Isosurface Extraction and View-Dependent Filtering from Time-Varying Fields Using Persistent Time-Octree (PTOT)

Cong Wang and Yi-Jen Chiang, *Member, IEEE*

Abstract—We develop a new algorithm for isosurface extraction and view-dependent filtering from large time-varying fields, by using a novel *Persistent Time-Octree (PTOT)* indexing structure. Previously, the Persistent Octree (POT) was proposed to perform isosurface extraction and view-dependent filtering, which combines the advantages of the interval tree (for optimal searches of active cells) and of the Branch-On-Need Octree (BONO, for view-dependent filtering), but it only works for *steady-state* (i.e., single time step) data. For time-varying fields, a 4D version of POT, 4D-POT, was proposed for 4D isocontour slicing, where slicing on the time domain gives *all* active cells in the queried time step and isovalue. However, such slicing is *not* output sensitive and thus the searching is *sub-optimal*. Moreover, it was not known how to support view-dependent filtering *in addition to* time-domain slicing.

In this paper, we develop a novel Persistent Time-Octree (PTOT) indexing structure, which has the advantages of POT and performs 4D isocontour slicing on the time domain with an *output-sensitive* and *optimal* searching. In addition, when we query the same isovalue q over m consecutive time steps, there is *no additional searching overhead* (except for reporting the additional active cells) compared to querying just the first time step. Such searching performance for finding active cells is asymptotically optimal, with asymptotically optimal space and preprocessing time as well. Moreover, our PTOT supports view-dependent filtering *in addition to* time-domain slicing. We propose a simple and effective out-of-core scheme, where we integrate our PTOT with *implicit occluders*, batched occlusion queries and batched CUDA computing tasks, so that we can greatly reduce the I/O cost as well as increase the amount of data being concurrently computed in GPU. This results in an efficient algorithm for isosurface extraction with view-dependent filtering utilizing a state-of-the-art programmable GPU for time-varying fields larger than main memory. Our experiments on datasets as large as 192GB (with 4GB per time step) having no more than 870MB of memory footprint in both preprocessing and run-time phases demonstrate the efficacy of our new technique.

Index Terms—Isosurface extraction, time-varying fields, persistent data structure, view-dependent filtering, out-of-core methods.

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 developing a novel algorithm for isosurface extraction and view-dependent filtering of large time-varying fields.

Isosurface extraction is one of the most important and widely used visualization techniques for volume datasets. Specifically, for time-varying fields, performing an isosurface query (q,t) is to extract and display all the points (a surface) in the volume whose scalar values at time step t are the *isovalue* q. Due to the importance of isosurfaces, a tremendous amount of work has focused on speeding up isosurface queries. For *steady-state* (i.e., single time step) data, most acceleration methods employ the following idea: producing for each cell its scalar value range (i.e., an interval $[\min, \max]$), the *active* cells intersected by the isosurface are exactly those cells whose intervals contain the isovalue q. This reduces the problem of finding active cells to that of *interval search*, and the interval tree approach [9] achieves optimal search time. Later, more aggressive approaches (e.g., [11, 19]) have been proposed to perform *view-dependent filtering*, so that only the *visible* portions of an (opaque) isosurface are extracted. In isosurface extraction with view-dependent filtering, essentially the extraction part is a query in the scalar-value domain, and the filtering part is a query in the volume-space domain.

It has been a challenge to develop a search structure that *simultaneously* supports both queries efficiently. Observe that the interval tree does not have space-domain information; extracting all active cells then filtering out invisible ones can result in extracting many active cells that are eventually discarded. On the other hand, using space-partitioning structures such as the Branch-On-Need Octree (BONO) [30] can perform filtering efficiently but the search on the value domain is not optimal. Recently, the Persistent Octree (POT) [23] was proposed to solve this problem nicely, which combines the advantages of the interval tree and of BONO, but POT only works for steady-state data. For time-varying fields, a 4D version of POT, 4D-POT, was proposed for 4D isocontour slicing [23], where slicing at t on the time domain gives *all* active cells in the queried time step t and isovalue q. However, such slicing is *not* output sensitive and thus the searching is *sub-optimal*. Moreover, it was not known how to support view-dependent filtering *in addition to* time-domain slicing.

