Out-of-Core Volume Rendering for Time-Varying Fields Using a Space-Partitioning Time (SPT) Tree

Zhiyan Du Yi-Jen Chiang Han-Wei Shen

Department of Computer Science and Engineering

Technical Report
TR-CSE-2009-01
07/31/2009

Out-of-Core Volume Rendering for Time-Varying Fields Using a Space-Partitioning Time (SPT) Tree*

Zhiyan Du
Polytechnic Institute of New York University

Yi-Jen Chiang
Polytechnic Institute of New York University

Han-Wei Shen
Ohio State University

Abstract

In this paper, we propose a novel out-of-core volume rendering algorithm for large time-varying fields. Exploring *temporal* and *spatial* coherences has been an important direction for speeding up the rendering of time-varying data. Previously, there were techniques that hierarchically partition both the *time* and *space* domains into a data structure so as to *re-use* some results from the previous time step in multiresolution rendering; however, it has not been studied on which domain should be partitioned first to obtain a better re-use rate. We address this open question, and show both *theoretically* and *experimentally* that partitioning the time domain first is better. We call the resulting structure (a binary *time tree* as the primary structure and an *octree* as the secondary structure) the *space-partitioning time (SPT) tree*. Typically, our SPT-tree rendering has a higher level of details, a higher re-use rate, and runs faster. In addition, we devise a novel *cut-finding* algorithm to facilitate efficient out-of-core volume rendering using our SPT tree, we develop a novel *out-of-core preprocessing* algorithm to build our SPT tree I/O-efficiently, and we propose modified error metrics with a *theoretical guarantee* of a *monotonicity* property that is desirable for the tree search. The experiments on datasets as large as 25GB using a PC with only 2GB of RAM demonstrated the efficacy of our new approach.

1 Introduction

The rapid growth of the data size in recent years has made scientific visualization of time-varying datasets a big challenge. The sheer size of the data often makes the task of interactive exploration impossible, as only a small portion of the data can fit into main memory, and the computation cost is often too high for an algorithm to run in real-time. In this paper, we address the issues of limited main memory and insufficient computing speed, by proposing a novel *out-of-core* volume rendering technique for large time-varying fields.

Exploring *temporal coherence* among time steps has been an important direction for speeding up the rendering of time-varying data. This has often been combined with the exploration of *spatial coherence* to facilitate multiresolution rendering. Previously, there were two major techniques that hierarchically partition both the *time* and *space* domains into a data structure so as to *re-use* some rendering results from the previous time step (by *temporal coherence*), in the context of multiresolution rendering: (a) *Finkelstein's tree* [8], which first partitions the time domain by a binary tree (we call it *time tree* in this paper for consistency) as a primary structure, and then for each time-tree node partitions the space domain into a quadtree (in the context of generating multiresolution *videos*), and (b) Shen's *TSP tree* [17] (and the follow-up work [6, 9]), which first partitions the space domain by an octree as the primary structure, and then for each octree node partitions the time domain by a time tree as the secondary structure. Although these two complementary schemes have been proposed for a long time, it has *never* been studied on which domain should be partitioned first to obtain a better re-use rate (the TSP scheme is still being employed and shown to be very effective in a more recent work [9], but this question is still not addressed). In fact, one main reason that the TSP tree chose to partition the space domain first was because volume rendering requires a *consistent breadth cut* through the octree, which is non-trivial if such cut has to go through a collection of secondary octrees (Finkelstein's tree [8] only supports a *fixed* cut through quadtrees (i.e., only for a *fixed* spatial error) and cannot work for *dynamic* error queries in run-time as needed); see [17]. The decision was not based on the re-use rate, however.

In this paper, we take on this line of work and make a novel extension along two *orthogonal* directions: (1) we study the open question of which domain should be partitioned first for a better re-use rate, and choose the better scheme as our data structure; (2) for the chosen data structure, we make it *out-of-core* so that we can perform out-of-core volume rendering, and in addition we develop an out-of-core preprocessing algorithm to build the data structure I/O-efficiently. Note that in the out-of-core setting, a better re-use rate means more savings in the I/O cost (via a better re-use of subvolume textures in hardware volume rendering), which is very important. Also, our out-of-core techniques in (2) can be applied to both partitioning schemes, hence (2) is orthogonal to (1).

For (1), we show, with both *theoretical analysis* of the tree structures and experiments on real datasets, that partitioning the time domain first is better. We call the resulting structure (*time tree* as the primary and *octree* as the secondary structures) the *space-partitioning time (SPT) tree*. It is important to observe that searching on the SPT and the TSP trees for subvolumes satisfying user-specified error tolerances can have different results. Intuitively, since the search is on the primary tree first and then the secondary trees, the SPT tree favors higher-level time-tree nodes (i.e., with *larger time spans* and hence *re-usable for more time steps*), while the TSP tree favors higher-level octree nodes (i.e., larger subvolumes) instead. Therefore the SPT tree has a better re-use rate. Moreover, since the SPT tree tends to select smaller (but more) subvolumes, we typically have a higher level of details, and yet the speed is still faster due to a higher re-use rate. In addition, the structural property of our SPT tree makes it extremely simple to cache subvolumes for future re-use.

As mentioned above, using our SPT tree for volume rendering needs to find a consistent/valid cut through a collection of secondary octrees, which is non-trivial and there was no algorithm before. We devise a novel *cut-finding* algorithm for this task, which exploits the *traversal coherence* among the octrees to optimize the search. In addition, we obtain a further speed-up by combining the *temporal coherence* when traversing the *time-tree* part of our SPT tree for subsequent time steps.

For (2), the original TSP tree can be easily adapted to work in the out-of-core setting in the *rendering* phase (and similarly for the SPT tree), but its *preprocessing* phase has been done *in-core* (i.e., requiring the entire dataset including *all* time steps to reside in main memory) using a brute-force approach. We develop a novel *out-of-*

*Research supported by NSF Grant CCF-0541255. Yi-Jen Chiang is also supported by NSF CAREER Grant CCF-0093373. Author Email: zdu@cis.poly.edu; yjc@poly.edu; hwshen@cse.ohio-state.edu.

core preprocessing algorithm to build our SPT tree, and the same algorithm (with just a very simple mapping) can build the TSP tree as well in the out-of-core setting. Our out-of-core preprocessing algorithm makes a good use of intermediate computing results, and is actually *much faster* than the in-core brute-force approach even when there is enough main memory (see Section 4).

We summarize our technical contributions as follows.

(i) We study the open question of which of the time and space domains should be partitioned first for a better re-use rate. We show both theoretically and experimentally that our SPT tree scheme is better. Typically, our SPT-tree rendering has a higher level of details, a higher re-use rate, and runs faster.
(ii) We devise a novel cut-finding algorithm to facilitate efficient out-of-core volume rendering using our SPT tree.
(iii) We develop a novel out-of-core preprocessing algorithm that can build both our SPT tree and the TSP tree I/O-efficiently. This algorithm is much faster than the original in-core approach even when there is enough main memory.
(iv) We propose modified error metrics and provide a *theoretical guarantee* of a *monotonicity* property that is desirable for both our SPT tree and the TSP tree (see Section 3.1).

Note that our major results are (i)-(iii), and they are *independent* of the underlying error metrics used. As for (iv), although *image-space* error metrics (e.g., those in [6]) can potentially result in a better coherence (by mapping different scalar values to the same color and opacity under a particular transfer function), *data-space* error metrics are still correct and actually more *conservative* (never treating different scalar values as equal). In addition, image-space error metrics need to be re-computed each time the transfer function is changed in the rendering phase, which is expensive for large, out-of-core datasets and does not pay off for the potential rendering speed-up gains. Therefore we opt for data-space error metrics, which are independent of the transfer function and thus we only need to perform out-of-core preprocessing *once*. The experiments on datasets as large as 25GB using a PC with only 2GB of RAM demonstrated the efficacy of our new approach.

