Volume Rendering of Time-Varying Scalar Fields on Unstructured Meshes

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

[1] Instituto de Informática, UFRGS

[2] Scientific Computing and Imaging Institute, University of Utah

Abstract

Volume Rendering of time-varying datasets is essential in several scientific applications. Due to the enormous amount of data involved, in datasets with static sampling regions it is common to consider only time-varying scalar fields (TVSFs). The use of Vector Quantization (VQ) to compress scalar fields has been shown to be quite effective when combined with texture-based volume rendering algorithms for structured grids. In this paper we discuss how to apply VQ to volume render unstructured grids (meshes of tetrahedra). We extended two of the fastest unstructured grid algorithms (both use programmable GPUs) to handle time-varying scalar fields, discuss advantages and disadvantages of each extension, and show results that allows us to interactively render meshes composed of nearly one million tetrahedra and several hundred time instances.

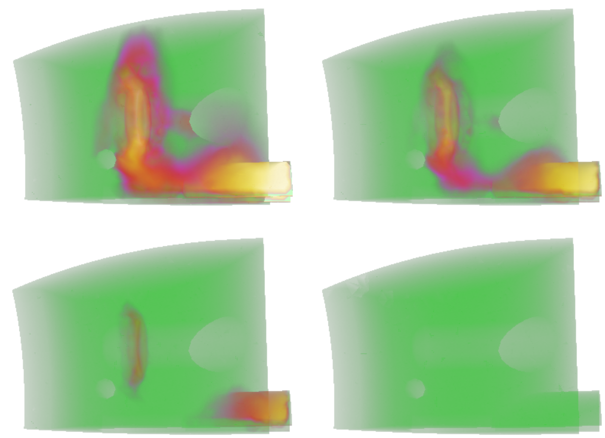

Figure 1. Different time steps for SPX dataset.

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.

However, at this time, there is a mismatch between the simulation 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 [8] 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.

In this paper, we propose an approach that couples the compression scheme proposed in [11] with rendering techniques proposed in [2, 1]. In our approach, the data is first compressed using hierarchical vector quantization [11], which helps reduce the amount of data being transferred through the AGP bus (in fact, we can handle an arbitrary number of time steps through page management of the compression tables). Then, it is rendered with the Hardware-Assisted Visibility Sorting (HAVS) algorithm or GPU-based ray casting.

2. Related 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 [8] and Ma and Lum [9].

Very little has been done for 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, 10, 12]. The Time-Space Partitioning (TSP) Tree used in those papers is based on an octree which is extended to encode one extra dimension [12] by storing a binary tree at each node that represents the evolution of the subtree through time. The TSP tree can also store partial subimages to accelerate ray-casting rendering.

More related to our work is the technique proposed by Westermann [14], where he compresses time-varying isosurface and associated volumetric data with a wavelet transform that allows for fast reconstruction and rendering. A follow-up of this work [11] describes a GPU implementation using Vector Quantization, reviewed in more details in the next section. Another technique is the hardware-accelerated rendering technique of Lum et al [5, 6]. The basic idea is based on the temporal encoding of indexed volumetric data that can be quickly decoded in hardware. Their rendering engine is based on texture-based volume rendering. They compressed the time-varying volume by breaking it up into "spans" and using the Discrete Cosine Transform (DCT). Every sample within the span is encoded as a single index. Then, they store the volume as a set of 2D paletted textures, which are decoded using a time-varying palette. 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 [15].

3. Compression of TVSF

Compression schemes are important to reduce the memory footprint of time-varying volumetric datasets. For real-time applications, the decompression time needs to be considered along with compression ratios when selecting a compression scheme. In this section we review the vector quantization (VQ) approach used by [11] to compress TVSF for structured grids, and extend it to handle unstructured grids.

3.1 Hierarchical Decomposition

In order to prepare for VQ, data is organized hierarchically using a multi-resolution approach. Consider a TVSF on a given vertex v of a mesh containing n time instances (i.e. n scalar values per vertex). Let V_l be a vector in R^n storing the different scalar values of v, and for simplicity consider n to be a power of b (i.e. $b^l = n$). We divide V_l into b disjoint subsets, compute the mean of its elements, and use the resulting averages to form a down-sampled vector V_{l-1} in $R^{n/b}$ of V_l. This process is repeated for l iterations until we obtain a vector V_0 in R containing a single value which corresponds to the mean of the entire vector V_l.