In this paper, we develop a novel Persistent Time-Octree (PTOT) indexing structure, which has the advantages of POT and performs 4D isocontour slicing on the time domain (i.e., a query (q,t)) with an *output-sensitive* searching, in $O(\log N + \log T + K)$ time, where there are N cells and T time steps in the dataset, and K active cells. Typically T is no bigger than N and the search time becomes $O(\log N + K)$, which is asymptotically optimal. The space and preprocessing time for PTOT are both asymptotically optimal too. In addition, when we query the same isovalue q over m consecutive time steps, which is typically the case, the query time is $O(\log N + \log T + \Sigma_{i=1}^{m} K_i)$ (or $O(\log N + \Sigma_{i=1}^{m} K_i)$ when $T \leq N$), where K_i is the number of active cells in the i-th time step queried. Observe that there is *no additional searching overhead* (except for reporting the additional active cells) compared to querying just the first time step. Such searching performance for finding active cells in time-varying data is asymptotically optimal, which was not known before to the best of our knowledge.[1] Moreover, our PTOT supports view-dependent filtering *in addition to*

• *The authors are with CSE Dept., Polytechnic Institute of New York University, Brooklyn, NY, USA. E-mail: cwang05@students.poly.edu; yjc@poly.edu.* *Cong Wang is supported by NSF grant CCF-0541255. Yi-Jen Chiang is supported in part by NSF CAREER Grant CCF-0093373 and NSF Grant CCF-0541255.*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

[1] As a comparison, using one interval tree per time step takes $O(m \log N + \Sigma_{i=1}^{m} K_i)$ time.

time-domain slicing.

To handle datasets larger than main memory, we employ a simple out-of-core scheme, partitioning the volume into meta-cells consisting of $k \times k \times k$ cells and using the PTOT to index the meta-cells (rather than the original cells), so that the PTOT can entirely fit in main memory; the meta-cells are kept on disk and read to main memory when needed. In this out-of-core setting, we want to reduce the I/O cost, for which it is equally important to reduce the *number of disk reads* and to reduce the *disk seek time*. Ideally, we would like to use view-dependent filtering to reduce the number of active meta-cells needed to be read, and at the same time we wish to read such meta-cells as *sequential* as possible (in terms of the *order they are stored on disk*) to minimize the disk seek time. However, view-dependent filtering typically needs to visit the active meta-cells in visibility order, which is often highly non-sequential (again in terms of the *storage order on disk*). We propose a simple and effective approach, which integrates the *implicit occluders* [19], so that we can strike a balance between the two objectives.[2] In addition, we use the CUDA hardware marching cubes [17], which is a popular, publicly available and highly efficient programmable-GPU isosurface engine, to extract isosurface triangles from active meta-cells, where we show how to *batch* the meta-cells for CUDA computation to increase the amount of data being concurrently computed in GPU. All these methods are integrated into an efficient algorithm for isosurface extraction with view-dependent filtering for time-varying fields larger than main memory. We remark that our view-dependent filtering approach under the out-of-core setting and the method for batching the CUDA computation are both general and not restricted to PTOT, and might be of an independent interest.

One limitation to our PTOT search structure, compared to the 4D-POT, is that 4D-POT is more general in that it can also support 4D isocontour slicing on the x-, y-, or z-domain, which is not supported in our PTOT. However, for time-varying datasets, the most common isocontour queries are of the type (q,t), for which our PTOT improves over 4D-POT to achieve optimal searching and to additionally support view-dependent filtering as mentioned above.

Our experiments on datasets up to 192GB (with 4GB per time step) having at most 870MB of memory footprint in both preprocessing and run-time phases demonstrate the efficacy of our new technique.

2 Previous Work

In this section, we review the previous work on isosurface extraction, including out-of-core and view-dependent filtering approaches. For out-of-core techniques in graphics and scientific visualization other than isosurface extraction, we refer to the survey [24].

