Application-Driven Compression for Visualizing Large-Scale Time-Varying Volume Data

Chaoli Wang, Hongfeng Yu, Kwan-Liu Ma
University of California, Davis

Abstract

We present an application-driven approach to compressing large-scale time-varying volume data. Our method identifies a reference feature to partition the data into space-time blocks, which are compressed with various precisions depending on their association to the feature. Runtime decompression is performed with bit-wise texture packing and deferred filtering. We show that our method achieves high compression rates and interactive rendering while preserving fine details surrounding regions of interest. Such an application-driven approach points us to a promising direction for coping with the large data problem facing computational scientists.

Keywords

Large data visualization, time-varying data visualization, importance-based compression, bit-wise texture packing, deferred filtering.

Introduction

Leveraging the power of high-performance supercomputers and the advancement of numerical algorithms, scientists are able to perform three-dimensional direct numerical simulations of many complex phenomena in unprecedented details, leading to new scientific discoveries. Nowadays, a typical scientific simulation may produce data that contain several hundred million voxels, hundreds of time steps, and tens of variables. The vast amounts of data generated from simulations post a challenge to data visualization. Even though the past several years witnessed the great advancement of commodity graphics hardware, the size of the video memory in the current generation of graphics hardware accelerators is still limited to several hundred megabytes. Therefore, transferring data from the disk to the main memory and from the main memory to the video memory remains a key performance bottleneck for large data visualization.

To address this issue, different approaches have been proposed to effectively reduce the data sent to the graphics pipeline. These include different multiresolution data representations and a wide variety of data reduction techniques, such as quantization and transform-based compression. Many existing algorithms, however, do not explicitly take any domain knowledge or the visualization tasks into account for data representation and reduction. Although these algorithms are general and easy to use for different data sets, they often suffer from the fact that they are not tightly coupled with specific visualization needs. Therefore, they are not an optimal solution in terms of reduction efficiency and visualization quality tradeoff.

We advocate an application-driven approach to compressing and rendering large-scale time-varying scientific simulation data. We notice that quite often scientists have certain domain knowledge and specific visualization tasks in their minds. In the context of time-varying, multivariate volume data visualization, such

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

knowledge could be the salient isosurface of interest for some variable and the visualization task could be observing spatio-temporal relationships among other variables in the neighborhood of that isosurface. We have attempted to directly incorporate such knowledge and tasks into the whole data reduction, compression, and rendering process. Experimental results on two large-scale time-varying, multivariate scientific data sets show that:

- our approach is applicable to visualization of large-scale time-varying data where regions of importance can be identified;
- our method achieves high compression rates (around 20x) on disk by taking into account features of interest and spatio-temporal coherence in the data;
- our GPU-based *bit-wise* decompression and rendering solution proves effective for performance speedup and visualization quality tradeoff.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

Compression Methods for Time-Varying Volume Data

There exists a wealth of research in volume data compression and rendering. Here, we only review related work on time-varying data visualization. Shen and Johnson[1] proposed differential volume rendering that utilizes temporal coherence between consecutive time steps to compress the data and accelerate volume animation. An approach dealing with large-scale time-varying fields was described by Shen et al.[2]. The data structure, called the time space partitioning (TSP) tree, captures both the spatial and temporal coherence of the data. The TSP tree uses an octree for spatial partitioning and a binary tree for storing temporal information at each octree node. Both these approaches treat the spatial and temporal dimension separately. Alternatively, temporal and spatial dimensions can also be treated uniformly. For example, Wilhelms and Van Gelder[3] encoded time-varying data using a multidimensional tree (i.e., a 4D tree).

Other research efforts in time-varying data visualization focused on transform-based compression and rendering. Guthe and Straßer[4] introduced an algorithm that uses the wavelet transform to encode each spatial volume, and then applies a motion compensation strategy to match the volume blocks in adjacent time steps. Lum et al.[5] proposed to use the discrete cosine transform (DCT) to encode individual voxels along the time dimension, and employed a color table animation technique to render the volumes using texture hardware. Sohn et al.[6] described a compression scheme where the wavelet transform is used to create intra-coded volumes and difference encoding is applied to compress the time sequence. Schneider and Westermann[7] presented a hierarchical vector quantization solution to compress time-varying volumetric data where both decompression and rendering are performed at runtime in graphics hardware. Fout et al.[8] also used vector quantization for time-varying, multivariate volume data reduction in which correlations among related variables are exploited.

Research work closely related to ours includes that of Ma et al.[9] and Bajaj et al.[10]. Ma et al. used octree encoding and difference encoding for spatial and temporal domain compression, respectively. They also investigated how quantization might affect further compression, rendering optimization, and image results. Bajaj et al. classified voxels according to their importance in visualization, and assigned weights to them. To compress the volume data, they used the Haar wavelet transform and defined weight functions for wavelet coefficients based on voxels' weights in the same spatial-frequency locations.

References

1. H.-W. Shen and C. R. Johnson, "Differential Volume Rendering: A Fast Volume Visualization Technique for Flow Animation," *Proc. IEEE Visualization Conf.*, 1994, pp. 180–187.
2. H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree," *Proc. IEEE Visualization Conf.*, 1999, pp. 371–377.
3. J. Wilhelms and A. Van Gelder, "Multi-Dimensional Trees for Controlled Volume Rendering and Compression," *Proc. IEEE Symposium on Volume Visualization*, 1994, pp. 27–34.
4. S. Guthe and W. Straßer, "Real-Time Decompression and Visualization of Animated Volume Data," *Proc. IEEE Visualization Conf.*, 2001, pp. 349–356.
5. E. B. Lum, K.-L. Ma, and J. Clyne, "Texture Hardware Assisted Rendering of Time-Varying Volume Data," *Proc. IEEE Visualization Conf.*, 2001, pp. 263–270.