2 PREVIOUS WORK

In this section, we review previous work on in-core and out-of-core techniques for volume visualization of time-varying scalar fields. For other out-of-core techniques in graphics and scientific visualization, we refer to the survey by Silva et al. [19].

Exploring data coherence has been an important direction for speeding up the visualization of time-varying fields. Shen and Johnson [18] proposed a differential volume rendering strategy, and Shen [16] utilized temporal coherence for fast isosurface extraction. As mentioned in Section 1, Shen et al. [17] developed the TSP tree to capture spatial and temporal coherences of the data for fast volume rendering, and prior to the TSP tree, *Finkelstein's tree* [8] was proposed in the context of generating multiresolution *videos*. Following up the TSP work, Ellsworth et al. [6] used the TSP tree for hardware volume rendering. More recently, Gao et al. [10] exploited temporal occlusion coherence to speed up volume rendering using visibility culling, and Younesy et al. [24] employed a differential time-histogram table for efficient volume rendering. Other work on the visualization of time-varying fields includes applying compression techniques (e.g., [15] and the references therein), feature tracking [13], parallel algorithms [14], high-dimensional approaches [23], dynamic view selection [12], and ray tracing [21].

The techniques mentioned so far are mainly main-memory approaches. For out-of-core volume visualization, Chiang and Silva [4] and Chiang et al. [5] developed out-of-core isosurface extraction algorithms, and Farias and Silva [7] proposed out-of-core volume rendering methods. Also, Bajaj et al. [1] proposed a parallel and out-of-core isosurface approach based on contour propagation from seed cells. All these out-of-core techniques are for steady-state datasets. As for time-varying fields, Sutton and Hansen [20], Gregorski et al. [11] and Waters et al. [22] developed out-of-core isosurface extraction methods for regular grids, and Chiang [3] developed an out-of-core isosurface approach for irregular grids. Also, Gao et al. [9] employed the TSP tree scheme [17] for distributed parallel volume rendering that addresses some I/O issues such as data caching and prefetching, with the main focus on distributed data management in parallel computing.

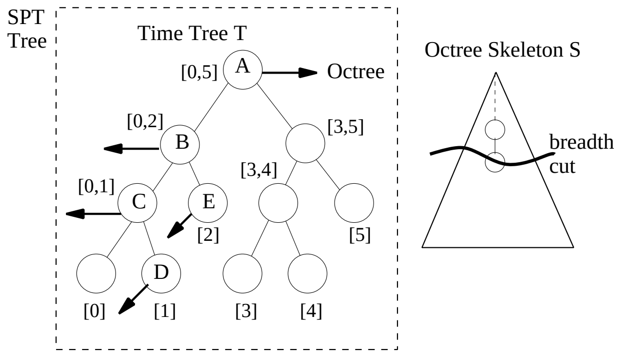

Figure 1: An example of the SPT tree for time interval $[0,5]$. In the time tree T, each internal node labeled $[t_1,t_2]$ covers the time span $[t_1,t_2]$, and each leaf labeled $[t]$ corresponds to time step t. The search path P on T for query $(\varepsilon_s,\varepsilon_t,t')$ with $t'=1$ is $P=(A,B,C,D)$, and P' with $t'=2$ is $P'=(A,B,E)$. At run-time, only T and the octree skeleton S are kept in main memory. We also show a *breadth cut* on S; any node and its ancestor cannot both exist in a valid cut.

3 OUR APPROACH

3.1 The SPT Tree Data Structure

We first give an overview of the SPT tree partitioning scheme, analyze its re-use rate from the tree structure, and then describe our modified error metrics, followed by the out-of-core organization and other details.

The primary structure of the SPT tree is a fully balanced binary tree T called *time tree*. The root of T corresponds to the time interval over the entire time steps, and we recursively partition the time interval into two equal halves for the two subtrees until the time interval becomes a single time step (see Fig. 1).

The secondary structure of the SPT tree is a standard complete octree, which recursively subdivides the input volume spatially until all octree leaves are at the same predefined depth D. We will specify D later.

For each time-tree node u with time span I_u, we have a secondary octree of the same structure (as described above); all such octree nodes have the same time span I_u in the time domain, but they represent different subvolumes in the space domain—the node α representing subvolume V_α means the resulting time-space partition is (I_u,V_α). In each such octree node α, we store both the *spatial error* and the *temporal error* of the partition (I_u,V_α). The spatial (resp. temporal) error serves as an indication of the spatial (resp. temporal) coherence of the subvolume; the lower the value, the higher the coherence. We will give our error metrics later.

In addition to the SPT tree, we have an auxiliary *octree skeleton* S to represent the (same) structure of all secondary octrees. This skeleton S will be used for efficient searches, and is also conceptually useful for analyzing the re-use rate. Now we first address the question of which of the time and space domains should be partitioned first to get a better re-use rate.

3.1.1 Structural Properties: High Re-Use Rate and Simple Caching

As mentioned in Section 1, the TSP tree [17] is similar to the SPT tree but reverses the partitioning order: it uses the octree S to partition the space domain as the primary tree, and then the time tree T to partition the time domain as the secondary tree. Let u be a time-tree node with time span I_u and α be an octree node with subvolume V_α; the time-space partition (I_u, V_α) in the SPT tree and the space-time partition (V_α, I_u) in the TSP tree are *exactly the same*.

In the rendering phase, the user specifies $(t', \varepsilon_s, \varepsilon_t)$ for volume rendering time step t' satisfying spatial and temporal error tolerances ε_s and ε_t respectively. When the user keeps ε_s and ε_t unchanged and only varies t' sequentially in subsequent queries, some previously selected subvolumes may be selected again and thus can be re-used. Clearly, if a selected subvolume has a larger time span, then it can be re-used more.

Although details are different (see Section 3.2), the search algorithms for the SPT and the TSP trees are based on a top-down search, first on the primary tree and then on the secondary tree: For the current primary-tree node, look at its secondary tree to find the *highest* node(s) satisfying $\varepsilon_s, \varepsilon_t$. If such node(s) cannot be found, then go down one level in the primary tree and repeat the process. Note that *higher-level primary-tree nodes* are always preferred, at the expense of choosing lower-level nodes in their corresponding secondary trees if possible. In our SPT tree, we favor higher-level time-tree nodes (with larger time spans), possibly splitting their space domain into more subvolumes of smaller sizes. In the TSP tree, higher-level octree nodes (larger subvolumes) are preferred, possibly splitting their time domain into smaller time spans. This intuitively explains why our SPT tree has a higher re-use rate. Moreover, since we tend to split into more subvolumes, our rendering typically has a higher level of details, and can still be faster due to a higher re-use rate (see Section 4).

Lemma 1: Our SPT tree has a re-use rate at least as good as, and possibly better than, the re-use rate of the TSP tree.

Proof: In order to perform volume rendering, the selected subvolumes of both methods must form a valid *breadth cut* on the octree skeleton S (see Fig. 1). For an octree node α (associated with time-tree node u of time span I_u) in the cut of the TSP tree, we look at how the space of subvolume V_α is covered in the octree cut of SPT. There are three cases.

(1) The octree cut of SPT goes through the same node α. In this case, since the partition (V_α, I_u) of the TSP tree satisfies $\varepsilon_s, \varepsilon_t$, in our SPT tree surely we can choose time-tree node u whose octree node α is satisfied. So our subvolume at least has the same time span I_u and hence the re-use rate is at least as good. (In fact by a symmetric argument, our subvolume has exactly the same time span I_u and hence the same re-use rate.)

