An Adaptive Framework for Visualizing Unstructured Grids with Time-Varying Scalar Fields

Fábio F. Bernardon [a,*], Steven P. Callahan [b], João L. D. Comba [a], Cláudio T. Silva [b]

[a] *Instituto de Informática, Federal University of Rio Grande do Sul, Brazil*
[b] *Scientific Computing and Imaging Institute, University of Utah*

Abstract

Interactive visualization of time-varying volume data is essential for many scientific simulations. This is a challenging problem since this data is often large, can be organized in different formats (regular or irregular grids), with variable instances of time (from hundreds to thousands) and variable domain fields. It is common to consider subsets of this problem, such as time-varying scalar fields (TVSFs) on static structured grids, which are suitable for compression using multi-resolution techniques and can be efficiently rendered using texture-mapping hardware. In this work we propose a rendering system that considers unstructured grids, which do not have the same regular properties crucial to compression and rendering. Our solution simultaneously leverages multiple processors and the graphics processing unit (GPU) to perform decompression, level-of-detail selection, and volume visualization of dynamic data. The resulting framework is general enough to adaptively handle visualization tasks such as direct volume rendering and isosurfacing while allowing the user to control the speed and quality of the animation.

Key words: volume rendering, unstructured grids, time-varying data

* Corresponding author.
Email addresses: fabiofb@inf.ufrgs.br (Fábio F. Bernardon), stevec@sci.utah.edu (Steven P. Callahan), comba@inf.ufrgs.br (João L. D. Comba), csilva@cs.utah.edu (Cláudio T. Silva).

1 Introduction

Advances in computational power are enabling the creation of increasingly sophisticated simulations generating vast amounts of data. Effective analysis of these large datasets is a growing challenge for scientists who must validate that their numerical codes faithfully represent reality. Data exploration through visualization offers powerful insights into the reliability and the limitations of simulation results, and fosters the effective use of results by non-modelers. Measurements of the real world using high-precision equipment also produces enormous amounts of information that requires fast and precise processing techniques that produce an intuitive result to the user.

However, at this time, there is a mismatch between the simulation and acquiring capabilities of existing systems, which are often based on high-resolution time-varying 3D unstructured grids, and the availability of visualization techniques. In a recent survey article on the topic, Ma [1] says:

> "Research so far in time-varying volume data visualization has primarily addressed the problems of encoding and rendering a single scalar variable on a regular grid. ... Time-varying unstructured grid data sets has been either rendered in a brute force fashion or just resampled and downsampled onto a regular grid for further visualization calculations. ..."

One of the key problems in handling time-varying data is the raw size of the data that must be processed. For rendering, these datasets need to be stored (and/or staged) in either main memory or GPU memory. Data transfer rates create a bottleneck for the effective visualization of these datasets. A number of successful techniques for time-varying regular grids have used compression to mitigate this problem, and allow for better use of resources. Most solutions described in the literature consider only structured grids, where exploiting coherence (either spatial or temporal) is easier due to the regular structure of the datasets. For unstructured grids, however, the compression is more challenging and several issues need to be addressed.

There are four fundamental pieces to adaptively volume render dynamic data. First, compression of the dynamic data for efficient storage is necessary to avoid exhausting available resources. Second, handling the data transfer of the compressed data is important to maintain interactivity. Third, efficient volume visualization solutions are necessary to provide high-quality images that lead to scientific insight. Furthermore, the framework has to be flexible enough to support multiple visualization techniques as well as data that changes at each frame. Fourth, maintaining a desired level of interactivity or allowing the user to change the speed of the animation is important for the user experience. Therefore, level-of-detail approaches that generally work on static datasets

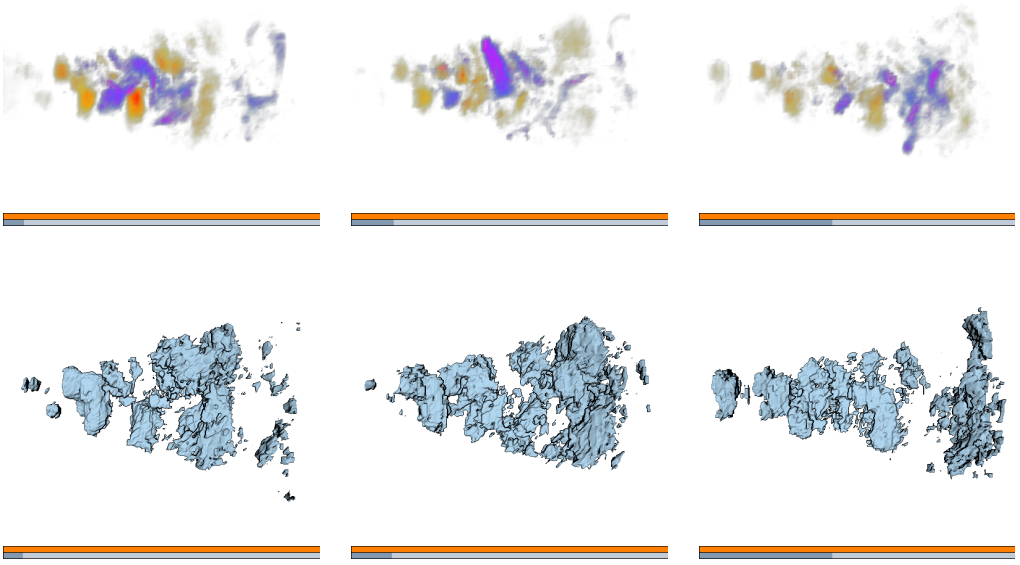

Fig. 1. Different time instances of the Turbulent Jet dataset consisting of one million tetrahedra and rendered at approximately six frames per second using direct volume rendering (top) and isosurfacing (bottom). Our user interface consists of an adjustable orange slider representing the level-of-detail and an adjustable gray slider representing the current time instance.

must be adapted to efficiently handle the dynamic case.

Since multiple CPUs and programmable GPUs are becoming common for desktop machines, we concentrate on efficiently using all the available resources. Our system performs decompression, object-space sorting, and level-of-detail operations with multiple threads on the CPUs for the next time-step while simultaneously rendering the current time-step using the GPU. This parallel computation results in only a small overhead for rendering time-varying data over previous static approaches.

To demonstrate the flexibility of our framework, we describe how the two most common visualization techniques for unstructured grids can be easily integrated. Both direct volume rendering as well as isosurface computation are incorporated into our adaptive, time-varying framework.

Though our goal is to eventually handle data that changes in geometry and even topology over time, here we concentrate on the more specific case of time-varying scalar fields on static geometry and topology. The main contributions of this paper are:

- We show how the data transfer bottleneck can be mitigated with compression of time-varying scalar fields for unstructured grids;

- We show how a hardware-assisted volume rendering system can be enhanced to efficiently prefetch dynamic data by balancing the CPU and GPU loads;
- We describe how direct volume rendering and isosurfacing can be incorporated into our adaptive framework;
- We introduce new importance sampling approaches for dynamic level-of-detail that operate on time-varying scalar fields.