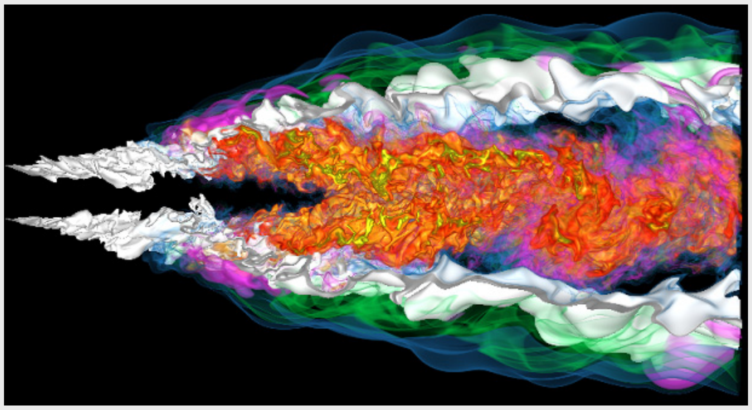
Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

6. B. S. Sohn, C. L. Bajaj, and V. Siddavanahalli, “Feature Based Volumetric Video Compression for Interactive Playback,” *Proc. IEEE Symposium on Volume Visualization*, 2002, pp. 89–96.
7. J. Schneider and R. Westermann, “Compression Domain Volume Rendering,” *Proc. IEEE Visualization Conf.*, 2003, pp. 293–300.
8. N. Fout, K.-L. Ma, and J. P. Ahrens, “Time-Varying Multivariate Volume Data Reduction,” *Proc. ACM Symposium on Applied Computing*, 2005, pp. 1224-1230.
9. K.-L. Ma, D. Smith, M.-Y. Shih, and H.-W. Shen, “Efficient Encoding and Rendering of Time-Varying Volume Data,” *Tech. Rep. ICASE Report No. 98-22*, Institute for Computer Applications in Science and Engineering, June 1998.
10. C. L. Bajaj, S. Park, and I. Ihm, “Visualization-Specific Compression of Large Volume Data,” *Proc. Pacific Graphics*, 2001, pp. 212–222.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

Turbulent Combustion Simulation

Scientists at the Sandia National Laboratories perform terascale turbulent combustion simulation to study the basic phenomena of reacting flows in the combustion process[1]. The scientific interest is on the main flame structure, which corresponds to the stoichiometric mixture fraction (*mixfrac*) surface at isovalue = 0.2. In particular, scientists would like to observe how other variables distribute along the main flame surface since knowing this is critical for evaluating the efficiency of the combustion process. As a matter of fact, this kind of analytical visualization is quite common in scientific data analysis. Such domain knowledge should be translated into a reference feature to guide our application-driven data compression and rendering. The intuition is that the closer a voxel to the surface of interest, the higher precision we should preserve in order to ensure reconstruction quality. In other words, the precisions of data should vary according to their associations to the reference feature.



Simultaneous rendering of two variables of the turbulent combustion simulation data set. The *mixfrac* surface (at the isovalue of 0.2) is rendered in white and the HO_2 variable is depicted in a volume rendering style.

References

1. E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen, “Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models,” *Journal of Physics: Conference Series (Proc. DOE SciDAC 2005 Conf.)*, vol. 16, 2005, pp. 65–79.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

Algorithm overview

Figure 1 illustrates the flowchart of our application-driven compression and rendering approach. The compression of a large time-varying data set is performed in a preprocessing stage. The compression starts with spatial partitioning and temporal grouping, which take into account domain knowledge provided by scientists. Accordingly, the time-varying data set is partitioned into space-time blocks and each space-time block is then compressed individually. At runtime, we first partially decode the compressed data blocks and perform bit padding in the CPU for texture loading. Then, we pack data blocks into graphics memory, perform deferred filtering to reconstruct and render the data in the GPU. The effectiveness of our solution lies in significant reduction in data transferring and efficient usage of limited graphics memory. This approach enables us to render large time-varying data sets interactively while preserving fine details around regions of interest for visual analysis.

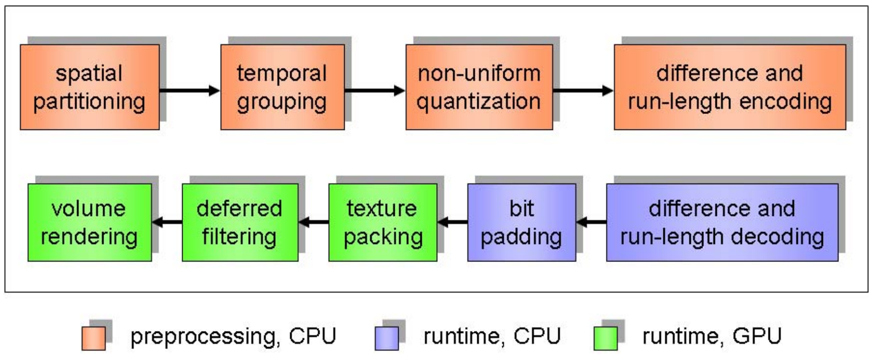


Figure 1 The flowchart of our application-driven compression and rendering approach.

Compression

Our compression scheme consists of three steps: first, we partition the volume data at each time step into a list of blocks with different sizes and importance values. The partition is based on the domain knowledge given by scientists. Then, we merge together spatial blocks at the same octree node along the time dimension to create space-time blocks. This allows us to utilize temporal coherence for compression. Finally, we encode each space-time data block in a bit-wise manner based on its data range and importance value.

Spatial partitioning. We assume that scientists know regions of interest in their data. An example of the turbulent combustion simulation data is given in the sidebar. To measure the distances of voxels to the region of interest, we create a distance volume for each time step, where the distance value of each voxel is the shortest distance of the voxel to the surface of interest. We implement an algorithm similar to the fast marching method (FMM)[1]. The FMM is a technique for computing the arrival time of a front expanding in the normal direction at a set of grid points. Instead of calculating the actual surface, we start with a front containing voxels that intersect with the surface and push the front outwards gradually. The voxel-wise shortest distance is calculated for each voxel on the current front as an approximation of the distance to the actual surface. The distance volume can be temporarily kept in memory for compression purpose. Or, it itself can be compressed and saved for runtime use, such as distance-based rendering.

Spatial partitioning works as follows: first, we build an octree skeleton where the block dimension for leaf nodes is predefined. Then, we start from the root node in the octree and partition the volume: if the data associated with the octree node does not include any isosurface voxels (i.e., voxels that intersect with the surface) or the data block contains more than a certain percentage of isosurface voxels, then we stop and do not partition the block any further. Otherwise, we partition the data block into eight subblocks and perform this process recursively until the leaf nodes are reached. In this way, the entire volume at a time step is partitioned into blocks with different sizes. The corresponding octree nodes constitute a cut through the full octree skeleton, as illustrated in Figure 2.

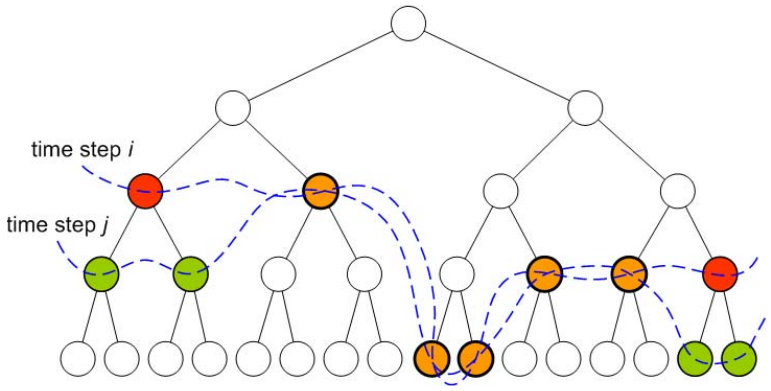


Figure 2 The volume data at a time step is partitioned into a list of blocks of different sizes. The corresponding octree nodes constitute a cut through the full octree skeleton. Two neighboring time steps *i* and *j* share a subset of data blocks (drawn in orange), which are merged into space-time blocks in temporal grouping. For illustration purpose, a binary tree instead of an octree is drawn in the figure.