The multi-resolution representation is created by computing l detail vectors D_{2^i} in R^{2^i} that corresponds to the difference between the original vector V_i (i:1..l) and the average used to create the down-sampled vector of V_i. The detail vectors combined with the mean stored in V_0 are sufficient to reconstruct each element of V_l. The reconstruction involves adding the mean V_0 with the proper coefficients of each detail vector D_{2^i}. In Figure 2 we show an example with $n = 64$, $b = 8$ and $l = 2$. The mean stored at V_0 is added to one component of the detail vectors D_8 and D_{64} to reconstruct the original data.

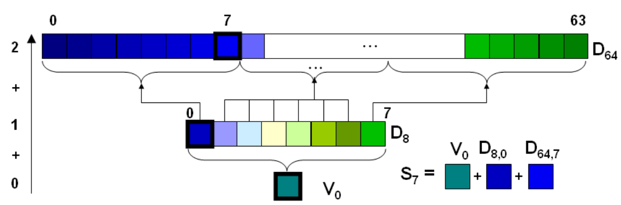

Figure 2. Vector Reconstruction.

3.2 VQ in structured grids

The VQ approach described in [11] to compress TVSF in structured grids is summarized here. First, consider one time instance of an $n \times n \times n$ structured grid. The volume is decomposed into $4 \times 4 \times 4$ sub-blocks, which was a suitable choice for their tests. Each sub-block contains 64 values represented as a vector in R^{64}. Down-sampled versions of this vector are obtained by considering $2 \times 2 \times 2$ and $1 \times 1 \times 1$ sub-blocks. Two detail vectors D_8 and D_{64} are computed for each sub-block.

Consider a structured grid with dimensions $128 \times 128 \times 128$, with 32768 $4 \times 4 \times 4$ sub-blocks. The 32768 detail vectors are processed using VQ quantized to generate two codebooks C_8 and C_{64}, each with the 256 most representative vectors of each set. VQ uses covariance analysis to find an initial codebook, which is then refined using a modification of the LBG-algorithm [4]. Original values are reconstructed by keeping, for each block, the mean V_0 and two indices i_8 and i_{64} into codebooks C_8 and C_{64}.

In order to handle time-varying data, they propose to use a new codebook for each time instance. Due to temporal coherence, a codebook generated in time t often can be used during quantization as initial codebook for the LBG-algorithm for the next time step, thus avoiding the covariance analysis at each time instance.

3.3 VQ in unstructured grids

Our solution to extend the VQ approach to unstructured grids arranges data for VQ in a different manner. Instead of using spatial coherence when forming vectors (as in structured grids), we focus our approach on temporal coherence. For each vertex of a structured mesh, we form vectors containing 64 scalar values, each corresponding to a consecutive time instance of the scalar field. The choice of 64 is for convenience only, and allows us to obtain a multi-resolution approach identical to the one used in [11]. The number of detail coefficients D_8 and D_{64} passed to VQ is given by the number of vertices in the mesh.

To reconstruct a scalar value, both codebook indices i_8 and i_{64} need to be stored per vertex as well as a mean value V_0. For a given time instance t, codebooks C_8 and C_{64} are accessed as described in Figure 3. Note that the same index t is used to index both C_8 and C_{64}.

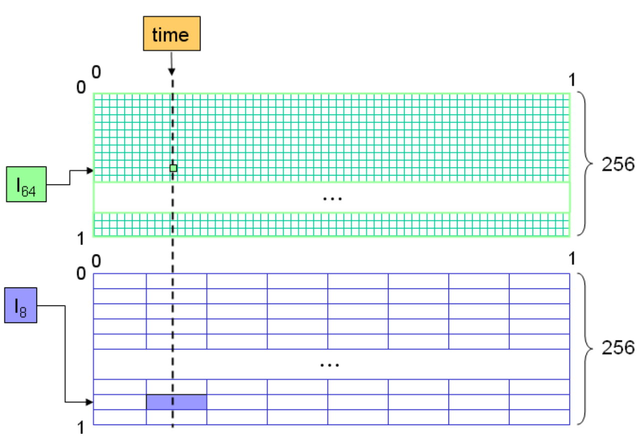

Figure 3. Codebooks access.

In the sections to follow we show how this compression scheme is combined with two GPU-based Volume Rendering algorithms.

4 Time-Varying HAVS

4.1 HAVS summary

The Hardware-Assisted Visibilty Sorting (HAVS) volume rendering system was proposed in [2]. Given an unstructured mesh, HAVS prepares the mesh faces for rasterization by sorting them by their centroids. This provides in most cases only a partial order of the faces in object-space since the mesh may contain faces of varying size or even visibility cycles. Upon rasterization, the fragments undergo an image-space sort in the GPU via a data structure called the k-buffer (Figure 4).