Figure 1 illustrates our system in action on an unstructured grid representation of the Turbulent Jet dataset. The rest of this paper is organized as follows. Section 2 surveys related previous work. Section 3 outlines our system for adaptively volume rendering unstructured grids with time-varying scalar fields. The results of our algorithm are shown in Section 4, a brief discussion follows in Section 5, and conclusions and future work are described in Section 6.

2 Previous Work

The visualization of time-varying data is of obvious importance, and has been the source of substantial research. Here, we are particularly interested in the research literature related to compression and rendering techniques for this kind of data. For a more comprehensive review of the literature, we point the interested reader to the recent surveys by Ma [1] and Ma and Lum [2].

Very little has been done for compressing time-varying data on unstructured grids, therefore all the papers cited below focus on regular grids. Some researchers have explored the use of spatial data structures for optimizing the rendering of time-varying datasets [3–5]. The Time-Space Partitioning (TSP) Tree used in those papers is based on an octree which is extended to encode one extra dimension by storing a binary tree at each node that represents the evolution of the subtree through time [5]. The TSP tree can also store partial sub-images to accelerate rendering by ray-casting.

The compression of time-varying isosurfaces and associated volumetric data with a wavelet transform was first proposed in [6]. With the advance of texture-based volume rendering and programmable GPUs, several techniques explored shifting data storage and decompression into graphics hardware. Coupling wavelet compression of structured grids with decompression using texturing hardware was discussed in work by Guthe *et al.* [7, 8]. They describe how to encode large, static datasets or time-varying datasets to minimize their size, thus reducing data transfer and allowing real-time volume visualization. Subsequent work by Strengert *et al.* [9] extended the previous approaches by employing a distributed rendering strategy on a GPU cluster. Lum *et al.* [10, 11] compress time-varying volumes using the Discrete Cosine Transform (DCT).

Because the compressed datasets fit in main memory, they are able to achieve much higher rendering rates than for the uncompressed data, which needs to be incrementally loaded from disk. Because of their sheer size, I/O issues become very important when dealing with time-varying data [12].

More related to our work is the technique proposed by Schneider *et al.* [13]. Their approach relies on vector quantization to select the best representatives among the difference vectors obtained after applying a hierarchical decomposition of structured grids. Representatives are stored in textures and decompressed using fragment programs on the GPU. Since the multi-resolution representations are applied to a single structured grid, different quantization and compression tables are required for each time instance. Issues regarding the quality of rendering from compressed data were discussed by Fout *et al.* [14] using the approach described by Schneider *et al.* as a test case.

Hardware-assisted volume rendering has received considerable attention in the research community in recent years (for a recent survey, see [15]). Shirley and Tuchman [16] introduced the Projected Tetrahedra (PT) algorithm for decomposing tetrahedra into renderable triangles based on the view direction. A visibility ordering of the tetrahedra before decomposition and rendering is necessary for correct transparency compositing [17]. More recently, Weiler *et al.* [18] describe a hardware ray caster which marches through the tetrahedra on the GPU in several rendering passes. Both of these algorithms require neighbor information of the mesh to correctly traverse the mesh for visibility ordering or ray marching, respectively. An alternative approach was introduced by Callahan *et al.* [19] which considers the tetrahedral mesh as unique triangles that can be efficiently rendered without neighbor information. This algorithm is ideal for dynamic data because the vertices, scalars, or triangles can change with each frame with very little penalty. Subsequent work by Callahan *et al.* [20] describes a dynamic level-of-detail (LOD) approach that works by sampling the geometry and rendering only a subset of the original mesh. Our system uses a similar approach for LOD, but adapted to handle time-varying data.

Graphics hardware has also been used in recent research to visualize isosurfaces of volumetric data. The classic Marching Cubes algorithm [21] and subsequent Marching Tetrahedra algorithm [22] provide a simple way to extract geometry from structured and unstructured meshes in object space. However, image-space techniques are generally more suitable for graphics hardware. One such algorithm was proposed by Sutherland *et al.* [23] which uses alpha-test functionality and a stencil buffer to perform plane-tetrahedron intersections. Isosurfaces are computed by interpolating alpha between front and back faces and using XOR operation with the stencil buffer. Later, Röttger *et al.* [24] revised this algorithm and extended the PT algorithm to perform isosurfacing. Instead of breaking up the tetrahedra into triangles, the algorithm creates

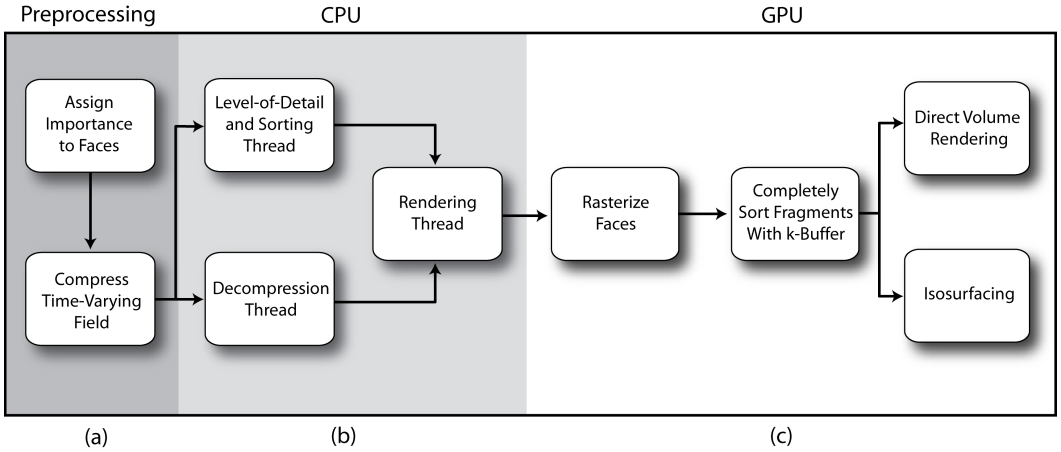

Fig. 2. An overview of our system. (a) Data compression and importance sampling for level-of-detail are performed in preprocessing steps on the CPU. (b) Then during each pass, level-of-detail selection, optional object-space sorting, and data decompression for the next step occur in parallel on multiple CPUs. (c) Simultaneously, the image-space sorting and volume visualization are processed on the GPU.

smaller tetrahedra which can be projected as triangles and tested for isosurface intersection using a 2D texture. More recent work by Pascucci [25] uses programmable hardware to compute isosurfaces. The algorithm considers each tetrahedron as a quadrilateral, which is sent to the GPU where the vertices are repositioned in a vertex shader to represent the isosurface. Our system takes advantage of recent hardware features to perform isosurface tests directly at the fragment level. Furthermore, our isosurfacing can be used directly in our time-varying framework and naturally benefit from our dynamic level-of-detail.

Several systems have previously taken advantage of multiple processors and multi-threading to increase performance. The iWalk system [26] by Correa *et al.* uses multiple threads to prefetch geometry and manage a working set of renderable primitives for large triangles scenes. More recent work by Vo *et al.* [27] extends this idea to perform out-of-core management and rendering of large volumetric meshes. Our approach is similar in spirit to these algorithms. We use multiple threads to prefetch changing scalar values, sort geometry, and prepare level-of-detail triangles at each frame.

