or codeword carries a vector of equal dimensionality that is used to reproduce the original one. Usually, no such codebook is known a priori. Hence encoding the input set means finding a partitioning of the input set that calculates the codebook. Because we aim at reproducing each input set as closely as possible vector quantization can be seen as a data fitting procedure that minimizes the residual distortion implied by the mapping with respect to some metric δ. The most common distortion metric used in data fitting processes is the squared-distance metric $\delta(x,y) \mapsto \|x-y\|_2^2$, which provides an intuitive measurement of the distortion.

Although vector quantization has been around for quite a long time, until now there was only a limited use of quantization schemes in computer graphics applications due to a number of drawbacks of existing algorithms. In our opinion the most serious drawback of vector quantization schemes is their inability to be used in time

critical applications. Usually, high quality quantizers show a significant lack in performance, which prohibits their application to high-resolution data. In addition, the extension of existing approaches to higher dimensions or multi-parameter data is not straight forward in general. This is due to the conceptual design of the algorithms and due to performance issues.

Linde, Buzo and Gray [Linde et al. 1980] developed one of the first vector quantization algorithms suitable for practical applications - the *LBG-algorithm* - which improved on the scalar quantizer proposed by Max [Max 1960] and by Lloyd [Lloyd 1982]. For an excellent introduction to and a comprehensive survey of vector quantization let us refer to [Gray and Neuhoff 1998; Sayood 2000]. While conceptually simple there are still some serious problems inherent to the LBG-algorithm, of which the most severe ones are execution speed and the so called *empty cell problem*.

In the following we will demonstrate that by integrating an enhanced splitting strategy to find optimal codebooks empty cells can be avoided automatically without any special treatment. In addition this approach makes vector quantization fast enough to establish it as attractive alternative to other compression schemes, yet providing excellent compression ratios at high fidelity. Our algorithm is easy to implement, and it is general enough to deal with high dimensional multi-parameter data. The latter property exposes our approach among previous ones, which are often restricted to applications in two or three dimensions.

5.1 LBG Revisited

Since our algorithm is a modification of the LBG-algorithm, let us start with a review of this algorithm.

1. Start with an initial codebook $C = \{Y_i^{(0)}\}_{i=1}^m \subset \Re^n$. Let $I \subset \Re^n$ be the set of input vectors. Set $k = 0$, $D^{(0)} = 0$ and select threshold ε.
2. Find quantization regions $V_i^{(k)} = \{X \in I : \delta(X,Y_i) < \delta(X,Y_j) \ \forall j \neq i\}$, where $j = 1,2,\ldots,m$.
3. Compute the distortion $D^{(k)} = \sum_{i=1}^m \sum_{X \in V_i^{(k)}} \delta(X,Y_i^{(k)})$
4. If $\frac{D^{(k-1)}-D^{(k)}}{D^{(k)}} < \varepsilon$ stop, otherwise, continue.
5. Increment k. Find a new codebook $\{Y_i^{(k)}\}_{i=1}^m$ by calculating the centroids of each cell $V_i^{(k-1)}$. Go to (2).

The final output of the algorithm is a codebook $C = \{Y_i^{(k)}\}_{i=1}^m$ and a partition $\{V_i^{(k)}\}_{i=1}^{2^r}$ of I, where r is the fixed bit-rate of the quantization. Each input vector is then replaced by the index of the associated quantization cell.

The initial codebook for step (1) is usually obtained by means of a so called *splitting technique*. First, the centroid of the entire input set is placed as a single entry into the codebook. Then, a second entry is generated by adding a random offset to the first entry, and the LBG-algorithm is executed until convergence. This procedure is repeated until the desired bit-rate is achieved

While repeated nearest neighbor searches in step (2) slow down its performance significantly, the LBG-algorithm also suffers from the *empty cell problem*. Empty cells are the result of collapsing codebook entries during refinement steps. These entries are not detected by the algorithm, and as a consequence many codebook entries might be wasted. Although it is possible to explicitly detect and delete these entries, this alternative results in even more LBG-steps because the number of codebook entries is not going to be doubled in each iteration. Starting with as many codebook entries as there are input vectors and merging these entries until the desired codebook size is achieved solves the problem as well. This, however, requires $\frac{1}{2}n^2$ nearest neighbor searches for n input vectors and clearly disqualifies the approach when it comes to speed.

