Preprint
UCRL-JC-200087

Adaptive Extraction of Time-varying Isosurfaces

Benjamin Gregorski, Joshua Senecal, Mark Duchaineau, and Kenneth I. Joy

This article was submitted to IEEE Transactions on Visualization and Computer Graphics

September, 2003

U.S. Department of Energy

Lawrence Livermore National Laboratory

DISCLAIMER

Adaptive Extraction of Time-varying Isosurfaces

Benjamin Gregorski, Joshua Senecal, Mark Duchaineau and Kenneth I. Joy

***Abstract*— We present an algorithm for adaptively extracting and rendering isosurfaces from compressed time-varying volume datasets. Tetrahedral meshes defined by longest edge bisection are used to create a multiresolution representation of the volume in the spatial domain that is adapted over time to approximate the time-varying volume. The reextraction of the isosurface at each time step is accelerated with the vertex programming capabilities of modern graphics hardware. A data layout scheme which follows the access pattern indicated by mesh refinement is used to access the volume in a spatially and temporally coherent manner. This data layout scheme allows our algorithm to be used for out-of-core visualization.**

***Index Terms*— Visualization techniques and methodologies, Graphics data structures and data types**

I. INTRODUCTION

High-performance computing on large shared memory machines and PC clusters makes it possible to study complex problems through large scale numerical simulations. These simulations, of such things as weather patterns, fluid dynamics, and material deformations, are being carried out on larger scales and are producing larger datasets that need to be visualized. An example is the simulation of a Richtmyer-Meshkov instability in a shock tube experiment [15] conducted at Lawrence Livermore National Laboratory. The output of the simulation is a series of Brick-of-byte (BOB) volumes that measure entropy. Each output file is divided into an $8 \times 8 \times 15$ grid of $256 \times 256 \times 128$ bricks. The resolution of a single time step is $2048 \times 2048 \times 1920$ or about 7.7 GB. There are 274 time steps for a total data size of about 2.1 TB. A full resolution isosurface of the mixing interface for a single time step produces a mesh with several hundred million triangles. The visualization of a single time step of these simulations presents a significant challenge because of the size and topological complexity of the surfaces. Since these problems are time-varying in nature a true understanding can only come from a time-varying visualization. Visualization of these datasets requires techniques that exploit view-dependent, hierarchical and out-of-core algorithms to minimize the amount of runtime computation and maximize the use of available resources.

Our method utilizes the diamond-based, crack-free refinement strategy presented in [6]. This algorithm uses a dual priority queue split/merge algorithm similar to the ROAM system [2] for view-dependent terrain visualization. The mesh refinement scheme has fast coarsening and refinement operations necessary for view-dependent and adaptive refinement. To improve the efficiency of the mesh refinement and the isosurface visualization, we represent a tetrahedron in the hierarchy as a set of tetrahedra called a subtree. Subtrees reduce the granularity of the refinement, allow for efficient temporal caching, and fast rendering from vertex arrays. Our data storage scheme aligns the data with the access pattern indicated by the mesh refinement. This storage scheme exploits spatial and temporal locality of reference. It allows us to progressively extract isosurfaces for single time steps and to quickly extract new surfaces as the time step changes. This storage scheme is coupled with a time-varying compression algorithm that allows the data to be progressively decoded over time and only in the regions needed for the isosurface visualization.

The input to our algorithm is a series of volume datasets that we refer to as *time steps*. We assume the intervals between time steps are constant. At runtime, for a given isovalue and time step, we extract a new surface from the tetrahedral mesh which existed from the previous time step. When the time step remains constant, the mesh is progressively refined in the visible, high error regions that contain the isosurface. This refinement strategy allows one to move through the time steps at a coarse resolution to get a general idea of how a surface changes over time or to move through the time steps at a high resolution and closely inspect the surface at each time step.

II. PREVIOUS WORK

The refinement of a tetrahedral mesh via longest-edge bisection described in [14] has been used in many scientific visualization applications. In [26], a fine-to-coarse merging of groups of tetrahedra is used to construct a multi-level representation of a dataset. Topology preservation and controlled topology simplification of an extracted isosurface is presented by Gerstner and Pajarola [3]. Roxborough and Nielson [17] adaptively model 3-dimensional ultrasound data. The mesh refinement is used to create a model of the volume that conforms to the complexity of the underlying data. In [6] an adaptive refinement scheme for tetrahedral meshes is used for view-dependent isosurface extraction. The hierarchical data layout scheme of [12], [16] is used for out-of-core visualization.

Techniques for visualizing time-varying datasets focus on exploiting spatial and temporal locality to reduce the time needed to load and render a time step. Spatial techniques focus on reducing the time to locate the regions needed for visualization and the amount of extra storage needed for these search structures. The Temporal Hierarchical Index Tree [18] exploits temporal coherence in time-varying datasets by adaptively joining cells whose extreme values do not change much over time. The cells are organized in a tree structure whose nodes are groups of cells that have small extreme value

Benjamin Gregorski, Joshua Senecal, and Mark Duchaineau are with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA, 94551
Email: {gregorski1,senecal1,duchaine}@llnl.gov

Kenneth I. Joy is with the Center for Image Processing and Integrated Computing, Department of Computer Science, University of California Davis, One Shields Avenue, Davis CA, 95616 Email: kijoy@ucdavis.edu

deviation over a particular time interval. For a given time step, the tree structure is used to quickly locate the regions of the dataset that contain the isosurface. The approximation of actual extreme values with extreme values over time can cause regions that do not contain the isosurface to be loaded. Once these regions are found, the actual values for the indicated time step are used to extract the isosurface.

In [22], [21] Sutton et al. extend the Branch-on-Need Octree(BONO) [25] to a Temporal Branch-on-Need Octree for time-varying isosurface extraction. They build a BONO for each time step of the dataset and separately store the extreme values and data values for each time step. Instead of exploiting temporal coherence, they minimize the I/O bottleneck, associated with reading in a new time step for isosurfacing, by dividing the dataset into bricks for better spatial coherence. Following the extreme values associated with the nodes of the BONO allows the regions of the volume containing the isovalue to be quickly loaded.

The Time-Space Partitioning Tree [19] extends the Branch-on-Need Octree [25] to take advantage of spatial and temporal coherence. The nodes in the octree are binary time trees which measure the spatial and temporal coherence of the region over time. The structure exploits spatial coherence by determining when regions of the volume can be approximated using coarser volume, and it exploits temporal coherence by determining when rendered images from one time step can be used to render another time step. The data structure is adapted for out-of-core visualization by dividing the dataset into bricks and utilizing a demand paging system [1].

Compression based techniques for visualizing time-varying data focus on reducing the size of the datasets for faster loading from disk and faster rendering. These algorithms use the coherence in the data values over time to reduce the data size. Westermann [24] separately encodes each time step using a wavelet transform, and uses the Lipschitz exponents to detect regions with low and high temporal variation. This method allows the volume to be decompressed locally in regions of interest for multiresolution and progressive rendering. In [7], Guthe and Strasser use wavelet transforms, run-length encoding, and arithmetic coding to compress time-varying volume datasets. Individual volumes are first encoded using a wavelet transform, quantized and then compressed using run-length and arithmetic coding. Finally a sequence of compressed volumes is further compressed using a process similar to MPEG to encode and compress the differences between time steps. The volumes are decompressed at run time and rendered using graphics hardware. Unlike [24] their technique decompresses the whole volume at each time step. Lum et al. [13] use a DCT and Lloyd-Max quantization to compress time-varying volumes. A group of values over time is first transformed and then compressed into bytes using a lossy quantization process. Rendering is performed directly from the compressed volumes using 2D textures and paletted texture lookups. This allows interactive rendering and interactive manipulation of colormaps.