Figure 4. HAVS sorts faces on the CPU and GPU and composites them into a final image.

The k-buffer is implemented using fragment shaders and keeps a fixed number of fragments (k) in each pixel of the framebuffer. As a new fragment is rasterized, it is compared with the other entries in the k-buffer, the two entries closest to the viewpoint (for front-to-back) are used to find the color and opacity for the fragment using a lookup table which contains the pre-integrated volume integral. The color and opacity are composited in the framebuffer, and the remaining fragments are written back to the k-buffer (see [2] for more detail).

4.2 TVSF HAVS

Our first approach was to store the codebooks in textures and reconstruct scalar values in the Vertex Shader using Shader Model 3.0 available on an NVIDIA 6800, which allows a texture access within the vertex shader. This approach, however, did not work well and was considerably slow, since the number of operations that are performed in the vertex shader are small to compensate for the latency necessary to fetch data from textures.

The second alternative was to perform the reconstruction in the fragment shader. Since the k-buffer approach renders faces of the original mesh, we would need to recover the means and codebook indices i_8 and i_{64} for each face vertex (requiring at least 3 texture fetches). Since this information changes per face, it requires sending constant information to the fragment shader which is hard to do in any other place than as a program argument, which can be slow if changed for each face.

Our most effective solution is to avoid using vertex buffer objects (VBOs) for explicit definition of vertex data. At each rendering pass we perform the decompression on the CPU and pass reconstructed scalar values as texture coordinates. Codebooks are stored in CPU memory and a simple paging mechanism allows us to render multiple instances of time.

5 Time-Varying Ray Casting

5.1 Ray Casting Summary

Our algorithm is based on the GPU-based ray-casting described in [1], which introduced several extensions to the work described by [13]. The idea is to compute ray intersections using the fragment shader of a GPU, advancing one intersection inside the mesh at a time (Figure 5) while evaluating the volume rendering integral. Two sets of textures are used to store current ray intersection information, as well as accumulated color.

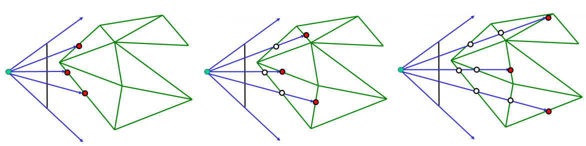

Figure 5. Ray-Casting Calculation

Computing the point that each ray leaves a tetrahedron is only possible if mesh data is available at the fragment shader, therefore mesh information (vertices, normals, connectivity, scalar fields) needs to be stored per tetrahedron in texture memory. Scalar data is stored using the gradient of the scalar field and a reference static value, in order to allow reconstruction of scalar values using interpolation. A depth-peeling approach is used to handle non-convex meshes in a more general way than [13].

5.2 GPU Storage

Since this approach already uses texture memory to store mesh data, adding TVSF data increases GPU memory usage even further. Our first action was to reduce usage for the static algorithm, removing the normals, since they can be reconstructed from vertex positions.

An important difference is the representation of scalar data. We apply the same VQ used in HAVS, which allows us to decompress scalar data for each vertex, instead of the gradient of the scalar field as in the static case. In order to perform interpolation, we store a matrix that allows us to calculate the gradient of the scalar field [7]. Codebook indices and mean are stored per vertex in a way similar to vertex positions. Figure 6 illustrates the data stored per tetrahedron (192B total). In addition, we need to store the codebooks using 8×256 and 64×256 floating-point textures.

