Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Interactive Volume Rendering of Unstructured Grids with Time-Varying Scalar Fields

Fábio F. Bernardon,[1] Steven P. Callahan,[2] João L. D. Comba,[1] and Cláudio T. Silva[2]

[1]Instituto de Informática, Federal University of Rio Grande do Sul, Brazil
[2]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 few 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 uses an encoding mechanism that is tightly coupled with our rendering system. Decompression is performed on the CPU while rendering for the next frame is processed. The rendering system runs entirely on the GPU, with an adaptive time-varying visualization that has a built-in level-of-detail that chooses the most significant aspects of the data.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

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 equipments also produces a lot 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 [Ma03] 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 memory either on 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 exploring coherence (either spatial or temporal) is easier due to the regular structure of the

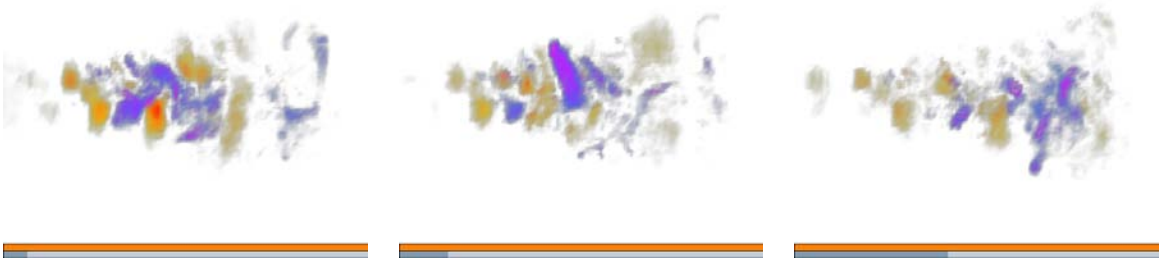

Figure 1: *Different time instances of the Turbulent Jet dataset consisting of one million tetrahedra and rendered at approximately four frames per second. Our user interface consists of an adjustable orange slider representing the level-of-detail and an adjustable gray slider representing the current time instance.*

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 rendering solutions are necessary to provide high-quality images. In addition, the volume rendering system needs to be flexible enough to handle data that may change at each frame. Finally, 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 must be adapted to efficiently handle the dynamic case.

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 introduce a new importance sampling approach for dynamic level-of-detail that operates 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 and conclusions and future work are described in Section 5.

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 [Ma03] and Ma and Lum [ML04].

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 [ECS00, MS00, SCM99]. The Time-Space Partitioning (TSP) Tree used in those papers is based on an octree which is extended to encode one extra dimension [SCM99] by storing a binary tree at each node that represents the evolution of the subtree through time. The TSP tree can also store partial sub-images to accelerate ray-casting rendering.

The compression of time-varying isosurface and associated volumetric data with a wavelet transform was first proposed in [Wes95]. 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 [GWGS02]. Lum *et al.* [LMC01, LMC02] compress time-varying volumes using the Discrete Cosine Transform (DCT). Due to the fact that 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 [YMW04].

More related to our work is the technique proposed by [SW03]. 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 into textures and decompressed using fragment programs on the GPU Since the multi-resolution representations are applied to a single unstructured grid, different quantization and compression tables are required for each time instance. Issues regarding the quality of rendering from compressed data were discussed in [FAM*05] using the approach described by [SW03] as a test case.

Hardware-assisted volume rendering has received considerable attention in the research community in recent years (for a recent survey, see [SCCB05]). Shirley and Tuchman [ST90] 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. More recently, Weiler *et al.* [WKME03] 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.* [CICS05] which treats 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.* [CCSS05] 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.

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 rendering, and dynamic level-of-detail for interactivity. Figure 2 shows the interplay of these components. To balance the workload, our algorithm efficiently takes advantage of both the CPU and GPU simultaneously, which improves the user experience.

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 [SW03], 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.

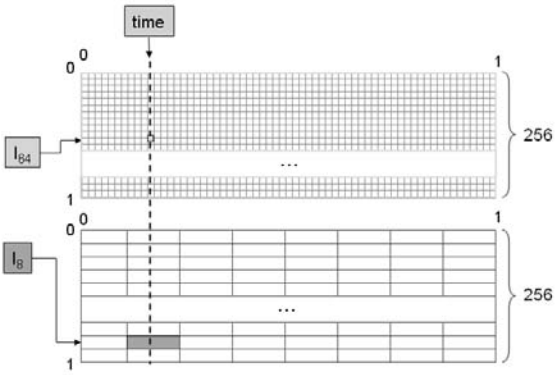

Figure 3: *Decompression of a given time instance for each mesh vertex requires adding the scalar mean of a block to the quantized differences recovered from the respective entries (i_8 and i_{64}) in the codebooks.*

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 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).