(2) The octree cut of SPT goes through a bigger octree node β than α (i.e., an ancestor of α). This is impossible. Suppose β is associated with time-tree node u'. This means that the partition $(V_\beta, I_{u'})$ in TSP tree would satisfy $\varepsilon_s, \varepsilon_t$. Since TSP tree favors bigger octree nodes, β would have been chosen (with time-tree node u'), a contradiction.

(3) The octree cut of SPT goes through smaller octree nodes than α (i.e., descendants of α). This means that at least one of such descendants, say γ, is associated with a time-tree node u' *higher* than u (and thus the time span $I_{u'}$ is larger (i.e., more re-usable) than I_u). Namely, when searching the time tree in SPT, we look at u' before u, and find that u' has an octree node γ that satisfies $\varepsilon_s, \varepsilon_t$ (otherwise, if no such γ exists, then in SPT search we would eventually reach time-tree node u and select the same octree node α, contradicting the case condition). For other descendants of α in question, their time spans are at least as big as I_u, since in the secondary octree of time-tree node u, octree node α already satisfies $\varepsilon_s, \varepsilon_t$ and surely the descendants of α are satisfied as well (because each child error is no larger than the parent error[1]). In summary, there is at least one γ that has a *better* re-use rate, and others have re-use rates at least as good.

Finally, over all possible cases, our re-use rate is always as good, and if some instances of case (3) occur then our re-use rate is better. □

In addition to re-use rate, another advantage of our SPT tree is that it is extremely simple to cache the subvolumes for future re-use. Referring to Fig. 1, the search path on the time tree T for time step $t' = 1$ is (A, B, C, D) and for time step $t' = 2$ is (A, B, E). The two paths *fork* at node B, i.e., they have a *common subpath* (A, B), which is from the beginning up to and including the fork node B. The search results on the secondary octrees are *all the same* (and can be re-used) except *after* the fork node B. Therefore, we can cache the subvolumes in order, and only replace the *last part* of the subvolumes by the new subvolumes that correspond to the new path *after* the fork (e.g, replacing (C, D) by E). Since the replacements always occur at the end, it is extremely simple to cache the subvolumes.

3.1.2 Modified Error Metrics

Now we consider the error metrics, defined for the subvolume V_α over time span I_u, where u and α are time-tree and octree nodes respectively.

As discussed at the end of Section 1, we choose to use data-space error metrics since they are correct, conservative, and *independent* of the transfer function. As mentioned, all other results in this paper are *independent* of the underlying error metrics used.

Our error metrics are modified from those given in [17]. The spatial error metric is the coefficient of variation and can be seen as a normalized version of the standard deviation:

$m = \frac{\sum_{i,\,t} v_{i,\,t}}{N}$, $s = \sqrt{\frac{\sum_{i,\,t} v_{i,\,t}^2}{N} - m^2}$, and

$$\text{Spatial Error} \quad = \quad \frac{1}{8^k}(s/m) \qquad (1)$$

where $v_{i,t}$ is the scalar value of grid point i at time step t, N is the total number of data points in the subvolume V_α across all time steps in the time span I_u, m is the mean value of the data points in question, s is the standard deviation, and finally k means the octree node α in question is at level k of the octree where the root level is 0. The spatial error defined above is always between 0 and 1. This spatial error is the same as that in [17] except for the term $\frac{1}{8^k}$. The TSP/SPT tree search algorithm assumes that the parent error is at least as large as the child error (otherwise when searching top-down to find the node(s) satisfying the specified error tolerance the child would never be chosen). We call this desirable condition the *monotonicity* property. This monotonicity property, however, is *not* guaranteed in the original metric of [17]. With the additional term $\frac{1}{8^k}$, we can now prove that such property is always guaranteed. Intuitively, when we go down one level in the octree, the node is split into 8 children, so there is a factor of 1/8 to the contribution, and hence the term $\frac{1}{8^k}$.

Lemma 2: Our spatial error metric as defined in Eq. (1) satisfies the monotonicity property.

Proof: See Appendix. □

For our temporal error, letting the time span I_u be $[t_1, t_2]$, we have $m(v_i) = \frac{\sum_{t=t_1}^{t=t_2} v_{i,\,t}}{t_2 - t_1 + 1}$, $s(v_i) = \sqrt{\frac{\sum_{t=t_1}^{t=t_2} v_{i,\,t}^2}{t_2 - t_1 + 1} - m(v_i)^2}$, $c(v_i) = s(v_i)/m(v_i)$,

[1]We call this condition the *monotonicity* property. This property is implicitly assumed but not guaranteed in [17]. We fix it by modifying the error metrics and giving a theoretical guarantee; see below.

and

$$\text{Temporal Error} \quad = \quad \frac{1}{8^k}\frac{\sum_i c(v_i)}{n} \qquad (2)$$

where $m(v_i)$ is the mean value of the grid point i over the time span I_u, $s(v_i)$ is the corresponding standard deviation, $c(v_i)$ is the coefficient of variation of i , n is the total number of grid points in the subvolume V_α, and finally k means that the node α in question is at level k of the octree. The temporal error defined above is always between 0 and 1. Again, this temporal error is the same as that in [17] except for the term $\frac{1}{8^k}$ for the same reason, and we can prove that with this term the monotonicity property is always guaranteed. The intuition to add this term is the same as before.

Lemma 3: Our temporal error metric as defined in Eq. (2) satisfies the monotonicity property.

Proof: See Appendix. □

3.1.3 Out-of-Core Organization and Other Details

Now we describe the out-of-core organization and other details of our SPT tree data structure. There are two parts: (1) the SPT tree itself, and (2) the data of simplified subvolumes associated with the time-space partitions induced by the SPT tree.

Recall that the primary tree of the SPT tree is the time tree T. We assume that the entire time tree T can fit in main memory, which is not a restriction since typically the number of time steps in large-scale time-varying datasets is just in the order of tens of thousands.

For the secondary octrees, recall that each of them recursively subdivides the input volume until all octree leaves are at the same predefined depth D. We predefine the parameter D according to the available main memory size, so that the skeleton of a single octree can fit in main memory. We assume that for a single time step, the input grid points in a single leaf subvolume can all fit in main memory. Again this is a reasonable assumption: suppose the main memory can fit $O(M)$ items (e.g., $O(M)$ grid points (in a leaf subvolume) or an octree skeleton of $O(M)$ nodes/leaves), then this means that we can handle datasets with $O(M^2)$ grid points in the input. Even for a main memory of size 128MB, M^2 is in the order of 10^{13}–10^{15}, showing that this assumption is clearly not restrictive.

For each secondary octree node α corresponding to the time-space partition (I_u, V_α), we store its spatial and temporal errors given above. In addition, we also store a pointer to the *simplified grid* that represents the data, which is a $k \times k \times k$ grid obtained by down-sampling the subvolume V_α. For each such grid point p, we let its scalar value be the *average scalar value* of p over the time span I_u. This simplified grid will be used for multiresolution volume rendering if node α satisfies the queried error tolerances and is selected for rendering. To support empty-space skipping during rendering, we record at node α the min, max scalar values of this simplified grid. If node α is an octree *leaf* associated with a time-tree *leaf* (a single time step), we additionally store a pointer to the *original grid* G_ℓ which is the subvolume V_α of the original input grid. We use G_ℓ when a full resolution rendering of V_α is needed. Again we record at α the min, max values of G_ℓ.

Finally, recall that we have an auxiliary octree skeleton S in addition to the SPT tree. In summary, the time tree T of the SPT tree represents the partition of the time domain, and the octree skeleton S represents the partition of the space domain. At run time we only keep T and S in main memory, and other structures are kept in disk and read to main memory when needed.