Mesh Data	Tex Format	Tex. Coord.		Texture Data			
		u	v	r	g	b	a
vertices	F32x4	t_u	t_v	$V0.x_t$	$V0.y_t$	$V0.z_t$	$V3.x_t$
vertices	F32x4	t_u	t_{v+dv}	$V1.x_t$	$V1.y_t$	$V1.z_t$	$V3.y_t$
vertices	F32x4	t_u	t_{v+2dv}	$V2.x_t$	$V2.y_t$	$V2.z_t$	$V3.z_t$
neighbor index	F32x4	t_u	t_v	$t_u(a_0)$	$t_v(a_0)$	$t_u(a_1)$	$t_v(a_1)$
neighbor index	F32x4	t_u	t_{v+dv}	$t_u(a_2)$	$t_v(a_2)$	$t_u(a_3)$	$t_v(a_3)$
gradient matrix	F32x4	t_u	t_{v+2dv}	a_{11}	a_{12}	a_{13}	a_{14}
gradient matrix	F32x4	t_u	t_v	a_{21}	a_{22}	a_{23}	a_{24}
gradient matrix	F32x4	t_u	t_{v+dv}	a_{31}	a_{32}	a_{33}	a_{34}
gradient matrix	F32x4	t_u	t_{v+2dv}	a_{41}	a_{42}	a_{43}	a_{44}
compressed	F32x4	t_u	t_v	$m_{0,t}$	$i8_{0,t}$	$i64_{0,t}$	$m_{3,t}$
compressed	F32x4	t_u	t_{v+dv}	$m_{1,t}$	$i8_{1,t}$	$i64_{1,t}$	$i8_{3,t}$
compressed	F32x4	t_u	t_{v+2dv}	$m_{2,t}$	$i8_{2,t}$	$i64_{2,t}$	$i64_{3,t}$

Figure 6. Mesh Data (per tetrahedron)

5.3 Managing Codebook Changes

The codebooks we used only handle 64 time instances. To avoid rendering stalls while switching codebooks, we again use a paging mechanism to keep in texture memory the current and next codebooks. The first two codebooks are loaded into GPU memory when rendering starts. After we access the last time instance stored in the first codebook, we swap texture references to the second set, already in memory. The rendering process continues, and simultaneously we load the next codebook in place of the first one, giving time to the new data to be loaded into the graphics memory before it is required. This avoids a stall in the graphics pipeline while rendering, and allow us to handle an arbitrary number of time steps without any noticeable performance loss.

5.4 Decompression

The scalar reconstruction runs on the fragment shader. First we recover for each vertex the mean and two codebook indices i_8 and i_{64}. Each codebook index is used as the *v* coordinate to access the codebook texture. The *u* coordinate contains the current time step, and it used for both codebooks. Once the four scalars are reconstructed, we calculate the gradient of the scalar field using the gradient matrix stored with the tetrahedron data. We finally compute the scalar value for both the current entry and exit points of the tetrahedron, using the algorithm described in ([7]).

6. Results

Experiments were performed on a PC computer with a 2.8 GHz Pentium 4 and a GeForce 6800GT with 256MB RAM. The HAVS code was written using OpenGL and the Ray-Casting Code uses DirectX 9.0c. A video showing some of the results can be found at `http://www.inf.ufrgs.br/~fabiofb/tvsf`.

6.1 Compression Results

Most TVSF used in our tests were procedurally generated with known unstructured grids. However, the Torso dataset shows the results of a rotating dipole in the mesh and the Brain dataset shows actual electroencephalograph (EEG) readings of a brain mapped onto a head. The number of time instances varies between 64 and 360. Scalar values passed to VQ are float numbers, with one codebook generated for each sequence of 64 time steps. We used the VQ code written by Schneider et al[11] to compress our TVSF data. The meshes we used in our tests are listed in Table 1.

Mesh	Vert	Tetra	Time Instances
SPX	19K	12K	64
SPX1	36K	101K	64
SPX2	162K	808K	64
BLUNT	40K	183K	64
TORSO	8K	50K	360
BRAIN	68K	387K	120

Table 1. TVSF mesh data

In Table 2 we summarize the compression results we obtained. In addition to the signal-to-noise ratio given by the VQ code, we also measured the minimum and maximum discrepancy between the original and quantized values. The procedurally generated datasets have more continuous variation on the scalar fields and give the higher SNR results. The brain dataset has the worst compression results since it has more complex and alleatory scalar field motion.

The storage of TVSF data without compression is given by $size_u = v \times t \times 4B$, where v is the number of vertex meshes, t is the number of time instances in each dataset, and a scalar value uses four bytes (float). The compressed size using VQ is equal to $size_{vq} = v \times c \times 3 \times 4B + c \times 72KB$, where c is the number of codebooks used ($c = t/64$), each vertex requires 3 values per codebook (mean plus codebook indices i_8 and i_{64}), and each codebook uses $72KB = 256 \times 64 \times 4B + 256 \times 8 \times 4B$ (Table 2).