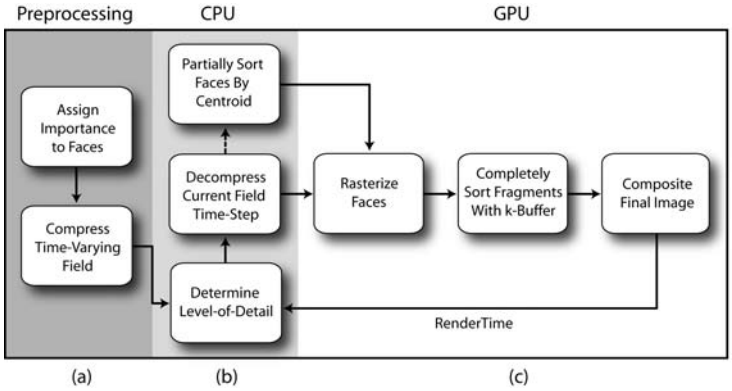

Figure 2: *An overview of our system. (a) Data compression and importance sampling for level-of-detail are performed in preprocessing steps on the CPU. (b) Then during each pass, level-of-detail selection, optional object-space sorting, and data decompression for the next step occur on the CPU. (c) Simultaneously,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

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 is 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}). This solution does not work well since 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.

A second approach is 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 interpolator hardware). Also, this computation requires accessing the mean and codebook indexes. Sending this information as a single vertex attribute is not possible since they are interpolated, and multiple-vertex attributes increase the amount of data transfer per vertex. Since our volume renderer runs in the fragment shader, this solution also increases the shader complexity and thus reduces performance of the system.

The final and our 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 by 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. In addition, decompression of the next time step can be performed on the CPU while the GPU renders the current frame. This provides a better distribution of the computation through all the system resources.

3.3. Volume Rendering

Our system is based on the Hardware-Assisted Visibility Sorting (HAVS) algorithm of Callahan *et al.* [CICS05]. Figure 2 shows how the volume rendering system handles time-varying data.

The HAVS algorithm 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 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.

3.4. Time-Varying Level-of-Detail

Recent work by Callahan *et al.* [CCSS05] introduces a new dynamic level-of-detail 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 level-of-detail 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 level-of-detail 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 histogram and use stratified sampling to stochastically select the triangles that cover the entire range of scalars. This approach works well for static geometry, but may miss important regions if applied to each time instance when considering time-varying data. In addition, attempting to assign a different importance for each time instance is not practical for interactive rendering. Our goal is to create a sampling strategy that works globally for every time step.

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:

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

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. 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. This 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). This heuristic works especially well 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. For datasets with regions that change frequently during some time steps and infrequently at others, this approach could be used on a subset of the time-steps and the importance of the triangles could be changed at a small cost several times throughout the animation.

To incorporate this level-of-detail strategy 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 level-of-detail on the fly to maintain interactivity. Our second type of interaction allows the user to use a slider to control the level-of-detail. This slider dynamically changes the speed of the animation by setting the level-of-detail manually. Since visibility ordering-dependent viewing transformations occur on the CPU in parallel to the GPU rendering, they do not change the level-of-detail or speed of the animation. Figure 2 shows the interaction of the level-of-detail algorithm with the time-varying data.

4. Results

In this section we report results obtained using as computational environment 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.

Mesh	Num Verts	Num Tets	Time Instances	Size TVSF	Size VQ	Compression Ratio	SNR Min	SNR Max	Max Error	Static FPS	Time-Varying FPS	Time-Varying Tets/s
SPX1	36K	101K	64	9.0M	504K	18.3	39.5	42.0	0.0045	32.2	30.7	3.1M
SPX2	162K	808K	64	40.5M	2.0M	20.6	39.2	42.0	0.0091	5.0	4.7	3.8M
SPXF	19K	12K	192	14.7M	2.0M	7.1	20.8	30.2	0.0144	125	83.3	1.0M
BLUNT	40K	183K	64	10.0M	552K	18.6	41.7	44.4	0.0046	23.1	16.7	3.0M
TORSO	8K	50K	360	11.2M	1.0M	11.4	20.5	28.1	0.0017	49.2	48.9	2.5M
TJET	160K	1M	150	93.8 M	2.7M	34.7	5.3	17.9	0.2042	3.7	3.6	3.6 M

Table 1: *Results of compression sizes, ratios, and error as well as performance comparisons for static and time-varying volume rendering.*

Datasets

Datasets used in our tests have diverse sizes and number of time instances. The SPX1, SPX2 and Blunt Fin (BLUNT) datasets were procedurally generated by linear interpolation that fades scalar values to zero. 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 Figs 1 and 6.