There is a very rich literature for isosurface extraction; we refer to [15] for an excellent and thorough review. In Marching Cubes [16], all cells in the volume dataset are searched for isosurface intersection. Techniques avoiding exhaustive scanning include using an octree (the *branch-on-need octree (BONO)* [30]), identifying a collection of *seed cells* and performing contour propagation from the seed cells [3, 13, 27], NOISE [15], and other nearly optimal isosurface extraction methods [20, 22]. The first *query-optimal* algorithm was given by Cignoni et al. using the interval tree [9]. Bordoloi and Shen [4] proposed a technique to reduce the space overhead of the indexing structure while maintaining an efficient search performance. Stockinger et al. [25] introduced the *query-driven visualization* approach using *bitmap indexing*, with a nice feature of being able to support multi-dimensional, multivariate queries. Such indexing structure is value-domain based and does not have space-domain information to support efficient view-dependent filtering.

The techniques mentioned above are main-memory methods. As for out-of-core approaches, Chiang and Silva [7] and Chiang et al. [8, 6] developed out-of-core isosurface extraction algorithms, by using I/O-optimal interval trees such as those given in [1, 8] and the *meta-cell* technique for irregular grids [8]. In addition, Bajaj et al. [2] proposed a parallel and out-of-core isosurface approach based on contour propagation from seed cells.

[2] In [23], the authors mentioned that the implicit occluders could be used together with the POT, but they did not discuss how to do it either in the in-core or the out-of-core setting.

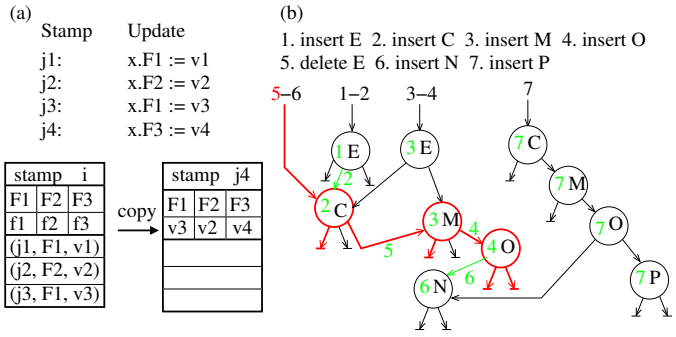

Fig. 1. Persistent structure. (a) The node-copying technique, where each persistent node has 3 extra fields. (b) Persistent binary search tree (where we use the alphabetical order to compare the keys (letters)) after simulating a sequence of updates. Each node has 1 extra field. The number associated with a node/pointer denotes its version stamp. The numbers 1 to 7 on the top horizontal line denote the entry-point array A. Version 5 is shown in red.

The techniques mentioned so far are for steady-state datasets. For time-varying fields, Shen [21] gave an in-core technique based on the *THI tree*. Out-of-core algorithms include the *temporal branch-on-need octree* method by Sutton and Hansen [26], the adaptive extraction approach by Gregorski et al [12], the *time-tree* algorithm by Chiang [5], and the *difference intervals* technique by Waters et al. [29].

Most of the above approaches try to extract the whole isosurface, while the *view-dependent filtering* techniques [14, 11, 19] try to extract only the visible portions of the isosurface to speed up the process. A variation of these approaches includes those based on ray tracing [18, 28]. Among these methods the one most closely related to us is the *implicit occluders* technique by Pesco et al. [19]; we review it in more detail in Section 3.2.2. As mentioned before, view-dependent filtering needs space-domain data structures such as octrees (e.g., BONO [30]), whose search on the value domain is not optimal, while the value-domain data structures (e.g., the interval-tree approach [9]) cannot perform filtering efficiently. The first approach to combine both advantages was the elegant *persistent octree (POT)* developed by Shi and JaJa [23], where the 4D extension, the *4D-POT*, was proposed for time-varying fields. In this paper we take on this direction and improve over 4D-POT, and in addition we integrate view-dependent filtering under the out-of-core setting.

3 Our Approach

As mentioned in Section 1, there are several technical components in our approach, including the Persistent Time-Octree (PTOT) to index cells/meta-cells, the out-of-core scheme, view-dependent filtering with I/O considerations, and batching the CUDA computation. We now describe them one by one.

3.1 Persistent Time-Octree (PTOT)

First we present our PTOT. Although eventually our PTOT is used to index the meta-cells, conceptually it is the same for the indexing structure whether the basic units being indexed are cells or meta-cells. For simplicity of discussions, we use "cells" to refer to the basic units being indexed in this subsection.