In this paper, we present algorithms for time-varying isosurface extraction which can extract an arbitrary isosurface at various levels of detail based upon user specified criteria. Additionally we are only interested in moving forwards and backwards in the temporal domain in discreet steps to *play* the volume from start to finish. Volumetric methods for time-varying isosurface extraction [22], [21], [18] can extract arbitrary isosurfaces, but because they do not incorporate level-of-detail approximations, they must extract the surface from the finest level cells in the volume even when a coarser approximation can be used (i.e view-dependent or other error-based rendering).

Our algorithm combines a hierarchical, volumetric data structure, an adaptive refinement strategy, and a cache coherent data ordering to perform time-varying isosurface extraction. This allows us to extract arbitrary time-varying isosurfaces and to adjust the resolution of the surface based upon user specified criteria. The extraction of new isosurfaces is accelerated by vertex programs on modern graphics hardware to perform the interpolation of positions and gradients.

III. Preprocessing

In a preprocessing phase, we compute the following information for each time step:

1) The isosurface approximation error, minimum data value, and maximum data value of the region enclosed by each diamond in the mesh.
2) The normalized gradient vector at data point. The surfaces are shaded using the gradient as a texture coordinate. At each frame the texture coordinates (i.e. gradients) are multiplied by the inverse of the rotation component of the modeling matrix, translated and scaled so that all components lie within the range $[0, 1]$. This is used to perform diffuse lighting and to highlight the regions where the gradient is orthogonal to the viewing direction. The texture used for rendering is shown in Figure 1.

The data values are bytes, and the gradients are normalized and quantized in 16 bits. For data compression purposes, we need a gradient representation that has temporal coherence and a meaningful delta between consecutive values. The gradients are quantized component-wise with eight bits for the x-component, seven for the y-component, and one bit for the z-component's sign. This 16 bit gradient is split into two 8 bit values; the first being the x-component and the second being the y-component with the sign of the z-component as the least significant bit. These are stored as three bytes per input data point. The errors are stored on a logarithmic scale and quantized using six bits. The min and max values for a diamond are not compressed and are stored as bytes. This data can be stored uncompressed through the use of subtrees described in Section V.

A. Gradient Processing

Gradients for shading are estimated from the volume dataset using central differences. These gradients can be very noisy and in constant-valued regions they are degenerate (i.e. their magnitude is zero). For our algorithm, which adaptively extracts isosurfaces from different levels of detail, noisy and degenerate gradients are especially bad for shading. Figure

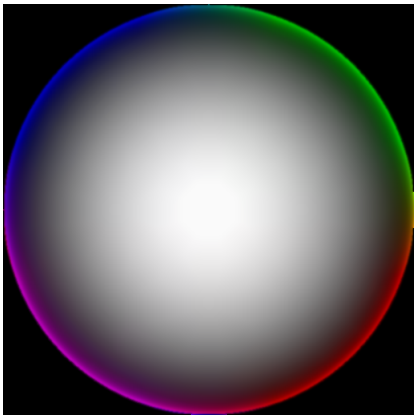

Fig. 1. Diffuse lit sphere texture map used for shading.

2 shows images of the raw and smoothed gradients from the Richtmyer-Meshkov dataset. The gradient (g_x, g_y, g_z) is converted to an rgb color as follows:

$$r = (g_x + 1.0) \times 0.5$$
$$g = (g_y + 1.0) \times 0.5$$
$$b = (g_z + 1.0) \times 0.5$$

Gray areas indicate regions where the gradient is $(0, 0, 0)$. In order to improve the quality of the gradients for shading, they are smoothed in the spatial and temporal domains.

The gradient smoothing process consists of two separate phases. The first phase is a symmetric diffusion of the gradients where the gradient at each data point is replaced with a weighted average of the surrounding gradients. In this phase, if the gradient at a data point is degenerate, it is not updated. The second phase is a constrained symmetric diffusion, the new gradient, a weighted average of the surrounding gradients, is not allowed to move to far from the value obtained after the first diffusion phase. The constrained diffusion is only performed where the original gradients are not degenerate. In the degenerate regions, the symmetric diffusion from the first phase is performed. Results of this process are illustrated in Figure 2 which shows the results after both smoothing phases from a single slice of the volume at different time steps.

Given an original, unnormalized gradient g_o and the new averaged gradient g_n, we compute the difference gradient g_d as:

$$g_d = g_n - g_o$$

g_n is given as

$$g_n = \begin{cases} |g_d| > \frac{K}{|g_o|} & : \quad g_o + \frac{g_d}{|g_d|} \times \frac{K}{|g_o|} \\ otherwise & : \quad g_n \end{cases}$$

Where K is a positive number. Regions where the original gradient has zero magnitude continue to undergo the diffusion process of the first phase. In both phases of the smoothing process, if the weighted average gradient has zero or near zero magnitude it is not used. This occurs in regions where the data varies very slowly.

Ideally, the gradient smoothing would be performed in the spatial and temporal domains on a four-dimensional (i, j, k, t) grid. However this computation can be extremely time consuming given the size of our test datasets, and so we approximate it with a three-dimensional smoothing in the spatial domain and a one-dimensional smoothing in the temporal domain.

1) For a dataset with k time steps, perform the gradient smoothing process for each time step separately until no degenerate gradients exist.
2) For each spatial location in the volume, smooth the gradients associated with this spatial location in the temporal domain.
3) Repeat the above steps until a sufficiently smooth gradient field is obtained.

IV. RUN TIME SYSTEM

A. Mesh Refinement for Static Volume

Given an error bound E, isovalue I_v, and time step T, the refinement process creates a set of tetrahedra S_t that approximates the regions of the dataset containing I_v at time T to within E.

Given a mesh at a time step T and an error tolerance E, the following steps are taken as long as the time step does not change:

1) Diamonds outside the view frustum and diamonds that do not contain I_v are assigned an error of zero. Errors are recomputed for all other diamonds in the split and merge queues.
2) The split/merge refinement process is used to create S_t. The refinement process is stopped when no more diamonds can be split or merged, or when the time allocated for the current frame has elapsed.
3) The isosurface is extracted from the tetrahedra that belong to the visible, non-empty diamonds in the split queue.

B. Changing The Time Step

When the time step changes the isosurface for the new time step is extracted by starting from the previous mesh as follows:

1) Get the error, min, and max values for the new time step for all visible diamonds in the split and merge queues.
2) Refine and coarsen the mesh as described in Section IV-A.
3) Extract a new isosurface from the tetrahedra in the current mesh.

Once the error, min, and max values have been updated, the refinement procedure for a static volume is used until the time step is changed again. Diamonds whose error, min, and max values were not updated in Step 1 are updated the next time they could be used for visualization. Each diamond stores the time step that its error, min, and max values correspond to. When the diamond is needed, the stored time step is checked against the current time step and the data is updated as necessary. For invisible diamonds in the split and merge queues, the values are updated when they become visible. For diamonds at coarser levels of the hierarchy, we perform a lazy evaluation and update the values only when they are added to the merge queue.

