# 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 [13] conducted at Lawrence Livermore National Laboratory which is used to test our algorithms. 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. The simulation was run for 27,000 time steps, and the output consists of 274 BOB files 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 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]. The mesh refinement scheme has fast coarsening and refinement operations necessary for adaptive, view-dependent 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, given an isovalue and a time step, we adapt the previous time step's mesh 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

Isocontouring algorithms such as [1], [22] precompute minimum and maximum values to quickly find cells that contain the isosurface. However, they extract the isosurface from the finest cells in the mesh. For large isosurfaces with several million triangles, extraction, storage and rendering are very expensive. Adaptive isocontouring is a valuable technique for interactively visualizing these large surfaces. Westermann et al. [20] use octrees to define a multiresoltion hierarchy on a volume dataset. The octree is adaptively refined and coarsened based on metrics such as distance from the viewpoint and local curvature. To fix the cracks introduced by the octree, triangle fans are created between adjacent cells at different levels of resolution. The refinement of a tetrahedral mesh via longest-edge bisection described in [12] has been used in many scientific visualization applications. In [23], 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 [5]. Roxborough and Nielson [15] 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

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

layout scheme of [10], [14] is used for out-of-core visualization. Grosso et al. [7] develop hierarchical algorithms for contouring multilevel meshes that contain hexahedra, prisms and tetrahedra. Cracks in the adaptive mesh are fixed by using conforming splits to add elements that eliminate hanging nodes. Lookup tables based on hanging node configurations are used to perform the isosurface extraction.

Techniques for visualizing time-varying datasets focus on exploiting spatial and temporal locality.Spatial techniques reduce 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 [16] exploits temporal coherence by adaptively joining cells whose extreme values do not change much over time. A tree structure is used to quickly locate the regions of the dataset that contain the isosurface. In [18], [19] Sutton et al. extend the Branch-on-Need Octree(BONO) [22] 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. The Time-Space Partitioning Tree [17] extends the Branch-on-Need Octree [22] 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 approximating regions with coarser volumes, and it exploits temporal coherence by reusing rendered images between time steps. The data structure is adapted for out-of-core visualization by dividing the dataset into bricks and utilizing a demand paging system [3].

Compression based techniques utilize coherence over time to reduce the size of the datasets for faster disk loads and rendering. Westermann [21] 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 [8], Guthe and Strasser use wavelet transforms, run-length encoding, and arithmetic coding to compress time-varying volume datasets. Individual volumes are encoded using a wavelet transform and compressed using run-length and arithmetic coding. A sequence of compressed volumes is further compressed using MPEG-style encoding and compression.Unlike [21] their technique decompresses the whole volume at each time step. Lum et al. [11] use a DCT and Lloyd-Max quantization to compress time-varying volumes. A group of values over time is first transformed and then compressed using a lossy quantization process. Rendering is performed directly from the compressed volumes allowing interactive rendering and 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 [16], [18], [19] 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. Tetrahedral Mesh Refinement

In this section we review the refinement of a tetrahedral mesh by longest-edge bisection and illustrate its connection with the z-order space filling curve [14] which we utilize to efficiently organize our data. Our mesh refinement has three phases and begins with a cube divided into six tetrahedra around its major diagonal. Figure 1 shows the three phases which occur on cube diagonals, face diagonals, and edges. The edge that is refined is the *split edge*, and the added vertex is the *split vertex*. To prevent hanging edges and cracks in the isosurface, tetrahedra that share a split edge are grouped into structures called *diamonds*. Diamonds are the basic unit of refinement. When an edge is split, all tetrahedra in the edge's diamond are split. In an adaptive mesh, this can require recursive splits of coarser diamonds. The three types of diamonds are shown in Figure 2. Each spatial location in a dataset is the split vertex of a diamond. The root diamond is located at the dataset's center. For a $64^3$ volume, the root diamond's split vertex is at $(32, 32, 32)$

The mesh is refined and coarsened using the *split* and *merge* operations. Splitting a diamond divides each of its tetrahedra into two child tetrahedra. Merging a diamond reverses the splitting process, and pairs of child tetrahedra are coalesced into a single tetrahedron. Splitting and merging diamonds ensures that the mesh is always crack-free. These operations are implemented with two error-based priority queues [4] [6], the split queue and the merge queue, which contain all diamonds that can be split and merged respectively. The split queue contains diamonds that have not been split (i.e. their tetrahedra are leaf tetrahedra in the mesh). The merge queue holds diamonds that have been split but whose child tetrahedra have not been split (i.e. their tetrahedra have children but no grandchildren). At runtime, diamonds in the split queue are refined, and diamonds in the merge queue are coarsened with respect to an error threshhold $E$.

Figure 3 illustrates the data access patterns of the mesh refinement and the z-order space filling curve. The order in which the mesh refinement accesses the data at each level is essentially the same as that given by the z-order curve. The difference is that the mesh refinement operates on $(2^k + 1)^3$ grids, and the z-order curve works on $(2^k)^3$ grids. The extra

points on the boundary of the tetrahedral mesh are naturally handled by the z-order curve which uniformly tiles the grid in space causing index $2^k$ to wrap to 0. In three dimensions, each point $P$ in the dataset is indexed as an $(i, j, k)$ triple which corresponds to a $z - order$ index on the one dimensional z-order curve. Conversion between $(i, j, k)$ and $z-order$ indices is fast and easy as detailed in [14]. This relationship between the mesh hierarchy and the z-order curve give excellent disk and memory coherence which makes them well suited for adaptive, out-of-core visualization of large datasets.