3.1.1 Building Block: Persistent Data Structure

An important building block for our PTOT is the technique [10] for making a dynamic linked data structure *persistent*, where the linked data structure consists of a set of nodes with a fixed number of pointers, such as a search tree with *bounded node degree*. An ordinary dynamic tree structure is called *ephemeral*, in the sense that making an update to the structure destroys the old version, leaving only the new version. A *persistent* data structure, on the other hand, keeps *all* versions resulting

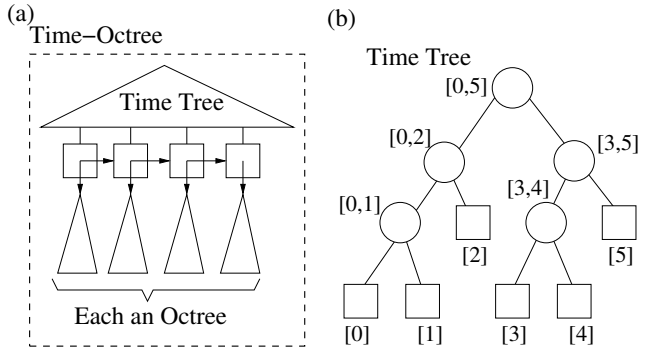

Fig. 2. (a) The time-octree. (b) An example of the time tree for time interval $[0,5]$. Each internal node labeled $[t_1,t_2]$ covers the time span $[t_1,t_2]$, and each leaf labeled $[t]$ corresponds to time step t.

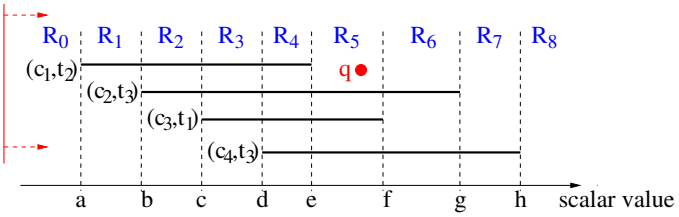

Fig. 3. Line-sweep process to insert/delete intervals $I_{c,t}$ to the time-octree. There are four time cells $(c_1,t_2),(c_2,t_3),(c_3,t_1),(c_4,t_3)$ whose $[\min,\max]$ intervals are respectively $[a,e],[b,g],[c,f],[d,h]$. The vertical red line is the sweep line. The $[\min,\max]$ interval endpoints subdivide the scalar-value range $(-\infty,\infty)$ into ranges $R_0=(-\infty,a)$, $R_1=[a,b)$, $R_2=[b,c)$, $R_3=[c,d)$, $R_4=[d,e]$, $R_5=(e,f]$, $R_6=(f,g]$, $R_7=(g,h]$, $R_8=(h,\infty)$, where version i of the time-octree corresponds to range R_i. Isovalue q lies in range R_5, and version 5 of the time-octree contains exactly time cells $(c_2,t_3),(c_3,t_1),(c_4,t_3)$, which are active for q; (c_2,t_3) and (c_4,t_3) belong to the octree of t_3 and (c_3,t_1) belongs to the octree of t_1.

from a sequence of updates, by providing a compact representation such that any version of the tree can be queried, and that querying version i takes asymptotically the same time as querying version i of the ephemeral tree. Also, if a sequence of updates makes a total of m structural changes to the ephemeral tree (which takes $O(m)$ time), then the same update sequence applied to the corresponding persistent tree results in $O(m)$ additional space, using $O(m)$ processing time, to record these changes in the persistent tree. To perform a search in a persistent tree, we need to first specify the *version number* i, so that we can proceed to query version i of the tree.