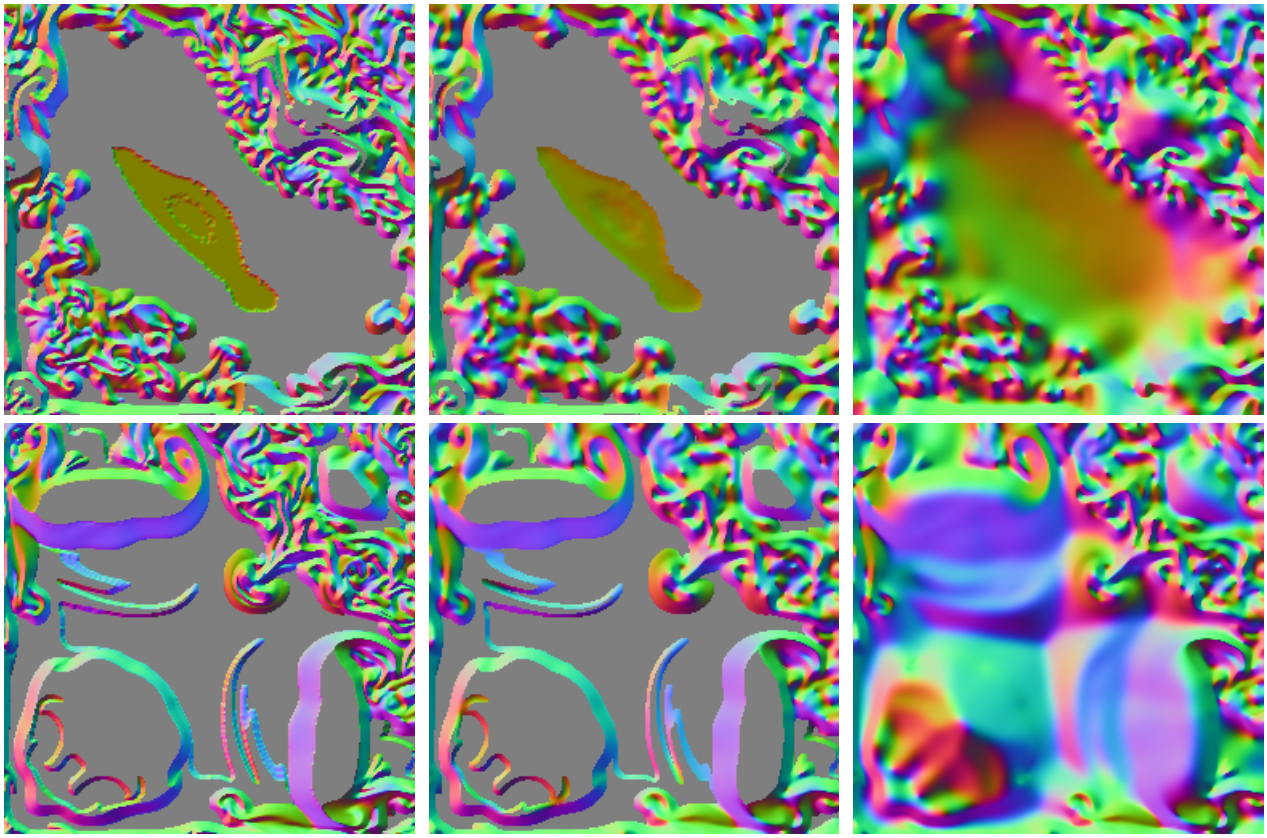

Fig. 2. Top and bottom: images of raw and smoothed gradients from a two-dimensional slice of a $256^3 \times 274$ dataset for time steps 160 and 273 at slice $z = 100$. Left to right: Original gradients, gradients after smoothing phase one, and after smoothing phase two.

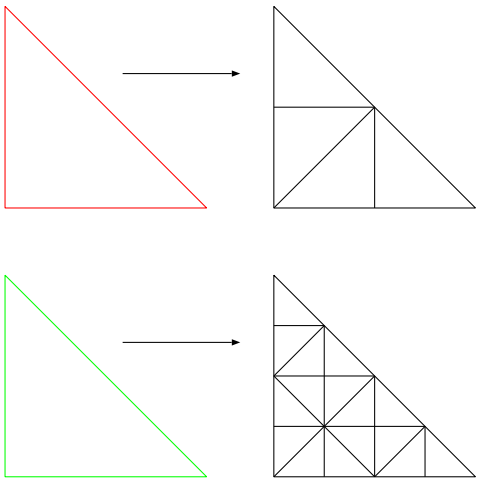

Fig. 3. Division of a triangle into subtrees of depth 1(top) and 2(bottom).

V. SUBTREES

In order to reduce the storage cost of the precomputed data and the granularity of the refinement process, we implicitly replace a single coarse tetrahedron T with a set of smaller tetrahedra S_T called a subtree. Figure 3 shows two-dimensional examples of a triangle divided into subtrees of depths one and two. The surface through the tetrahedra in S_T is extracted instead of the surface through T. T's min and max

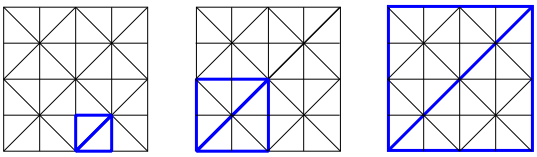

Fig. 4. From left to right: Leaf nodes for subtrees of depth zero, one, two for a two-dimensional 5x5 grid. The bold lines indicate the leaf diamonds which are at levels zero, one, and two respectively.

values remain the same and T's error value is given by:

$$e_T \quad = \quad \max(e_t) \;\; \forall \, t \in S_T.$$

The size of the subtrees is 2^{3d} where d is called the depth of the subtree. A depth of two divides one tet into 64 tets. In general, the subtrees do not need to be a power of two as long as adjacent edges and faces have the same tessellation. The division of tetrahedra into subtrees occurs at all levels of the mesh. The finest level of the mesh, i.e. the one with the smallest tetrahedra, is level zero. For a 2^{3k} mesh the root diamond is at level k.

The division of the mesh into subtrees has several important performance benefits. Subtrees reduce the size of the mesh for which data is precomputed by a factor of 2^{3d}. Figure 4 shows how the leaf nodes move to coarser and coarser levels of the mesh as the depth of the subtrees increases. Tetrahedra at finer(smaller) levels than the leaf nodes cannot be divided

into a subtree because all of the data points at the vertices introduced by the subtree do not exist. Error, min, and max values do not need to be computed for diamonds smaller than the leaf diamonds. Thus for a 1024^3 volume only 256^3 errors, min, and max values need to be computed. Subtrees reduce the storage cost of this information by a factor of 2^{3d}. Furthermore, it reduces the granularity of the mesh refinement, the number of splits and merges that need to be performed, and the size of the runtime data structures. This reduced granularity means that a tetrahedron tends to remain in the mesh over several time steps.

The use of fixed-size subtrees easily allows us to cache the extracted triangles and gradients in vertex arrays for improved rendering performance. A subtree of depth two has 64 tetrahedra with at most 128 triangles and 92 vertices. With these fixed sizes, we allocate fixed-size chunks of triangles and vertices, and use lookup tables to quickly place the extracted vertices and triangles into vertex arrays. These chunks of triangles are cached in AGP memory and can be rendered extremely quickly using graphics hardware [10].

VI. HARDWARE ACCELERATED ISOCONTOURING

When we move from one time step to the next, the isosurface needs to be reextracted from the tetrahedra in the current mesh. This process is very computationally intensive since it must touch every tetrahedron. For linear interpolation, the equations for positions and gradients along an edge are:

$$\begin{aligned} p &= (1.0-\alpha)\times p_0 + \alpha \times p_1 \\ g &= (1.0-\alpha)\times g_0 + \alpha \times g_1 \\ \alpha &= \frac{(isovalue - d_0)}{d_1 - d_0} \end{aligned}$$

where $p_0, p_1, g_0, g_1, d_0, d_1$ are the positions, gradients and data values at the end points. Performing the interpolations on the CPU requires a lot of data movement and a large number of floating point operations. Modern graphics cards are optimized to perform these vector operations in parallel, and thus it makes sense to perform these interpolations on the graphics card.

When interpolations are performed on the CPU, six floating point values, three for position and three for gradient, are stored for one vertex. To perform these interpolations on the graphics card, we need to send it two positions, two gradients, two data values, and the isovalue for a total of 15 values. However, sending 15 floating point values per vertex increase the amount of data that needs to be transfered by 250% which significantly degrades performance.