5.2 Covariance Analysis

To obtain the initial codebook an improved splitting technique needs to be integrated. A splitting based on a principal component analysis (PCA) followed by a relaxation-based optimization phase is one alternative solution to find an initial codebook. It enables us to choose an optimal splitting plane with regard to the variances of the disjunct subregions. The exploitation of such a splitting process has been described in various applications ranging from data clustering to load balancing, and it is essentially the technique used in [Pauly et al. 2002; Lensch et al. 2001] for the hierarchical clustering of point sets and scanned BRDFs.

The splitting technique proceeds as follows. We start with a single quantization cell V_1, which contains the entire input set I. The respective codebook entry Y_1 is the centroid of the entire set, and the distortion of this quantization cell is computed as $D_1 = \sum_{X \in V_1} \delta(X,Y_1)$. We then construct a double-linked "to-do" list by inserting the new "group" defined by $(D_1, Y_1, \Im_1 = \{i \in \Im : X_i \in V_1\})$ into this list. After subsequent splits this list is sorted in descending order with respect to the stored distortions D_j. In each iteration the element j with the largest residual distortion D_j is selected and split further on. We use this heuristic to predict the actual maximum decrease in distortion, without that we have to explicitly compute the gain. Because we no longer need to perform one split in advance, a considerable speed-up can be achieved in addition to improved quality of the codebook generated this way.

To perform one split, we proceed as follows:

1. Pick the group j with largest residual distortion D_j from the to-do list.
2. Calculate the auto-covariance matrix $M = \sum_{i \in \Im_j} (X_i - Y_j) \cdot (X_i - Y_j)^t$
3. Calculate the eigenvector e_{max} that corresponds to the largest eigenvalue λ_{max} of M
4. Split the original group into a "left" and a "right" group:
   $\Im_{left} = \{i \in \Im_j, \ < (Y_j - X_i), e_{max} > \ < 0\}$
   $\Im_{right} = \{i \in \Im_j, \ < (Y_j - X_i), e_{max} > \ \geq 0\}$
5. Calculate new centroids Y_{left} and Y_{right} along with new residual distortions D_{left} and D_{right}
6. Insert the two new groups into the to-do list.
7. If number of groups equals 2^r, stop, else go to 1.

This procedure essentially adds one codebook entry per split (see Figure 3). Since the splitting hyperplane passes through the old centroid, the new distortions will be small compared to the old one. Once the splitting procedure terminates, the codebook along with the residual distortion can be directly obtained from the to-do list.

The benefits of this approach are manifold. First, a cell with low residual distortion will never be split because it is always appended to the end of the list. In particular this includes cells that only contain one data point and thus have a residual distortion of 0. Second, if a cell is split it is divided into two sub-cells of roughly equal residual distortions. Third, applying some LBG-steps as post-refinement to relax the centroids quickly generates stable Voronoi regions (see Figure 4). Fourth, in all our examples we did never observe any empty cells during the LBG post-refinement. This is due to the fact that we always place centroids into densely populated cells. Fifth,

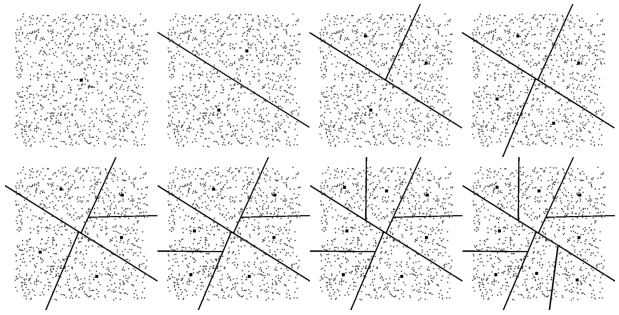


Figure 3: A series of PCA-Splits to obtain a first codebook.

our algorithm is extremely fast because it avoids expensive LBG-steps during the splitting process. It has a runtime of $O(N \cdot \log_2 m)$, where N is the number of input vectors and $m = 2^r$ is the number of codebook entries. Sixth, the algorithm is easy to implement and can be extended straight forwardly to any dimension. Moreover, since the numerical complexity is dominated by distortion evaluations the algorithm offers a huge potential for further optimizations based on latest SIMD-technology, such as SSE or 3DNow

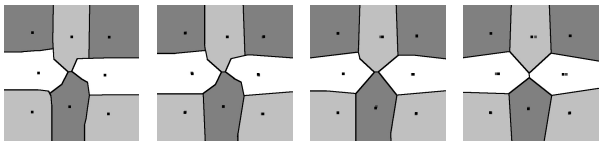


Figure 4: A series of post-refinements by means of LBG-steps applied to the codebook from Figure 3. Current centroids are marked by dark points, while bright points show the centroids from some previous iteration.