After spatial partitioning, we calculate the importance value of each data block which is proportional to one over the average distance values of all the voxels in that block. Our experiments show that regions closer to the surface of interest usually get finer partitioning. Therefore, rather than of using the conservative minimum distance value of all voxels in a data block for the importance value calculation, we use the average distance for a more aggressive compression afterwards. In our implementation, the importance value is further scaled by a ratio γ (<1.0) which keeps decreasing linearly as the average distance increases. Essentially, we use γ to steer the compression rate of space-time blocks.

Temporal grouping. The result of spatial partitioning is a list of blocks at each time step with different sizes and importance values. In general, the volume data at consecutive time steps exhibit strong temporal coherence in local neighborhoods. This means that there is a large degree of node overlap in the block lists for neighboring time steps, as illustrated in Figure 2. For example, experimental results on the combustion data set show that the average percentage of node overlap for any two consecutive time steps is 90%. To utilize this temporal coherence for compression, for each octree node, we merge spatial blocks at consecutive time steps into space-time blocks. In the meanwhile, a maximum window size w is specified to control the tradeoff between compression rate and decompression speed. An example of temporal grouping on an octree node is sketched in Figure 3. In essence, temporal grouping consolidates data blocks at different time steps into space-time blocks, which become the basic units for the following encoding.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

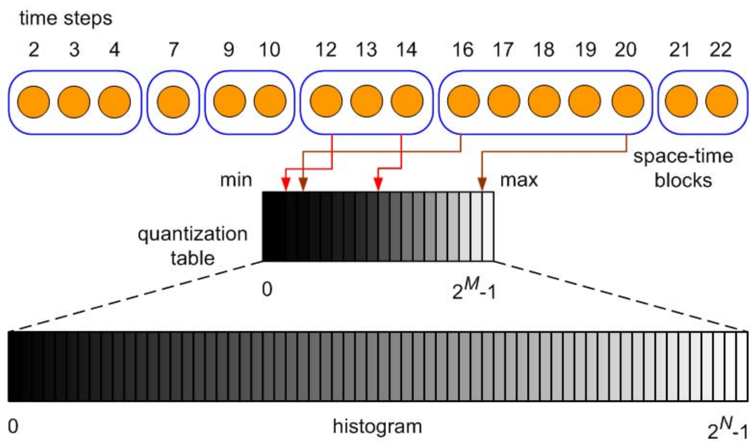


Figure 3 Temporal grouping merges spatial blocks in the same octree node at consecutive time steps into space-time blocks. In this example, the maximum window size is five. For each space-time block, we record the beginning and ending indices to the non-uniform quantization table and the offset index for each voxel in the block. In the figure, a time step (e.g., 5) which does not appear in the octree node stays on one of its ancestor or descendant nodes instead.

Encoding. In the encoding step, we compress all the space-time blocks using non-uniform quantization together with difference and run-length encoding, resulting in a highly compacted data representation. We first create a histogram (with 2^N entries) from the volume data at all time steps. Then, we build a non-uniform quantization table (with 2^M entries, where $M < N$) from the histogram. Many solutions can be applied for histogram quantization. One solution[2] we find simple yet effective is to partition the histogram into 2^M parts with equal areas (i.e., accumulated bin counts), and pick the data value with the highest bin count (i.e., the most frequently occurring value) in each part as the quantized value. Furthermore, our quantization scheme also incorporates data-specific domain knowledge, such as ranges of interest, into quantization. This is to ensure that particular data ranges of interest are sufficiently sampled and represented in the quantization table.

The encoding works as follows: for each space-time block, we use its minimum and maximum values to acquire the beginning and ending indices to the quantization table, as shown in Figure 3. For each voxel in the space-time block, recording the offset index (I_o) with respect to the beginning index (I_b) is sufficient to look up the corresponding quantized value. Moreover, from the beginning and ending indices, we can calculate the number of bits (denoted as B_n) needed to encode the offset index for each voxel. The number of bits is further modulated by the average distance of the space-time block to the surface of interest. That is, the farther a block is away from the surface, the less number of bits we use to represent each voxel in the block. Adjusting relevant parameters such as the ratio γ gives different bit modulations for space-time blocks, thus different compression levels. Let the number of bits after distance modulation be B_m. We have $B_m \leq B_n$. The actual index I to the quantization table is:

$$I = I_b + I_o \times 2^{B_n - B_m} . \qquad (1)$$

After the quantization step, each voxel in the same space-time block is represented with a few bits. The entire space-time block can be further compressed using a combination of difference and run-length encoding: first, we calculate the differences of index values for neighboring time steps in each space-time block; then, the difference values can be compressed using a run-length encoding scheme to exploit temporal coherence.

Texture Packing

To effectively utilize limited graphics memory, Kraus and Ertl[1] introduced adaptive texture maps with locally adaptive resolution. They demonstrated the usage of adaptive texture maps for packing data blocks of different resolutions. This technique allows us to represent fine details in images and volumes without the need to increase the resolution of the whole texture map. Binotto et al.[2] developed a similar approach for texture packing and compression of sparse time-varying volume data into 3D textures. During rendering, the decompression is performed by the fragment shader in the GPU. Li et al.[3] studied texture partitioning and packing for skipping empty space and accelerating slice-based volume rendering. In their approach, the entire volume is first partitioned into sub-volumes with similar properties. Sub-volumes are then packed and stitched together into larger textures for rendering. Akiba et al.[4] used data packing for time-varying data reduction. Data packing was achieved by discarding data blocks with values outside the data interval of interest and encoding the remaining data in a way so that they can be efficiently decoded in the GPU.

References

1. M. Kraus and T. Ertl, "Adaptive Texture Maps," *Proc. ACM SIGGRAPH/Eurographics Conf. on Graphics Hardware*, 2002, pp. 7–15, 2002.
2. A. P. D. Binotto, J. L. D. Comba, and C. M. D. Freitas, "Real-time Volume Rendering of Time-Varying Data Using a Fragment-Shader Compression Approach," *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003, pp. 69–75.
3. W. Li, K. Mueller, and A. E. Kaufman, "Empty Space Skipping and Occlusion Clipping for Texture-Based Volume Rendering," *Proc. IEEE Visualization Conf.*, 2003, pp. 317–324.
4. H. Akiba, K.-L. Ma, and J. Clyne, "End-to-end Data Reduction and Hardware Accelerated Rendering Techniques for Visualizing Time-Varying Non-uniform Grid Volume Data," *Proc. International Workshop on Volume Graphics*, 2005, pp. 31-39.