Since we are working with byte data on a regular grid, the vertex positions and data values are integers. The gradient components as described in Section III are also integers. The positions, gradients, and data are packed into seven values as follows:

$$\begin{aligned} v_{packed} &= v_0 + v_1 \times 2^k \\ g_{packed} &= g_0 + g_1 \times 2^k \\ d_{packed} &= d_0 + d_1 \times 2^k \end{aligned}$$

Since all the data are integers and we are always scaling by a power of two, the packing is done using only shifts, masks, and logical operations. These packed values are converted to floating point and stored in vertex arrays as described in Section V. The interpolation is performed on the graphics card using a vertex program. The positions, gradients, and data values are unpacked using mod operations. The gradient components are then transformed to the range $[-1, 1]$ and normalized. The α for linear interpolation is computed from the data values by passing the current isovalue as a parameter to the program. By packing the data and decoding it on the graphics card, we send 7 floating point values per vertex which increases the amount of transfered data by 17%.

VII. DATA COMPRESSION

A. Data Transformation

Many types of data cannot be compressed easily because the number of different data values present is high and the corresponding redundancy in the data is low. For better compression, data is often transformed from its regular domain to another domain in order to increase its redundancy. This redundancy is exploited and greater compression results. Data transforms include the FFT, DCT [4], and the well-known families of wavelet transforms [5], [20].

For data that varies gradually and smoothly a simple difference transform, which assumes piecewise constant behavior, is very effective. Given a set of data D with n values the difference transform T of any value i is the difference between it and the previous data value.

$$T(i) = D(i) - D(i-1), 0 < i < n. \tag{1}$$

The first data value at $i = 0$ is passed along unchanged.

The difference transform $T\prime$ used in our compression calculates the difference between two consecutive difference values calculated using equation 1. This transform assumes that the data varies linearly and thus that consecutive difference values should be similar.

$$T\prime(i) = \begin{cases} T(i) & : \quad i = 1 \\ T(i) - (T(i-1)) & : \quad 1 < i \end{cases} \tag{2}$$

The first data value at $i = 0$ is passed along unchanged.

Figure 5 shows the coefficient histogram of a $256^3 \times 274$ time-varying dataset before and after this difference transformation. Even though the magnitude range of the transformed coefficients has increased, a much larger number of the magnitudes are closer to zero. This fact indicates that the transformed coefficients will compress better than the original data values.

B. Compression and Decompression

Figure 6 shows a block diagram of the general data compression process. In the *transform* stage, the data is transformed to another representation to decrease its entropy for better compression. In the *modeling* stage, the data is scanned to collect information on its contents. This information is used in the *code generation* stage to generate codewords which are used in the *encoding* stage to compress the data. Not all coders may follow this process exactly. In particular the modeling

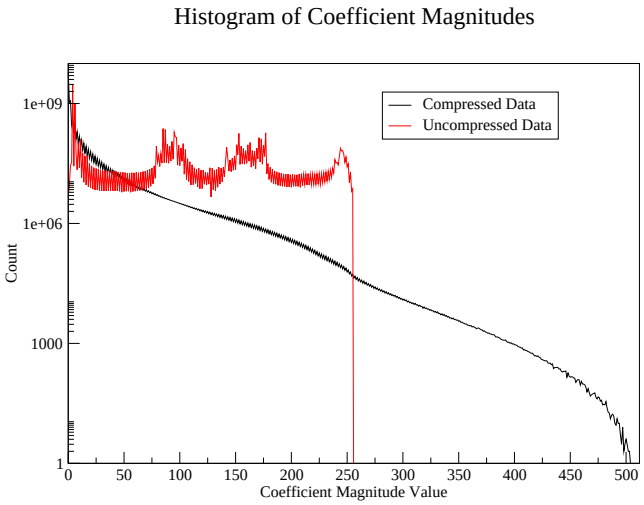

Fig. 5. Histogram of coefficient magnitudes in a 256^3 volume with 274 timesteps. The vertical axis is on a logarithmic scale.

stage often can be integrated with the transformation stage, and in adaptive coders the coding process repeatedly loops through the modeling, code generation, and encoding stages. Since we are developing a system for visualizing data that has

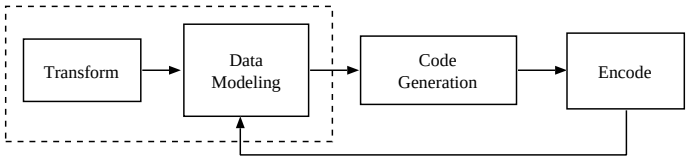

Fig. 6. Flow diagram of the basic encoding process.

been processed and encoded, our main focus is on simplifying and streamlining the decoding process. Furthermore because the datasets have already been quantized, we are interested only in lossless compression techniques. We choose to avoid more sophisticated methods and use the minimum number of operations needed, keeping the operations used to the most basic; thus we use a simple direct table lookup scheme. In a table lookup scheme, assuming there are N possible input symbols, a prefix–free codeword is assigned to each symbol (often using the Huffman [8] algorithm) and an encoding lookup table (LUT) is created with N entries, each entry giving a symbol–codeword association. The encoding process takes an input symbol, uses the symbol as an index into the LUT, and the code given in the LUT entry is written to output. Decoding is similar; assuming the longest codeword is L bits long, a decoding LUT is created with 2^L entries, each entry giving a codeword–symbol association. Then the encoded bits are read in blocks of size L, and those L bits are used as an index into the LUT, which gives the associated symbol and the number of those L bits actually used.

One problem with using a direct table lookup scheme is that the decoding LUT must be a size that is a power of 2 of the longest code length. In the worst case the longest code is equal to the number of possible input symbols minus 1. The difference transform on 8–bit values, shown in Equation 2, yields 512 possible transform coefficient magnitudes, and a possible worst–case code length of 511 bits, although in practice this code length is not likely to be reached. Even if the longest code length is of a more "reasonable" length, it may be that the decode table is larger than desired. This is especially important if memory is tight or performance is critical. Another problem is that the longest code length may be longer than a standard word size (32 bits on most common architectures). Handling these longer code lengths requires extra operations that slow down the encoding and decoding processes.

In order to have small deciding tables for fast decoding, length–limited codes must be used. This can be accomplished using codes generated by the Package–Merge algorithm [9] and its variants [11], [23], or by reducing the number of possible input symbols. Our algorithm follows the latter approach, and reduces the number of possible input symbols.

When examining a set of transform coefficient magnitudes in binary representation, most of the redundancy in a coefficient lies in the position of its leading 1. For a given 9–bit transform coefficient, say 000001010, the leading 000001 compresses rather well, while the following 010 does not. Our encoding scheme, known as "Lead–1 encoding" takes advantage of this observation. Rather than trying to accommodate all 512 possible transform coefficient magnitudes, we encode only the position of the magnitude's leading 1. Because this reduces the total number of input symbols to 10, our longest possible code length is nine bits, and the decoding tables can have at most 512 entries. Each decode LUT entry stores the length of the codeword that indexes to that entry, and the leading 1 position indicated by the code. This information easily fits into a standard byte, and it yields a decoding table size of 256 bytes.

To use Lead–1 encoding we treat difference transform coefficients as a 10–bit signed–magnitude representation, nine bits for the magnitude and, if the magnitude is nonzero, a sign bit. When encoding the transformed coefficients the 9–bit magnitude is read and used as the index into the encoding LUT. The table entry gives the position of the coefficient's leading 1, the length of the associated codeword, and the codeword itself. The codeword is output, and if the coefficient magnitude is nonzero the sign bit is placed on output also. Finally, if the magnitude is greater than 1 all bits following the leading 1 in the magnitude are placed on output unchanged. The decoding process is similar, and is shown in Figure 7. Table I shows the lead–1 counts for transformed and untransformed data. After the difference transform a much larger number of the lead–1 positions are zero, indicating that the data will compress nicely and that the decoding loop will be efficient.