5.3 Performance Optimization

Regardless the mentioned improvements the total run-time of the algorithm is still dominated by the LBG-steps, which are employed to relax the codebook entries. On the other hand, these refinements are necessary because they significantly increase the resulting fidelity. As a consequence we further improved the performance of the enhanced LBG-algorithm as follows.

We restrict the expensive nearest-neighbor search that is performed during the LBG-refinement to a subset of the entire codebook entries. In the literature this is commonly referred to as *fast searching*. While there exist different approaches to determine the minimal codebook subset we favor a fast but solid heuristic. By observing that the possibility is high that during one LBG-step each data point migrates from the initial quantization cell into adjacent cells, we restrict the search to the *k-neighborhood* of the initial cell. Mutual distortions are calculated for each pair of centroids, and references to the k nearest neighbors are established for each entry. The k nearest neighbors are found by means of a modified Quicksort algorithm. This algorithm is essentially the one that is used to find the k smallest entries. In contrast to the full Quicksort, this sorting procedure only recurses for the partition that overlaps the k^{th} entry and consequently runs in linear time. Now the nearest neighbor search is restricted to the list of adjacent centroids and the adjacency information is updated after each LBG-step. Because for small k this procedure converges to a suboptimal distortion the search radius is continuously increased by some value whenever the convergence rate drops below 5%. This seems to be a reasonable choice, and it allows us to save up to 85% of the execution time compared to an exhaustive search. In both cases the algorithm terminates when no gain in distortion could be achieved or when a search radius k_{max} or an user-defined minimum distortion ε is reached.

For higher dimensions our improvement relies on the concept of *partial searches*. Because the distortion measure is essentially a scalar product $< X,Y >$ that increases with each evaluated dimension, the computation is stopped whenever the next contribution $X_i \cdot Y_i$ leads to a higher value than an initial distortion $\delta(X,Y_{init})$. Obviously, the correct selection of Y_{init} is very important because it triggers the number of computations to be performed. Since it is very probable for a data point not to change the associated quantization point, we initialize the distortion with respect to the nearest quantization point. In all our experiments this approach allows us to save about 50% of the accumulated calculations.

5.4 Region Encoding

One of the drawbacks of vector quantization as described is that as soon as the bit-rate decreases below 8 bpp artifacts become apparent. In addition, the compression ratio is limited to roughly 3:1 because each RGB texel is represented by one 8 bit color index. On the other hand, we know that encoding contiguous pixel regions can significantly improve the compression ratio, yet resulting in even better quality. This phenomenon can be explained by the fact that the number of possible colors is increased due to larger codewords at the same time decreasing the spatial resolution by exploiting coherences. Using this approach the number of index bits can be reduced considerably, while the length of the generated codewords is increased. This makes the approach very appropriate for large data sets, since they profit most from a compact index set.

In correspondence to that observation the quantizer was modified such as to accept vectors of arbitrary length as input set. This allows us to interpret enlarged 3D regions as (n^3)D vectors, which are fed into the quantizer to compute appropriate indices and corresponding codebooks. Note that scalar or RGBα samples can be handled in exactly the same way without any coding modifications.

Now, the set of texels within a region is represented by the same codebook index. The compressed texture map is of length N_x/n x N_y/n x N_z/n, where N_x, N_y and N_z is the size of the original data set in each dimension. The codewords, however, consist of n^3 consecutive scalar or color samples.

As described in Chapter 3, the proposed quantization scheme can be easily integrated into a hierarchical setting to generate a multiresolution representation. Fed with the appropriate input vectors, the quantizer produces codebooks at different resolution levels. Codewords can then be combined to produce the best fit for a particular input vector.