3.2 Run-Time Volume Rendering Using the SPT Tree

We now describe our run-time volume rendering technique. We start by reading the time tree T and the octree skeleton S to main memory. The user specifies $(t', \varepsilon_s, \varepsilon_t)$ for rendering time step t' satisfying spatial and temporal error tolerances ε_s and ε_t. When the user keeps ε_s and ε_t unchanged and only varies t' in a series of queries, as typically the case, we can take advantage of the coherence and speed up the rendering. We remark that in main memory we have a place holder for a *single* subvolume only. Each new subvolume needed will be read from disk to this place holder and then cached in the texture memory of GPU. We refer to such texture cache as the *GPU buffer* $\mathscr{B}$. Note that using the programmable GPU, changing the transfer function only requires us to re-load the 1D texture for the transfer function [2], while the cached subvolume textures can still be re-used in hardware volume rendering.

3.2.1 The Cut-Finding Algorithm

Performing volume rendering using our SPT tree essentially performs a *breadth cut* of the underlying octree (see Fig. 1) so that the octree nodes in the cut collectively cover the entire volume. These octree nodes α in the cut may come from different secondary octrees S_u of different time-tree nodes u, where each node α satisfies both ε_s and ε_t. There are three major tasks for each volume rendering query. First, we find the appropriate octree nodes α in the cut. In the process, for each such α, if its subvolume V_α is not an empty space (checked by the min, max values with the transfer function) and has not been cached, we read V_α from disk to main memory and cache it in the GPU buffer $\mathscr{B}$. Finally, we perform a standard hardware volume rendering using texture mapping on octree subvolumes, where the visibility sorting of the subvolumes is easily done by just sorting their octree-node IDs. At any time, we only cache the subvolumes of the most recent query.

The key task is to find the octree nodes α in the cut. Our goals are the following. First, we want to find α whose corresponding time-tree node is as high as possible, so that the subvolume V_α can be re-used as much as possible. Secondly, we want α itself to be as high as possible in the octree, so that we use the most simplified subvolume possible to speed up the rendering. To achieve these goals, we first search on the time tree T and find a root-to-leaf path P such that each node on P has its time span containing the query time step t' (see Fig. 1, for example $P = (A, B, C, D)$ for $t' = 1$). Next, we process the nodes of P one at a time starting from the root, where we say that processing one node of P is one *round*. For each current-round node u, we load its secondary octree S_u from disk to main memory to identify the cut nodes in S_u. Recall that S_u has spatial and temporal errors for each node. We perform the actual cut-finding on the *octree skeleton* S, which records the global cut-finding progress from different secondary octrees S_u, so that at the end we complete a breadth cut in S.

We now discuss the main idea in this cut-finding process. Naively, in each round we might want to find the highest nodes in the octree satisfying both $\varepsilon_s, \varepsilon_t$. However, this does not work, since we must respect the cut nodes found in the *previous* rounds to form a *valid* cut at the end. For example, suppose in the first round (for the root r of T) we already identify some cut nodes α in the secondary octree S_r (and the corresponding nodes α in the skeleton S). In the next round, it is possible that an ancestor $a(\alpha)$ of α satisfies both $\varepsilon_s, \varepsilon_t$, since now the time span is shorter. However, any node and its ancestor cannot simultaneously exist in a valid cut (see Fig. 1). Clearly, α is *already* a cut node and has a *priority* over $a(\alpha)$, and this is the key property: as soon as α becomes a cut node, all its ancestors are ruled out from being a cut node in the future rounds. In other words, a node β can be a candidate cut node in the next round only if so far *no descendants* of β satisfy both $\varepsilon_s, \varepsilon_t$. Since we want to find the highest possible cut, the next round should start from the *highest* such candidate nodes β in S.

At the end of each round corresponding to node w in P, we create two lists for w: (a) the *cut list CL*, maintaining the cut nodes found in this round, and (b) the *next-round starting list NR*, maintaining the highest nodes β mentioned above so that the next round starts from each node in this list. We use a marking scheme to mark the nodes of S, with three types: (i) "cut", meaning that the node satisfies both $\varepsilon_s, \varepsilon_t$ and is a cut node found; (ii) "not candidate", meaning that the node has a "cut" descendant and cannot be a candidate cut node in the next round; (iii) "candidate", meaning that this node has no descendant satisfying both $\varepsilon_s, \varepsilon_t$ and hence is a candidate node β for the next-round cut nodes; the highest such nodes will be put to the list *NR*.

In the initial round for the root r of T, we perform **Algorithm Find_Cut** below on nodes of S recursively starting from the *root* of S. We describe **Find_Cut (s)** for a generic node s of S:

Algorithm Find_Cut (s)

```
0. Unmark $s$.
1. If $s$ satisfies both $\varepsilon_s, \varepsilon_t$, mark $s$ "cut", put $s$ to the cut list CL of
   the current time-tree node and return.
2. Otherwise, $s$ does not satisfy both error tolerances. If $s$ is a leaf,
   mark $s$ "candidate" and return; otherwise, perform Find_Cut recursively
   on each child of $s$. When these recursions return, distinguish the following cases.
   Case i All children of $s$ are marked "candidate": this means that all
      children are the $\beta$ nodes, and thus $s$ is a $\beta$ node as well. Therefore
      we mark $s$ "candidate" and return. Observe how the "candidate"
      mark is propagated in a bottom-up fashion in the entire recursive process.
   Case ii At least one child of $s$ is marked "cut" or "not candidate":
      this means that $s$ has a "cut" descendant and thus $s$ is ruled out from
      being a candidate cut-node in the next round. Therefore we mark
      $s$ "not candidate". Note that the "not candidate" mark is eventually
      propagated bottom up for all ancestors of a "cut" node in the recursive
      process. In addition, any child of $s$ marked "candidate" must
      be the highest candidate now, since $s$ and all ancestors of $s$ are "not
      candidate". Therefore, we put each "candidate" child of $s$ to the
      next-round starting list NR of the current time-tree node and return.
```

For the next round corresponding to node u on P, we start by applying **Find_Cut** recursively on *each node* in the list *NR* of r, and create the two lists of u. In the yet next round, we apply **Find_Cut** on each node in the list *NR* of u, and so on. Finally, in the last round (for a leaf time-tree node), we have to complete a breadth cut on S. In the **Find_Cut** process of this round, in case a leaf ℓ of S is reached but ℓ still does not satisfy the two error bounds, we put ℓ as a cut node but will instead use its *original grid* G_ℓ for the volume rendering, which has zero errors and surely satisfies the error bounds. Except for this special case, for each cut node we use its simplified $k \times k \times k$ grid for the rendering.

In the above cut-finding process, as soon as an octree node α is found as a cut node, if its corresponding grid is not an empty space (checked by the min, max scalar values with the transfer function) and is not already cached in the GPU buffer $\mathscr{B}$, we load this subvolume grid from disk to main memory and cache it in $\mathscr{B}$. We will see later that due to search-path coherences in our approach, we actually do *not* need to check whether a subvolume has been cached or not. The caching is done sequentially, putting the new subvolume to the next available place in $\mathscr{B}$. The resulting effect is that the subvolumes are cached in $\mathscr{B}$ in groups, each group for a node in path P. For example, in Fig. 1, path P for $t' = 1$ is (A,B,C,D). Then the subvolumes discovered for A are in the first group, those discovered for B are in the second group, and so on. We maintain the current path P, and for each node in P we maintain a pointer to the starting position of this group in the GPU buffer $\mathscr{B}$.