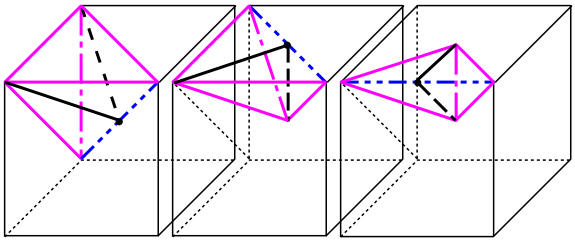

Fig. 1. Left to right: phases two, one, and zero of the mesh refinement.

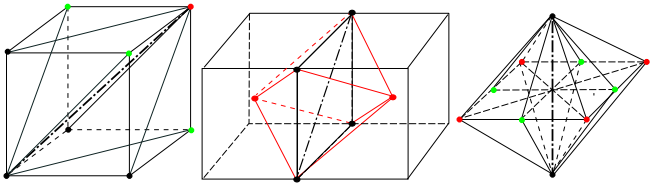

Fig. 2. Left to right: diamonds of phases two, one, and zero containing six, four and eight tetrahedra respectively. The split edge is the bold, circle-segment dashed line.

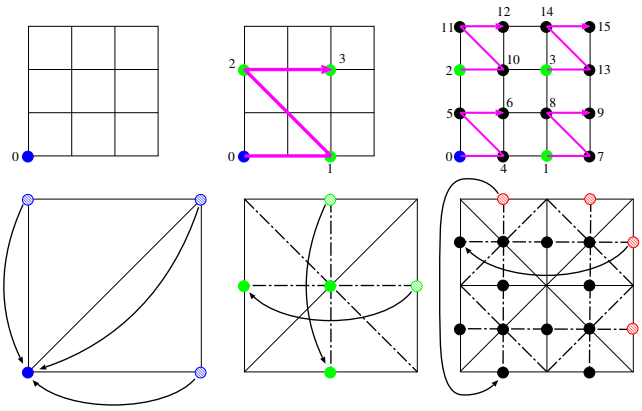


Fig. 3. Two-dimensional comparison of the data access patterns indicated by the mesh refinement and the z-order space filling curve. Top: the order of the data points on the z-order curve. Bottom: The data points introduced by the mesh refinement at each hierachy level.

Our crack fixing method differs from methods such as [20] that add extra geometry to fix cracks, and it is similar to methods such as [7] which use extra splits to create a conformant mesh. We choose to fix the cracks using the extra splits of the tetrahedral mesh for several reasons. First, the regular structure of the mesh allows these splits to be done very efficiently using lookup tables. Second, the tetrahedral mesh refinement allows us to leverage the performance benefits of the z-order space filling curve which is one of the keys for interactive, out-of-core visualization with of large datasets. While this can be done for quadtrees and octrees, the tetrahedral mesh introduces the data points at a slower rate allowing more localized refinement. Last, fixing cracks with extra splits makes contouring a tetrahedron a local algorithm which easily allows parallel implementations. It is not necessary to use extra neighbor information which can require extra pointers or extra tree traversals. Subtrees described in Section VI allow alternative contouring algorithms, such as dual contouring or contouring more complex elements such as prisms and hexahedra, to be utilized in local regions to improve the quality of the isosurface.

## IV. 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 (Section III) in the mesh.
2) The normalized gradient vector at each data point. The gradient is used as a texture coordinate to perform diffuse lighting and to highlight the regions orthogonal to the viewing direction. The texture used for rendering is shown in Figure 4.

For a tetrahedron $T$, the isosurface approximation error is the maximum difference between the isosurface generated using the scalar values at $T$'s vertices from the true isosurface passing through $T$. It is proportional to the inverse of the gradient magnitude of the linear interpolant defined over $T$. The min and max data values are used to quickly eliminate regions that do not contain the isosurface. 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 VI.

The data values are bytes, and the gradients are normalized and quantized in 16 bits. Many algorithms exist for quantizing gradients. The gradients can be uniformly distributed over a cube or sphere or a codebook based quantization algorithm can be used to create an optimal set of gradients. In these techniques, the gradient value is an index into a lookup table. Gradients quantized in this manner often do not have a lot of coherence. Two gradients that are very similar, as measured by their dot product for example, can have very different quantized values. For data compression purposes, we need a gradient representation that has more coherence and a meaningful delta between consecutive values over time. To meet these requirements, the gradients are quantized component-wise with eight bits for $x$, seven for $y$, and one bit for $z$'s sign. This format gives high accuracy to two components and less accuracy to the third. The max x, y, and z component errors are 0.004, 0.008, and 0.13. Higher z-component errors occurs when it is near 0. The maximum error in our quantized gradients, as measured by the deviation of the dot product between the original and quantized vectors, is 0.012 (i.e. the

dot product is 0.988). This 16 bit value is split into two 8 bit values; the first being $x$ and the second being $y$ with $z$'s sign as the least significant bit.