Some results of the proposed quantization scheme are shown in Figure 5, where a part of one slice of the Visible Human RGB data set and a 3D confocal microscopy scan of size 512^2x32 were encoded using the hierarchical compression scheme. As one can see, by using the hierarchical region encoder we achieve high fidelity at good compression ratios of 25.3:1 and 31.2:1, respectively. These ratios include the memory overhead required to store the dependent textures. In both examples, dependent textures were of size 256x16 and 256x4. The RGB index texture used to store the mean value and indices into the dependent textures was composed of 8 bit components.

5.5 Quantization of Time-Varying Sequences

To encode non-stationary data, we employ a so-called codebook-retraining algorithm. For a good survey of such algorithms, that are at the core of adaptive vector quantization schemes, let us refer to [Fowler 1996]. In general these algorithms partition the data stream

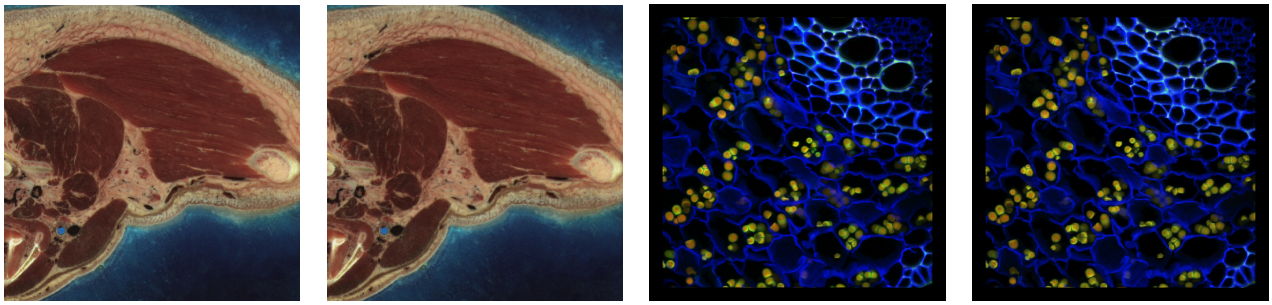


Figure 5: *This sequence demonstrates the effectiveness of the hierarchical quantization scheme. First, a 24 bpp true color image (first) was encoded in 0.95 bpp (second). Next, a 3D 32 bpp confocal microscopy scan (third) was encoded in 1 bpp.*

into frames that are to be encoded one at a time, but each frame can reuse the codebooks of previous frames in different ways. Firstly, a frame can be encoded separately, producing an index texture and 2 local codebooks for every time step. Secondly, codebooks can be retrained from the previous frame. Retraining proceeds by taking the previous codebooks as initial codebooks to the LBG algorithm. Since not only the codebooks but also the index sets can be reused to obtain a first guess to the final quantization, fast and partial searches are possible throughout the entire process.

Obviously, choosing the second alternative for all but the first time step produces the best results in terms of performance, because it only computes the PCA-Split for the first frame. For all other frames fast searches are employed to speed up the remaining LBG-relaxations. On the other hand, after some number of frames the current codebooks can become sub-optimal, since the LBG-algorithm depends on the choice of the initial codebook. It has thus become common practise to select special frames that are encoded separately, similar to key-frames, while the quantization for all other frames is obtained by retraining. This scheme shows remarkably good results. For instance, in figure 6 we show the same time step out of a shock wave simulation. In the first example the time step was encoded separately, while the second image was obtained using the 10th retrained codebook from the last key-frame.

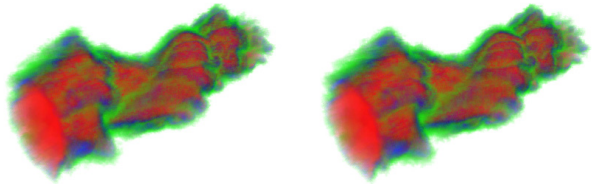


Figure 6: *This example demonstrates progressive encoding of time-varying sequences using I- and P-frames. On the left, time step 65 was separately encoded. On the right, the same time step was encoded using the initial codebook of time step 55, and by performing the LBG-relaxation on this codebook. Encoding time decreased from 29 minutes to 13 minutes for the entire sequence.*

6 Results and Comparison

In the following we will discuss the proposed quantization and rendering scheme in more detail, and we will give performance and quality measures for a variety of different data sets. All our experiments were compiled and run under WindowsXP on a P4 2.8 GHz processor equipped with 512 MB main memory and an ATI 9700.

