Asynchronous Rendering of Time-Variant Volumes with Workload-Based Data Shuffling

Antonio Garcia*
Ohio State University

Han-Wei Shen†
Ohio State University

Keywords: brick, partition, distribution, shuffling

Abstract

Complex time-varying phenomena usually produce volume datasets that are not static but dynamic in time. The amounts of storage and the computing power needed to visualize such datasets can easily exceed what could be offered by a single processor. Therefore, networks of computers are ideal to render them. However, data distribution schemes can be highly vulnerable to how the user wants to visualize the data, which could be stored in bricks that have several levels of resolutions. This scenario is highly influenced by the error estimates that low resolution data accumulate to represent high resolution data. Different set of error tolerances can produce rendering sequences that make the workload of processors imbalance.

We present the asynchronous rendering of time-variant volume datasets under a shuffled distribution, which is based on the expected number of bricks to render. The data is subdivided into partitions that hold many bricks; a metric is then calculated for every partition. The partitions are then mapped to processors based on the metric, so that high concentration of bricks go where low concentration have been previously assigned; and vice versa. Because of this behavior, asynchronous rendering is required; otherwise, processors stand idle instead of rendering frames.

Our results compare our method against the traditional synchronous rendering found in most of the parallel rendering literature. In all cases, our method balances the workload better in terms of number of bricks.

1 Introduction

Scientists are often presented with complex phenomena that change through time. Fluid dynamics, for instance, is an area of intensive research where scientists perform numerical simulations with high precision to study the phenomena. The output of these studies, often in the form of time-varying volume datasets, or TVVD, can be effectively visualized in animations through volume rendering.

*e-mail: agarcia@cis.ohio-state.edu
†e-mail: hwshen@cis.ohio-state.edu

The size of a high resolution time-varying volume dataset is often overwhelmingly large. A single processor is not capable of dealing with such huge datasets; therefore, a PC-cluster where several processors are connected with a high-speed network are more suitable for volume rendering them. PC-clusters also provide a distributed environment, which means that the data has to be partitioned and shipped to the different processors that take part in the rendering. This distribution is done prior to the rendering and data does not migrate from processor to processor.

The task of data distribution has to be accomplished with great care because it is very sensitive to load imbalances. A major reason for these imbalances is the setting of the run-time rendering parameters, for example: where the user is looking from or what materials he is interested in. These parameters are unknown until the visualization starts. Re-distributing the data upon changing the parameters is not an option, because data distribution by itself takes a long time to complete and thus hinders interactivity.

In this paper, we propose the shuffling distribution of data to balance the rendering of time-variant volume datasets. We distribute data by considering a metric that is based on the average number of bricks rendered by each partition, it also considers the error tolerances of bricks such as spatial and temporal deviations. Using this metric, processors switch from low to high concentration of bricks. Furthermore, processors must not wait for their neighbor processors to start rendering a new time step, because the behavior of shuffling can yield idle time; thus asynchronous rendering is necessary for the success of our distribution. It also overlaps the stages of rendering and compositing of the partial frames produced by the processors.

We compare our method against the traditional synchronous rendering that uses the fixed distribution, which puts processors in charge of the same spatial partition for all time steps. Our results show that the asynchronous rendering with data shuffling distribution was consistently better than the traditional approach.

The remainder of the paper is divided into the following sections: section 2 discusses the previous work; section 3 presents the basis of both synchronous rendering and asynchronous rendering; section 4 presents the shuffling distribution scheme; section 5 shows results; and section 6 discusses our conclusions and presents suggestions to extend our work.