3 Adaptive Time-Varying Volume Rendering

Our system for adaptive time-varying volume rendering of unstructured grids consists of four major components: compression, data transfer, hardware-assisted volume visualization, and dynamic level-of-detail for interactivity. Figure 2 shows the interplay of these components.

Desktop machines with multiple processors and multiple cores are becoming increasingly typical. Thus, interactive systems should explore these features to leverage computational power. For graphics-heavy applications, the CPU cores can be used as an additional cache for the GPU, which may have limited memory capabilities. With this in mind, we have parallelized the operations on the CPU and GPU with multi-threading to achieve interactive visualization.

There are currently three threads in our system to parallelize the CPU portion of the code. These threads run while the geometry from the previous time-step is being rendered on the GPU. Therefore, the threads can be considered a prefetching of data in preparation for rasterization. The first thread handles decompression of the compressed scalar field (see Section 3.1). The second thread handles object-space sorting of the geometry (see Section 3.3). Finally, the third thread is responsible for rendering. All calls to the graphics API are done entirely in the rendering thread since only one thread at a time may access the API. This thread waits for the first two threads to finish, then copies the decompressed scalar values and sorted indices before rendering. This frees the other threads to compute the scalars and indices for the next time-step. These three threads work entirely in parallel and result in a time-varying visualization that requires very little overhead over static approaches.

3.1 Compression

Compression is important to reduce the memory footprint of time-varying data, and the consideration of spatial and temporal coherence of the data is necessary when choosing a strategy. For example, it is common to exploit spatial coherence in time-varying scalar fields defined on structured grids, such as in the approach described by Schneider *et al.* [13], where a multi-resolution representation of the spatial domain using vector quantization is used. This solution works well when combined with texture-based volume rendering, which requires the decompression to be performed at any given point inside the volume by incorporating the differences at each resolution level.

In unstructured grids, the irregularity of topological and geometric information makes it hard to apply a multi-resolution representation over the spatial domain. In our system we apply compression on the temporal domain by considering scalar values individually for each mesh vertex. By grouping a fixed number of scalar values defined over a sequence of time instances we obtain a suitable representation for applying a multi-resolution framework.

Our solution collects blocks of 64 consecutive scalars associated with each mesh vertex, applies a multi-resolution representation that computes the mean of

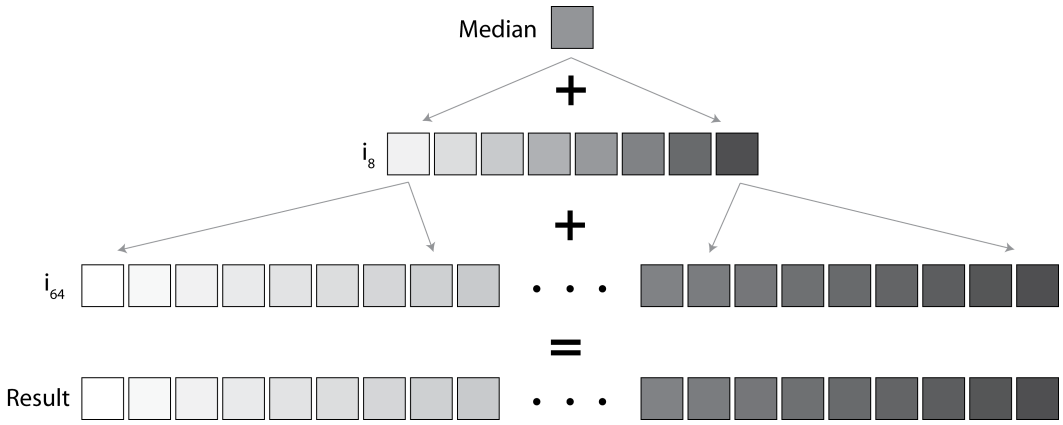

Fig. 3. Decompression of a given time instance for each mesh vertex requires adding the scalar mean of a block to the quantized differences recovered from the respective entries (i_8 and i_{64}) in the codebooks.

each block along with two difference vectors of size 64 and 8, and uses vector quantization to obtain two sets of representatives (codebooks) for each class of difference vectors. For convenience we use a fixed number of time instances, but compensate for this by increasing the number of entries in the codebooks if temporal coherence is reduced and leads to compression errors.

This solution works well for projective volume rendering that sends mesh faces in visibility ordering to be rendered. At each rendering step, the scalar value for each mesh vertex in a given time instance is decompressed by adding the mean of a given block interval to two difference values, which are recovered from the codebooks using two codebook indices i_8 and i_{64} (Figure 3).

3.2 Data Transfer

There are several alternatives to consider when decompressing and transferring time-varying data to the volume renderer. This is a critical point in our system and has a great impact on its overall performance. In this section we discuss the alternatives we explored and present our current solution. It is important to point out that with future CPU and GPU configurations this solution might need to be revisited.

Since the decompression is done on a per-vertex basis, our first approach was to use the vertex shader on the GPU. This requires the storage of codebooks as vertex textures, and the transfer for each vertex of three values as texture coordinates (mean and codebook indices i_8 and i_{64}). In practice, this solution does not work well because the current generation of graphics hardware does not handle vertex textures efficiently and incurs several penalties due to cache misses, and the arithmetic calculations in the decompression are too simple

to hide this latency.

Our second approach was to use the GPU fragment shader. Since computation is done at a fragment level, the decompression and the interpolation of the scalar value for the fragment is necessary. This requires three decompression steps instead of a single step as with the vertex shader approach (which benefits from the interpolation hardware). Also, this computation requires accessing the mean and codebook indices. Sending this information as a single vertex attribute is not possible due to interpolation, and multiple-vertex attributes increase the amount of data transfer per vertex. As our volume renderer runs in the fragment shader, this solution also increases the shader complexity and thus reduces performance of the system.

Our final (and current) solution is to perform the decompression on the CPU. Since codebooks usually fit in CPU memory—a simple paging mechanism can be used for really large data—the main cost of this approach is to perform the decompression step and send scalar values through the pipeline. This data transfer is also necessary with the other two approaches. The number of decompression steps is reduced to the number of vertices, unlike the vertex shader approach which requires three times the number of faces.

3.3 Volume Rendering

Our system is based on the Hardware-Assisted Visibility Sorting[1] (HAVS) algorithm of Callahan *et al.* [19]. Figure 2 shows how the volume rendering system handles time-varying data. Our framework supports both direct volume rendering as well as isosurfacing.

The HAVS algorithm is a general visibility ordering algorithm for renderable primitives that works in both object-space and image-space. In object space the unique triangles that compose the tetrahedral mesh are sorted approximately by their centroids. This step occurs entirely on the CPU. In image-space, the triangles are sorted and composited in correct visibility order using a fixed size A-buffer called the k-buffer. The k-buffer is implemented entirely on the GPU using fragment shaders. Because the HAVS algorithm operates on triangles with no need for neighbor information, it provides a flexible framework for handling dynamic data. In this case, the triangles can be stored on the GPU for efficiency, and the scalar values as well as the object-space ordering of the triangles can be streamed to the GPU at each time instance.