3.2.2 Re-Using Subvolumes by Search-Path Coherences

The major advantage of our approach is the *re-use* of the subvolumes. Typically, the search paths P and P' on the time tree T for two consecutive time steps have a long *common subpath* at the beginning. For example, in Fig. 1 P for $t' = 1$ is (A,B,C,D) and P' for $t' = 2$ is (A,B,E). The two paths *fork* at node B, with a common subpath (A,B). It is easy to see that the search results on the secondary octrees will be *exactly the same* (and thus can be *re-used*) for the common subpath (A,B), and we only need to replace the part *after* the fork node B (e.g., replacing (C,D) with E). Now all we need is to update the cut starting from the first node w *after* the fork (w is node E in our example). This is essentially to *resume* the above cut-finding process starting from the round of w. Observe that our scheme readily supports this task: we now apply Algorithm **Find_Cut** on each node in the *next-round starting list NR* of the fork node, the node immediately before w. We remark that at step 0 of **Find_Cut** we first unmark each node of S visited, which serves to initialize the marking of the nodes in S as needed.

Finally, since the common (re-used) subvolumes all appear at the beginning of the GPU buffer $\mathscr{B}$, and the new subvolumes all appear after the fork-node group, it is extremely simple to cache/replace the subvolumes.

3.3 Out-of-Core Preprocessing

We present our out-of-core preprocessing algorithm for building the SPT tree. The computation is highly non-trivial in the out-of-core setting, especially in computing the spatial and temporal errors defined in Eqs. (1) and (2). We develop the *slice accumulation* algorithm for computing these errors, and the *slice distribution* algorithm for computing the simplified scalar data for the simplified grids of the secondary octree nodes. Although our discussion of the *slice accumulation* algorithm is based on the errors defined in Eqs. (1) and (2), the main theme is to re-order the computation and data so that the data values are available when needed for computing in the out-of-core setting, and hence our algorithm can be easily adapted for other error definitions. Moreover, observe that each secondary-tree node of the TSP tree is uniquely identified by $(\text{octree\_ID}, \text{time-tree\_ID})$, which is exactly the secondary-tree node $(\text{time-tree\_ID}, \text{octree\_ID})$ of our SPT tree. Thus the same out-of-core preprocessing algorithm can also compute the TSP tree by just performing an additional simple mapping step at the end.

We now describe how to compute the errors of Eqs. (1), (2). The main task is to compute the sum and the sum of squares in these equations. Typically the input dataset is organized by groups of increasing time steps, one file per time step, where in each such file the grid-point scalar values are given in slices of increasing z-coordinates. First we create a scratch file for each node of the time tree T as follows. Starting from the leaf level, at each leaf (a single time step t_i) we create a scratch file by augmenting the input file of t_i such that each grid-point scalar value f is replaced by (f, f^2). In the next level up, for each internal node u with two children, we create a scratch file of u by summing the corresponding data values from the two child scratch files. Namely, if for the same grid point p its data values in the two children are (f_1, f_1^2) and (f_2, f_2^2), then the data values of p in the file of u are $(f_1 + f_2, f_1^2 + f_2^2)$. Note that the grid points appear in the same order for all scratch files, so that this step can be easily done by simultaneously scanning through the two child files. This process is repeated level by level up to the root of T. Moreover, as soon as the file F_u for a node u has been used to create its parent file, we replace each tuple (f, f^2) by $(f/|I_u|, f^2/|I_u|)$ in F_u, where $|I_u|$ is the number of time steps in the time span of u. Comparing with Eq. (2), we see that $f/|I_u|$ is $m(v_i)$ and $f^2/|I_u|$ is ready for use to compute $s(v_i)$, for each *individual* grid point $p = v_i$, using the scratch file F_u for each node u.

To complete the computation for Eqs. (1) and (2), what we need is to distribute the appropriate grid points to the subvolumes defined by the secondary octree (and accumulate the suitable data values of these grid points within the subvolumes). Specifically, for each time-tree node u, we use its scratch file F_u to compute the errors of Eqs. (1) and (2) for each subvolume of its secondary octree S_u using the following *slice accumulation* algorithm. We repeat the process

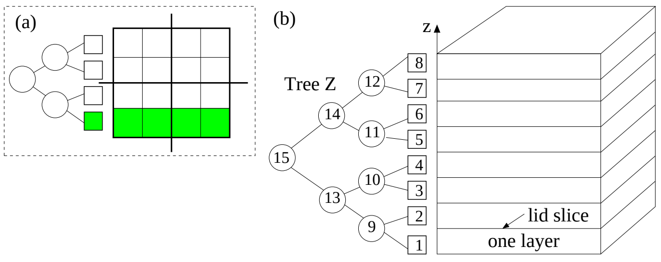

Figure 2: Illustration for the *slice accumulation* algorithm. The (conceptual) tree Z is a binary tree obtained by viewing the octree in the z-dimension only. (a) A 2D illustration, where we replace the octree with a quadtree. The green leaf of the binary tree corresponds to the union of the four green quadtree leaves. (b) An example of tree Z.

for each time-tree node u one by one; since the underlying octree structures are all the same, the same *slice accumulation* algorithm is used for all of them.

3.3.1 The Slice Accumulation Algorithm

Now we describe the *slice accumulation* algorithm for octree S_u associated with a generic time-tree node u. Recall that the file F_u organizes the grid points in slices of increasing z-values. The octree S_u, when viewed just in the z-dimension, is a fully balanced *binary* tree Z on the z-dimension (see Fig. 2(a)(b)). Each leaf of Z corresponds to a *layer*, which is the union of the subvolumes of the octree leaves having the same z-span (see the green leaves in Fig. 2(a)).

We will read from F_u to main memory once slice at a time to an *input-slice buffer*, and for each level of tree Z we also have a *slice buffer* in main memory to store one slice. Note that tree Z has the same height as the octree, and thus our main memory requirement is just a small, fixed number (e.g., 5 in our experiments) of slices. Intuitively, the algorithm proceeds as follows. We load one slice at a time in the order of increasing z. For the current layer, we maintain its *accumulation slice AC* so far. Initially, *AC* is just the bottom slice of this layer. When a new slice comes in, it is "squashed" into *AC* by adding the data values of each grid point to those of *AC* having the same (x,y) coordinates. When the current layer is finished, the corresponding leaf of Z is ready, and we move on to the next layer. When both children of a node of Z are ready, we make this node ready as well by squashing the two *AC*'s from the children in the same way. For example, in Fig. 2(b), the nodes of Z are ready in the following order: 1, 2, 9, 3, 4, 10, 13, 5, 6, 11, 7, 8, 12, 14, 15. When a node τ of Z is ready, we can compute the errors of Eqs. (1), (2) for all the octree nodes corresponding to τ: for each such octree node, we take its 2D (x,y)-range R from the *AC* slice of τ, and use the data values of the points in R to finish computing the two errors, which now can be done very easily.

There are still some technical details, to be discussed next. Overall, each slice is read only once and the results are written out once, which is I/O-optimal.

Additional Technical Details
There are still some technical details that we need to address. Consider the common slice between layers 1 and 2 (i.e., between nodes 1 and 2 of tree Z) in Fig. 2(b)—by the octree partitioning scheme, this slice is included in *both* layers 1 and 2 (and thus is duplicated) so that both layers 1 and 2 are complete. By the above method, this slice is added to both nodes 1 and 2 of tree Z in Fig. 2(b), which is correct, but in node 9, this slice is added *twice*, which is incorrect. In fact, *every* common slice between two layers creates such problem. To fix this, we introduce the notion of *no-lid AC* for a node τ of tree Z: it is the *AC* obtained by squashing all slices of τ except for the last (i.e., topmost) slice; we call such topmost slice the *lid slice* (see Fig. 2(b)). As soon as τ obtains its no-lid *AC*, we propagate this no-lid *AC* to its parent $p(\tau)$. At $p(\tau)$, if there is no *AC* yet, then the propagation stops there (e.g., propagating the no-lid *AC* of node 1 to node 9); if there is already a propagated no-lid *AC* at $p(\tau)$, then the two propagated no-lid *AC*'s at $p(\tau)$ are squashed together to become the no-lid *AC* of $p(\tau)$, and this triggers the propagation of the no-lid *AC* of $p(\tau)$ to its own parent $p(p(\tau))$ recursively. For example, when we propagate from node 2 to node 9, since there is already a propagated no-lid *AC* at node 9 (from node 1), this results in forming the no-lid *AC* of node 9, which in turn is propagated to node 13; propagating from node 4 goes all the way through nodes 10, 13 to node 15.