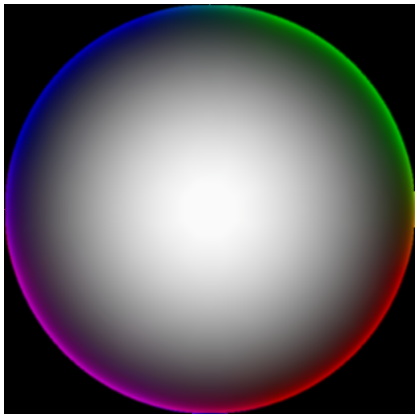

Fig. 4. Diffuse lit sphere texture map used for shading.

### 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 isosurfaces extracted from different levels of detail, noisy and degenerate gradients are especially bad for shading. In order to improve the quality of the gradients for shading purposes, they are smoothed in the spatial and temporal domains to diffuse non-degenerate gradients into the degenerate regions.

Figure 5 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:

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

Gray areas indicate regions where the gradient is $(0, 0, 0)$.

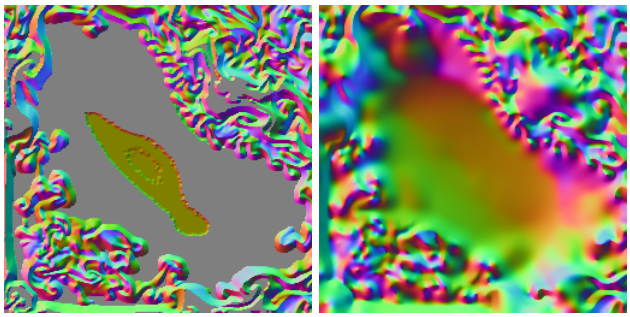

Fig. 5. Left to Right: Images of raw and smoothed gradients.

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 itself and the surrounding gradients. The second phase is a constrained symmetric diffusion, the new gradient, again a weighted average of itself and the surrounding gradients, is not allowed to move to far from the value obtained after the first diffusion phase. In the degenerate regions, the symmetric diffusion from the first phase is performed. Results of this process are illustrated in Figure 5.

## V. RUNTIME

### A. Mesh Refinement for Static Volume

Given an error bound $E$, isovalue $I$, and time step $T$, the refinement process creates a set of tetrahedra $S_t$ that approximates the regions of the dataset containing $I$ 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 $I$ are assigned an error of zero. Errors are recomputed for all other diamonds in the split and merge queues.
2) The split and merge refinement process (Section III) 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 V-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.

Adaptive refinement over time allows us to exploit temporal coherence in the volume. Tetrahedra with low temporal variation will remain in the mesh over several time steps. When moving from time step $T$ to $T+1$, many diamonds from time step $T$ will have an error that is still within the error tolerance, and a small fraction of the diamonds must be split or merged to satisfy the error tolerance. While the isosurface for $T+1$ still needs to be extracted from all tetrahedra in the current

mesh, initiating the refinement process for time step $T+1$ with the mesh from time step $T$ requires fewer splits and merges than beginning from the root diamond and allows speculative paging of data from future time steps (Section IX). Similarly as the viewing parameters change from frame $F$ to $F+1$, tetrahedra where the isosurface is simple will remain in the mesh for several frames, and tetrahedra where the isosurface is more complex with be refined and coarsened as they move toward and away from the view point.

The performance of the refinement process as the time step and viewing parameters change depends on the spatial and temporal coherence of the volume, the error threshold, and the number of error calculations. When the viewing parameters change slowly and smoothly, the system takes advantage of frame-to-frame coherence as many tetrahedra remain in the mesh between frames and much of the data needed to extract the new isosurface is already loaded from disk. This is true even when the isosurface is being extracted at a high resolution as the amount of mesh reuse between frames is high although it often decreases as the resolution of the extracted surface increases. Additionally, the isosurface resolution affects system performance because the cost of extracting the new isosurface is directly proportional to its resolution. Conversely large changes in the viewing parameters are often very incoherent and will subsequently cause performance to decrease. The cost of the error calculations performed each frame is reduced by the use of Subtrees (Section VI).

### C. Changing The Isovalue

When the isovalue is changed, the new isosurface can be extracted by starting at the root diamond or starting from the current mesh as shown in Figure 6. In the first case, the split queue, merge queue and isosurface are invalidated and initialized with the root diamond. Refinement then begins from this initial configuration. In the second case, the split queue and merge queue remain the same, but the old isosurface is discarded. Isosurface errors are computed for those diamonds that contain the new isovalue, and the mesh refinement continues from this new configuration. Regions that contained the old surface but do not contain the new surface will be coarsened since they are not needed, and regions that contain the new surface will be refined as coarsened as necessary to meet the error threshold.

The performance and effectiveness of both of these methods depends on the distance between the old and new isosurfaces in the mesh hierarchy. Given two isosurfaces $I_0$ and $I_1$ and their corresponding meshes $M_0$ and $M_1$, the distance between the isosurfaces is measured as the number of splits and merges necessary to convert $M_0$ to $M_1$. Starting from the current configuration makes sense if they are close together, and starting from the base configuration makes sense if they are far apart. Similarly if a new isosurface is close to the existing surface, extracting the new surface and refining the mesh will be very coherent since much of the necessary data is already in memory and the data structures do not need to be rebuilt. On the other hand, if the two surfaces are far apart, extracting the new surface will be very incoherent and will probably generate a large number of page faults as the system loads data from regions stored on disk.