Our algorithm extends the HAVS algorithm with time-varying data with virtually no overhead by taking advantage of the HAVS architecture. Since work

[1] Source code available at http://havs.sourceforge.net

performed on the CPU can be performed simultaneously to work on the GPU, we can leverage this parallelization to prefetch the time-varying data. During the GPU rendering stage of the current time instance, we use the CPU to decompress the time-varying field of the next time-step and prepare it for rendering. We also distinguish user provided viewing transformations that affect visibility order from those that do not and perform visibility ordering only when necessary. Therefore, the object-space centroid sort only occurs on the CPU during frames that have a change in the rotation transformation. This avoids unnecessary computation when viewing time-varying data.

To manage the time-stepping of the time-varying data, our algorithm automatically increments the time instance at each frame. To allow more control from the user, we also provide a slider for interactive exploration of the time instances.

3.3.1 Direct Volume Rendering

The HAVS algorithm performs direct volume rendering with the use of pre-integration [28]. In a preprocess, a three-dimensional lookup table is computed that contains the color and opacity for every set of scalars and distances between them (s_f, s_b, d). Then, as fragments are rasterized, the k-buffer is used to retrieve the closest two fragments and the front scalar s_f, back scalar s_b, and distance d between them is calculated. The color and opacity for the gap between the fragments is determined with a texture lookup, then composited into the framebuffer. More details on direct volume rendering with the HAVS algorithm can be found in [19].

3.3.2 Isosurfaces

We extend the HAVS algorithm to perform isosurfacing in our time-varying framework. The fragments are sorted using the k-buffer as described above. However, instead of compositing the integral contribution for the gap between the first and second fragments, we perform a simple test to determine if the isosurface value lies between them. If so, the isosurface depth is determined by interpolating between the depths of the two fragments. The result is a texture that contains the depth for the isosurface at each pixel (*i.e.*, a depth buffer), which can be displayed directly as a color buffer or post-processed to include shading.

There are several differences between our isosurfacing using HAVS and previous hardware-assisted approaches. First, with the advent of predicates in the fragment shader, a direct isosurface comparison can be performed efficiently, without the need of texture lookups as in previous work by Röttger *et al.* [24]. Second, the k-buffer naturally provides the means to handle multiple transpar-

ent isosurfaces by compositing the isosurface fragments in the order that they are extracted. Third, to handle lighting of the isosurfaces, we avoid keeping normals in our k-buffer by using an extra shading pass. One option is a simple depth-based shading [29], which may not give sufficient detail of the true nature of the surface. Another option is to use a gradient-free shading approach similar to work by Desgranges *et al.* [30], which uses a preliminary pass over the geometry to compute a shadow buffer. Instead, we avoid this extra geometry pass by using screen-space shading of the computed isosurface through central differencing on the depth buffer [31]. This results in fast and high quality shading. The isosurface figures shown in this paper were generated with the latter shading model. Finally, since our isosurfacing algorithm is based on our direct volume rendering algorithm, the same level-of-detail strategies can be used to increase performance. However, level-of-detail rendering may introduce discontinuities in the depths that define the isosurface, adversely affecting the quality of the image-space shading. Thus, we perform an additional smoothing pass on the depths using a convolution filter, which removes sharp discontinuities, before the final shading. This extra pass is inexpensive and typically only necessary at low levels-of-detail.

3.4 Time-Varying Level-of-Detail

Recent work by Callahan *et al.* [20] introduces a new dynamic level-of-detail (LOD) approach that works by using a *sample-based simplification* of the geometry. This algorithm operates by assigning an importance to each triangle in the mesh in a preprocessing step based on properties of the original geometry. Then, for each pass of the volume renderer, a subset of the original geometry is selected for rendering based on the frame rate of the previous pass. This recent LOD strategy was incorporated into the original HAVS algorithm to provide a more interactive user experience.

An important consideration for visualizing time-varying data is the rate at which the data is progressing through the time instances. To address this problem, our algorithm uses this LOD approach to allow the user to control the speed and quality of the animation. Since we are dealing with time-varying scalar fields, heuristics that attempt to optimize the quality of the mesh based on the scalar field are ideal. However, approaches that are based on a static mesh can be poor approximations when considering a dynamically changing scalar field.

Callahan *et al.* introduce a heuristic based on the scalar field of a static mesh for assigning an importance to the triangles. The idea is to create a scalar histogram and use stratified sampling to stochastically select the triangles that cover the entire range of scalars. This approach works well for static geome-

try, but may miss important regions if applied to a time-step that does not represent the whole time-series well. Recent work by Akiba *et al.* [32] classifies time-varying datasets as either *statistically dynamic* or *statistically static* depending on the behavior of the time histogram of the scalars. A statistically dynamic dataset may have scalars that change dramatically in some regions of time, but remain constant during others. In contrast, the scalars in statistically static datasets change consistently throughout the time-series. Because datasets vary differently, we have developed two sampling strategies for dynamic LOD: a local approach for statistically dynamic datasets and a global approach for statistically static datasets.

To incorporate these LOD strategies into our time-varying system, we allow two types of interactions based on user preference. The first is to keep the animation at a desired frame-rate independent of the data size or viewing interaction. This dynamic approach adjusts the LOD on the fly to maintain interactivity. Our second type of interaction allows the user to use a slider to control the LOD. This slider dynamically changes the speed of the animation by setting the LOD manually. Since visibility ordering-dependent viewing transformations occur on the CPU in parallel to the GPU rendering, they do not change the LOD or speed of the animation. Figure 2 shows the interaction of the LOD algorithm with the time-varying data.

3.4.1 Local Sampling

Datasets with scalars that vary substantially in some time regions, but very little in others benefit from a local sampling approach. The general idea is to apply methods for static datasets to multiple time-steps of a time-varying dataset and change the sampling strategy to correspond to the current region. We perform local sampling by selecting time-steps at regular intervals in the time-series and importance sampling the triangles in those time-steps using the previously described stochastic sampling of scalars. Since the LOD selects a subset of triangles ordered by importance, we create a separate list of triangles for each partition of the time sequence. Then, during each rendering step, the region is determined and the corresponding list is used for rendering. Since changing the list also requires the triangles to be resorted for rendering, we perform this operation in the sorting thread to minimize delays between frames.

Statistically dynamic datasets benefit from this approach because the triangles are locally optimized for rendering. The disadvantage of this approach is that because the active set of renderable triangles may change between time intervals, flickering may occur. However, this is a minor issue when using dynamic LOD because the number of triangles drawn at each frame may be changing anyway, to maintain interactivity.