Decompression and rendering

At runtime, the compressed time-varying data are first partially decompressed with difference and run-length decoding. As illustrated in Figure 1, this step is conducted in the CPU and the results are space-time blocks storing offset indices to the quantization table. To render a volume data at a time step, its corresponding data blocks are categorized, padded, and packed into texture memory. Finally, the volume is rendered with the deferred filtering scheme.

Bit padding. Let us assume that the non-uniform quantization table has up to 1024 entries. Thus, the offset indices stored in a space-time block could range from 1 to 10 bits. Note that the number of bits to represent offset indices varies from block to block due to its data range and importance value. Since the standard OpenGL only defines a list of fixed formats of the pixel data for texture loading, we specifically pick the formats listed in Table 1. Clearly, this bit padding step involves space overhead and increases texture memory consumption. However, our experiments show that, the size increase due to bit padding is around 10%, which is quite affordable.

original # bits	padded # bits	# elements in a texel	overhead (%)	data type for pixel data	internal texture format
1, 2	2	3	166.7, 33.3	GL_UNSIGNED_BYTE_3_3_2	GL_R3_G3_B2
3, 4	4	4	33.3, 0.0	GL_UNSIGNED_SHORT_4_4_4_4	GL_RGBA4
5	5	3	6.67	GL_UNSIGNED_SHORT_5_5_5_1	GL_RGB5_A1
6 , 7, 8	8	4	33.3, 14.3, 0.0	GL_UNSIGNED_INT_8_8_8_8	GL_RGBA8
9, 10	10	3	18.5, 6.67	GL_UNSIGNED_INT_10_10_10_2	GL_RGB10_A2

Table 1 A space-time block is padded into one of the five different OpenGL pixel data formats for texture loading.

Texture packing. After bit padding, data blocks are ready for texture packing into one of the five index textures (each for one of the internal texture formats). The purpose of texture packing is to reduce the number of textures used as well as the overall texture memory consumption. Texture packing can be performed as usual by treating each data block as a 3D array and by utilizing a greedy algorithm to optimize block arrangement. It is important to note that when deferred filtering is used in decompression, volume slices are being reconstructed voxel by voxel using the nearest neighbor interpolation rather than the trilinear interpolation. Therefore, in our scenario, texture packing does not have to be performed directly in 3D. As an alternative, we can treat each 3D data block as a 1D array and pack it into a 3D texture in the form of a cube. The dimension of the packed 3D texture is calculated as follows:

$$L = \lceil \sqrt[3]{\Sigma \parallel A \parallel} \rceil, \qquad (2)$$

where $||A||$ denotes the length of array A. This treatment greatly simplifies data packing and yields a better utilization of the texture memory. The only overhead is that we need to map (x, y, z) tuples to 1D indices for texture lookup.

Deferred filtering and volume rendering. At runtime, each voxel in the volume is decompressed for rendering, where the trilinear interpolation is performed on a per-fragment basis. Note that a single voxel may be needed multiple times for neighboring sample reconstructions and gradient calculations. To avoid redundant decompressions, we can cache a proxy geometry (i.e., a small subset of decompressed volume) first and then use the subset for conventional rendering. This deferred filtering technique[3] separates decompression and interpolation into two passes so that a voxel needs to be decompressed only once no matter how many times it is needed for interpolation. In our case, the proxy geometry is multiple axis-aligned slices assembled from volume partitions, as illustrated in Figure 4. To render a single slab, we first decompress two consecutive slices of the volume in the first pass. In the second pass, we render sampling slices as usual using trilinearly interpolated samples. The volume is thus rendered slab by slab. The main advantage of using axis-aligned slices is that data blocks in the volume do not need to be padded to ensure seamless rendering along block boundaries.

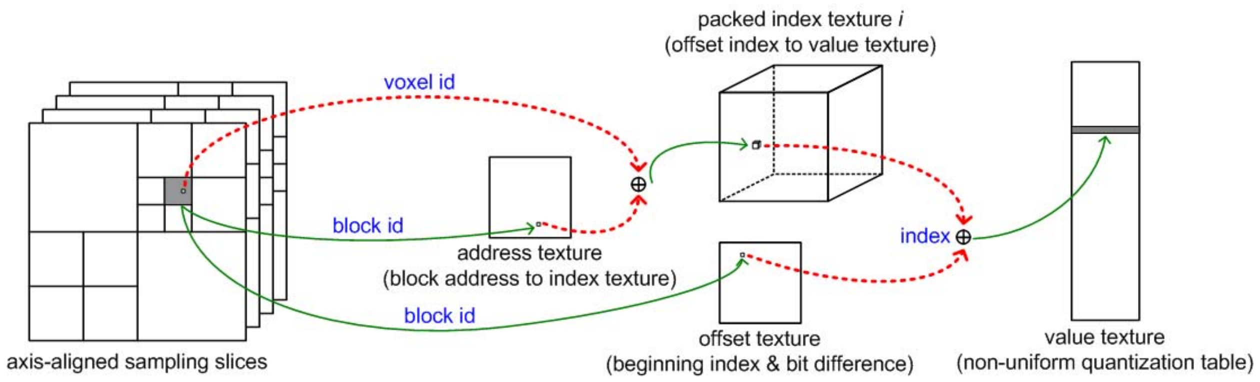

packed index texture *i*
(offset index to value texture)
voxel id
block id
address texture
(block address to index texture)
index ⊕
block id
axis-aligned sampling slices
offset texture
(beginning index & bit difference)
value texture
(non-uniform quantization table)

Figure 4 Texture lookups with the deferred filtering scheme. The address texture stores data blocks' addresses to the packed index texture. The offset texture stores data blocks' beginning indices and bit differences. The packed index texture stores voxels' offset indices to the value texture, i.e., the non-uniform quantization table. Red dashed lines correspond to texture fetches and green solid lines correspond to texture lookups. To reconstruct a data block on a sampling slice, two texture lookups are needed on a per block basis; and two texture lookups are needed on a per voxel basis.