Mesh	Size TVSF	Size VQ	Compr Ratio	SNR Min	SNR Max	MAX Error
SPX	4.75M	300K	16.21	39.44	42.08	0.0041
SPX1	9.00M	504K	18.29	39.45	41.96	0.0045
SPX2	40.50M	1.97M	20.57	39.24	41.88	0.0091
BLUNT	10.00M	552K	18.55	41.70	44.36	0.0046
TORSO	11.25M	1008K	11.43	20.53	28.12	0.0017
BRAIN	31.87M	1.73M	18.38	2.96	10.24	1.0632

Table 2. Compression Results

6.2 HAVS results

Results were obtained by using a fixed number of viewpoints. Rendering rates were nearly the same for most datasets, and only 40% slower on the largest dataset (see Table 3).

Mesh	Min Time Static(ms)	Max Time Static(ms)	Min Time TVSF(ms)	Max Time TVSF(ms)
SPX	31	47	31	47
SPX1	109	125	110	125
SPX2	703	813	1016	1157
BLUNT	156	312	218	266
TORSO	62	79	62	79
BRAIN	438	500	578	625

Table 3. HAVS Results

6.3 Ray Casting results

Due to GPU memory limitation, we were only able to run the smaller datasets with the Ray Casting code. Table 4 shows the performance of our algorithm when compared with a static rendering.

Mesh	Min Time Static(ms)	Max Time Static(ms)	Min Time TVSF(ms)	Max Time TVSF(ms)
SPX	156	265	203	235
SPX1	297	500	406	672
BLUNT	94	1062	125	1125

Table 4. Ray-Casting Results

7. Conclusions

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 the performance for most cases. We have described how vector quantization can be employed in two state-of-the-art, GPU-assisted volume rendering systems to achieve interactive rendering rates. Our algorithm is simple, easily implemented, and most importantly, it closes the gap between rendering time-varying data on unstructured and structured grids.

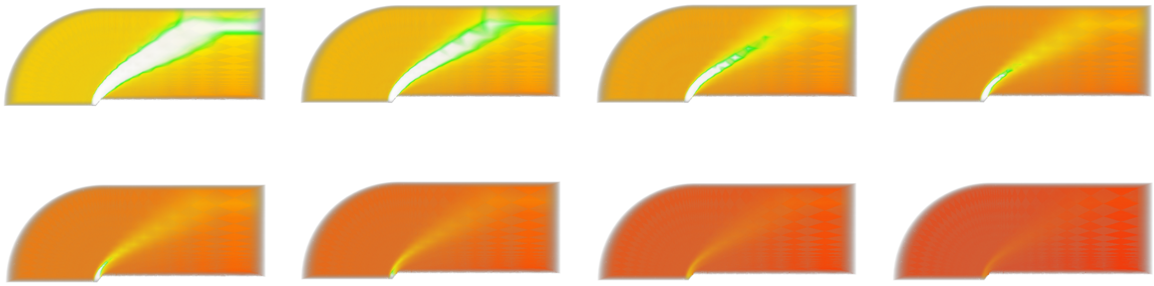

Figure 7. Different time steps of Blunt using HAVS.

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. Finally, we would like to explore solutions for time-varying geometric data.

Acknowledgments

The authors thank J. Schneider for the VQ code, Mike Callahan and the SCIRun team at the University of Utah for the brain and torso datasets, Bruno Notrosso (Electricite de France) for the SPX dataset, and NVIDIA from donated hardware. Steven Callahan is supported by the Department of Energy (DOE) under the VIEWS program. The work of Fábio Bernardon and João Comba is supported by a CNPq grant 540414/01-8 and FAPERGS grant 01/0547.3. Cláudio Silva is partially supported by the DOE under the VIEWS program and the MICS office, the National Science Foundation under grants CCF-0401498, EIA-0323604, and OISE-0405402, and a University of Utah Seed Grant.

References

[1] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. Gpu-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, to appear. Also available as SCI Institute Technical Report UUSCI-2004-006.

[2] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva. Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[3] D. Ellsworth, L.-J. Chiang, and H.-W. Shen. Accelerating time-varying hardware volume rendering using tsp trees and color-based error metrics. In *Proceedings of 2000 Volume Visualization Symposium*, pages 119–128, 2000.

[4] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 1:84–95, Jan. 1980.

[5] E. Lum, K.-L. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of IEEE Visualization 2001*, pages 263–270, 2001.

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

[7] C. Lürig, R. Grosso, and T. Ertl. Implicit Adaptive Volume Ray-Casting. In S. Klimenko, Y. Bayakovsky, and V. Galaktionov, editors, *Proceedings of GraphiCon '97*, pages 114–120, 1997.

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

[9] K.-L. Ma and E. Lum. Techniques for visualizing time-varying volume data. In C. D. Hansen and C. Johnson, editors, *Visualization Handbook*. Academic Press, 2004.