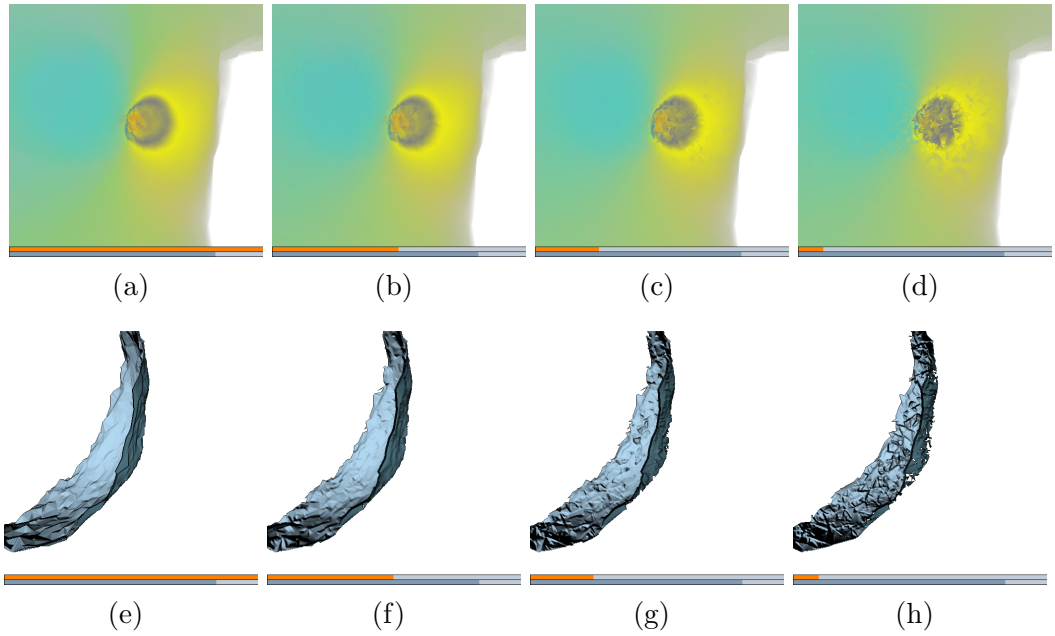

Fig. 4. Time-varying level-of-detail(LOD) strategy using the coefficient of variance for the Torso dataset (50K tetrahedra and 360 time steps). For a close-up view using direct volume rendering: (a) 100% LOD at 18 fps, (b) 50% LOD at 40 fps, (c) 25% LOD at 63 fps, and (d) 10% LOD at 125 fps. For the same view using isosurfacing: (e) 100% LOD at 33 fps, (f) 50% LOD at 63 fps, (g) 25% LOD at 125 fps, and (h) 10% LOD at 250 fps. Front isosurface fragments are shown in light blue and back fragments are shown in dark blue.

Certain datasets may contain regions in time where none of the scalars are changing and other regions where many scalars are changing. These datasets would benefit from a non-linear partitioning of the time sequence (*i.e.*, logarithmic). A more automatic approach to partitioning the time-steps is to greedily select areas with the highest scalar variance to ensure that important changes are not missed. In our experiments, a regular interval is used because our experimental datasets do not fall into this category.

3.4.2 Global Sampling

A global strategy is desirable in datasets that are statistically static due to its simplicity and efficiency. For global sampling, one ordering is determined that attempts to optimize all time-steps. This has the advantage that it does not require changing the triangle ordering between frames and thus gives a smoother appearance during an animation.

A sampling can be obtained globally using a statistical measure of the scalars. For n time-steps, consider the n scalar values s for one vertex as an independent random variable X, then the expectation at that position can be expressed as

$$E[X] = \sum_{1}^{n} s(Pr\{X = s\}),$$

where $Pr\{X = s\} = 1/n$. The dispersion of the probability distribution of the scalars at a vertex can then be expressed as the variance of the expectation:

$$\begin{aligned} Var[X] &= E[X^2] - E^2[X] \\ &= \sum_{1}^{n} (\frac{s^2}{n}) - (\sum_{1}^{n} \frac{s}{n})^2 \end{aligned}$$

In essence, this gives a *spread* of the scalars from their expectation. To measure dispersion of probability distributions with widely differing means, it is common to use the coefficient of variation C_v, which is the ratio of the standard deviation to the expectation. This metric has been used in related research for transfer function specification on time-varying data [32, 33] and as a measurement for spatial and temporal error in a time-varying structured grids [5]. Thus, for each triangle t, the importance can be assigned by calculating the sum of the C_v for each vertex as follows:

$$C_v(t) = \sum_{i=1}^{3} \frac{\sqrt{Var[X_{t(i)}]}}{E[X_{t(i)}]}$$

This results in a dimensionless quantity that can be used for assigning importance to each face by directly comparing the amount of change that occurs at each triangle over time.

The algorithm provides good quality visualizations even at lower levels-of-detail because the regions of interest (those that are changing) are given a higher importance (see Figure 4). The described heuristic works especially well in statistically static datasets if the mesh has regions that change very little over time since they are usually assigned a lower opacity and their removal introduces very little visual difference.

4 Results

In this section we report results obtained using a PC with Pentium D 3.2 GHz Processors, 2 GB of RAM, and an NVidia 7800 GTX GPU with 256 MB RAM. All images were generated at 512×512 resolution.

Table 1
Results of compression sizes, ratios, and error.

Mesh	Num Verts	Num Tets	Time Instances	Size TVSF	Size VQ	Comp. Ratio	SNR Min	SNR Max	Max Error
SPX1	36K	101K	64	9.0M	504K	18.3	39.5	42.0	0.005
SPX2	162K	808K	64	40.5M	2.0M	20.6	39.2	42.0	0.009
SPXF	19K	12K	192	14.7M	2.0M	7.1	20.8	30.2	0.014
Blunt	40K	183K	64	10.0M	552K	18.6	41.7	44.4	0.005
Torso	8K	50K	360	11.2M	1.0M	11.4	20.5	28.1	0.002
TJet	160K	1M	150	93.8 M	2.7M	34.7	5.3	17.9	0.204

4.1 Datasets

The datasets used in our tests are diverse in size and number of time instances. The time-varying scalars on the SPX1, SPX2 and Blunt Fin datasets were procedurally generated by linear interpolating the original scalars to zero over time. The Torso dataset shows the result of a simulation of a rotating dipole in the mesh. The SPX-Force (SPXF) dataset represents the magnitude of reaction forces obtained when a vertical force is applied to a mass-spring model that has as particles the mesh vertices and as springs the edges between mesh vertices. Finally, the Turbulent Jet (TJet) dataset represents a regular time-varying dataset that was tetrahedralized and simplified to a reduced representation of the original. The meshes used in our tests with their respective sizes are listed in Table 1 and results showing different time instances are shown in Figures 1, 4, 5, and 6.

4.2 Compression

The compression of Time-Varying Scalar Field (TVSF) data uses an adaption of the vector quantization code written by Schneider *et al.* [13], as described in Section 3.1. The original code works with structured grids with building blocks of $4 \times 4 \times 4$ (for a total of 64 values per block). To adapt its use for unstructured grids it is necessary to group TVSF data into basic blocks with the same amount of values. For each vertex in the unstructured grid, the scalar values corresponding to 64 contiguous instances of time are grouped into a basic block and sent to the VQ code.

The VQ code produced two codebooks containing difference vectors for the first and second level in the multi-resolution representation, each with 256 entries (64×256 and 8×256 codebooks). For our synthetic datasets this configuration led to acceptable compression results as seen on Table 1. However, for the TJet and SPXF datasets we increased the number of entries in the codebook due to the compression error obtained. Both datasets were

compressed using codebooks with 1024 entries.