Figure 4 sketches our texture lookups with the deferred filtering scheme. Axis-aligned slices that are most perpendicular to the viewing direction are dynamically reconstructed. The address of each data block to the packed index texture is stored in the address texture. For each data block on a sampling slice, we first look up its block address and the beginning index and bit difference ($B_d = B_n - B_m$) in the address texture and offset texture, respectively. Then, for each voxel in the data block, we look up its offset index in the corresponding packed index texture using its voxel id and the block address. Finally, the voxel's offset index and the block's beginning index and bit difference are used to compute the actual index to the value texture (Equation 1). Note that depending on the number of bits used, a data block is classified into one of the five different texture formats (as listed in Table 1), and accordingly, texture lookup is performed on its corresponding packed index texture.

Results

We experimented with our algorithm on two floating-point data sets obtained from scientific simulations, as listed in Table 2. All tests were performed on a PC with a 2.33GHz Intel Xeon processor, 4GB main memory, and an nVidia GeForce 8800 GTX graphics card with 768MB video memory.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

The combustion data set was provided by scientists at the Sandia National Laboratories. The combustion simulation ran thousands of time steps, and at each time step it output dozens of variables representing different chemical species. A subset of the data set we used here has four different variables. They are scalar dissipation rate (*chi*), stoichiometric mixture fraction (*mixfrac*), hydroperoxy radical (HO_2), and hydroxyl radical (*OH*). The scientific interest for the combustion data is on the main flame structure, which corresponds to the *mixfrac* surface with isovalue = 0.2.

The hurricane data set was obtained from the National Center for Atmospheric Research. The hurricane modeled in the simulation is Hurricane Isabel, a strong hurricane in the West Atlantic region during September 2003. We picked four variables, namely, pressure (*P*), cloud moisture (*CLOUD*), total precipitation (*PRECIP*), and water vapor (*QVAPOR*) for our experiment. We focused on the region that has very low pressure ($P \approx 0$), which corresponds to the center of the hurricane.

data set	combustion	hurricane
volume dimension	(800, 686, 215)	(500, 500, 100)
# time steps	53	48
# variables	4	4
data size	92.3GB	17.9GB
block dimension	(64, 64, 32)	(32, 32, 16)
avg # node	278	275
avg node overlap	90%	62%
compressed size (after quantization)	15.12GB	3.6GB
compressed size (after diff and RLE)	4.53GB	900MB
data reduction on disk	20.57x	20.37x
compression time	3hrs	40mins
padding overhead	11%	12%
tex reduction on GPU	82%	77%
frame rate	12.5fps	28.5fps

Table 2 The two data sets and their experimental results. We report the two-stage compression results, i.e., quantization and difference and run-length encoding (RLE) encoding, respectively. The frame rate is measured for rendering one variable with a 512^2 viewport and a sampling rate of 1.0.

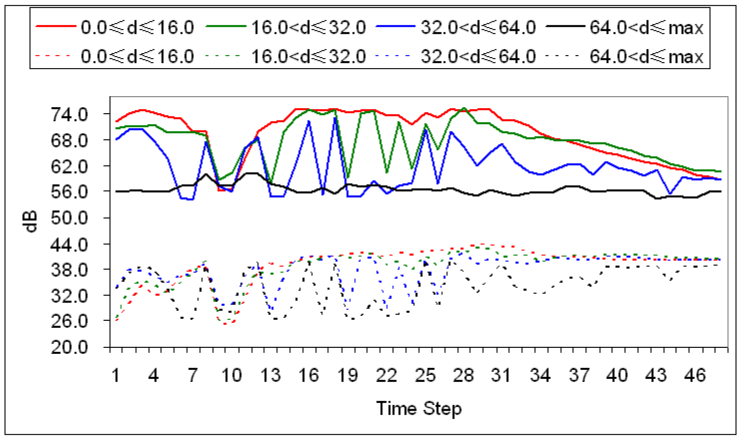
Compression. To construct the octree skeleton, we set the block dimension for leaf nodes as (64, 64, 32) and (32, 32, 16) for the combustion and hurricane data sets, respectively. For spatial partitioning, the threshold for the percentage of isosurface voxels in data blocks was chosen as 5% for both data sets. The decisions of block size for leaf nodes and the percentage threshold for isosurface voxels determine how many blocks are generated in spatial partitioning.

With such configurations, the average number of octree nodes that have non-empty data blocks in a time step is around 270. On the other hand, the average percentages of node overlap for consecutive time steps are 90% and 62% for the two data sets, respectively. This indicates a great degree of coherence for compression. We set the maximum window size $w = 5$ in temporal grouping. For both data sets, we chose $N = 16$ and $M = 10$ for the non-uniform quantization, which allows the histogram to be sufficiently sampled in the 1024-entry quantization table.

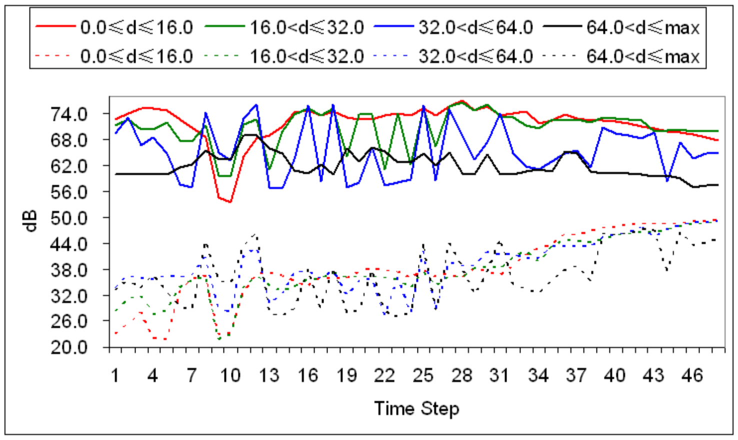
It took us three hours to compress the 93.2GB combustion data set (on average, less than one minute per variable per time step, which is around 450MB). The compressed data size is 4.53GB so we achieved a compression rate of 20.57x. This means that on average each variable in the time sequence is compressed to about 1.6 bits per voxel. A comparable compression performance was achieved for the hurricane data set.

In Figure 5, we show the signal-to-noise ratio (SNR) and the peak signal-to-noise ratio (PSNR) of the compressed hurricane data sets. Different curves in Figure 5 (a)-(c) correspond to different distance ranges from the surface of interest. It can be seen that generally the regions with smaller distances (i.e., closer to the surface) get higher SNRs or PSNRs (i.e., less distortion). At some time steps (such as time steps 1 to 8 for the SNR curves and time steps 40 to 48 for the PSNR curves), this observation does not always hold. This is because our compression scheme is block-wise instead of voxel-wise, and the average distance instead of the minimum distance is used to calculate a block's importance value. With these settings, the choice of block size also matters since all voxels in a block use the same encoding scheme. On the other hand, Figure 5 shows that the *QVAPOR* variable has higher SNRs and lower PSNRs than the *CLOUD* and *PRECIP* variables. This suggests that different variables may require customized bit modulations for compressing their space-time blocks in order to balance the overall rate-distortion across all the variables.