Compression

The compression of TVSF data uses the vector quantization code written by Schneider *et al.* [SW03]. The only caveat while using this code is that it works with structured grids with building blocks of $4 \times 4 \times 4$ (for a total of 64 values per block). In order 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 multiresolution 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 SPX2F 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 given by the VQ code, we also measured the minimum and maximum descrepancy between the original and quantized values. Results show that procedurally generated datasets have a higher SNR and smaller discrepancy, since they have a linear variation on their scalar fields. The TJET dataset has smaller SNR values since it represents a real fluid simulation, but it also led to higher compression ratios since codebook sizes are fixed.

Rendering

The rendering system allows the user to interactively inspect the time-varying data, play continuously 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.

Rendering time statistics were produced using a fixed number of viewpoints. In Table 1 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. The dynamic overhead is minimal even for the larger datasets. Note that for the smaller datasets, it is difficult to accurately measure the frame-rates when they exceed 60 frames per second.

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 (Figure 5)

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 even the largest dataset in our experiments. In addition, there is no noticeable

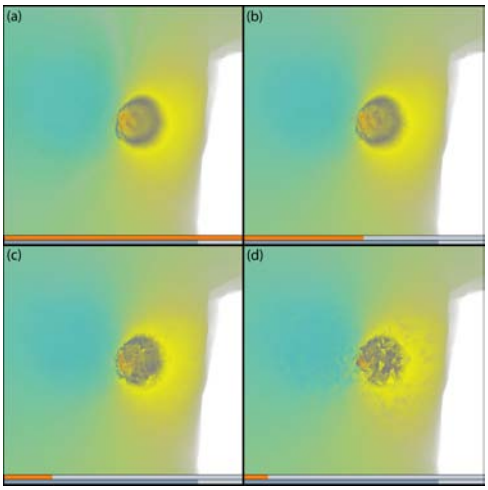

Figure 4: *Time-varying level-of-detail(LOD) strategy using the coefficient of variance. For a close-up view, (a) 100% LOD at 18 fps, (b) 50% LOD at 33 fps, (c) 25% LOD at 63 fps, and (d) 10% LOD at 125 fps.*

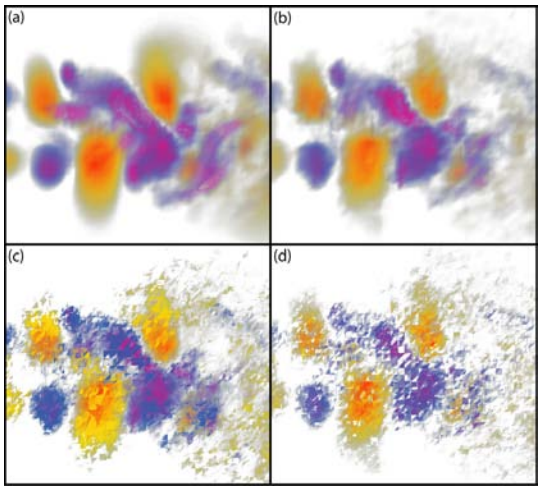

Figure 5: *Comparison of rendering using (a) uncompressed and (b) compressed data. Level-of-Detail is compared at 5% using (c) our time-varying approach and (d) a published technique for static datasets based on the field [CCSS05].*

overhead in adjusting the level-of-detail at a per frame basis because only the number of triangles in the current frame is computed (see [CCSS05]). Figure 4 shows the results of our level-of-detail strategy on the TORSO dataset at decreasing levels-of-detail. Figure 5 shows the increase in image quality when using our time-varying level-of-detail approach instead of a static approach that is based on sampling the field of one time-step. For comparison, we used the field-based approach from [CCSS05] which gave the best image quality for this dataset. In our experiments, the frame-rates increase at the same rate as the level-of-detail decreases (see Figure 4).

Limitations

Our algorithm successfully balances CPU and GPU workloads. However, with GPU speeds increasing faster than CPU speeds, this balance will need to be revisited to push more work onto the GPU or shift the CPU burden to multiple cores. Another limitation of our approach is that because our compression exploits temporal coherence, it may not be suitable for datasets with abrupt changes between time instances.

5. 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 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 a new time-varying approach for dynamic level-of-detail that improves 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.