In figure 7 the quality of the proposed quantization scheme is demonstrated for different compression modes - 4-color quantization, 2^2 region encoding and hierarchical encoding of 4^2 pixel blocks. We show the compression of a 2048x1216 24 bpp slice from the Visible Human data set. Processing time was 2.2, 5.0 and 10.0 seconds, respectively. Compression ratios were 12:1, 19.18:1 and 25.29:1, resulting in respective SNRs of 16.25dB, 25.97dB and 26.02dB. As one can see, the hierarchical quantization scheme yields slightly better results compared to 2^2 region encoding and generates a significantly better compression ratio. The images also show nicely the benefits of region encoding compared to color quantization of separate pixels.

The figures in 8 and 9 demonstrate the application of the hierarchical quantization scheme to volumetric data sets. All our examples are rendered using nearest neighbor interpolation, because linear interpolation within codebooks is not possible. Trilinear interpolation could be achieved by decoding 8 adjacent neighbors around each fragment in the shader program. On the other hand, this approach would considerably slow down performance and has not been considered here.

As in the previous 2D example, the results of different quantization modes are shown and compared to each other. From left to right, we show the original data set, region encoding using 2^3 and 4^3 texel blocks, and hierarchical encoding of 4^3 blocks. For the engine data set the compression rates are 7.98:1, 56.89:1 and 20.38:1. The SNR is 24.29 dB, 19.04 dB and 21.87 dB. For the skull data set the respective rates are 7.98:1, 60.24:1 and 20.84:1. The SNR is 14.86 dB, 9.96 dB and 11.70 dB.

Obviously, 4^3 region encoding yields the best results in terms of compression ratio, but it also produces significant quantization artifacts. Hierarchical encoding, on the other hand, achieves significantly better texture fidelity at reasonable compression rates. Rendering performance drops to roughly 1/3 and 1/2 of the performance of standard 3D texture based rendering. Due to our improved rendering scheme, which employs the early z-test to skip empty regions, performance strongly depends on the consistency of the data sets. The engine data set, for instance, exhibits noise even in those regions that have finally been suppressed be the selected transfer function. If noisy structures are removed from the data in advance, a considerable speed up can be achieved.

Hierarchical quantization took 19.0 and 50.6 seconds for the engine and skull data sets,respectively. Compared to other quantization tools, we achieve a significant speed up due the proposed performance optimizations. For instance, the Open Source vector quantization toolbox QccPack [Fowler 2000] is of a factor of 40 slower than the proposed scheme; a difference that is of important relevance if a scheme is to be used for the compression of time-varying data sets.

In the last figure 10 we demonstrate the effectiveness of the hierarchical quantization scheme and the GPU-based rendering approach for the display of time-varying data sets. Two different sequences are shown: a shock wave simulation consisting of 89 time steps, each of size 256^3, and a vortex flow simulation consisting of 100 time steps of size 128^3. Both sequences were compressed using the proposed progressive quantizing scheme. In this way, memory requirement dropped from 1.4 GB to 70 MB for the shock wave simulation, and from 200 MB to 11.1 MB for the vortex simulation. Due to progressive encoding, quantization of the vortex data set took roughly 5 minutes.

We should note here, that the original vortex data was quantized to 8 bits in advance. Thus, we can encode the mean value into an 8 bit color component without introducing large distortions. On the other hand, we could easily use the α-channel in the index texture to encode 16 bit mean values in the R- and α-component. In this way, for higher resolved data samples, fidelity can be improved significantly.

Rendering the sequences on our target architecture is performed with 24 fps and 16 fps, respectively. Note that in the first example, a considerable speed-up is achieved by taking advantage of the advanced rendering method that allows us to skip empty space. In this case, we are yet slightly faster than standard 3D texture based rendering.

7 Conclusion and future work

In this work, we have outlined a basis for volumetric texture compression and hardware accelerated texture decoding and rendering. Therefore, we have developed a hierarchical vector quantization scheme that is able to efficiently encode static and time-varying multi-dimensional data. With regard to performance, the proposed scheme significantly improves previous vector quantization schemes, and it achieves compression rates and fidelity similar to wavelet based compression. For the quantization of sequences, we have presented an acceleration technique that effectively takes advantage of temporal coherence between consecutive time steps. In this way, performance gains up to a factor of 3 could be demonstrated.