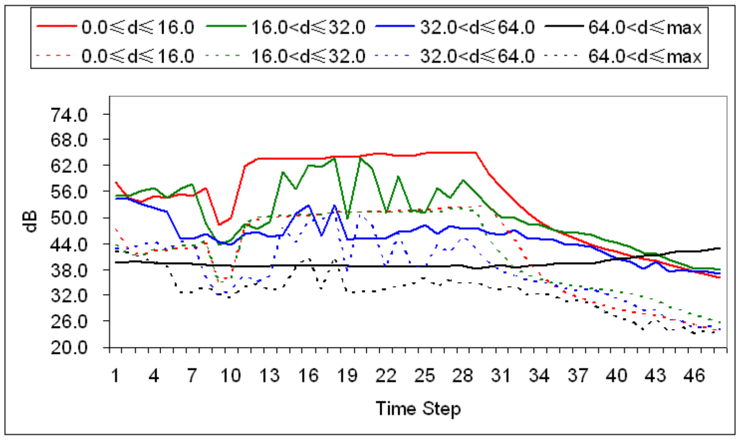
Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE



(a) *CLOUD*



(b) *PRECIP*

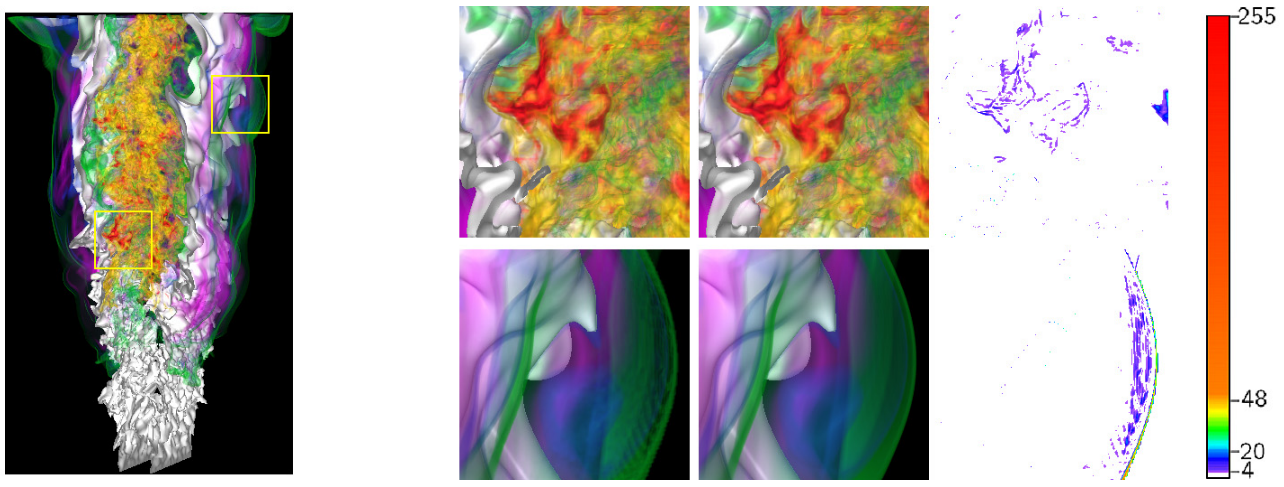


(c) *QVAPOR*

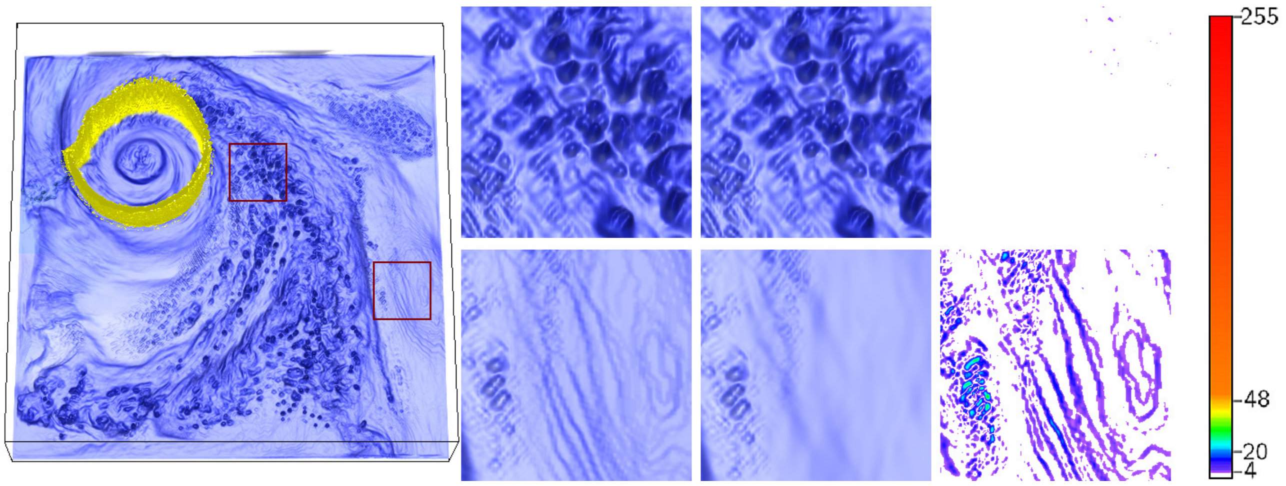
Figure 5 SNR and PSNR curves of the application-driven compressed hurricane data sets. In (a)-(c), four SNR (dashed lines) and PNSR (solid lines) curves are shown with different distance ranges from the surface of interest. Note that we used the theoretical signal peak as a reference in PSNR calculation. In general, the regions close to the surface yield higher SNRs or PSNRs, and thus less distortion.

difference and run-length decoding	2.37 s
bit padding	2.58 s
texture packing	1.27 s
deferred filtering and volume rendering	2.95 s

Table 3 Timing breakdown for rendering the *CLOUD* variable of the 48 time steps hurricane data set.