[10] K.-L. Ma and H.-W. Shen. Compression and accelerated rendering of time-varying volume data. In *Proceedings of 2000 International Computer Symposium – Workshop on Computer Graphics and Virtual Reality*, pages 82–89, 2000.

[11] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proceedings of IEEE Visualization 2003*.

[12] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying field using a time-space partitioning (tsp) tree. In *Proceedings of IEEE Visualization 1999*, pages 371–377, 1999.

[13] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, 2003.

[14] Westermann. Compression time rendering of time-resolved volume data. In *Proceedings of IEEE Visualization 1995*, pages 168–174, 1995.

[15] H. Yu, K.-L. Ma, and J. Welling. I/O strategies for parallel rendering of large time-varying volume data. In *Eurographics/ACM SIGGRAPH Symposiumm Proceedings of Parallel Graphics and Visualization 2004*, pages 31–40, 2004.

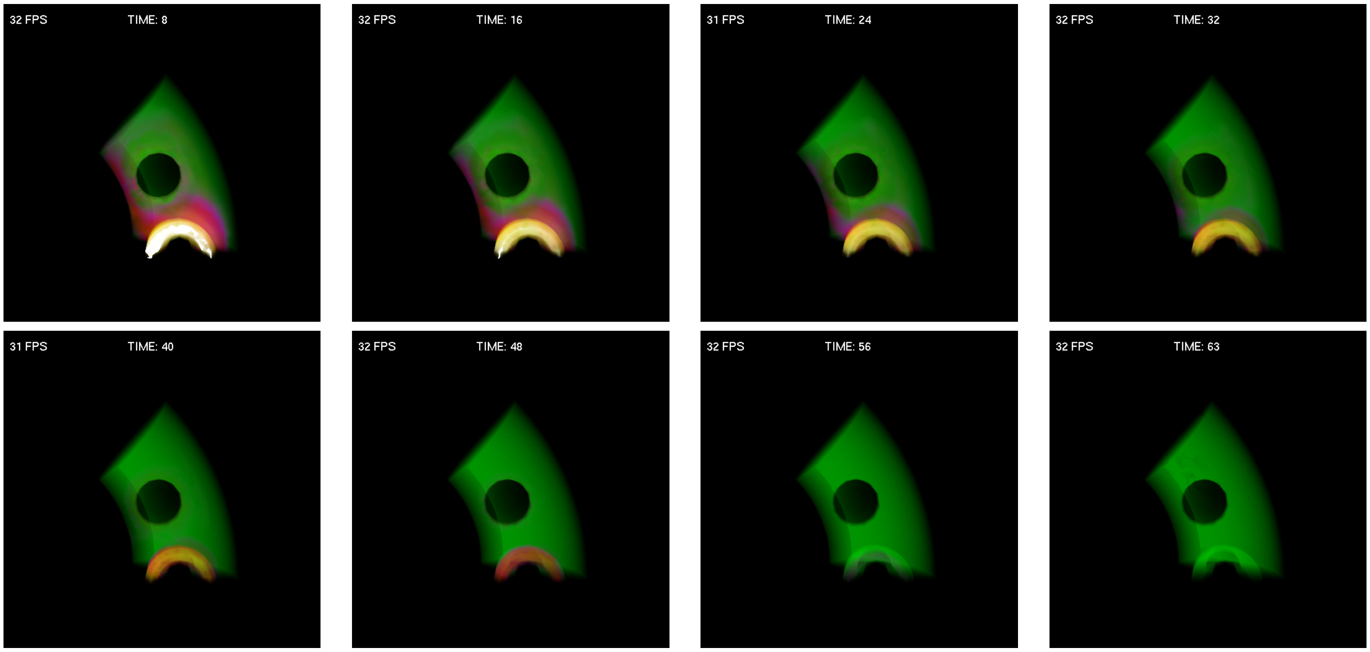

Figure 8. Different time steps of SPX using HAVS.