## VI. SUBTREES

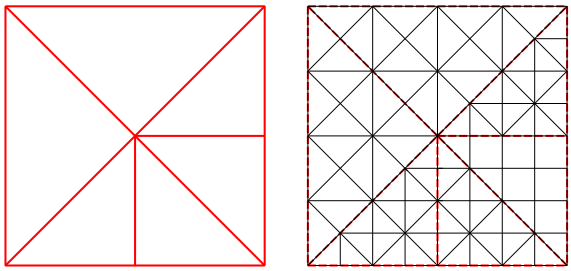

Fig. 8. Two-dimensional example of subtrees in an adaptively refined mesh. A coarse adaptively refined mesh's elements are replaced with subtrees. Coarse triangles are implicitly replaced with a set of finer elements, and contouring is performed on the finer mesh.

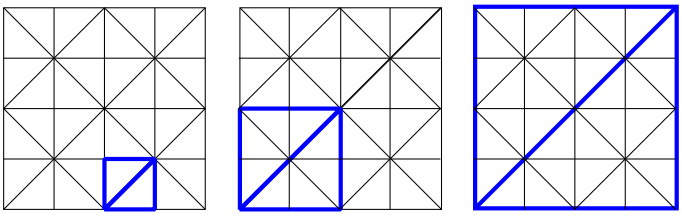

Fig. 9. Left to right: Leaf nodes for subtrees of depth zero, one, two for a two-dimensional 5x5 grid. The bold lines indicate the leaf diamonds.

In order to reduce the storage cost of the precomputed data and the granularity of the refinement process, a single coarse tetrahedron $T$ is implicitly replaced with a set of smaller tetrahedra $S_T$ called a *subtree* or a *chunk*. Figure 7 shows two-dimensional examples of a triangle divided into subtrees of depths one and two. This is similar to aggregate triangle structures or chunks used to improve performance in terrain rendering applications [9] [2]. We precompute $S_t$ and replace the coarse tetrahedron with $S_t$ when the isosurface is extracted. The mesh refinement is performed on the coarser level and the isosurface extraction is performed on the finer level indicated by the subtrees. Isosurfaces can still be extracted from coarse mesh levels just not levels coarser than the subtree depth. $T$'s min and max values remain the same, and $T$'s error value is the maximum of its subtree's tetrahedras' errors. The error value is given as:

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

In general, a subtree does not have to be composed of triangles or tetrahedra. It can use any element type, hexahedra, prisms etc., and any algorithm can be used to extract the isosurface from the subtree elements as long as there is a consistent tesselation of the boundary edges and faces. Figure 8 shows an adaptively refined mesh before and after the use of subtrees. Two different types of subtrees are used but because the boundaries are consistent there will be no cracks in the contour. One triangle on the lower right has a subtree

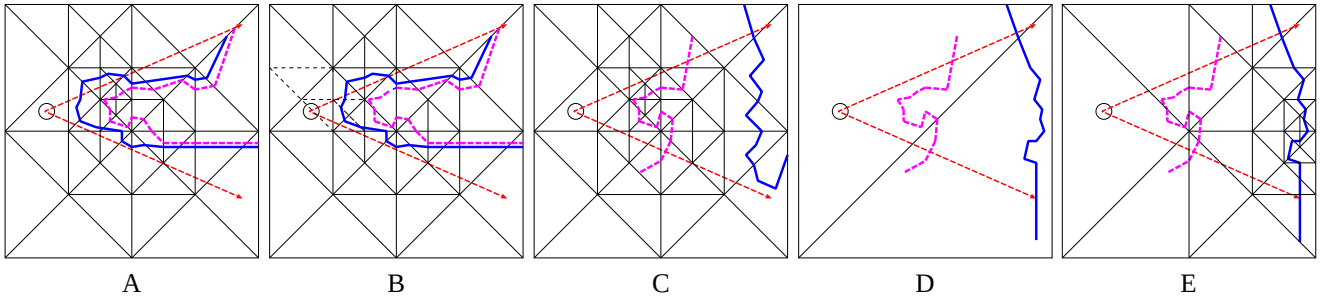


Fig. 6. A-B: The new contour is close to the old contour. The new contour is extracted from the current mesh and refinement continues. C-E: The two contours are far away. The old mesh is discarded, and refinement starts from the base mesh.

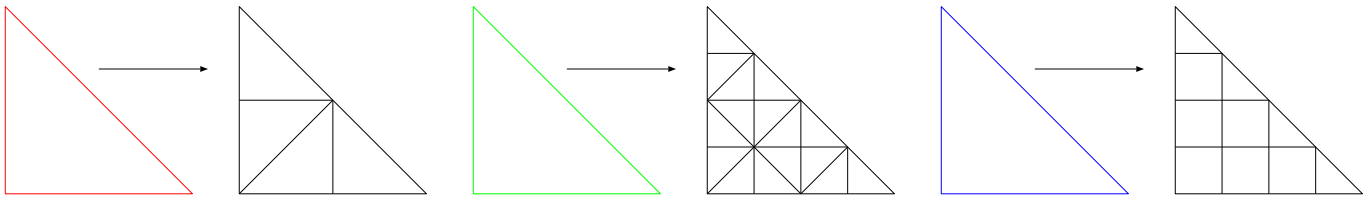

Fig. 7. Division of a triangle into different subtrees. Left to right: Subtree of depth 1, subtree of depth 2, and subtree using triangular and quadrilateral elements that maintains the same boundary refinement as the depth 2 subtree.

composed of six quadrilaterals and four triangles, and the other triangles have subtrees composed of 16 triangles. The isosurface is extracted from the elements in $S_T$ instead of $T$ itself.