C. Improving the Coding Rate

The table generation step for the basic Lead–1 coder simply collects the number of occurrences of each leading–1 position in the transformed data and uses those counts to create

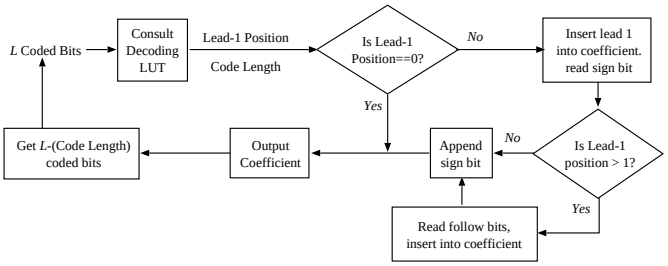

Fig. 7. Coding Loop of basic Lead–1 decoding.

Lead–1	Untransformed	Transformed
0	4743194	6093309314
1	9297339	1075445689
2	61897415	1453200739
3	4119536406	1010756643
4	408098714	880931513
5	282209288	704350384
6	412069938	479577928
7	2753591178	251551499
8	3977820400	78543126
9	0	1597037

TABLE I
LEAD–1 POSITION COUNTS.

the coding tables. This method of data collection and table generation answers the following question: "given a coefficient magnitude, what is the probability that the leading 1 position will be x?" To improve the coding rate we modify the simple coder to use one position of context. When collecting data about the leading 1 positions the table generator conditionally separates the lead–1 counts based on the leading 1 position of the previous coefficient. This method answers the question "given that the previous lead–1 position was x, what is the probability that the current one is y?" Using context in this manner allows us to take advantage of relationships in the data that were not visible with the simple coding method. Table II shows a selection of the conditional counts for a 256^3 dataset with 274 timesteps. For example, if the previous lead–1 position was a 2 the current position is most likely to be a 0. Instead of having a single encoding and decoding table, we now have one for each context (i.e. previous position). The decoding table requires 2560 bytes.

	Prev Lead–1		
Next Lead–1	0	1	2
0	4257612902	440481133	689695448
1	649598731	71511545	112961554
2	525024182	321374553	216536125
3	232651401	109045949	176819114
4	170946802	64646151	122123553
5	125473481	39082448	76907193

TABLE II
CONDITIONAL LEAD–1 POSITION COUNTS.

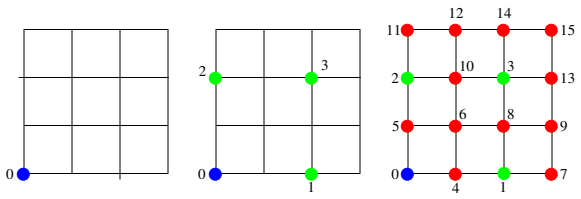

Fig. 8. Hierarchical layout of data points in two dimensions. From left to right: the order of the input data points in a one-dimensional array.

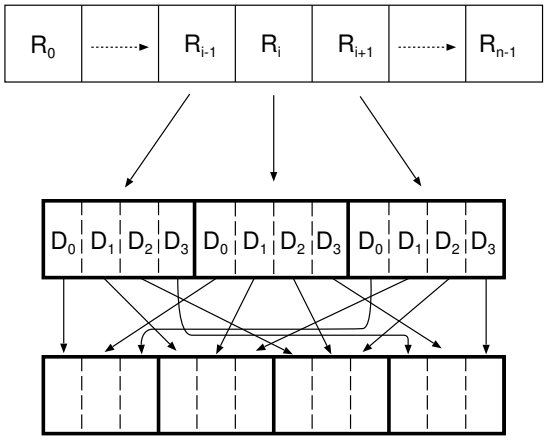

Fig. 9. Top: Meta-records on disk in z-order, Middle: The meta-records are grouped into pages. Bottom: The values in each page are interleaved in time.

VIII. MEMORY LAYOUT

To improve memory performance, the data is arranged in a hierarchical z-layout that follows the mesh refinement structure. The performance benefits of this layout scheme are well known and it has been used for interactive isosurface extraction [6], interactive terrain rendering [12], and slicing of large volumes [16].

Figure 8 shows a 2D example of the layout for a static dataset. The data points are ordered first by levels of detail defined by quadtrees in 2D and octrees in 3D and then by geometric proximity within each level. For static volumes, the dataset is stored first by level-of-detail (i.e. quadtree or octree level) and then by geometric proximity within each level. For time-varying data, we have a sequence of datasets that needs to be stored in a spatially and temporally coherent manner. As shown in in Figure 9, each data point is a meta-record which stores the data values for all time steps at the given spatial location. On disk the records are grouped into pages of $n = 2^k$ elements. Since our data layout scheme orders the data points first by level of detail and then by geometric proximity, each of these pages corresponds to a spatially coherent region within the data hierarchy. Within these pages, the data is interleaved in time so that time steps can be effectively prefetched for the spatial region covered by the page. The size of the pages is a tunable parameter that can be optimized for the capabilities of different storage systems and specific datasets.

Figure 10 illustrates the two extremes of this storage scheme. For $k = 0$ and $n = 1$, each page contains a single element and thus all timesteps for a single data point are stored

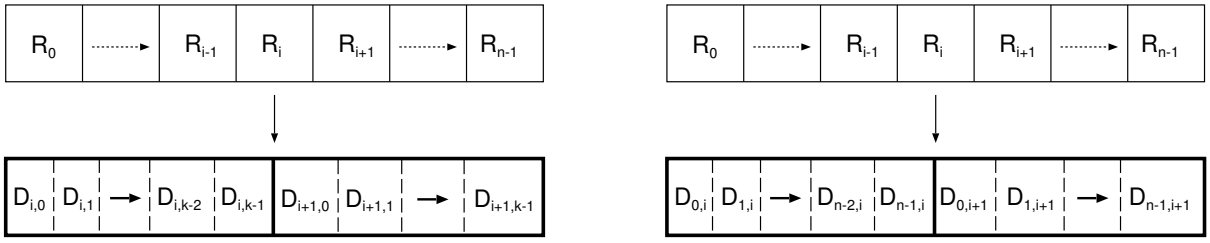

Fig. 10. Left: Page size $k = 0, n = 1$: all timesteps for single data point are store together. Right: Page size $k = i, n = 2^i$, where 2^i is the size of the dataset: all data points for a single timestep are stored together. $D_{i,j}$ indicates data at index i time step j.

together. In this case, the storage scheme has extrememly fine granularity allowing data values to be prefetched over time only for a single data point. While this minimizes the amount of unsed data that is loaded from disk, prefetching time steps for a single data point requires a large number of small disk reads which is very inefficient. If we load several pages at once in a single disk read, all of the time steps for a single data point must be loaded from disk and kept in memory at the same time. This creates a large amount of wasted space as time step 200 must be loaded even if timestep 10 is currently being visualized.

When $k = 3i, n = 2^{3i}$ where 2^{3i} is the size of the dataset, each page contains all data points for a single time step and the storage scheme has coarse granularity allowing all data values for a single time step to be loaded at once. This storage scheme minimizes the number of disk reads, but results in a large amount of unused data being loaded since certain regions will not contain the isosurface and others may be visualized at coarser resolutions. This layout also makes it more expensive to do temporal prefetching for a given spatial region since a disk read is required for each prefetched time step and data for these time steps can be far apart on disk even if the dataset is stored in contiguous disk blocks. Thus, the page size must be selected to balance the disk and memory performance tradeoffs between these two extremes and to allow for efficient prefetching in the temporal domain.

Page faults occur when the requested page or time step is not in memory. The number of time steps loaded when a page fault occurs is slightly randomized to prevent a large number of faults from occurring at the same time. A fixed amount of memory is allocated for the pages, and they are loaded from disk as needed and replaced with a least-recently-used replacement scheme. Since the amount of space needed for k compressed time steps is not known at runtime, for each page that resides in main memory, we create fixed-size buffers for storing compressed and uncompressed data. When compressed data is loaded from disk, the compressed data buffer is filled and the range of loaded time steps is saved. Similarly, when the compressed data is decompressed, we uncompress a fixed number of time steps into the uncompressed data buffer.