Node Copying The main persistent-tree technique is *node-copying* [10], summarized below. For an ephemeral-tree node x, the corresponding persistent-tree node $\bar{x}$ has all fields of x, plus some constant number of extra fields to record some future updates. Suppose the i-th update creates node x; in node $\bar{x}$ we record the same field values as those in x, plus the *version stamp* i. Subsequent updates to x are recorded to the extra fields of $\bar{x}$ if available (see Fig. 1(a)). When all extra fields are used up, the next update to x causes a *node-copying*—we create a new persistent node $c(\bar{x})$, which has the new version stamp and the *latest* field values of x (see Fig. 1(a)). In addition, any predecessor node pointing to $\bar{x}$ should now change its pointer to $c(\bar{x})$. Again, such updates to predecessors are "simulated" by the above method, which may trigger new copying actions for the predecessors if needed. Finally, note that different versions may have different entry points (roots) to the persistent tree. For example, if for version j the root $\bar{r}$ is copied into a new root $c(\bar{r})$, then $\bar{r}$ is the entry point for versions 1 to $j-1$ and $c(\bar{r})$ is the entry point for version j. We maintain an entry-point array A so that $A[i]$ points to the root of version i. To access version i of the tree, we follow $A[i]$ to the root, and for each node visited, the desired field values are those with the *largest version stamp not exceeding* i. In Fig. 1(b), we give an example of persistent binary search tree. Note that the last update triggers more than one copying. Also, version 5 (shown in red) does not include the pointer from node (4 O) to (6 N) since its version stamp (6) exceeds the version number (5).

3.1.2 Persistent Time-Octree (PTOT)

Suppose there are N cells and T time steps in the time-varying regular-grid dataset. For each time step t, we produce for each cell c an interval $I_{c,t}=[\min,\max]$ representing the scalar-value range of c at time step t. We would like to use our data structure to index these NT intervals, so that we can efficiently find active cells for an isosurface query (q,t).

We use the persistent-tree technique in the following fashion. First, we develop a suitable *ephemeral* search structure, called the *time-octree*, which supports insertion, deletion, and query. We will discuss how to perform a sequence of insertion/deletion operations on the time-octree in a *line-sweep* process, and how to perform queries. Secondly, we apply the persistent-tree technique on the time-octree to make it persistent, called *persistent time-octree* (PTOT). In the preprocessing phase, we simulate the line-sweep process on the PTOT so that all versions are recorded into the PTOT. This is a construction phase to build our PTOT search structure, which then becomes *static* and there is no more update afterwords. In the run-time phase, for a given query we first find the right version number i, and proceed to query version i of the PTOT.

Time-Octree We first develop the *time-octree* as our base, *ephemeral* data structure (see Fig. 2(a)). The top part of the time-octree is a fully balanced binary tree called *time tree*. The root of the time tree 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. 2(b)).

The bottom part of the time-octree consists of a collection of octrees, where there is one octree per time-tree leaf, with a pointer from the time-tree leaf to its octree root. It is convenient to talk about a "cell c at time step t", called the *time cell* (c,t), which corresponds to the interval $I_{c,t}=[\min,\max]$ mentioned above. For each time-tree leaf of time step t, the corresponding octree stores the time cells (c,t). For now, we can simply think of each such octree as a standard complete octree, which recursively subdivides the input volume spatially until all octree leaves correspond to the grid cells. Finally, for each time-tree leaf t whose octree is not empty, we maintain a pointer from t to the next time-tree leaf t' whose octree is also not empty (see Fig. 2(a)).

Line Sweep Now we discuss a *line sweep* process for inserting/deleting intervals $I_{c,t}=[\min,\max]$ to/from our time-octree. Recall that there are NT such intervals. We first sort the $2NT$ interval endpoints, and then use a "sweep line" to sweep, from $-\infty$ to ∞, through the 1D segments/intervals (see Fig. 3). Initially, the time-octree is empty. During the sweep, when the sweep line encounters the left endpoint of an interval $I_{c,t}$, we insert $I_{c,t}$ to the time-octree, and when the right endpoint of $I_{c,t}$ is encountered, we delete $I_{c,t}$ from the time-octree.

To insert $I_{c,t}$, we first work on the time domain, using t to locate the leaf t in the time tree. In the process, we create/grow a root-to-leaf path on the fly. Secondly, we work on the space domain, going from the time-tree leaf t to the corresponding octree root, and use c to locate the leaf cell c in this octree. Again, in the process we create/grow a root-to-leaf path on the fly (see Fig. 4(a)).

Similarly, deleting $I_{c,t}$ is the reverse operation. First, we locate the leaf t of the time tree, and also locate the leaf cell c in the corresponding octree. We then delete the leaf cell c in the octree, and also remove any "extra" internal nodes along the *leaf-to-root* path from c in the octree, where an internal node on the path is "extra" if originally (i.e., before deleting c) it has only one child (see Fig. 4(a) right, the path from C to B). In addition, if the octree becomes empty, then we will delete the leaf t of the time tree, using essentially the same algorithm.