Furthermore, we have described a method to directly render hierarchically encoded data on programmable graphics hardware. We have employed DirectX9-level hardware to decode the data set and to achieve interactive frame rates even for large data sets. By effective use of the early z-test, we have considerably increased rendering performance. In this way, for sparse data sets we achieve performance rates similar or even better than those that can be achieved by rendering the uncompressed data sets. Although rendering is restricted to nearest neighbor interpolation, the proposed method allows for the interactive rendering of large data sets that would not have fit into texture memory otherwise.

In the future, we will investigate how to use our scheme to compress and render vector valued data. In particular, we will try to effectively compress large vector fields with regard to vector field topology. In addition, new rendering modes can be developed based on this technique, for instance palette based animation or template based flow representation.

Acknowledgements

Special thanks to K.-L. Ma and D. Silver for providing the Vortex sequence.

References

BAJAJ, C., IHM, I., AND PARK, S. 2001. 3D RGB image compression for interactive applications. *ACM Transactions on Graphics (TOG) 20*, 1, 10–38.

BUHMANN, J., FELLNER, D., HELD, M., KETTERER, J., AND PUZICHA, J. 1998. Dithered color quantization. In *EUROGRAPHICS '98*, 219–231.

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, 91–98.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*.

FISHER, Y., Ed. 1995. *Fractal Image Compression: Theory and Application*. Springer Verlag, New York.

FOWLER, J. 1996. *Adaptive Vector Quantization for the Coding of Nonstationary Sources*. PhD thesis, The Ohio State University.

FOWLER, J. E. 2000. Qccpack: An open-source software library for quantization, compression, and coding. In *Applicationd of Digital Image Processing XXIII (Proc. SPIE 4115)*, A. G. Tescher, Ed., 294–301. see also http://qccpack.sourceforge.net.

GERVAUZ, M., AND PURGATHOFER, W. 1990. *Graphics Gems*. Academic Press, ch. A simple method for color quantization: octree quantization, 287–293.

GHAVAMNIA, M., AND YANG, X. 1995. Direct rendering of laplacian pyramid compressed volume data. In *Proceedings of IEEE Visualization 1995*, 192–199.

GRAY, R., AND NEUHOFF, D. 1998. Quantization. *IEEE Transactions on Information Theory 44*.

GROSS, M., LIPPERT, L., DREGER, A., AND KOCH, R. 1995. A new method to approximate the volume rendering equation using wavelets and piecewise polynomials. *Computers and Graphics 19*, 1.

GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. 2002. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization 2002*, 104–115.

HECKBERT, P. 1982. Color image quantization for frame buffer displays. *Computer Graphics (SIGGRAPH 82 Proceedings)*, 297–307.

HEIDRICH, W., LENSCH, H., COHEN, M., AND SEIDEL, H.-P. 1999. Light field techniques for reflections and refractions. *Rendering Techniques '99 (Proceedings of Eurographics Rendering Workshop)*, 59–66.

IHM, I., AND PARK, S. 1998. Wavelet-based 3D compression scheme for very large volume data. In *Graphics Interface*, 107–116.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2001. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of IEEE Visualization 2001*, 255–262.

KNISS, J., PREMOZE, S., HANSEN, C., AND EBERT, D. 2002. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, 168–176.

KRAUS, M., AND ERTL, T. 2002. Adaptive texture maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, 7–15.

LENSCH, H., KAUTZ, J., GOESELE, M., HEIDRICH, W., AND SEIDEL, H.-P. 2001. Image-based reconstruction of spatially varying materials. In *Proceedings of the 12th Eurographics Workshop on Rendering*, 104–115.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. *Computer Graphics (SIGGRAPH 96 Proceedings)*, 31–42.

LI, W., AND KAUFMAN, A. 2002. Accelerating volume rendering with bounded textures. In *IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics 2002*.

LINDE, Y., BUZO, A., AND GRAY, R. 1980. An algorithm for vector quantizer design. *IEEE Transactions on Communications COM-28*, 1 (Jan.), 84–95.

LLOYD, S. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory 28*, 129–137.

LUM, E., MA, K.-L., AND CLYNE, J. 2001. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of IEEE Visualization 2001*.