2 Previous Work

Visualization of time varying volume data has received attention in the past. [Ma and Lum 2002] outlines the strategies for visualizing time-varying volume data. They mention encoding and compression as very important to accelerate the retrieval of data from disk. [Shen and Johnson 1995] exploit data coherence between consecutive time steps. Other encodings that are applied to the whole dataset are run-length encoding [Atherton et al. 2000] and discrete cosine transform combined with vector quantization [Ma et al. 2001]. Hierarchical representations, such as [Laur and Hanrahan 1991], allow for rendering of static volume data using different resolutions levels. [Westermann 1995] uses a hierarchy that encodes each time step using wavelets. [Shen et al. 1999] proposes the time-partitioning tree (TSP), which captures both the temporal and spatial coherence of time-varying volume data and results in considerable savings at rendering time. [Ellsworth et al. 2000] extends the TSP structure using hardware acceleration and color-based error metrics. [Shen et al. 1999] merge trees using quantization, octree encoding and difference encoding to reduce storage requirements. [Guthe and Strasser 2001] use motion compensation to visualize animated volume data. All the aforementioned publications present sequential algorithms.

In parallel visualization, [Lee et al. 1996] presents an algorithm where stages overlap. Rendering and compositing hide each other for the current frame, but processors move in unison to the next frame. [Chuieh and Ma 1997] presents the intra-volume, inter-volume and hybrid pipelines. The first pipeline uses all processors to render one volume at a time, the next pipeline has one processor assign to one volume, and the last assigns groups of processors to one volume. The pipelines are synchronous in nature, but inter-volume and hybrid are able to overlap I/O with rendering thus reducing time. In this work, we present an asynchronous approach closer to what [Lee et al. 1996] accomplished.

Parallel rendering strives for linear speedup -that is as the number of processors increases, so the rendering time decreases by a proportional amount- and many efforts have been devoted to reduce the imbalance. [Samanta et al. 1999], [Samanta et al. 2000], [Whitman 1994], [Neumann 1994] and [Molnar et al. 1994] present the case for static scenes. In our case, load balancing happens in time and not in space as these approaches have done.

Finally, network compositing is a key component for the actual behavior of this work. The binary swap [Ma et al. 1994] is one of the best known compositing algorithms for networks with switching capabilities. Its divide-and-conquer strategy keeps processors as busy as possible during compositing. In this work, we use it for the synchronous rendering scheme.

3 Rendering Schemes

Synchronous rendering is the most commonly used scheme for parallel rendering ([Ma et al. 1994]), but without some balancing mechanism ([Samanta et al. 2000]) it is prone to high imbalances. Synchronous rendering proceeds through the stages of the visualization pipeline in unison; no processor is at a different stage, either every processor is rendering or every processor is performing network operations, such as compositing and gathering. The synchronization is usually achieved by a checkpoint that every processor must reach in order to continue processing. It is easy to see that the slowest processor dictates the performance of this rendering scheme.

Asynchronous rendering aims for overlapping the rendering and network stages. With the partitions in place, asynchronous rendering starts by having every processor render its partitions. When a processor completes a partial frame, it tests the readiness of the processor with which it must composite; if it is the case, then sending of the partial frame proceeds and compositing follows. If not, the processor continues to render the next time step, but it does so in quotas; after each quota the processor returns to see if the partial frame has arrived to start the compositing. The processor that is supposed to send the partial frame keeps rendering until it completes, after which sending starts. Since there is no checkpoint (known in the parallel literature as a global barrier [Pacheco 1997]), it is possible for processors to be in different stages, thus overlapping their computation.

In any scheme, it is typical that before rendering starts, the data is generally preprocessed to take advantage of certain features inside the different parts of a volume; typical operations are formatting the data into octrees or compressing it or both. Then the data is partitioned and shipped to the different processors that participate in the rendering. Figure 1 shows a typical data setup for parallel rendering. The features will also give us the values for the averages presented in the next section.

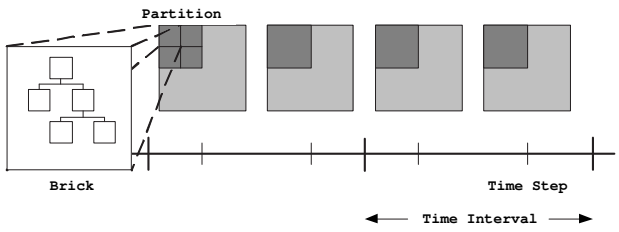

Figure 1: Before the rendering stage, the time-varying volume dataset is subdivided into time intervals. Each time interval contains several time steps, and each is associated to a volume that is formatted for the visualization pipeline. Each volume after preprocessing has a partition for each processor. Inside the partitions, there are bricks of data that are available at different resolutions.