Finally, we describe how to deal with a lid slice. Each lid slice can be viewed as the separating boundary between two leaves of tree Z, and this separating boundary uniquely corresponds to an internal node of Z—the *least common ancestor* of the two leaves being separated. For example, the lid slice between nodes 1, 2 corresponds to node 9, the lid slice between nodes 2, 3 corresponds to node 13, and the lid slice between nodes 4, 5 corresponds to node 15. We call such internal node the *lid node*. Now when the lid slice comes in, we keep it in the input-slice buffer, and use it to "close up" all the no-lid *AC*'s from the current leaf to its ancestors all the way up before reaching the lid node (i.e., *not* including the lid node), where "closing up" means adding the lid slice to the no-lid *AC* to complete that *AC* and make that node ready. After we use the completed *AC* of the ready node to finish computing the errors of Eqs. (1), (2) for the corresponding octree nodes as described above, we clear this slice buffer so that it is ready to be used for the next node at the same level of tree Z. At the leaf level, we start the next layer—the upper of the two separated layers, and put the lid slice as the bottom slice of this new layer. For example, the lid slice between nodes 1, 2 closes up node 1 and becomes the bottom slice of node 2; the lid slice between nodes 4, 5 closes up nodes 4, 10, 13 and becomes the bottom slice of node 5. The closing-up path is actually easy to compute: it is exactly the no-lid *AC* propagation path except the last node, which is the lid node (e.g., compare the propagation path consisting of nodes 4, 10, 13, 15); we can just follow the leaf-to-root path until we reach a node that we *cannot close up*, i.e., whose *AC* is *not yet* the no-lid *AC* (and this is the *lid node*). It is easy to verify that now each slice is added *exactly once* at each node of tree Z containing that slice, and hence the algorithm is correct. Moreover, we can see that each slice is read only once and the results are written out once, which is I/O-optimal.

3.3.2 The Slice Distribution Algorithm

Now we describe how to compute the simplified scalar data for the simplified $k \times k \times k$ grids of the secondary octree nodes. For each time-tree node u, we use its scratch file F_u to compute for its secondary octree S_u using the following *slice distribution* algorithm. Recall that in F_u we have for each grid point $p = v_i$ its average scalar value $m(v_i)$ over the time span of u, which is all we need from F_u. The *slice distribution* algorithm works in a manner similar to the *slice accumulation* algorithm, but is much simpler. Now we use a *layer buffer* in main memory big enough to hold all slices of just *one* layer. Instead of "squishing" the slices read, we keep the slices in this buffer until all slices of the current layer are available. Then we distribute the current-layer slices to the octree leaves belonging to this layer. This gives the original grid G_ℓ for each such octree leaf ℓ, and we take sub-samples to obtain the simplified $k \times k \times k$ grid. If u is a time-tree leaf, we store both grids to disk; otherwise we only store the simplified grid. Note that only the octree leaves of the *current layer* are active. We repeat this process for each layer; after all layers are done, we have completed the task for all octree leaves. We then work on each octree internal node by merging the simplified grids from its eight children and take sub-samples to obtain its own simplified grid, in a bottom-up, level-by-level fashion.

Data	# time steps	Dimensions	Size
Jets	200	128x128x128	1.56GB
Turb	150	104x129x129	990MB
Turb2-10	10	413x513x513	4GB
Turb2-30	30	413x513x513	12GB
TComb	122	480x720x120	19GB
Jets2	50	509x509x509	25GB

Table 1: Statistics of our test datasets.

Data	Turb2-30 (12GB)	TComb (19GB)	Jets2 (25GB)
SPT tree size	7MB	30MB	12MB
Original grids G_ℓ	13.5GB	24.4GB	27GB
Simplified grids	8.3GB	16.3GB	17GB
Total size	21.8GB	40.8GB	44GB
Size increase	82 %	115%	63%
Disk scratch space	3.5GB	2GB	5GB
SA memory footprint	38MB	62MB	45MB
Simp. memory footprint	150MB	73MB	182MB
SA time	2297s	3686s	4709s
Simp. time	660s	1057s	1257s
Total time	2957s	4743s	5967s

Table 2: Preprocessing results. The upper table shows the space statistics of the resulting data structure in disk. The lower table shows the execution performance of the preprocessing. The underlying octree has 5 levels (including the root). The dimensions of the simplified grids are: Turb2-30: 14x17x17, TComb: 16x24x5, and Jets2: 17x17x17.

Since we do this one node at a time, the main memory requirement is very small.

4 Results

We have implemented our technique in C/C++ and ran our experiments on a Dell Precision PC with 2GB of RAM, two 3GHz Intel Xeon CPUs, Nvidia Quadro FX 4500 graphics (512MB graphics memory), and 300GB SCSI 10K rpm disk, running under RedHat Enterprise 64bit Linux OS. The datasets we tested are listed in Table 1, where a pair such as (Jets, Jets2) means they correspond to the same volume data but sampled at different resolutions and taken with different numbers of time steps; Turb2-10 and Turb2-30 only differ in the number of time steps. Our main focus was on experimenting with the three largest datasets (12GB–25GB); the smaller datasets were only used to compare with in-core approaches.

Preprocessing
We ran our out-of-core preprocessing algorithm and built the SPT tree; the results are shown in Table 2. In the upper table we show the space statistics of the resulting data structure in disk. We see that the SPT tree itself is very small, and the total size increase ranges from 63% to 115%, showing that our data structure is very space efficient. In the lower table of Table 2, we show the execution performance of our algorithm, where SA means our *slice accumulation* algorithm, and Simp. means our *slice distribution* algorithm. The disk scratch space refers to all the scratch files F_u. It can be seen that such scratch space is small, and that both SA and Simp. have very small memory footprint (at most 182MB), making them very effective in the out-of-core setting. The total preprocessing time is quite fast, for example processing a 25GB dataset in 99.45 minutes (5967s). Recall from Section 3.3 that our preprocessing algorithm can also build the TSP tree by a simple mapping. In the experiments we also built the TSP tree out-of-core, which had the *same* run-time and space statistics as in Table 2.

Data	Jets	Turb	Turb2-10
SA time	345s	205s	765s
Simp. time	153s	81s	211s
Total time	498s	286s	976s
SA-MM time	217s	132s	No VM
BF-MM time	3880s	2249s	26.65h

Table 3: Preprocessing time comparison with in-core approaches. “No VM” means not enough virtual memory.

To study the effectiveness of SA, we also implemented two other methods for the same task: SA-MM, which is the same as our SA algorithm but performs all tasks *in main memory* instead, and BF-MM, which is the brute-force approach of directly applying Eqs. (1), (2) in main memory—so far this has been the method for the TSP tree. We compared our algorithm with SA-MM and BF-MM on the three smaller datasets; the results are shown in Table 3. It is interesting to see that BF-MM is quite inefficient due to repeated computations; it was the slowest, and in fact much slower than SA even when there was enough main memory (3880s vs. 345s for Jets and 2249s vs. 205s for Turb) albeit SA payed extra I/O costs. SA-MM was the fastest when there was enough main memory, but for the larger dataset (Turb2-10) it ran out of virtual memory (it needed 8.91GB of virtual memory) and could not finish. Comparing SA with BF-MM on Turb2-10 (when there was not enough main memory), we see that SA made a huge improvement from 26.65 hours to 12.75 minutes (765s)!