(a) the *mixfrac* surface (in white) + the HO_2 variable



(b) the *P* surface (in yellow) + the *QVAPOR* variable

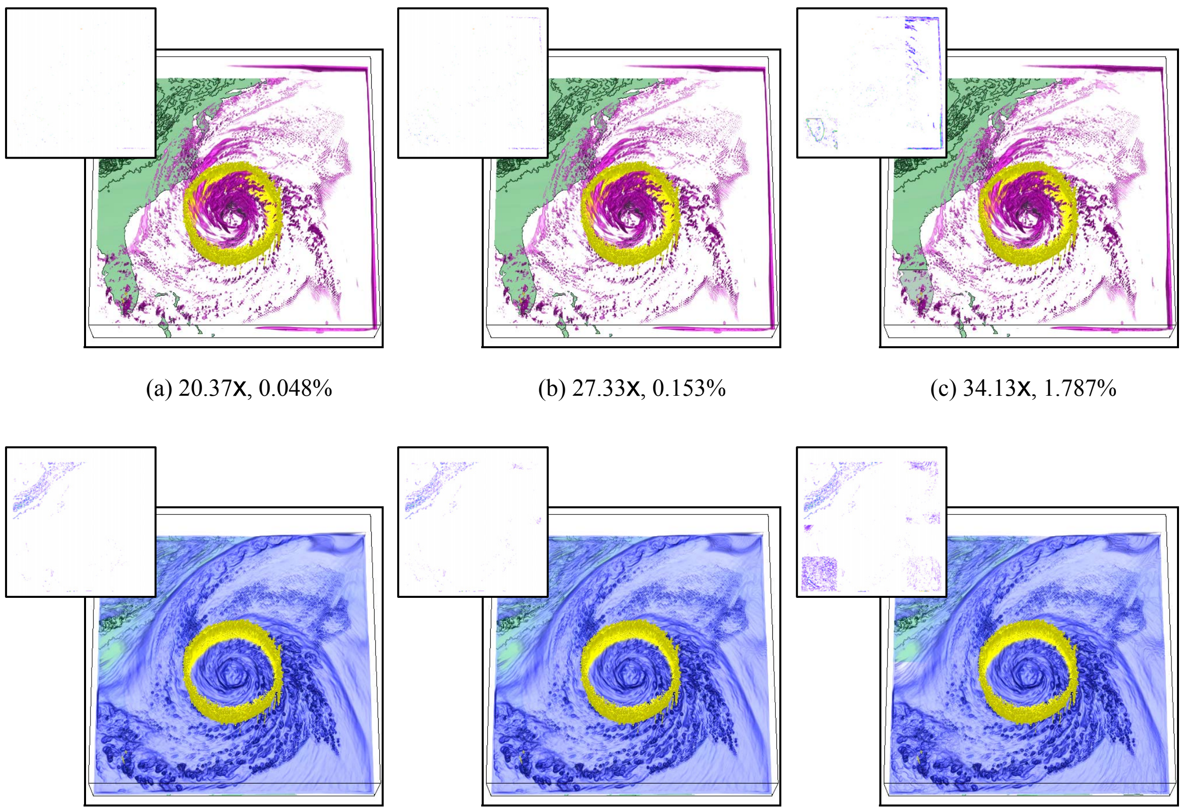
Figure 6 Rendering comparisons of the compressed (a) combustion and (b) hurricane data with their original data. In (a) and (b), from left to right: rendering compressed data for an overview, zoom-in of compressed data, zoom-in of original data, image difference of the compressed and original data calculated in the CIELUV color space, and color map. Note that regions farther away from the surface of interest show more quantization artifacts in rendering.

Decompression and rendering. At runtime, the partially-decoded data are padded and loaded into texture memory. As listed in Table 2, bit padding only incurs a slight increase of memory usage. The overall texture reduction with respect to loading the original data is 82% and 77% for the combustion and hurricane data sets,

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

respectively. Due to difference and run-length decoding and bit padding, the compression rates in the texture memory reduce to 5.6x and 4.3x for the combustion and hurricane data sets, respectively.

With a viewport of 512^2 and a regular sampling rate of 1.0 (i.e., one sample per voxel), we achieved 12.5fps and 28.5fps (including deferred filtering and volume rendering) for rendering one variable from the combustion and hurricane data sets, respectively, which is comparable to conventional volume rendering. Our application-driven compression and rendering solution makes possible interactive visualization of large-scale time-varying data while dramatically reducing data transferring between the memory hierarchies. The saving in data transferring greatly shortens the time to animate time-varying data. For example, the total time (including I/O and rendering) to animate the *CLOUD* variable of the 48 time steps hurricane data set reduces from 36.96 seconds to 9.17 seconds. Table 3 shows a breakdown of timing for each of the stages: decoding, bit padding, texture packing, deferred filtering and volume rendering. Note that the decoding part includes the time to read compressed data from disk. Thus, we improved the frame rate from 1.3fps (without compression) to 5.2fps (with compression), achieving a much desirable level of interactivity.

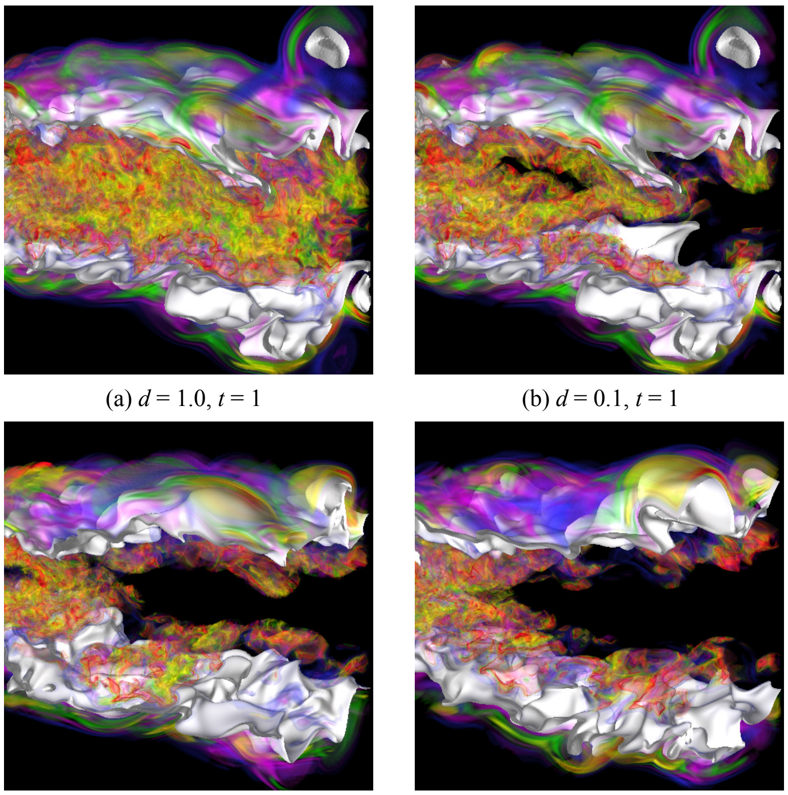


(a) 20.37x, 0.048% (b) 27.33x, 0.153% (c) 34.13x, 1.787%

(d) 16.13x, 0.810% (e) 20.37x, 0.877% (f) 27.33x, 2.547%

Figure 7 Rendering the compressed hurricane data set at time step 24 with different compression levels. By adjusting the parameters for bit modulation, we can change the number of bits allocated to each of the space-time blocks and compress the data differently. The near zero pressure surfaces with the *CLOUD* and *QVAPOR* variables are shown in (a)-(c) and (d)-(f), respectively. The corresponding difference image with respect to the original image is displayed on the corner. The compression rate and the percentage of noticeable pixel difference are listed in each caption.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE



(a) $d = 1.0$, $t = 1$ (b) $d = 0.1$, $t = 1$

(c) $d = 0.1$, $t = 27$ (d) $d = 0.1$, $t = 53$

Figure 8 Rendering the compressed combustion data with distance control. By changing the distance threshold (normalized to [0, 1]), the user can interactively control the amount of information displayed around the surface to better observe variable relationships. The *mixfrac* surface with the HO_2 variable is shown in the figure.