A. Compressed Disk Layout

To reduce the storage space and the number of the disk reads, the data and gradients are stored in a compressed form on disk and decoded at runtime. The difference transformation described in Section VII-A is applied to the data values and the two components of the gradient (see Section III) with respect to the temporal domain. The data for every kth time step is stored uncompressed to avoid decompressing time steps $[0, t-1]$ when time step t is requested. The decompression for time step t starts at time step $(t - t \bmod k)$ unless time step $t-1$ has already be decompressed. For each data point, in place of the original uncompressed data, we have uncompressed data and difference transformed values. This data is stored in z-order, grouped into pages, and interlaced as shown in Figure 9. The interlaced difference values are compressed using the algorithm described in Section VII.

IX. RESULTS

Our test machine is a 2 Ghz Pentium with 1 GB of main memory and a GeForce4 Ti 4600 graphics card. We have tested our algorithm on a $256^3 \times 274$ chunk of the Richtmyer-Meshkov instability dataset cropped from the full resolution dataset. The data values measure the entropy of the mixture. They are created by scaling the full range values into the range $[0, 255]$ such that 128 represents the value halfway between the minimum and maximum. Isosurfaces of the dataset contain a large number of small topological components and features as well as larger components with intricate mixing features.

A subtree depth of 2 is used; the uncompressed data and gradients take up 12.844 GB or 48 MB per time step, and the error, min, and max values take up 205 MB. Disk pages for the data and gradients have 2^{12} data points with 274 time steps and 3 bytes per time step for a total of 3.21 MB per page. The compressed data and gradients take up 6.99 GB or 54% of the original size. As described in Section VIII-A, we store every 8th time step uncompressed to allow for progressive decompression. This results in 34 uncompressed time steps which takes up 1.6 GB. The runtime lookup tables for the compressed volume require $1.12MB$. The size of the error, min, and max values is $\frac{1}{64}$ the size of the data and gradients because subtrees of depth 2 perform 1 to 64 refinement. The error, min, and max values are stored uncompressed and loaded into memory at runtime.

Figure 11 shows contours from time step 260 at different levels of resolution. At finer resolutions one can see that the surface contains a large number of small topological components. As we move to coarser resolutions, the overall shape of the surface is maintained, but the topology is not preserved and many of the small topological components

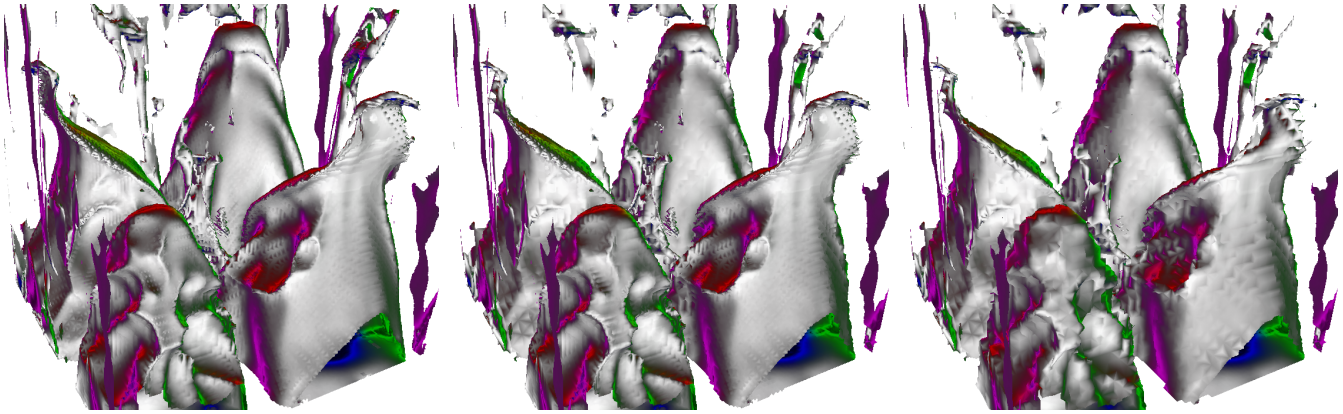

Fig. 11. Different levels of resolution for a fixed viewpoint. Time step = 260, Isovalue = 223.5. From left to right: Error = 0.78, 1.11M Triangles. Error = 1.2, 420K Triangles. Error = 1.8, 235K Triangles.

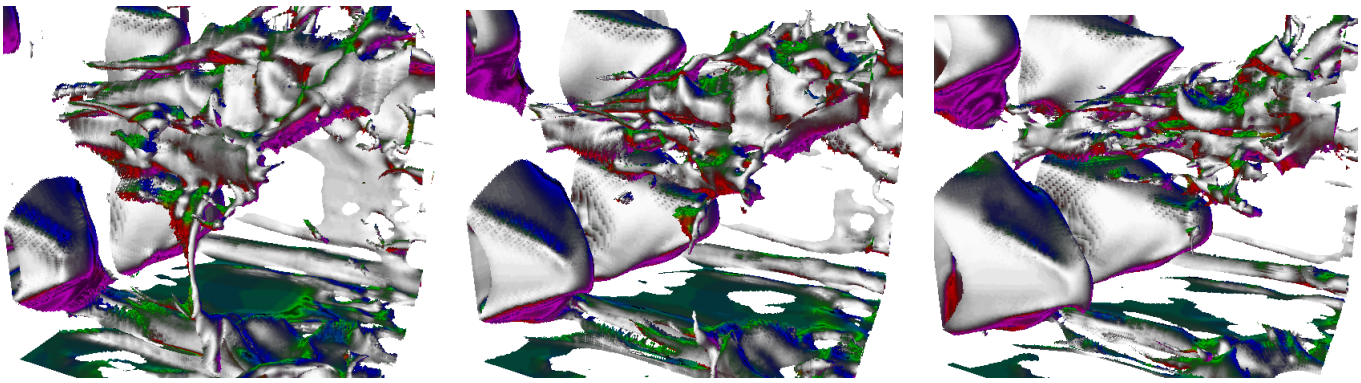

Fig. 12. Time-varying isosurfaces extracted from a 256^3 block of the Richtmyer-Meshkov dataset. Isovalue = 227.9 , Error = 0.5. From left to right: Time Steps 160, 180, and 200.

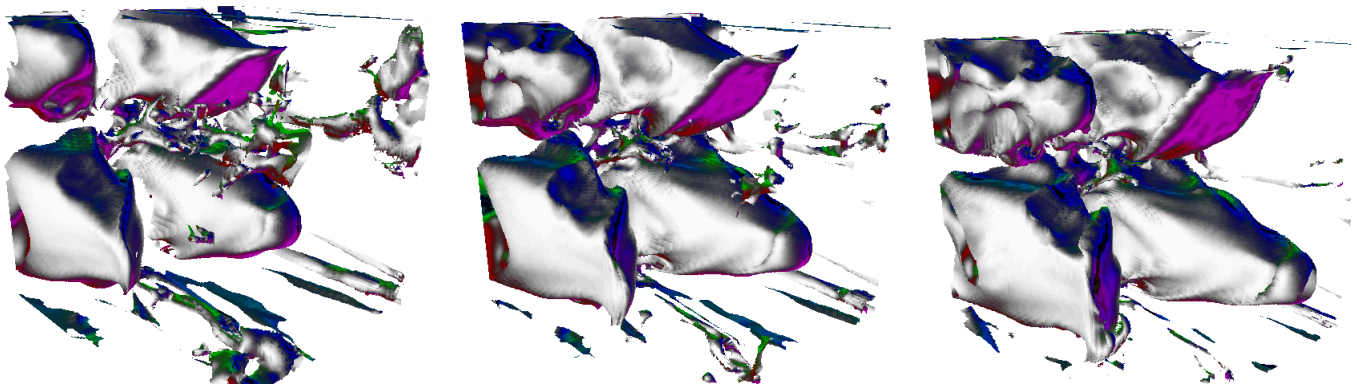

Fig. 13. Time-varying isosurfaces from a 256^3 block of the Richtmyer-Meshkov dataset. Isovalue = 227.9 , Error = 0.85. From left to right: Time Steps 223, 248, and 273.