6. Acknowledgments

The authors thank J. Schneider for the VQ code, Mike Callahan and the SCIRun team at the University of Utah for the torso dataset, Bruno Notrosso (Electricite 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,

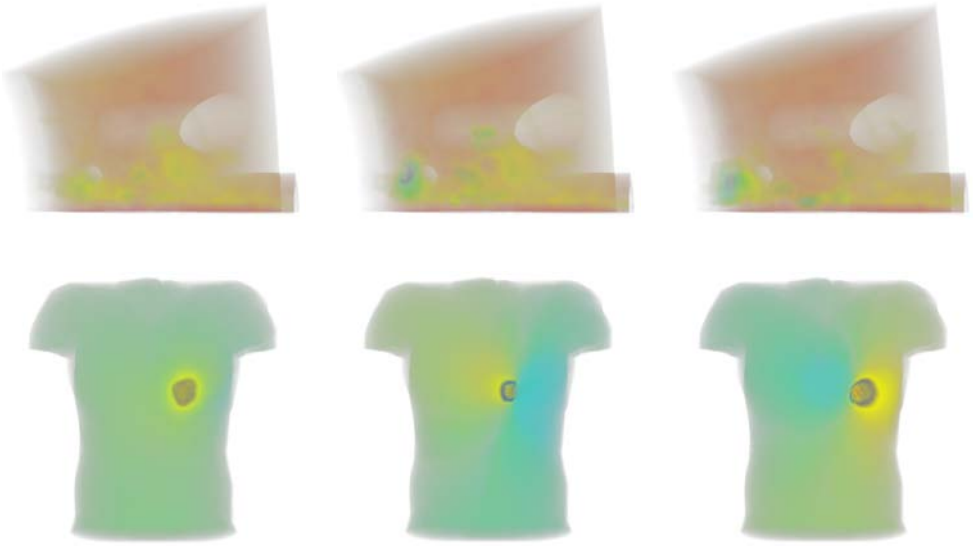

Figure 6: *Different time instances of the SPXF (above) and Torso (below) datasets at 100% LOD.*

and CCF-0528201, the Department of Energy, an IBM Faculty Award, the Army Research Office, and a University of Utah Seed Grant.

References

[CCSS05] CALLAHAN S. P., COMBA J. L. D., SHIRLEY P., SILVA C. T.: Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05* (2005), pp. 199–206.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L., SILVA C. T.: Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[ECS00] ELLSWORTH D., CHIANG L.-J., SHEN H.-W.: Accelerating time-varying hardware volume rendering using TSP trees and color-based error metrics. In *Volume Visualization Symposium* (2000), pp. 119–128.

[FAM*05] FOUT N., AKIBA H., MA K.-L., LEFOHN A., KNISS J. M.: High-quality rendering of compressed volume data formats. In *EuroVis '05* (2005).

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *IEEE Visualization '02* (2002), pp. 53–60.

[LMC01] LUM E., MA K.-L., CLYNE J.: Texture hardware assisted rendering of time-varying volume data. In *IEEE Visualization '01* (2001), pp. 263–270.

[LMC02] LUM E., MA K.-L., CLYNE J.: A hardware-assisted scalable solution of interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (2002), 286–301.

[Ma03] MA K.-L.: Visualizing time-varying volume data. *Computing in Science & Engineering 5*, 2 (2003), 34–42.

[ML04] MA K.-L., LUM E.: Techniques for visualizing time-varying volume data. In *Visualization Handbook*, Hansen C. D., Johnson C., (Eds.). Academic Press, 2004.

[MS00] MA K.-L., SHEN H.-W.: Compression and accelerated rendering of time-varying volume data. In *International Computer Symposium Workshop on Computer Graphics and Virtual Reality '02* (2000), pp. 82–89.

[SCCB05] SILVA C. T., COMBA J. L. D., CALLAHAN S. P., BERNARDON F. F.: A survey of GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA) 12*, 2 (2005), 9–29.

[SCM99] SHEN H.-W., CHIANG L.-J., MA K.-L.: A fast volume rendering algorithm for time-varying field using a time-space partitioning (TSP) tree. In *IEEE Visualization '99* (1999), pp. 371–377.

[ST90] SHIRLEY P., TUCHMAN A.: A polygonal approximation to direct scalar volume rendering. *San Diego Workshop on Volume Visualization 24*, 5 (1990), 63–70.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *IEEE Visualization '03* (2003).

[Wes95] WESTERMANN: Compression time rendering of time-resolved volume data. In *IEEE Visualization '95* (1995), pp. 168–174.

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-Based Ray Casting for Tetrahedral Meshes. In *IEEE Visualization '03* (2003), pp. 333–340.

[YMW04] YU H., MA K.-L., WELLING J.: I/O strategies for parallel rendering of large time-varying volume data. In *Eurographics Symposium on Parallel Graphics and Visualization '04* (2004), pp. 31–40.