MAX, J. 1960. Quantization for minimum distortion. *IRE Transactions on Information Theory IT-6*, 7–12.

MURAKI, S. 1993. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications 13*, 4, 50–56.

NGUYEN, K., AND SAUPE, D. 2001. Rapid high quality compression of volume data for visualization. *Computer Graphics Forum 20*, 13.

NING, P., AND HESSELINK, L. 1992. Vector quantization for volume rendering. In *Proceedings Workshop on Volume Visualization '92*, 69–74.

OPENGL ARB. The OpenGL Architecture Revision Board. http://www.opengl.org/developers/about/arb.html.

ORCHARD, M., AND BOUMAN, C. 1991. Color quantization of images. *IEEE Transactions on Signal Processing 39*, 2677–2690.

PAULY, M., GROSS, M., AND KOBBELT, L. 2002. Efficient simplification of point-sampled surfaces. In *Proceedings of IEEE Visualization 2002*, 171–177.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND T., E. 2000. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, 109–119.

SAUPE, D., HAMZAOUI, R., AND HARTENSTEIN, H., 1996. Fractal image compression - an introductory overview. ACM Siggraph '96 Course Note.

SAYOOD, K. 2000. *Introduction to Data Compression*, second ed. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA.

SHEN, H.-W., AND JOHNSON, C. 1994. Differential volume rendering; a fast volume rendering technique for flow animation. In *Proceedings of IEEE Visualization 1994*, 180–187.

SHEN, H.-W., CHIANG, L., AND MA, K.-L. 1999. A fast volume rendering algorithm for time-varying fields using time-space partitioning. In *Proceedings of IEEE Visualization 1999*, 371–377.

TARINI, M., CIGNONI, P., ROCCHINI, C., AND SCOPIGNO, R. 2000. Real time, accurate, multi-featured rendering of bump mapped surfaces. In *EUROGRAPHICS '00*, 112–120.

VAN GELDER, A., AND KIM, K. 1996. Direct volume rendering with shading via three-dimensional textures. In *Proceedings Symposium on Volume Visualization '96*, 23–31.

WESTERMANN, R., AND ERTL, T. 1998. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, 291–294.

WESTERMANN, R. 1994. A multiresolution framework for volume rendering. In *1994 Symposium on Volume Visualization*, ACM SIGGRAPH, A. Kaufman and W. Krüger, Eds., 51–58.

WESTERMANN, R. 1995. Compression domain volume rendering. In *Proceedings of IEEE Visualization 1995*, 168–176.

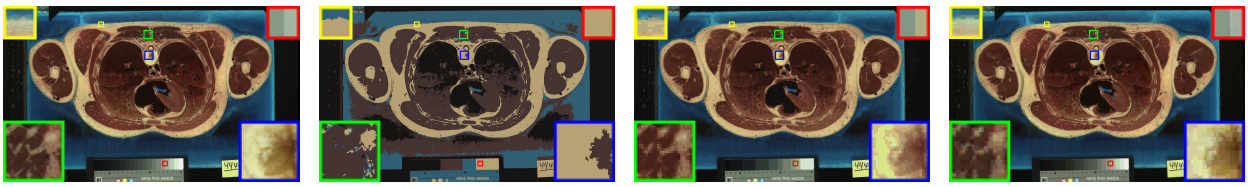


Figure 7: *First, the original RGB 24 bpp slice from the Visible Human data set is shown. Second, pixel encoding with 2 bpp is shown. Third, 2x2 regions are encoded in 1.25 bpp. Fourth, the data was hierarchically quantized and encoded in 0.94 bpp.*