The size of TVSF data without compression is given by $size_u = v \times t \times 4B$, where v is the number of mesh vertices, t is the number of time instances in each dataset, and each scalar uses four bytes (float). The compressed size using VQ is equal to $size_{vq} = v \times c \times 3 \times 4B \ + \ c \times size\_codebook$, where c is the number of codebooks used ($c = t/64$), s is the number of entries in the codebook (256 or 1024), each vertex requires 3 values per codebook (mean plus codebook indices i_8 and i_{64}), and each codebook size corresponds to $s \times 64 \times 4B \ + \ s \times 8 \times 4B$.

In Table 1 we summarize the compression results we obtained. In addition to the signal-to-noise ration (SNR) given by the VQ code, we also measured the minimum and maximum discrepancy between the original and quantized values. Results show that procedurally generated datasets have a higher SNR and smaller discrepancy, since they have a smoother variation in their scalars over time. The TJet dataset has smaller SNR values because it represents a real fluid simulation, but it also led to higher compression ratios due to its fixed codebook sizes. In general, the quality of the compression corresponds with the variance in the scalars between steps. Thus, datasets with smoother transitions result in less compression error. A limitation of the current approach is that because it exploits temporal coherence, it may not be suitable for datasets with abrupt changes between time instances. In this case, compression methods that focus more on spatial coherence may be necessary.

4.3 Rendering

The rendering system allows the user to interactively inspect the time-varying data, continuously play all time instances, and pause or even manually select a given time instance by dragging a slider. Level-of-detail changes dynamically to achieve interactive frame rates, or can be manually set using a slider. Changing from direct volume rendering to isosurfacing is accomplished by pressing a key. Similarly, the isovalue can be interactively changed with the press of a key and incurs no performance overhead.

Rendering time statistics were produced using a fixed number of viewpoints. In Table 2 we show timing results for our experimental datasets. To compare the overhead of our system with the original HAVS system that handles only static data, we also measure the rendering rates for static scalar fields without multi-threading. The dynamic overhead is minimal even for the larger datasets. In fact, for some datasets, our multi-threading approach is faster with dynamic data than single threading with static data. Note that for the smaller datasets, we do not report frame-rates greater than 60 frames per second since it is

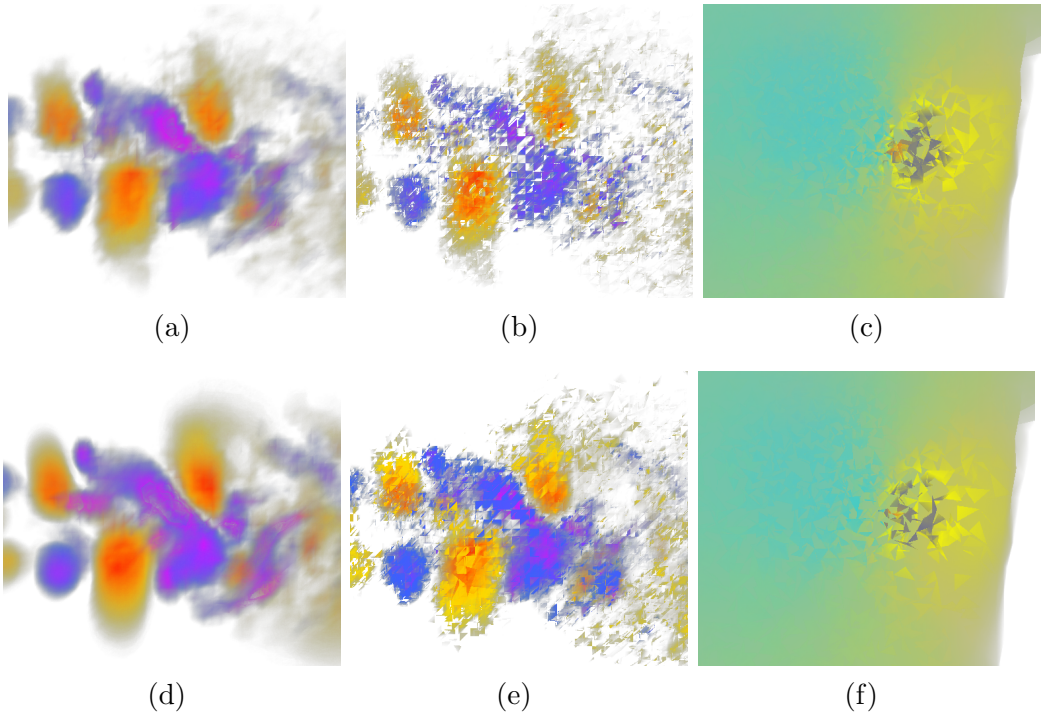

Fig. 5. Comparison of direct volume rendering using (a) uncompressed and (d) compressed data on the TJet dataset(1M tetrahedra, 150 time steps). Level-of-Detail is compared at 5% for the TJet and 3% for the Torso dataset(50K tetrahedra, 360 time steps) using (b)(c) our local approach and (e)(f) our global approach.

difficult to accurately measure higher rates.

In addition to the compression results described above, we evaluate the image quality for all datasets by comparing it against the rendering from uncompressed data. For most datasets the difference in image quality was minimal. However, for the TJet dataset (the one with the smaller SNR values), there are some small differences that can be observed in close-up views of the simulation (see Figure 5).

4.4 Level-of-Detail

Our sample-based level-of-detail for time-varying scalar fields computes the importance of the faces in a preprocessing step that takes less than two seconds for the global strategy or for the local strategy using six intervals, even for the largest dataset in our experiments. In addition, there is no noticeable overhead in adjusting the level-of-detail at a per frame basis because only the number of triangles in the current frame is computed [20]. Figure 4 shows the results of our global level-of-detail strategy on the Torso dataset at decreasing levels-of-detail. Figure 5 shows a comparison of our global and local sampling strategies. To capture an accurate comparison, the local sampling results are

Table 2
Performance measures for static and time-varying (TV) scalar fields for direct volume rendering (DVR) and isosurfacing (Iso). Static measurements were taken without multithreading. Performance is reported for each dataset with object-space sorting (during rotations) and without object-space sorting (otherwise).

Mesh	**Sort**	**DVR Static FPS**	**DVR TV FPS**	**DVR TV Tets/s**	**Iso Static FPS**	**Iso TV FPS**	**Iso TV Tets/s**
SPX1	Y	24.2	32	3.3M	26.4	28.2	2.9M
SPX1	N	42.6	41.7	4.3M	51.1	43.5	4.5M
SPX2	Y	2.8	2.9	2.4M	2.9	2.9	2.4M
SPX2	N	7.6	7.5	6.2M	8.2	8.2	6.8
SPXF	Y	>60	>60	0.7M	>60	>60	0.7M
SPXF	N	>60	>60	0.7M	>60	>60	0.7M
Blunt	Y	16.1	20.4	3.8M	15.9	19.5	3.6M
Blunt	N	25.6	27.5	5.2M	31.2	31.1	5.8M
Torso	Y	40.6	31.2	1.6M	44.8	31.2	1.6M
Torso	N	>60	>60	3.1M	>60	>60	3.1M
TJet	Y	2.3	2.1	2.1M	2.2	2.4	2.4M
TJet	N	6.1	5.8	5.8M	5.7	6	6.0M

shown in the middle of an interval, thus showing an average quality. In our experiments, the frame-rates increase at the same rate as the level-of-detail decreases (see Figure 4) for both strategies.