Run-Time Rendering
To study the re-use rate in practice, and to see how the re-use rate reflects the real running time, we would like to compare out-of-core volume rendering using both the SPT-tree and the TSP-tree schemes. Therefore we have also implemented another volume rendering approach, which uses the same out-of-core organization but replaces our SPT tree with the TSP tree. As mentioned, we used our preprocessing algorithm to build both data structures out-of-core. We remark that conceptually we treat the simplified grids to have equal dimensions $k \times k \times k$. But in our implementation we used the OpenGL shading language to handle texture mapping so that we can deal with subvolume textures of unequal dimensions easily (the dimensions are shown in the caption of Table 2).

We performed out-of-core volume rendering using both trees on the largest three datasets. For each set of $\varepsilon_s, \varepsilon_t$, we always rendered every time step from the beginning to the end. The results are listed in Table 4, and some representative images are shown in Fig. 3.

In Table 4, each average is taken over all time steps. The “cut size” means the number of subvolumes in the breadth cut of the octree. Since each subvolume is a grid of the same dimensions, more subvolumes in the cut means the rendering is at a higher resolution. Thus “cut size” gives a *quantitative indication* of the rendering resolution, the larger the higher. Also, “re-use rate” means the fraction of the subvolumes in the current cut that are also in the previous cut, *regardless* of whether the subvolumes are empty space or not. This coincides with our concept of the re-use rate discussed in Section 3.1, which only depends on the *structural properties* of the trees and does *not* depend on the transfer function. The “load rate” means the fraction of the subvolumes in the cut that actually need to be loaded from disk, i.e., they are *neither cached nor empty space*. Note that “load rate” times “cut size” gives the number of subvolume I/Os, and thus is an indication of the running time (the higher number, the slower).[2] It is important to observe that “re-use

[2]The actual volume rendering was very fast, less than 1% of the running

Turb2-30	$\varepsilon_s = 0.00002$		$\varepsilon_s = 0.00001$	
Err. in query	$\varepsilon_t = 0.000001$		$\varepsilon_t = 0.0000008$	
	SPT	TSP	SPT	TSP
Avg cut size	1873	338	2012	428
Avg re-use rate	90.9%	39.2%	89.9%	41.6%
Avg load rate	7.3%	44%	7.6%	42.9%
Avg time	0.59s	0.75s	0.75s	0.97s
Total time	17.8s	19.5s	22.5s	29.1s
TComb	$\varepsilon_s = 0.005$		$\varepsilon_s = 0.002$	
Err. in query	$\varepsilon_t = 0.00002$		$\varepsilon_t = 0.00001$	
	SPT	TSP	SPT	TSP
Avg cut size	1879	1360	3182	2958
Avg re-use rate	16.3%	7.3%	9.4%	7.7%
Avg load rate	21%	30%	18.9%	21.2%
Avg time	0.25s	0.33s	0.4s	0.53s
Total time	30.7s	40.26s	48.9s	65.88s
Jets2	$\varepsilon_s = 0.00002$		$\varepsilon_s = 0.00001$	
Err. in query	$\varepsilon_t = 0.0000002$		$\varepsilon_t = 0.0000001$	
	SPT	TSP	SPT	TSP
Avg cut size	3732	407	4096	863
Avg re-use rate	96.8%	70.5%	89.9%	71.6%
Avg load rate	1.6%	18.5%	2.76%	15.8%
Avg time	0.25s	0.32s	0.38s	0.61s
Total time	12.5s	16.1s	19.1s	30.7s

Table 4: Run-time statistics of our SPT tree and the TSP tree techniques. Note that "re-use rate" plus "load rate" is *not* necessarily 100% due to the empty-space effect; see text.

rate" plus "load rate" is *not* necessarily 100% because of the empty-space effect: for example, there might be *new* empty spaces not in the previous cut (thus cannot be "re-used") and yet they need not be loaded.

From Table 4, we see that our SPT tree always had a larger cut size, and always had a higher re-use rate; the differences were typically high (e.g., 90.9% vs. 39.2% for Turb2-30, left part). This confirms with our theoretical analysis of Lemma in Section 3.1, i.e., our SPT tree tends to select smaller subvolumes with larger time spans, resulting in a higher rendering resolution and a higher re-use rate. We also see that our SPT tree always had a smaller load rate, typically with a big difference too (e.g., 7.3% vs. 44% for Turb2-30 left, and 1.6% vs. 18.5% for Jets2 left). We observed that our SPT tree, in favor of selecting more, smaller subvolumes, resulted in a more refined selection that could also capture empty spaces better and skip them. This, combined with a higher re-use rate, made the load rate even better. Therefore, even though our cut size was larger, the smaller load rate made our actual I/O cost smaller, and hence a faster running time, as can be seen in Table 4. In summary, our SPT tree typically resulted in a higher rendering resolution, higher re-use rate, smaller load rate, and faster running time.

We also compared with a basic in-core approach which for each query loads the scalar values of one single time step of the entire volume from disk to main memory and performs volume rendering using the same rendering engine. We found that it needed 15s *per time step* for Jets2,[3] which was the best case for the approach since one single time step volume could still entirely fit in main memory (otherwise it would have been much slower). Still, this is significantly slower compared to our interactive frame rates of always much less than 1s in the average times shown in Table 4. We remark that for both SPT and TSP methods, the memory footprint was no more than 100MB, showing the efficacy of the techniques for out-of-core volume rendering.

time and hence negligible. Also, since we used hardware texture mapping, the volume rendering time was independent of the graphics window size.

[3]A 512^3 texture was our graphics-hardware limit. Turb2-30 and TComb exceeded this limit and needed bricking, which would need a preprocessing.

5 Conclusions

We have presented a novel out-of-core volume rendering algorithm for large time-varying datasets using the SPT tree. We address the open question of which of the time and space domains should be partitioned first to obtain a better re-use rate, both in theory and in practice. We have developed a novel *cut-finding* algorithm to facilitate out-of-core volume rendering with the SPT tree, proposed modified error metrics with a theoretical guarantee of a monotonicity property, and devised a novel out-of-core preprocessing technique that can build both our SPT and the TSP trees I/O-efficiently. Compared with the existing in-core brute-force approach, our algorithm is much faster even when there is enough main memory, and achieves a huge speed-up when there is not enough main memory.

We believe that our new techniques such as the *slice accumulation*, the *slice distribution*, and the *cut-finding* algorithms are quite general, and might be useful for other out-of-core computations.

References

[1] C. Bajaj, V. Pascucci, D. Thompson, and X. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proc. Sympos. Parallel Visualization and Graphics*, pages 97–104, 1999.

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

[3] Y.-J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proc. IEEE Visualization*, pages 217–224, 2003.

[4] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[5] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, pages 167–174, 1998.

[6] D. Ellsworth, L.-J. Chiang, and H.-W. Shen. Accelerating time-varying hardware volume rendering using TSP trees and color-based error metrics. In *Proc. Sympos. Volume Visualization*, pages 119–128, 2000.

[7] R. Farias and C. Silva. Out-of-core rendering of large unstructured grids. *IEEE Computer Graphics & Applications*, 21(4):42–51, 2001.

[8] A. Finkelstein, C.E. Jacobs, and D.H. Salesin. Multiresolution video. In *Proc. ACM SIGGRAPH '96*, pages 281–290, 1996.

[9] J. Gao, J. Huang, C. Johnson, S. Atchley, and J. Kohl. Distributed data management for large volume visualization. In *Proc. IEEE Visualization*, pages 183–189, 2005.

[10] J. Gao, H.-W. Shen, J. Huang, and J. Kohl. Visibility culling for time-varying volume rendering using temporal occlusion coherence. In *Proc. IEEE Visualization*, pages 147–154, 2004.

[11] B.F. Gregorski, J.G. Senecal, M.A. Duchaineau, and K.I. Joy. Adaptive extraction of time-varying isosurfaces. *IEEE Trans. Vis. Comput. Graph.*, 10(6):683–694, 2004.