disappear. Figures 12 and 13 show a series of contours from the same physical region in the simulation extracted at different time steps. These pictures illustrate the formation of the numerous large and small scale features that are present in the mixing interface. Figure 14 shows adaptive refinement of the isosurface over a series of time steps. Images from time steps 150, 211, and 270 are shown at different levels of resolution for a fixed viewpoint as the volume is played forward in time from timestep 0 to 273. The surface at time step 150 is refined with an error of 1.5 to capture some of the intricate folds and topological components of the surface. As time progresses, the large surface patch in the front breaks apart into smaller topologically separate surfaces. To capture these smaller components, the error is changed to 0.5, and finally as the surface becomes less complex, as shown in the image for time step 270, the error is changed to 2.7.

A. Memory Layout Performance

To compare the performance of our algorithm using compressed and uncompressed volumes, we play the volume from

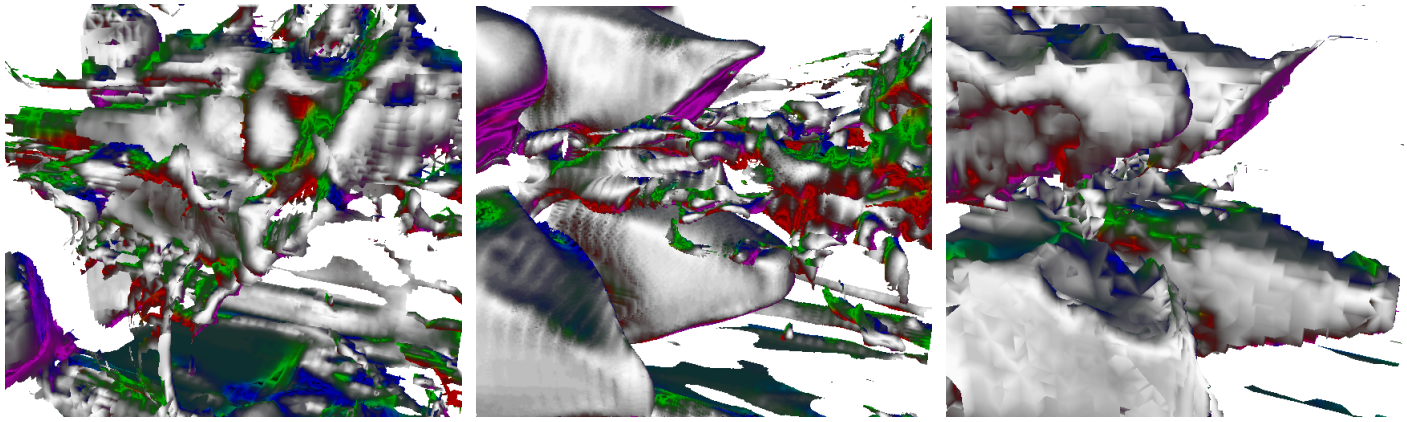

Fig. 14. Adaptive error-based refinement of an isosurface over time for a fixed viewpoint. Isovalue = 225. From left to right: Time step 150, Error = 1.5, 220K Triangles. Time step 211, Error = 0.5, 850K Triangles. Time step 270, Error = 2.7, 51K Triangles.

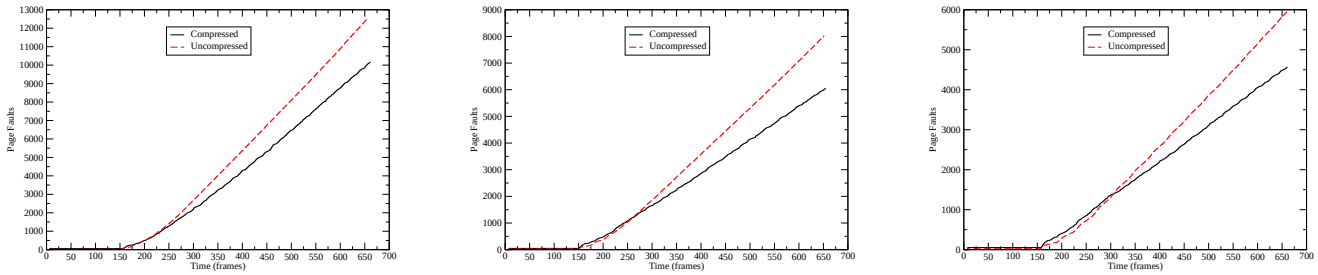

Fig. 15. Comparison of page faults over time for compressed and uncompressed volumes for different load sizes. From left to right: results for disk read sizes of 10, 15, and 20 uncompressed time steps.

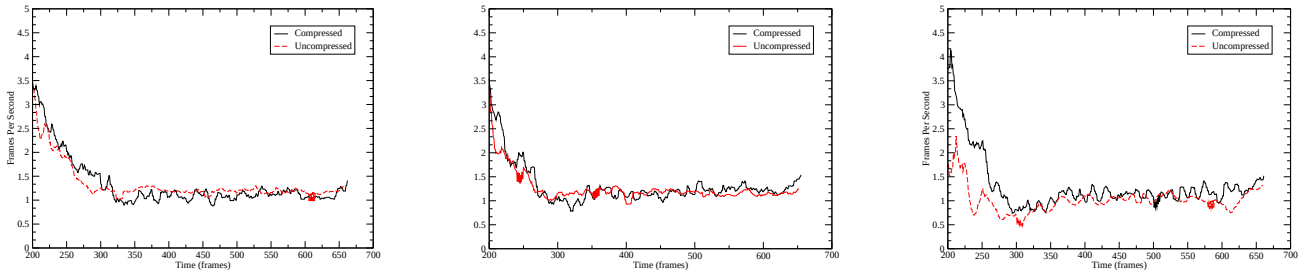

Fig. 16. Comparison of frame rate over time for compressed and uncompressed volumes for different load sizes. From left to right: results for disk read sizes of 10, 15, and 20 uncompressed time steps.

$t = 0$ to $t = 273$ from a fixed viewpoint with a small error value of 0.7. A small error ensures that a high resolution surface is extracted and that the finest levels of the hierarchy must be accessed for the isosurface extraction. To evaluate the amount of disk I/O, the tests are run so that the amount of data read from disk is roughly the same when compressed and uncompressed volumes are used. The amount of data loaded is a multiple of the size of a single time step in an uncompressed page. A single time step in an uncompressed page has 2^{12} or 4096 elements with 3 bytes per element for a total of 12288 bytes. In our tests, we use read sizes of 10, 15, and 20 uncompressed times steps. When the compressed volume is used, 7 time steps are decompressed at a time and kept in memory in addition to the compressed time steps loaded from disk. Figures 15 compares the total number of page faults over time for compressed and uncompressed volumes. Time is the number of frames recorded as the volume is played from start to finish. There are between 660 and 670 frames for each playback sequence. As is expected, the use of the compressed volume reduces the amount of data read from disk and the frequency of the disk reads. The linear increase in the amount of data read and the number of page faults during the playback

Load Size	10	15	20
Compressed	208904	270344	331784
Uncompressed	122888	184328	245768

TABLE III

SIZE IN BYTES OF MAIN MEMORY PAGES.

Load Size	10	15	20
Compressed	1.05(15%)	0.9(12.87%)	0.87(12.4%)
Uncompressed	1.43(11.1%)	1.38(10.72%)	1.36(10.6%)

TABLE IV

THE TOTAL AMOUNT OF DATA(GB) LOADED FOR EACH TEST RUN.