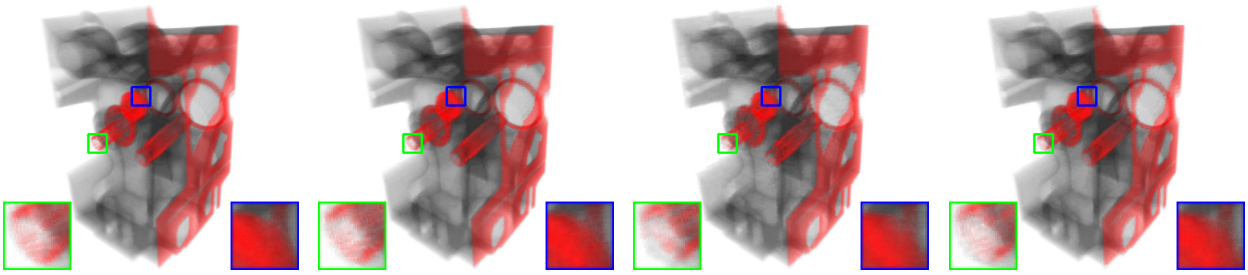


Figure 8: *The* 256^2x128 *engine data set (left) is quantized using different quantization modes. Second and third,* 2^3 *and* 4^3 *region encoding results in compression ratios of 7.99:1 and 56.89:1, respectively. Fourth, hierarchical quantization and encoding results in a compression ratio of 20.38:1. Rendering performance (*512^2 *viewport) is 19 fps, 14 fps, 16 fps and 12 fps, respectively.*

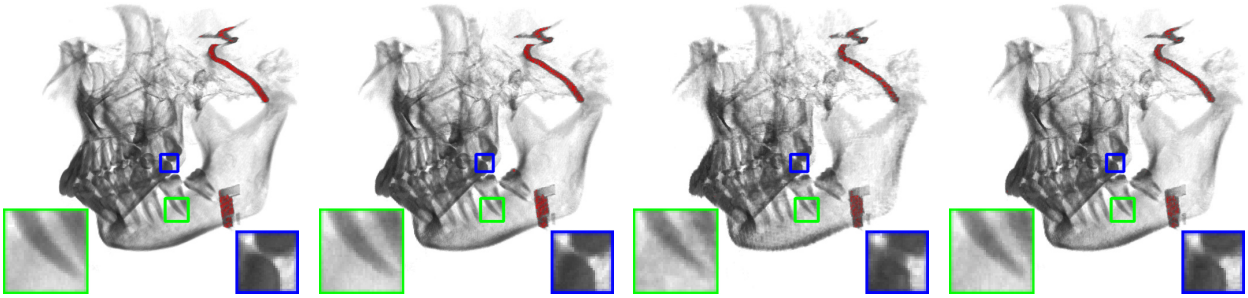


Figure 9: *The* 256^3 *skull data set (left) is quantized using different quantization modes. Second and third,* 2^3 *and* 4^3 *region encoding results in compression ratios of 7.98:1 and 60.24:1, respectively. Fourth, hierarchical quantization and encoding results in a compression ratio of 20.84:1. Rendering performance (*512^2 *viewport) is 14 fps, 10 fps, 11 fps and 11 fps, respectively.*

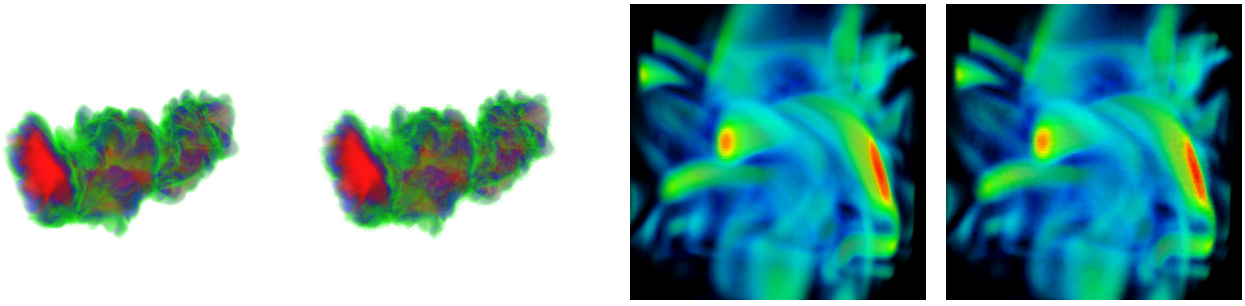


Figure 10: *Time steps from two different sequences are shown. First, time step 85 of a* 256^3x89 *shock wave simulation (1.4 GB) is shown. Second, from the hierarchically encoded data stream (70 MB), the same time step is directly rendered at 24 fps. Third, the first time step of a* 128^3 *x 100 (200 MB) vortex simulation is shown. Fourth, the sequence was encoded in 11.1 MB, and the compressed time step was directly rendered at 16 fps.*