4 Shuffling

Asynchronous rendering improves the performance of the overall visualization pipeline, because it is able to overlap stages; but the load imbalance is still postponed towards the end. If every processor renders always the same partition for every time step, as it happens in a fixed distribution, no gain is achieved by rendering asynchronously; instead we need a smarter distribution for the partitions.

We propose the shuffling distribution, which makes processors render different partitions at different time steps. We do this based on the observation that bricks in a partition do not render equally. For example, a uniform brick, where all elements have the same value and usually represent empty space, is rendered faster than a non-uniform brick. The later kind of bricks represent the visual complexities of the data and rendering is highly dependant upon them. In a partition, calculating how many of non-uniform bricks are rendered on average is our metric for shuffling.

4.1 Metric Function

First, let's consider what happens when rendering a time step. After a preprocessing step, the data is broken up into bricks and each brick has several levels of resolution. The bricks, both in space and in time, are contiguously grouped into partitions. The number of partitions matches the number of processors. Then, at every level, a spatial error estimate [Shen et al. 1999] is calculated to indicate how well the low resolution brick represents its high resolution children. Also, in order to accelerate the rendering, bricks that hardly change during some periods of time ([Shen and Johnson 1995], [Shen et al. 1999]) can be exploited, and thus temporal error estimates are calculated.

When rendering starts, the bricks at the lowest resolution are visited in viewing order. If the spatial error of the brick conforms to the user's tolerance, the brick is selected for rendering; if not, the children are recursively visited until the bricks with original data are reached. Furthermore, it is possible for a brick to be a good enough representation of its following counterparts in time, in which case, the brick is cached and reused, thus avoiding the processing of the brick's data in subsequent time steps.

The next set of equations follows the described procedure for calculating our metric. Equation 1 calculates the number of bricks expected when starting at brick i at time step t. It uses the spatial error of the brick, σ_i^t, as a probability weight to average the number bricks that result from selecting the actual brick and selecting its children.

$$mS_i^t = (1-\sigma_i^t)mT_i^t + \sigma_i^t \sum_{j \in children(i)} mS_j^t \qquad (1)$$

Equation 2 takes into account the case of cached bricks for temporal reuse during dt time steps. The bricks at the beginning of a time interval use their temporal error, ρ_i^t, for averaging the brick and its following counterparts. These bricks make the expression $t\%dt = 0$ in the equation true, where % is the remainder function. Notice that for the other bricks, mT_i^t is simply 1.

$$mT_i^t = 1 + (\rho_i^t \sum_{s=t+1}^{t+dt-1} mS_i^s) * (t\%dt = 0) \qquad (2)$$

At this point, we can calculate the metric for the whole partition. Since a partition contains a set of bricks in space and time, the individual calculations are summed up to find the metric for the whole partition as shown in equation 3. Notice that the calculations always start at the beginning of a time interval (i.e. $dt * t$).

$$metric_{partition}^t = \sum_{i \in partition} mS_i^{dt*t} / \sum_{s=t}^{t+dt-1} tot_i^s \qquad (3)$$

The purpose of the function tot is to normalize the metric with respect to the total number of bricks in a partition. Two partitions can on average render the same number of bricks, but have different total number of bricks; this suggests the scenario where partitions have good low resolution bricks (i.e. $\sigma_i^t = 0$), making them less variant than other partitions. Equation 4 shows the calculation of the function tot:

$$tot_i^t = 1 + (\sigma_i^t \neq 0) \sum_{j \in children(i)} tot_j^t \qquad (4)$$

4.2 Distribution

Once estimated the metrics for every time partition, we can use these them to perform the shuffling. Initially, every processor is assigned a 0 metric. The partitions that belong to the first time interval and their respective metrics are sorted from highest to lowest. Then they are assigned to processors and the metric of every processor is added to the metric of the partition assigned. The resulting tuples (processor, partition, metric) are saved to a table; then, the processors and their metrics are sorted from lowest to highest for the next iteration. The process repeats with the partitions of the next time interval. The table where tuples are stored becomes the tool for distribution. By looking at the sequence of the tuples, one can determine what partitions each processor gets and ship them for rendering.