5 Discussion

This paper extends on our previously published work [34]. There are three main contributions that we present here that were not in the original paper. First, we describe a multi-threaded framework to improve performance and distribute computation between multiple cores of the CPU and the GPU. Second, we expand our description of time-varying level-of-detail to include both global and local approaches to more accurately handle a variety of data types. Third, we generalize our rendering algorithm to handle isosurfacing as well as direct volume rendering. This provides a fast time-varying solution to isosurfacing with level-of-detail and is important to show that our framework can handle the two most common visualization techniques for unstructured grids. Apart from these larger changes, many minor changes in the text, figures, and results were made to correspond with these new contributions.

An important consideration for the framework we have presented is the scalability of the solution on current and future hardware configurations. Development of faster processors is reaching physical limits that are expensive and difficult to overcome. This is leading the industry to shift from the pro-

duction of faster single-core processors to multi-core machines. On the other hand, graphics hardware has not yet reached these same physical limitations. Even so, hardware vendors are already providing multiple GPUs along with multiple CPUs. The use of parallel technology ensures that the processing power of commodity computers keeps growing independently of some physical limitations, but new applications must be developed considering this new reality to take full advantage of the new features. In particular, for data-intensive graphics applications, an efficient load balance needs to be maintained between resources to provide interactivity. In this paper, we focus on this multi-core configuration that is becoming increasingly popular on commodity machines instead of focusing on traditional CPU clusters.

To take advantage of multi-core machines with programmable graphics hardware, we separate the computation into three components controlled by different threads. For the largest dataset in our experiments, the computation time for the three components is distributed as follows: 2% decompression, 55% sorting, and 45% rendering. For the smaller datasets, the processing time shifts slightly more towards rendering. With more available resources, the rendering and the sorting phases could benefit from additional parallelization.

To further parallelize the sorting thread, the computation could be split amongst multiple cores on the CPU. This could be done in two ways. The first is to use a sort-first approach that performs parallel sorting on the geometry in screen space (see Gasarch *et al.* [35]), then pushes the entire geometry to the graphics hardware. The second is a sort-last approach that breaks the geometry into chunks that are sorted separately, and sent to the graphics hardware for rendering and compositing (see Vo *et al.* [27]).

Because of the parallel nature of modern GPUs, the vertex and fragment processing automatically occurs in parallel. Even so, multiple-GPU machines are becoming increasingly common. The effective use of this technology, however, is a complex task, especially for scientific computing. We have performed some tentatives tests of our framework on a computer with an NVidia SLI configuration. SLI, or Scalable Link Interface, is an NVidia technology developed to synchronize multiple GPUs (currently two or four) inside one computer. This technology was developed to automatically enhance the performance of graphics applications, and offer two new operation modes: Split Frame Rendering (SFR) and Alternate Frame Rendering (AFR). SFR splits the screen into multiple regions and assigns each region to a GPU. AFR renders every frame on a different GPU, cycling between all available resources. Our experiments with both SFR and AFR on a dual SLI machine did not improve performance with our volume rendering algorithm. The SFR method must perform k-buffer synchronization on the cards so frequently that no performance benefit is achieved. With AFR, the GPU selection is controlled by the driver and changes at the end of each rendering pass. With a multi-pass ren-

dering algorithm, this forces synchronization between GPUs and results in the same problem as SFR. In the future, as more control of these features becomes available, we will be able to take advantage of multiple-GPU machines to improve performance.

6 Conclusion

Rendering dynamic data is a challenging problem in volume visualization. In this paper we have shown how time-varying scalar fields on unstructured grids can be efficiently rendered using multiple visualization techniques with virtually no penalty in performance. In fact, for the larger datasets in our experiments, time-varying rendering only incurred a performance penalty of 6% or less. We have described how vector quantization can be employed with minimal error to mitigate the data transfer bottleneck while leveraging a GPU-assisted volume rendering system to achieve interactive rendering rates. Our algorithm exploits both the CPU and GPU concurrently to balance the computation load and avoid idle resources. In addition, we have introduced new time-varying approaches for dynamic level-of-detail that improve upon existing techniques for static data and allows the user to control the interactivity of the animation. Our algorithm is simple, easily implemented, and most importantly, it closes the gap between rendering time-varying data on structured and unstructured grids. To our knowledge this is the first system for handling time-varying data on unstructured grids in an interactive manner.

In the future, we plan to explore the VQ approach to find a general way of choosing its parameters based on dataset characteristics. Also, when next generation graphics cards become available, we would like to revisit our GPU solution to take advantage of new features. Finally, we would like to explore solutions for time-varying geometry and topology.

7 Acknowledgments

The authors thank J. Schneider for the VQ code, Louis Bavoil for the isosurfacing shaders, Mike Callahan and the SCIRun team at the University of Utah for the Torso dataset, Bruno Notrosso (Électricité de France) for the SPX dataset, Kwan-Liu Ma for the TJet dataset, Huy Vo for the tetrahedral simplification, and NVIDIA from donated hardware. The work of Fábio Bernardon and João Comba is supported by a CNPq grant 478445/2004-0. The work of Steven Callahan and Cláudio Silva has been supported by the National Science Foundation under grants CCF-0401498, EIA-0323604, OISE-0405402, IIS-0513692, and CCF-0528201, the Department of Energy, an IBM

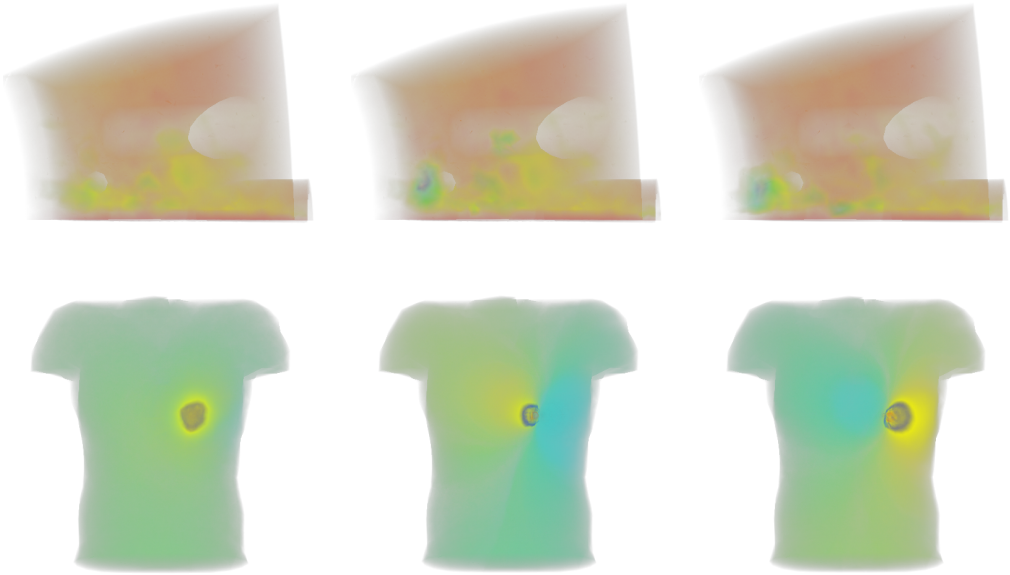

Fig. 6. Different time instances of the SPXF (above, 12K tetrahedra and 192 time steps) and Torso (below, 50K tetrahedra and 360 time steps) datasets volume rendered at full quality.