The size of a subtree 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 9 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. Thus, the error, min, and max values do not need to be computed and stored for the diamonds that exist at finer levels than the leaf diamonds. For a $1024^3$ volume only $256^3$ errors, min, and max values need to be computed. Subtrees reduce the storage cost of the precomputed errors, min, and max values (Section IV) by a factor of $2^{3d}$. Furthermore, subtrees reduce 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. Since once tetrahedron actually represents $64$ smaller tetrahedra, its error is smaller and thus larger changes in the error bound and viewing parameters must occur before it it removed from the mesh. This reduced granularity also 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 [9].

## VII. 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 IV are also integers. The positions, gradients, and data are packed into seven values as follows:

$$\begin{aligned} p_{packed} &= p_0 + p_1 \times 2^k \\ g_{packed} &= g_0 + g_1 \times 2^k \\ d_{packed} &= d_0 + d_1 \times 2^k \end{aligned}$$

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 VI. 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 $\alpha$ 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%.

## VIII. DATA COMPRESSION

### A. Data Transformation

For data that varies gradually and smoothly a difference transform between adjacent values is very effective for increasing the data's redundancy or decreasing its entropy. The difference transform $T_L$ that we use assumes that the data varies linearly.

$$T_C(i) = D(i) - D(i-1), 0 < i < n. \quad (1)$$

$$T_L(i) = \begin{cases} T_C(i) & : \quad i = 1 \\ T_C(i) - T_C(i-1) & : \quad 1 < i \end{cases} \quad (2)$$

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

Data transformation and compression are performed in the temporal domain. The set of data $D$, transformed using Equation 2, is the sequence of data and gradient values over time for a single spatial location. Figure 10 shows the coefficient histogram of a $256^3 \times 274$ time-varying dataset before and after the 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

Since we are visualizing data that has been processed and encoded, we focus on simplifying and streamlining the decoding process. Because the datasets have already been quantized, we are interested only in lossless compression. When examining a set of transform coefficient magnitudes

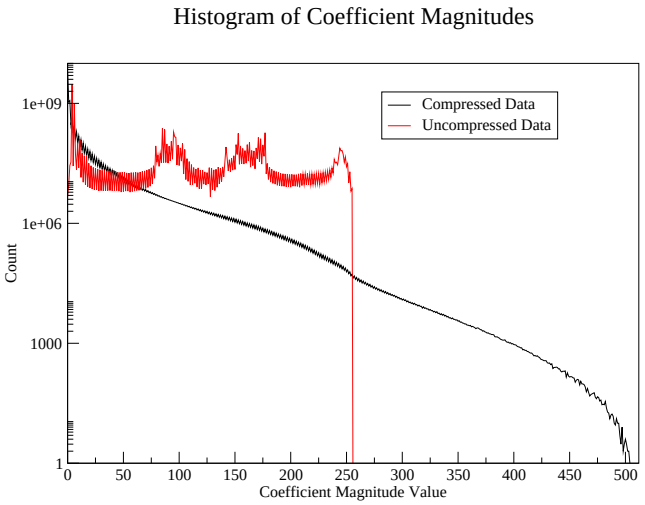


Fig. 10. Histogram of coefficient magnitudes in a $256^3$ volume with 274 timesteps. The vertical axis is on a logarithmic scale.

in binary representation, most of the redundancy in a coefficient lies in the position of its leading 1. Given a 9–bit transform coefficient, such as 000001010, the leading 000001 compresses rather well, and the following 010 does not. We utilize Lead–1 encoding to take advantage of this. Instead of encoding all 512 possible transform coefficients, we encode only the position of the magnitude's leading 1. This reduces the total number of input symbols that must be coded from 512 to 10 (nine locations for the leading 1 and the zero magnitude). The codewords are generated using the Huffman algorithm which cannot generate a code word greater than $n - 1$ bits when generating codes for $n$ symbols. Since we have only 10 symbols to encode, codewords are at most nine bits, and the decoding table has at most $2^9$ entries. The difference transform coefficients are treated as a 10–bit signed–magnitude representation, nine bits for the magnitude and, if the magnitude is nonzero, a sign bit. The encoded output consists of the codeword for the leading 1 position, and, if the magnitude is non-zero, a sign bit and all bits following the leading 1. Table I shows the Lead–1 counts for the transformed and untransformed data.

### C. Design Considerations

Lead-1 encoding and Huffman codes were chosen for several reasons. Encoding and especially decoding needs to be fast. Our algorithm uses direct table lookups which satisfies this criteria. To minimize the size of the encoding and decoding lookup tables, we need to decrease the number of symbols that need to be encoded. The generated Huffman codes are limited in length by the number of input symbols. If there are $n$ possible symbols to be encoded, the longest code length will be at most $n - 1$ bits. Lead–1 encoding reduces the number of symbols that need to be encoded thus reducing the size of the codes and the table size. An entry in the decoding table stores the length of the codeword that indexes to that

| 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 FOR A $256^3 \times 274$ TIME-VARYING DATASET.