This sorting ensures that a processor with low average estimation receives a high average estimation for the next time interval, and vice versa. The processors are balanced in terms of these metrics and we used them for our results.

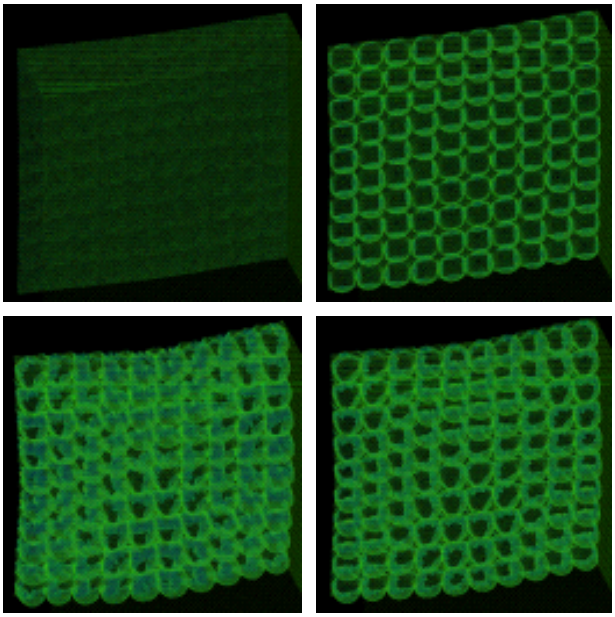

Figure 2: Time steps 0, 11, 21 and 31 of the Ritchmeyer-Meshkov instability dataset. The dataset starts out by showing the wire mesh that divides the mixing of substances; and as time progresses, it gets obscured by the turbulence of the mixing. All images are rendered at a resolution of 512x512 pixels using orthographic projection.

5 Results

We compared the fixed distribution using synchronous rendering against our shuffling distribution using asynchronous rendering. We performed our tests using the Richtmyer-Meshkov instability dataset (the Gordon Bell Prize Winner) with 32 of the original time steps. Each time step has a volume of 2048x2048x1920 8-bit scalar values or voxels, and it is arranged into an multiresolution tree with 2 levels of detail. The trees are comprised of 3D bricks of 128x128x128 voxels, and in order to exploit time coherence, time intervals have a duration of 2 time steps (i.e. $dt = 2$). Examples of the time steps are shown in figure 2.

Under the viewing parameters to generate the images in figure 2 and setting the error tolerances to 10% for both space and time, we performed rendering tests for 8, 16 and 32 processors that counted the number of non-uniform bricks rendered to see how far apart the slowest and fastest processors performed. Tables 1, 2 and 3 show the results.

Distribution	Avg #Bricks	Dev #Bricks	Fastest/Slowest
Fixed	212.50	218.34	15/576
Shuffled	212.50	61.56	133/304

Table 1: Test for 8 processors

Distribution	Avg #Bricks	Dev #Bricks	Fastest/Slowest
Fixed	106.25	105.90	0/288
Shuffled	106.25	32.36	52/168

Table 2: Test for 16 processors

Distribution	Avg #Bricks	Dev #Bricks	Fastest/Slowest
Fixed	53.125	58.25	0/154
Shuffled	53.125	17.12	28/90

Table 3: Test for 32 processors

We immediately see that the shuffled distribution makes an impact. The number of non-uniform bricks overall is lower for the shuffled distribution. The deviation of the shuffled distribution is also better than that of the fixed distribution, which translates into better load balancing. Also the difference between the slowest and the fastest for the shuffled distribution is not as pronounced as that of the fixed.