Faculty Award, the Army Research Office, and a University of Utah Seed Grant.

References

[1] K.-L. Ma, Visualizing Time-Varying Volume Data, Computing in Science & Engineering, 5(2), 2003, pp. 34–42.

[2] K.-L. Ma, E. Lum, Techniques for Visualizing Time-Varying Volume Data, in: C. D. Hansen, C. Johnson (Eds.), Visualization Handbook, Academic Press, 2004, pp. 511–531.

[3] D. Ellsworth, L.-J. Chiang, H.-W. Shen, Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics, Volume Visualization Symposium, 2000, pp. 119-128.

[4] K.-L. Ma, H.-W. Shen, Compression and Accelerated Rendering of Time-Varying Volume Data, International Computer Symposium Workshop on Computer Graphics and Virtual Reality, 2000, pp. 82–89.

[5] H.-W. Shen, L.-J. Chiang, K.-L. Ma, A Fast Volume Rendering Algorithm for Time-Varying Fields Using A Time-Space Partitioning (TSP) Tree, IEEE Visualization, 1999, pp. 371–377.

[6] R. Westermann, Compression Time Rendering of Time-Resolved Volume Data, IEEE Visualization, 1995, pp. 168–174.

[7] S. Guthe, W. Straßer, Real-time Decompression and Visualization of Animated Volume Data, IEEE Visualization, 2001, pp. 349–356.

[8] S. Guthe, M. Wand, J. Gonser, W. Straßer, Interactive Rendering of Large Volume Data Sets, IEEE Visualization, 2002, pp. 53–60.

[9] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, T. Ertl, Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters, Eurographics Symposium on Parallel Graphics and Visualization, 2004, pp. 41–48.

[10] E. Lum, K.-L. Ma, J. Clyne, Texture Hardware Assisted Rendering of Time-Varying Volume Data, IEEE Visualization, 2001, pp. 263–270.

[11] E. Lum, K.-L. Ma, J. Clyne, A Hardware-Assisted Scalable Solution for Interactive Volume Rendering of Time-Varying Data, IEEE Transactions on Visualization and Computer Graphics, 8(3), 2002, pp. 286–301.

[12] H. Yu, K.-L. Ma, J. Welling, I/O Strategies for Parallel Rendering of Large Time-Varying Volume Data, Eurographics Symposium on Parallel Graphics and Visualization, 2004, pp. 31–40.

[13] J. Schneider, R. Westermann, Compression Domain Volume Rendering, IEEE Visualization, 2003, pp. 293–300.

[14] N. Fout, H. Akiba, K.-L. Ma, A. Lefohn, J. M. Kniss, High-Quality Rendering of Compressed Volume Data Formats, Eurographics/IEEE VGTC Symposium on Visualization, 2005.

[15] C. T. Silva, J. L. D. Comba, S. P. Callahan, F. F. Bernardon, A Survey of GPU-Based Volume Rendering of Unstructured Grids, Brazilian Journal of Theoretic and Applied Computing, 12(2), 2005, pp. 9–29.

[16] P. Shirley, A. Tuchman, A Polygonal Approximation to Direct Scalar Volume Rendering, San Diego Workshop on Volume Visualization, 24(5), 1990, pp. 63–70.

[17] P. L. Williams, Visibility-Ordering Meshed Polyhedra, ACM Transactions on Graphics, 11(2), 1992, pp. 103–126.

[18] M. Weiler, M. Kraus, M. Merz, T. Ertl, Hardware-Based Ray Casting for Tetrahedral Meshes, IEEE Visualization, 2003, pp. 333-340.

[19] S. P. Callahan, M. Ikits, J. L. D. Comba, C. T. Silva, Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering, IEEE Transactions on Visualization and Computer Graphics, 11(3), 2005, pp. 285–295.

[20] S. P. Callahan, J. L. D. Comba, P. Shirley, C. T. Silva, Interactive Rendering of Large Unstructured Grids Using Dynamic Level-of-Detail, IEEE Visualization, 2005, pp. 199-206.

[21] W. E. Lorensen, H. E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, ACM SIGGRAPH, 1987, pp. 163–169.

[22] A. Doi, A. Koide, An Efficient Method of Triangulating Equivalued Surfaces By Using Tetrahedral Cells, IEICE Transactions Communication, Elec. Info. Syst., E74(1), 1991, pp. 214–224.

[23] I. E. Sutherland, R. F. Sproull, R. A. Schumacker, A Characterization of Ten Hidden Surface Algorithms, ACM Computing Surveys, 6(1), 1974, pp. 1–55.

[24] S. Röttger, M. Kraus, T. Ertl, Hardware-Accelerate Volume Rendering and Isosurface Rendering Based on Cell-Projection, IEEE Visualization, 2000, pp. 109–116.

[25] V. Pascucci, Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement, and Tetrahedral Stripping, Eurographics/IEEE VGTC Symposium on Visualization, 2004, pp. 293–300.

[26] W. Corrêa, J. Klosowski, C. Silva, iWalk: Interactive Out-of-Core Rendering of Large Models, Princeton University Technical Report TR-653-02, 2002.

[27] H. T. Vo, S. P. Callahan, N. Smith, C. T. Silva, W. Martin, D. Owen, D. Weinstein, iRun: Interactive Rendering of Large Unstructured Grids, SCI Institute Technical Report, 2006.

[28] K. Engel, M. Kraus, T. Ertl, High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading, Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2001, pp. 9–16.

[29] M. Vannier, J. Marsh, J. Warren, Three Dimensional Computer Graphics for Craniofacial Surgical Planning and Evaluation, Computer Graphics and Interactive Techniques, 1983, pp. 263–273.

[30] P. Desgranges, K. Engel, G. Paladini, Gradient-Free Shading: A New Method for Realistic Interactive Volume Rendering, Vision, Modeling, and Visualization, 2005.

[31] Y. Livnat, X. Tricoche, Interactive Point Based Isosurface Extraction, IEEE Visualization, 2004, pp. 457–464.

[32] H. Akiba, N. Fout, K.-L. Ma, Simultaneous Classification of Time-Varying Volume Data Based on the Time Histogram, Eurographics/IEEE VGTC Symposium on Visualization, 2006, pp. 171–178.

[33] T. J. Jankun-Kelly, K.-L. Ma, A Study of Transfer Function Generation for Time-Varying Volume Data, Eurographics/IEEE TVCG Workshop on Volume Graphics, 2001, pp. 51–68.

[34] F. F. Bernardon, S. P. Callahan, J. L. D. Comba, C. T. Silva, Interactive Volume Rendering of Unstructured Grids with Time-Varying Scalar Fields, Eurographics Symposium on Parallel Graphics and Visualization, 2006, pp. 51–58.

[35] S. G. Akl, Parallel Sorting Algorithms, Academic Press, Inc., Orlando, 1990.