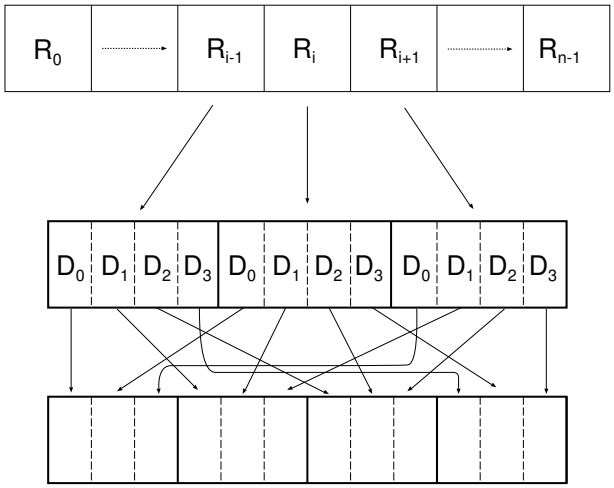


Fig. 11. Top to bottom: Meta-records on disk in z-order are grouped into pages and then interleaved in time.

entry and the leading 1 position indicated by the code. This information is packed into a single byte yielding very small decoding tables that can fit into cache memory. Since our goal is to decompress data as it is needed for contouring, we choose to have faster decoding over increased compression. Compared with arithmetic and arithmetic-style encoders which have better compression performance, our encoding scheme using Lead–1 encoding and Huffman codes achieves decent compression with superior speed and throughput.

## IX. MEMORY LAYOUT

To improve memory performance, the data is arranged in hierarchical z-order (Section III). The performance benefits of this layout scheme are well known and it has been used for interactive isosurface extraction [6], interactive terrain rendering [10], and slicing of large volumes [14]. 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 11, 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 12 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 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 unused data that is loaded from disk, prefetching time steps for several data points 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 $k = 3i$ and $n = 2^{3i}$, $2^{3i}$ 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 VIII-A is applied to the data values and the two components of the gradient (see Section IV) with respect to the temporal domain. The data for every $k$th time step is stored uncompressed to avoid decompressing time steps $[0, t-1]$ when time step $t$ is requested. The decompression for

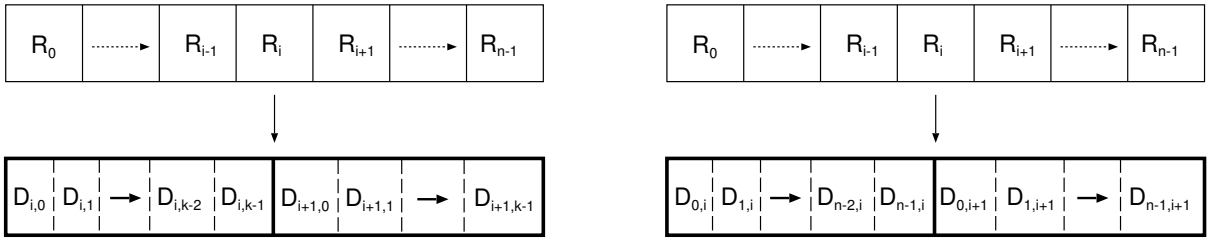


Fig. 12. Left: Page size $k = 0, n = 1$. Right: Page size $k = i, n = 2^i$, where $2^i$ is the size of the dataset.$D_{i,j}$ indicates data at index $i$ time step $j$.

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 11. The interlaced difference values are compressed using the algorithm described in Section VIII.

| Load Size | 10 | 15 | 20 |
|---|---|---|---|
| Data Loaded(C) | 1.05(GB)(15%) | 0.9(12.87%) | 0.87(12.4%) |
| Data Loaded(U) | 1.43(GB)(11.1%) | 1.38(10.72%) | 1.36(10.6%) |
| Page Size(C) | 208904(bytes) | 270344 | 331784 |
| Page Size(U) | 122888(bytes) | 184328 | 245768 |

TABLE II
TOTAL DATA LOADED AND MAIN MEMORY PAGE SIZES FOR EACH RUN.

## X. RESULTS

Our test machine is a 2 Ghz Pentium IV 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 from Lawrence Livermore National Lab. The simulation is of a shock passing through two gases of different densities and equal pressures initially separated by a membrane. In the simulation, the membrane is modeled as a contact discontinuity. As the shock passes through the gases, the mixing becomes more turbulent and the surfaces describing the mixing interface become very complex. As described in Section I, the full resolution simulation contains 27,000 time steps and the output consists of only 274 time steps. Compared to the actual simulation, our test dataset has a much coarser temporal resolution and thus reduced coherence between time steps. The coarse temporal resolution decreases the amount of temporal coherence that can be exploited for compression and during mesh refinement.

Our test dataset is 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 IX-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 13 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 disappear. Figures 14 and 15 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 16 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 $t = 0$ 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 loaded 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 $2^{12}$