In Figure 6, we compare the rendering of the compressed data with the original data. To render more than one variable simultaneously over time, each variable is independently decompressed and loaded into the graphics card. For objective comparison, we calculated pixel-wise differences (i.e., the Euclidean distances) of images generated from the compressed and original data in the CIELUV color space. The noticeable pixel differences (with $\Delta E \geq 4.0$) were mapped to non-white colors (differences greater than 255 were clamped). It can be observed that our application-driven solution preserves fine details near the regions of interest, while maintaining the overall quality well. Some visual differences between the rendering of the compressed and the original data can be perceived. These correspond to regions that are far away from the reference feature and therefore lose more precision in the quantization.

Figure 7 shows the rendering of the compressed hurricane data set at different compression levels. Adjusting the parameters for bit modulation leads to compression of the data with different reduction rates. From Figure 7, we can observe how the quality degrades with the increase of the compression rate. Compared with the *QVAPOR* variable of the same compression level, the *CLOUD* variable gives less degradation in visual quality. The $P \approx 0$ surface highlights the center region of the hurricane. We rendered the P surface with two other variables: *CLOUD* and *QVAPOR*. The rendering is informative, which allows the scientists to focus on the hurricane center and track other flow properties around the surface of interest. For the combustion data set

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

where the variable rendering could occlude the surface, we can utilize the distance volume to perform flexible rendering by changing the distance threshold, as shown in Figure 8. Only the voxels within the given distance threshold are assigned non-zero opacity values. The user can interactively control the amount of information displayed around the surface to better observe variable relationships at runtime. In the accompanying videos, we show side-by-side rendering of the original and compressed combustion and hurricane data sets over all time steps.

Discussion

In our work, we opted for scalar quantization instead of vector quantization for data reduction. This is because in vector quantization, the time to generate the codebook could be prohibitively long for a large time-varying, multivariate data set. The non-uniform quantization scheme we implemented is a simple and fast solution, which also gave us good compression and reconstruction results for the two test data sets. With this quantization scheme, however, we may miss details for underrepresented scalar values. This happens when the transfer function maps underrepresented scalar values to high opacity values. Other quantization schemes, such as the Lloyd's quantizer which guarantees to converge to a local minimum in the L_2 metric, could also be used. There is a need of further research on the quantization scheme that couples compression with visualization to strive for a better tradeoff between reduction performance and rendering quality.

Our approach resembles the importance-driven volume rendering work by Viola et al.[4]; however, in our case, the importance values of data blocks in relation to the surface of interest are utilized in compression and rendering. The limitation of our approach is that since the reference feature derived from domain knowledge is explicitly incorporated into data reduction, we have to redo the compression process if the scientific interest changes. We assume that this kind of change is less likely to happen. An extension of this work is to encode all possible input from scientists. This would allow shifting features of interest at runtime but at the expense of compression efficiency.

Conclusion

Our application-driven approach clearly suggests a viable direction for addressing the data challenge presented by large-scale scientific simulations. The solution presented can be applied to other domains where the reference features are in different forms, such as vortices in the flow data. The bottom line is that as long as the identified regions of interest only occupy a small percentage of the volume space, our method would remain effective. In the future, we would like to incorporate our data reduction scheme with multiresolution techniques to support flexible level-of-detail rendering. We will also consider a parallel implementation of our solution for compressing and rendering large-scale time-varying data in a massively parallel computing environment.

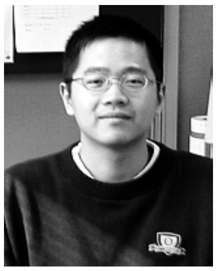
Acknowledgements

This research was supported in part by the NSF ITR program and the DOE SciDAC program. We would like to thank Jacqueline H. Chen at the Sandia National Laboratories for providing the combustion data set and sharing her domain knowledge. We also thank the reviewers for their constructive suggestions, which helped improve this article.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE

References

1. M. W. Jones, J. A. Bærentzen, and M. Srámek, "3D Distance Fields: A Survey of Techniques and Applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, 2006, pp. 581–599.
2. A. E. Shortt, T. J. Naughton, and B. Javidi, "Histogram Approaches for Lossy Compression of Digital Holograms of Three-Dimensional Objects," *IEEE Transactions on Image Processing*, vol. 16, no. 6, 2007, pp. 1548–1556.
3. N. Fout, H. Akiba, K.-L. Ma, A. E. Lefohn, and J. Kniss, "High Quality Rendering of Compressed Volume Data Formats," *Proc. Eurographics/IEEE VGTC Symposium on Visualization*, 2005, pp. 77–84.
4. I. Viola, A. Kanitsar, and M. E. Gröller, "Importance-Driven Volume Rendering," *Proc. IEEE Visualization Conf.*, 2004, pp. 139–145, 2004.



Chaoli Wang is a postdoctoral researcher in the VIDI research group at the University of California, Davis. His current research interest focuses on large-scale data analysis and visualization. Wang has a PhD in computer and information science from The Ohio State University. He is a member of the IEEE. Contact him at wangcha@cs.ucdavis.edu.

Hongfeng Yu has recently graduated from the University of California, Davis with a PhD in computer science and joined Sandia National Laboratories as a postdoctoral researcher. His research interests include scientific visualization and parallel computing. Contact him at hfyu@ucdavis.edu.

Kwan-Liu Ma is a professor of computer science at the University of California, Davis, and he directs the DOE SciDAC Institute for Ultrascale Visualization. His research spans the fields of visualization, high-performance computing, and user interface design. Ma received a PhD in computer science from the University of Utah. He presently serves on the editorial board of IEEE TVCG and IEEE CG&A. He is a senior member of the IEEE and a member of the ACM. Contact him at ma@cs.ucdavis.edu.

Digital Object Indentifier 10.1109/MCG.2009.104 0272-1716/$26.00 © 2009 IEEE