Error	Triangles	Hardware Interpolation	Reextract Tris/Sec
1.8	75-100K	no	755K
		yes	2.02M
1.3	200-300K	no	812K
		yes	1.9M
0.8	650-750K	no	625K
		yes	1.08M

TABLE V

COMPARISON OF TRIANGLE EXTRACTION RATES FOR HARDWARE AND SOFTWARE INTERPOLATION FOR DIFFERENT ERROR BOUNDS.

indicates that the data storage scheme, which balances spatial and temporal locality, does a good job of streaming the needed data from disk. A total of 510 different disk pages were loaded during playback. Tables IV and IV summarize the size of the main memory pages and the amount of data read for the compressed and uncompressed volumes at the different load sizes. Even when viewing a surface at a very high resolution, for many time steps the extracted surface has over a million triangles, the actual amount of data loaded is relatively small compared to the total size of the datasets. As expected, the percentage of data loaded when using the compressed volume is higher because it uses less storage, and the amount of data loaded in a single read is the same between compressed and uncompressed playback.

The frame rates over time for the compressed and uncompressed volumes are shown Figure 16. The data for the initial 200 frames is not shown because the surfaces in those frames contain fewer triangles and have higher frame rates which are not indicative of the overall performance. The compressed and uncompressed volumes both playback at 1-2 frames per second. The extra cost of decompressing and reconstructing the compressed volume is roughly equivalent to the extra cost of the disk reads when using the uncompressed volume. Using the compressed volume, we achieve the same performance as the uncompressed volume using about half of the disk space.

B. Hardware Accelerated Contouring

Table V summarizes the improvements in triangle extraction rate for hardware interpolation versus software interpolation for different error bounds and triangle counts. The numbers are calculated by playing the uncompressed volume from time step 0 to time step 273, advancing the time step at regular intervals, and recording the time to generate the new isosurface from the current mesh.

The isosurface is rendered from a fixed viewpoint in a 570×530 screen. To ensure consistent memory and disk performance, the page table and all caches are cleared at the start of each test. For error bounds of 1.8 and 1.3, hardware interpolation is two to three times faster than software interpolation. For an error bound of 0.8, hardware interpolation is about 70% faster. At higher errors, there are fewer page faults and thus the performance benefits of hardware over software interpolation dominate the performance of the extraction process. At lower errors, the number of page faults increases as more data is loaded to extract the higher resolution surface. At finer resolutions (i.e. lower errors), the performance benefits of hardware extraction start to be overshadowed by the slower performance of loading data from disk. Despite the increased cost of data access at low error bounds, the hardware interpolation still improves the extraction rate by about 455K triangles per second. Recalculating the interpolations every frame does reduce the raw rendering performance of the hardware by about 35%. For static scenes, where the view point and time step do not change and the mesh is not refined, we can render between 16 and 18 million textured triangles a second. When hardware interpolations are used, the extra cost of the vertex program causes the steady state rendering performance to drop to between 10 and 12 million triangles a second. For a typical surface with 750K triangles, this corresponds to a increase in the time to render a frame from 0.047 to 0.07 seconds. However, as graphics card performance improves, the effect of the vertex program will become less and less significant.

X. CONCLUSIONS

We have presented a new algorithm for adaptively extracting contours from compressed, time-varying datasets. Our error-based refinement scheme allows us to adaptively refine and coarsen our approximation over time to focus on the spatial locations with high error. A hierarchical data layout scheme allows us to exploit spatial and temporal locality in the data access so that consecutive time steps for different spatial regions can be prefetched as the volume is played forwards or backwards in time. This layout scheme improves memory performance, and allows our algorithm to operate out-of-core. The reextraction of a new isosurface at each time step is accelerated by the use of a hardware vertex program to perform the interpolations.

Our future work is focused on the following areas:

- Faster isosurface extraction when the time step changes using parallel processing and graphics hardware.
- Improved data compression that uses spatial as well as temporal information to further reduce disk and memory usage.
- Surface shading techniques that indicate temporal properties such as gradient magnitude and changes in the gradient.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. We also thank the people at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California Davis for their help and suggestions.

REFERENCES

[1] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization*, pages 235–244, 1997.

[2] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *IEEE Visualization '97*, pages 81–88. IEEE Computer Society Press, October 1997.

[3] Thomas Gerstner and Renato Pajarola. Topology Preserving And Controlled Topology Simplifying Multiresolution Isosurface Extraction. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 259–266. IEEE Computer Society Press, 2000.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison–Wesley, 1992. See Chapter 3.

[5] Amara Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2), 1995.

[6] Benjamin Gregorski, Mark A. Duchaineau, Peter Lindstrom, Valerio Pascucci, and Kenneth I. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings of the IEEE Visualization 2002*, 2002.

[7] Stefan Guthe and Wolfgang Staser. Real-time decompression and visualization of animated volume data. In *Proceedings of IEEE Visualization 2001*, pages 349–358, 2001.

[8] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[9] Lawrence Larmore and Daniel Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the Association for Computing Machinery*, 37(3):464–473, Jul 1990.

[10] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings of IEEE Visualization 2002*, pages 259–265, 2002.

[11] Mike Liddell and Alistair Moffat. Incremental calculation of minimum–redundancy length–restricted codes. In *Proceedings of the Data Compression Conference (DCC 2002)*, 2002.

[12] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In T. Ertl, K. Joy, and A. Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 363–370. IEEE Computer Society Press, 2001.

[13] Eric Lum, Kwan-Liu Ma, and John Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of IEEE Visualization 2001*, pages 263–270, 2001.

[14] J. Maubach. Local bisection refinement for n-simplicial grids generated by reflections. In *SIAM Journal on Scientific and Statistical Computing*, volume 6, pages 210–227, 1995.

[15] Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, William P. Dannevik, Andris M. Dimits, Mark A. Duchaineau, D. E. Eliason, Daniel R. Schikore, S. E. Anderson, D. H. Porter, , and Paul R. Woodward. Very High Resolution Simulation Of Compressible Turbulence On The IBM-SP System. In *Proceedings of SuperComputing 1999*. (Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237), 1999.

[16] Valerio Pascucci. Multi-Resolution Indexing For Out-Of-Core Adaptive Traversal Of Regular Grids. In G. Farin, H. Hagen, and B. Hamann, editors, *Proceedings of the NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometric Methods for Scientific Visualization*. Springer-Verlag, Berlin, Germany, (to appear)., 2002. (Available as LLNL technical report UCRL-JC-140581).

[17] Tom Roxborough and Gregory M. Nielson. Tetrahedron Based, Least Squares, Progressive Volume Models With Application To Freehand Ultrasound Data. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings Visualization 2000*, pages 93–100. IEEE Computer Society Press, 2000.

[18] Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 159–166. IEEE, 1998.

[19] Han-Wei Shen, Ling-Jan Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 371–378, San Francisco, 1999.

[20] Stollnitz, DeRose, and Salesin. *Wavelets for Computer Graphics*. Morgan Kaufffmann, 1996.

[21] P. M. Sutton and C. D. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, April/June 2000.

[22] Philip M. Sutton and Charles D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In David Ebert, Markus Gross, and Bernd Hamann, editors, *EEE Visualization '99*, pages 147–154, San Francisco, 1999.

[23] Andrew Turpin and Alistair Moffat. Efficient implementation of the package-merge paradigm for generating lengthlimited codes. In *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 187–195, Jan 1996.

[24] Rudiger Westermann. Compression domain rendering of time-resolved volume data. In *Proceedings of IEEE Visualization 1995*, pages 168–175, 1995.

[25] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[26] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution Tetrahedral Framework For Visualizing Regular Volume Data. In *Proceedings of IEEE Visualization 1997*, pages 135–142. IEEE Computer Society Press, 1997.

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551