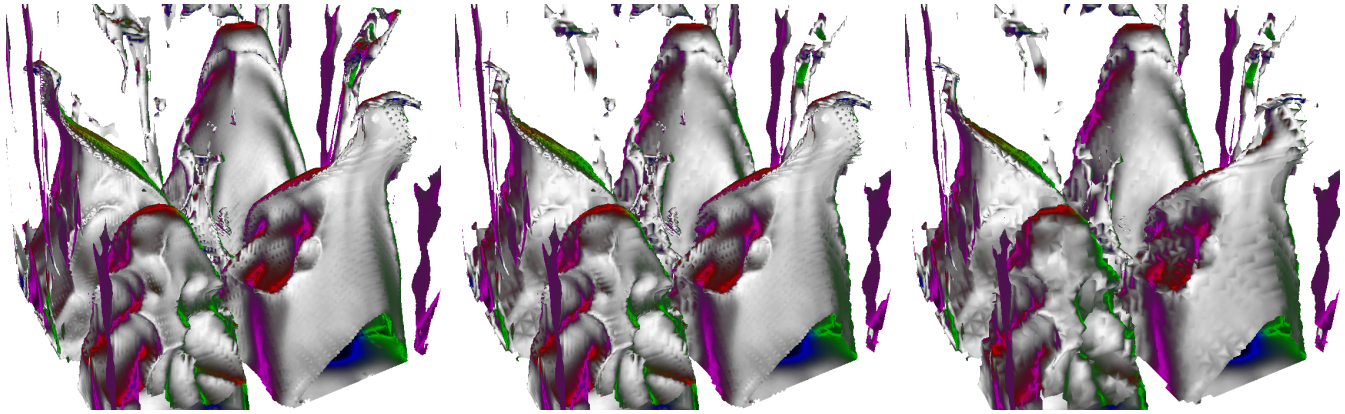

Fig. 13. Error based rendering with fixed viewpoint at time step 260, isovalue 223.5. Left to right (Error,Triangles): (0.78,1.11M), (1.2,420K), (1.8,235K).

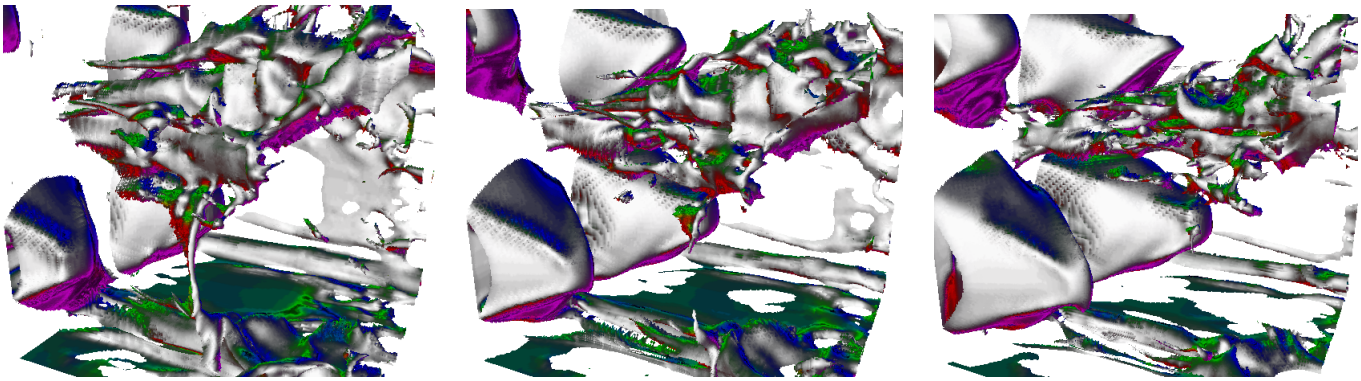

Fig. 14. Left to right: Isosurfaces extracted from time steps 160, 180, and 200 at isovalue 227.9, error 0.5.

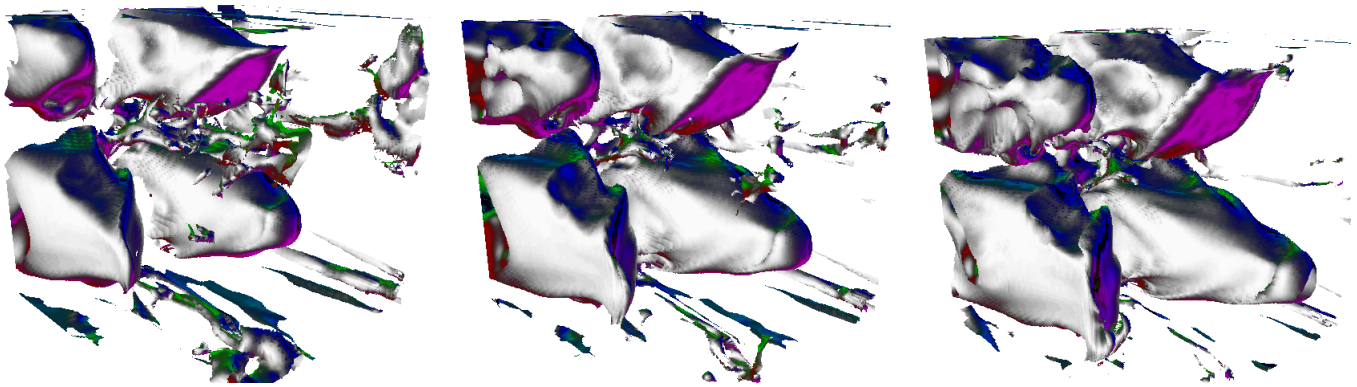

Fig. 15. Left to right: Isosurfaces extracted from time steps 223, 248, and 273 at isovalue 227.9, error 0.85.

or 4096 elements with 3 bytes per element for a total of 12288 bytes. In our tests, we use load 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. Table II summarizes 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. Figure 17 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 indicates that the data storage scheme, which balances spatial