Another big impact comes in the storage requirements, which is presented in table 4. The shuffled distribution also has a lower deviation that helps to keep a balanced storage, which is important since larger datasets can make the use of fixed distribution impossible. The local disks of processor can have enough space for a partition that is close to the average requirements, but because of storage imbalances, some disks allocate very little, while others allocate too much. Our implementation uses a variant of wavelet compression [Westermann 1995] to help reducing the amount of local disk, which for our tests had 20GB, although the whole space was not always entirely available since the disks belong to a public cluster.

# Processors	Avg #MB	Fixed Dev	Shuffled Dev
8	326.50	338.21	149.40
16	163.25	273.71	49.10
32	81.62	134.80	25.39

Table 4: Storage requirements for different number of processors. Fixed and Shuffled deviations are in MB.

A more detailed view of the actual total distribution per processor is given in figure 3, which shows the total number of non-uniform bricks for 8, 16 and 32 processors. Because partitions with different number of bricks can result in similar metric values, the total distribution is not entirely balanced for either distribution. It's noticed that the concentration of uniform bricks around non-uniform bricks is high. However, as more processors participate, the shuffled distribution stabilizes, while the fixed distribution remains highly imbalanced.

6 Conclusions and Future Work

In this paper we have presented the asynchronous rendering of time-varying volume datasets under a shuffling distribution. We compared our approach against the traditional approach in the literature, the synchronous rendering using the fixed distribution. The asynchronous approach with the shuffled distribution balances the workload better than the fixed one. Using the time dimension asynchronously, processors can render ahead, which is a departure from traditional parallel volume rendering. Since the assignment of workload is based on averages, high concentration of bricks processors have the chance to catch up with low-cost processors in the next time interval. This is due to the shuffling effect when distributing.

Our research will move on to test granularity of the partitions. In our metric estimations we assume that our distributions ship bricks that are contiguous. An alternative will be to distribute bricks that are not necessarily neighbors. We predict that this further helps load balancing for a time step, but requires more bandwidth for compositing.

More rendering parameters are also under study such as: change of transfer function and zooming. The first alters the outcome of the average number of bricks since non-uniform bricks can become transparent given a mapping of colors. The second alters viewing frustum, which throws away bricks outside the final image. Both are handled to some degree in the shuffled distribution, but a more thorough study is required.

References

ATHERTON, T. J., ANAGNOSTOU, K., AND WATERFALL, A. E. 2000. 4d volume rendering with the shear-warp factorization. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, 129–137.

CHUIEH, T. C., AND MA, K. L., 1997. A parallel pipeline renderer for time-varying volume data. NASA/CR-97-206275. ICASE Report No. 97-70.

ELLSWORTH, D., CHIANG, L., AND SHEN, H. W. 2000. Accelerating time-varying hardware volume rendering using tsp trees and colored-based error metrics. In *Proceedings of the 2000 Symposium on Volume Visualization ACM SIGGRAPH*.

GUTHE, S., AND STRASSER, W. 2001. Real-time decompression and visualization of animated volume data. In *IEEE Visualization 2001*.

LAUR, D., AND HANRAHAN, P. 1991. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proceedings of SIGGRAPH 91 ACM SIGGRAPH*, 285–287.

LEE, T. Y., RAGHAVENDRA, C. S., AND NICHOLAS, J. B. 1996. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3, 202–217.

MA, K. L., AND LUM, E. 2002. Strategies for visualizing time-varying volume data. In *Proceedings of the conference on Visualization '02*, 53–60.

MA, K. L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. 1994. Parallel volume rendering using binary-swap image composition. IEEE Computer Society Press, vol. 14, 59–68.

MA, K. L., LUM, E., AND CLYNE, J. 2001. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of the Visualization '01 Conference*, 263–270.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. IEEE Computer Society Press, vol. 14, 23–32.

NEUMANN, U. 1994. Communication costs for parallel volume rendering algorithms. *IEEE Computer Graphics and Applications 14*, 4, 49–58.