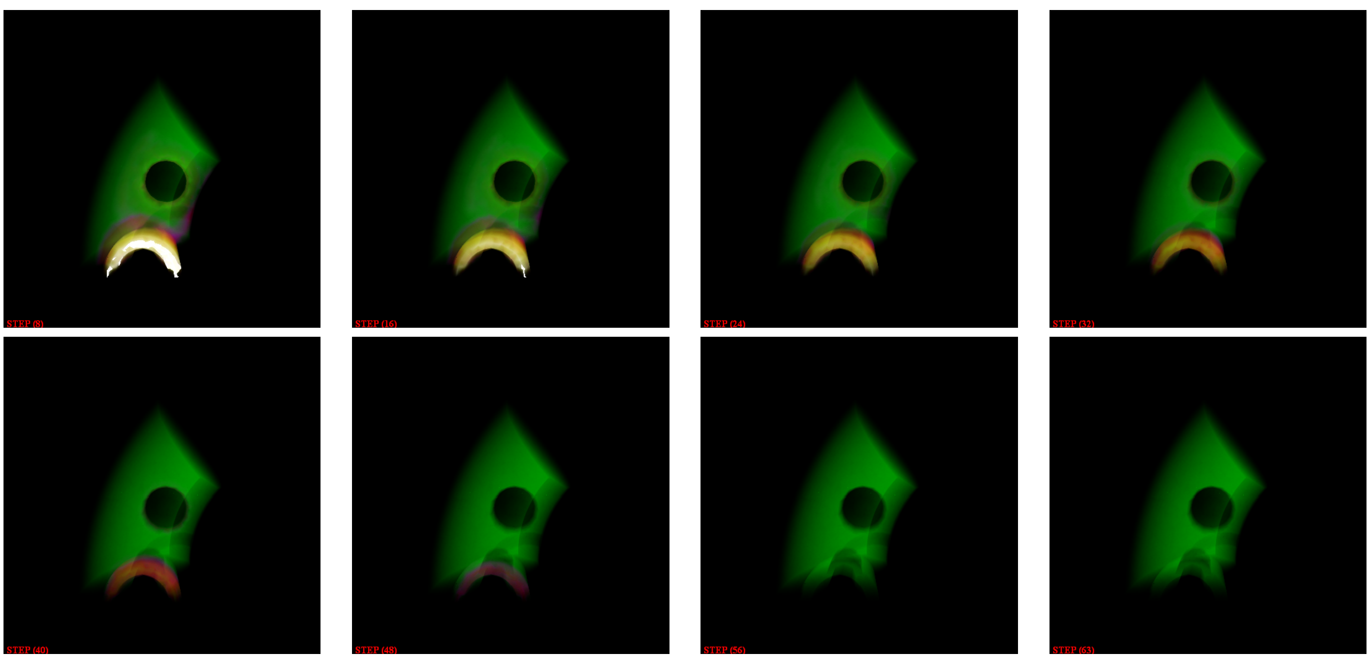

Figure 9. Different time steps of SPX using Ray Casting.