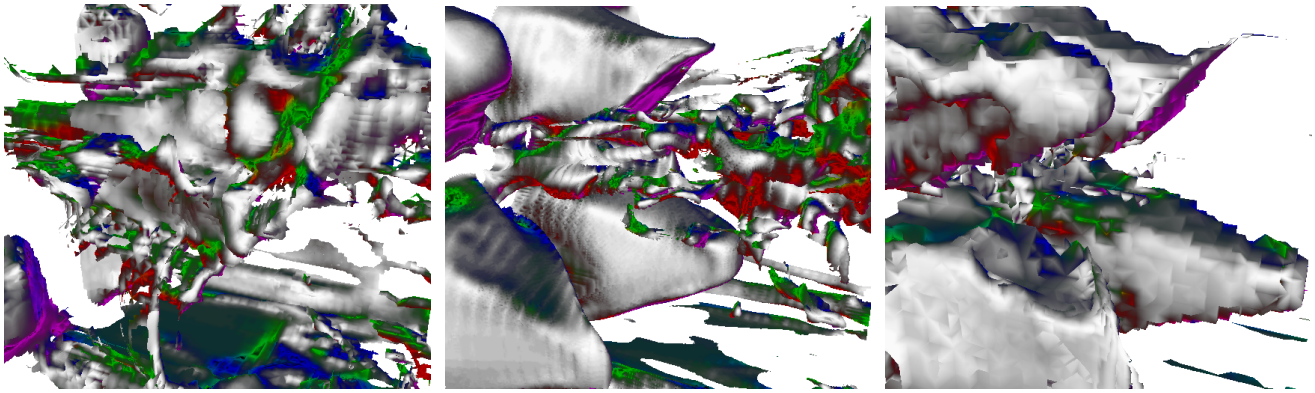

Fig. 16. Error based refinement over time, isovalue = 225. Left to right (Time step,Error,Triangles): (150,1.5,220K), (211,0.5,850K), (270,2.7,51K).

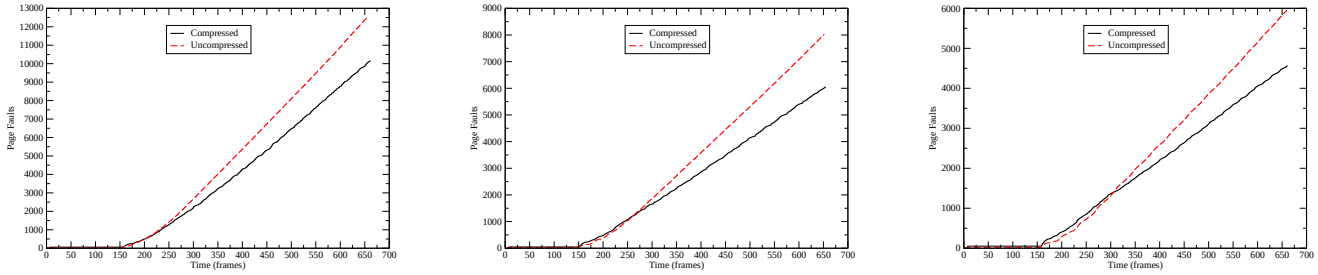


Fig. 17. Page faults over time for compressed and uncompressed volumes. Left to right: disk read sizes of 10, 15, and 20 uncompressed time steps.

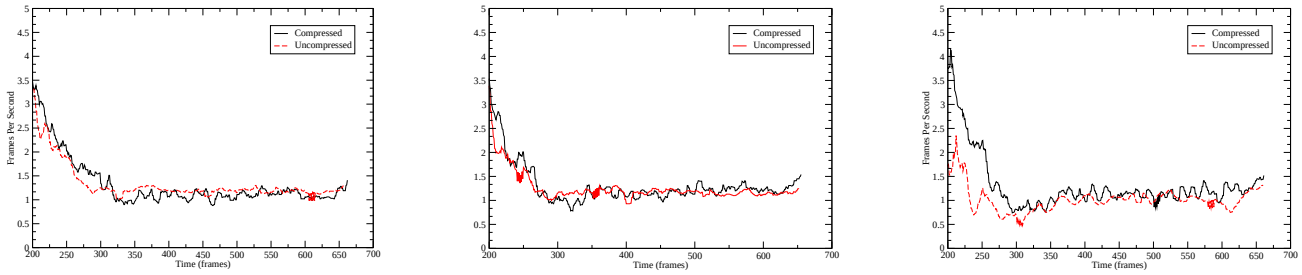


Fig. 18. Frame rate over time for compressed and uncompressed volumes. Left to right: disk read sizes of 10, 15, and 20 uncompressed time steps.

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. The frame rates over time for the compressed and uncompressed volumes are shown Figure 18. 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 III 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 $570 \times 530$ 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%

| 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 III
TABLE OF HARDWARE AND SOFTWARE TRIANGLE EXTRACTION RATES.

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 performance benefit decreases as more data must be accessed from memory and 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 reduces the 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 16–18 million textured triangles a second. When hardware interpolations are used, the rendering performance is 10–12 million triangles a second.

## XI. 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] Udeepta D. Bordoloi and Han-Wei Shen. Space efficient fast isosurface extraction for large datasets. In *Proceedings of IEEE Visualization 2003*, 2003.

[2] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *Proceedings of IEEE Visualization 2003*, pages 147–154, 2003.

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

[4] 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.

[5] 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.

[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] R. Grosso and G. Soza. Real-time exploration of scalar data on multilevel meshes. In *Proceedings of VMV 2002*, 2002.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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).

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. In *The Visual Computer*, pages 100–111, 1999.

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

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

[23] 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.