PACHECO, P. S. 1997. *Parallel Programming with MPI*. Morgan Kaufmann Publishers.

SAMANTA, R., ZHENG, J., FUNKHOUSER, T., LI, K., AND SINGH, J. P. 1999. Load balancing for multi-projector rendering systems. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 107–116.

SAMANTA, R., FUNKHOUSER, T., LI, K., AND SINGH, J. P. 2000. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware ACM Press*, 99–108.

SHEN, H. W., AND JOHNSON, C. 1995. Differential volume rendering: A fast volume visualization technique for flow animation. In *Proceedings of the Visualization '94 Conference*, 180–187.

SHEN, H. W., CHIANG, L., AND MA, K. L. 1999. A fast volume rendering algorithm for time-varying field using a time-space partitioning (tsp) tree. In *Proceedings of Visualization '99*, "IEEE Computer Society Press".

WESTERMANN, R. 1995. Compression domain rendering of time-resolved volume data. In *In Proceedings of the Visualization '95 Conference*, 168–174.

WHITMAN, S. 1994. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications 14*, 4, 41–48.

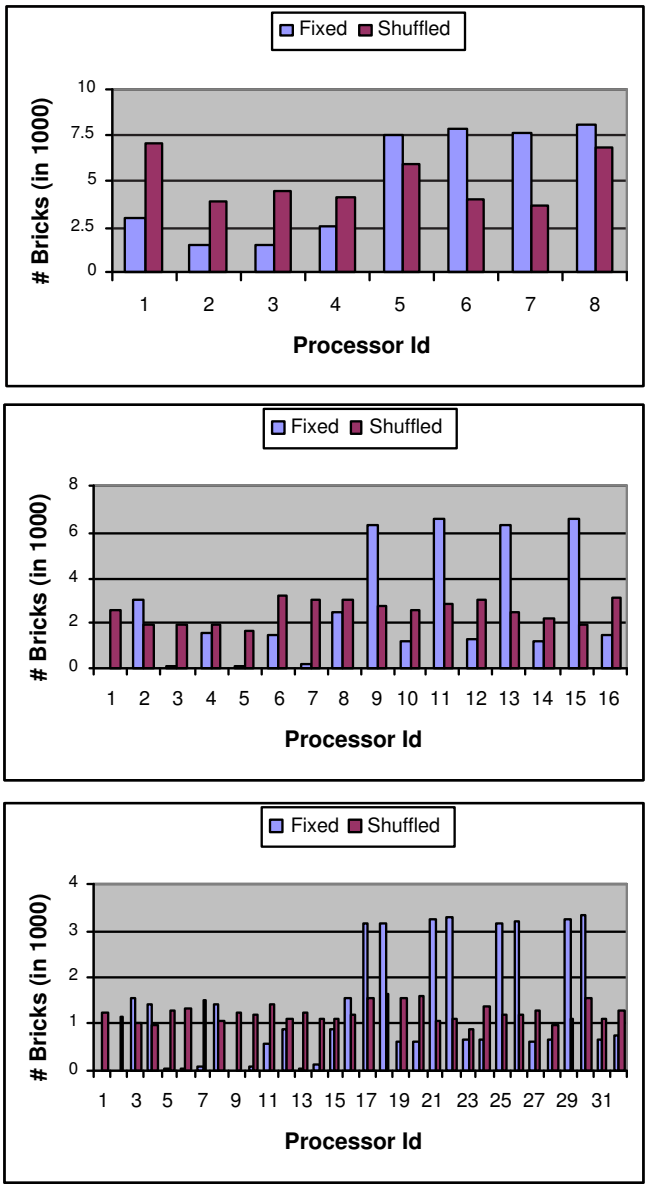

Figure 3: The charts show the total distribution of non-uniform bricks for 8, 16 and 32 processors under fixed and shuffled distributions. Because partitions with different number of bricks can result in similar metric values, the total distribution is not entirely balanced, but as more processors are added it stabilizes. The partitions are from the 32 time steps of the Ritchmyer-Meshkov instability dataset.