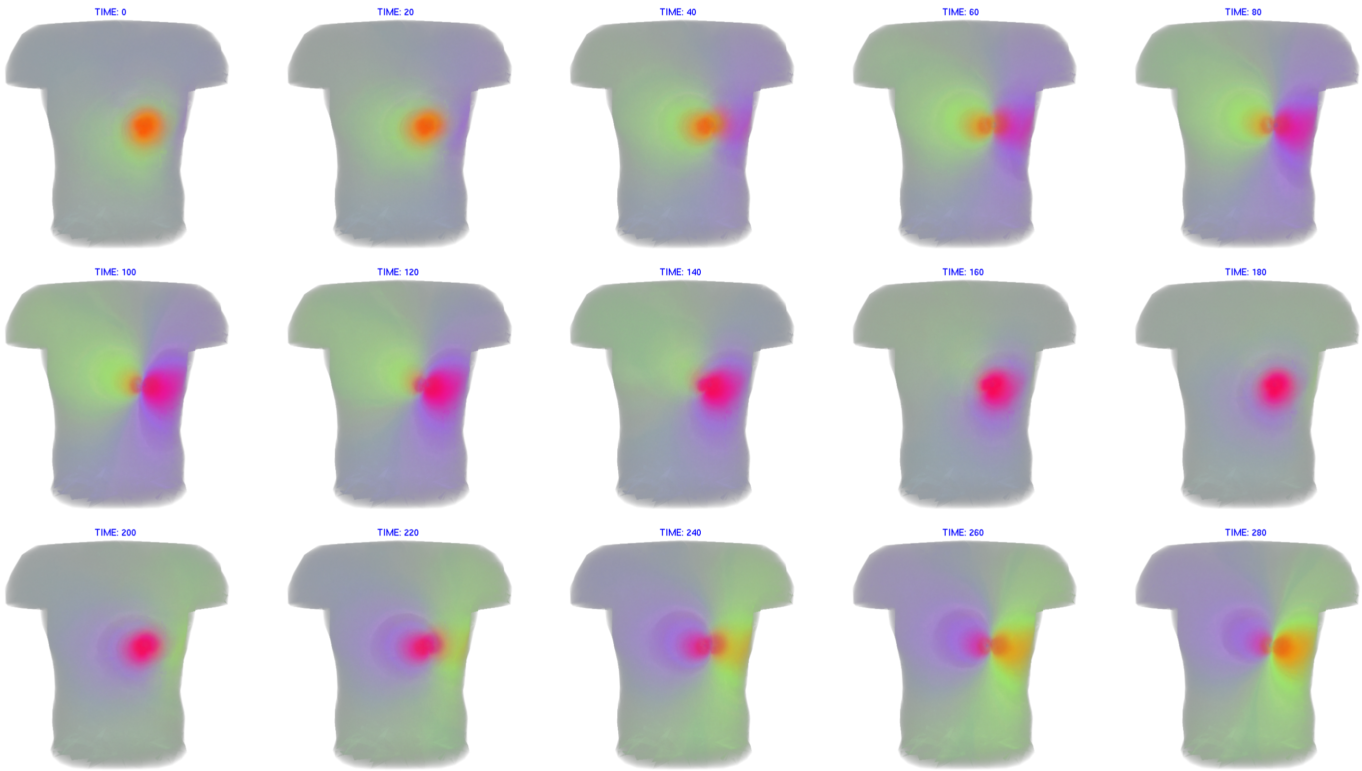

Figure 10. Different time steps of Torso dataset using HAVS.

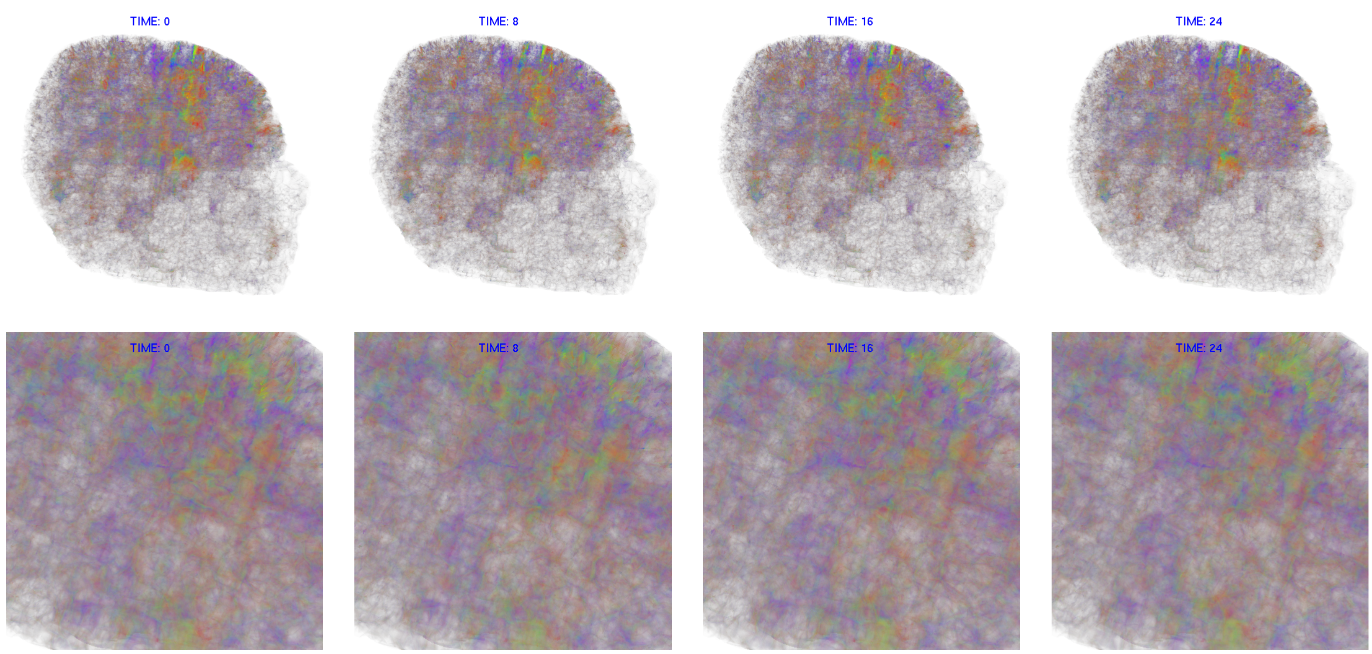

Figure 11. Different time steps of Brain using HAVS.