[12] G. Ji and H.-W. Shen. Dynamic view selection for time-varying volumes. *IEEE Trans. Vis. Comput. Graph. (Vis'06)*, 12(5):1109–1116, 2006.

[13] G. Ji, H.-W. Shen, and R. Wenger. Volume tracking using higher dimensional isosurfacing. In *Proc. Visualization*, pages 209–216, 2003.

[14] K.-L. Ma and D. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Proc. ACM/IEEE Supercomputing*, pages 59–59, 2000.

[15] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proc. IEEE Visualization*, pages 293–300, 2003.

[16] H.-W. Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proc. IEEE Visualization*, pages 159–166, 1998.

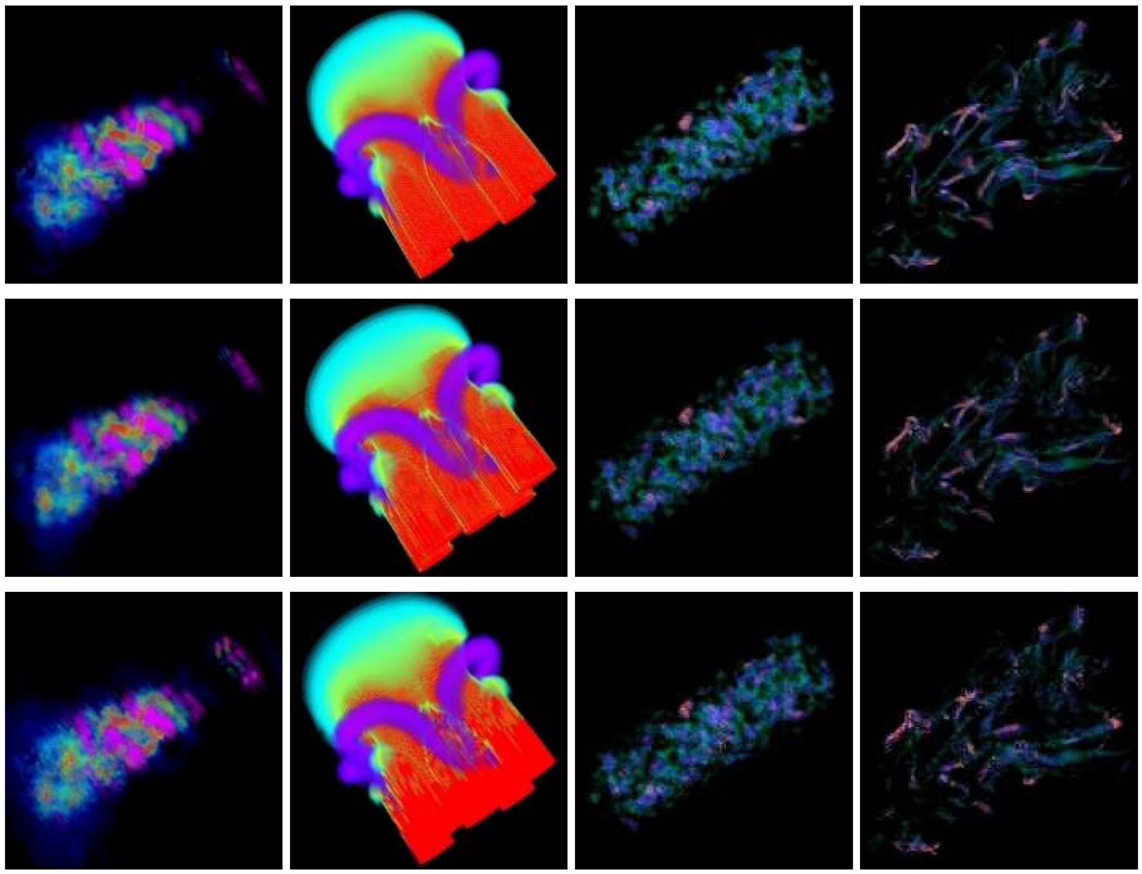

Figure 3: Representative volume rendering results. The datasets from left to right (and time step): Turb2-30 (1), Jets2 (50), TComb (1), TComb (105). Top row: *exact* images, by SPT (or TSP) tree with *no* errors ($\varepsilon_s = \varepsilon_t = 0$), which is the same as out-of-core *bricking* on *original input*. Middle row: by our SPT tree; bottom row: by the TSP tree. The error bounds $(\varepsilon_s, \varepsilon_t)$ for these two rows (from left to right): (0.0001, 0.000002), (0.00002, 0.0000002), and (0.001, 0.000001) (same for both time steps of the same dataset TComb). TSP resulted in more artifacts than our SPT.

[17] 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 *Proc. IEEE Visualization*, pages 371–377, 1999.

[18] H.-W. Shen and C.R. Johnson. Differential volume rendering: A fast volume visualization technique for flow animation. In *Proc. IEEE Visualization*, pages 180–187, 1994.

[19] C. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics, 2002. Tutorial Course Notes, IEEE Visualization 2002. http://cis.poly.edu/chiang/Vis02-tutorial4.pdf.

[20] P. Sutton and C. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transaction in Visualization and Computer Graphics*, 6(2):98–107, 2000.

[21] I. Wald, H. Friedrich, A. Knoll, and C.D. Hansen. Interactive isosurface ray tracing of time-varying tetrahedral volumes. *IEEE Trans. Vis. Comput. Graph. (Vis'07)*, 13(6):1727–1734, 2007.

[22] K.W. Waters, C.S. Co, and K.I. Joy. Using difference intervals for time-varying isosurface visualization. *IEEE Trans. Vis. Comput. Graph. (Vis'06)*, 12(5):1275–1282, 2006.

[23] J. Woodring, C. Wang, and H.-W. Shen. High dimensional direct rendering of time-varying volumetric data. In *Proc. IEEE Visualization*, pages 417–424, 2003.

[24] H. Younesy, T. Möller, and H. Carr. Visualization of time-varying volumetric data using differential time-histogram table. In *Proc. Volume Graphics*, pages 21–29, 2005.

Appendix

Proof of Lemma 2: Let $P(A)$ be the parent of node A in the octree, and we want to show that the smallest possible error$(P(A))$ is still as large as error(A). Let m be the mean value of A. The smallest possible error$(P(A))$ occurs when all other 7 siblings of A have every $v_{i,t}$ value equal to m, since this minimizes the variance of the data points in $P(A)$ (and the mean value is m). Simplifying the notation of $v_{i,t}$ of A to v, we want to show that $8\sqrt{(\Sigma v^2 + 7Nm^2)/(8N) - m^2} \geq \sqrt{\Sigma v^2/N - m^2}$, which is equivalent to $\Sigma v^2 - Nm^2 \geq 0$. The last inequality is equivalent to $\sqrt{\Sigma v^2/N - m^2} \geq 0$, which is true since the left-hand side is the standard deviation of the data points in A. □

Proof of Lemma 3: Letting $P(A)$ be the parent of node A in the octree and node A be at level k, we want to show that the smallest possible error$(P(A))$ is still as large as error(A) (i.e., $\min(\text{error}(P(A))) = \text{error}(A)$), so that $\text{error}(P(A)) \geq \text{error}(A)$. But $\min(\text{error}(P(A)))$ occurs when for all other 7 siblings of A their standard deviations $s(v_i)$ are all 0. Then $\min(\text{error}(P(A))) = \frac{1}{8^{k-1}} \cdot [\Sigma_{i=1}^{8n} s(v_i)/m(v_i)]/(8n) = \frac{1}{8^k} \cdot [\Sigma_{i=1}^{n} s(v_i)/m(v_i) + \Sigma_{n+1}^{8n} 0]/n = \text{error}(A)$. □