The $2NT$ interval endpoints subdivide the entire scalar-value range $(-\infty,\infty)$ into $2NT+1$ ranges, where each range R_i corresponds to a version i of the time-octree (see Fig. 3). Observe that version i

of the time-octree contains *exactly* those time cells whose $[\min, \max]$ intervals *cover* R_i. For example, in Fig. 3, version 5 corresponds to range $R_5 = (e, f]$, and the time cells stored in version 5 are $(c_2, t_3), (c_3, t_1), (c_4, t_3)$, each with an interval covering $(e, f]$. This means that if the query isovalue q lies in range $R_5 = (e, f]$, then version 5 of the time-octree contains exactly those *active* time cells if we ignore the queried time step t. These time cells are further grouped according to the time steps, with $(c_2, t_3), (c_4, t_3)$ stored in the octree of time step t_3 and (c_3, t_1) in the octree of time step t_1. In other words, these octrees store *exactly* the active time cells for the queries (q, t_3) and (q, t_1) respectively.

PTOT Now we are ready to apply the persistent-tree technique to our time-octree, which becomes our *persistent time-octree* (PTOT). In the preprocessing phase, we simulate the above line-sweep process on the PTOT so that all versions are kept. This is the construction phase to build our PTOT; recall that there is no more update to the PTOT from now on. In the run-time phase, given an isosurface query (q, t), we first perform a binary search to locate the range R_i containing q, and access the corresponding version, version i of the PTOT. We then search on the time tree part of this version to locate the leaf t, and follow the pointer to visit the octree of time step t, where *all* leaves in this octree are active cells. Moreover, since these active cells are already organized in an octree, we can in addition perform *view-dependent filtering* using the octree structure. Also, when we query the same isovalue q over m consecutive time steps, since the same q means the same version number for the PTOT, after reporting for the first time step, we can just follow the pointer from the current time-tree leaf to the next time-tree leaf whose octree is not empty, report the leaves of that octree, and repeat the process (recall the structure from Fig. 2 (a)).

The query time (without filtering) is $O(\log N + \log T + K)$ for a single query (q, t) where K is the number of active cells, and $O(\log N + \log T + \Sigma_{i=1}^{m} K_i)$ for querying the same isovalue q over m consecutive time steps where K_i is the number of active cells in the i-th queried time step. Typically T is no bigger than N and the $\log T$ term disappears in both bounds, which are both asymptotically optimal.

Final Structure: Compact Representation

In the above scheme of the ephemeral time-octree, inserting an interval $I_{c,t}$ may create an entirely new root-to-leaf path in either the time tree or the octree or both (see Fig. 4(a)), causing an insertion of $O(\log T + \log N)$ new nodes to the time-octree; similarly deleting $I_{c,t}$ may cause a deletion of $O(\log T + \log N)$ nodes. Therefore each interval insertion/deletion causes $O(\log(NT))$ structural changes; with $2NT$ interval insertions/deletions in the line-sweep process, there are $O((NT)\log(NT))$ structural changes, and hence the persistent time-octree has space $O((NT)\log(NT))$ (recall from Section 3.1.1 that $O(m)$ structural changes in the ephemeral tree result in $O(m)$ additional space in the persistent tree), which is non-linear.

To address this issue, we use a *compact* representation (similar to [23]) in both the time tree and the octrees (of the ephemeral structure): any *internal* node u other than the root is removed if u has only one child. Referring to Fig. 4(b), the path P between nodes V and W is replaced by a pointer from V directly to W. To retain the information of path P, in W we store a bit sequence of the path from the root to W (e.g., one bit per level in the time tree to specify left/right, and three bits per level in an octree to indicate one of the 8 children). Such bit sequence can be viewed as a canonical node ID for W in the standard tree. In this way, a new internal node u is created only when u is a new node with *two* children; we call u a *degree-2 fork* node. Note that inserting a leaf can create *at most one* such fork node, which is exactly where the new path joins some existing path (see Fig. 4(a) and (c), where A is such fork node). Therefore inserting an interval $I_{c,t}$ to the time-octree can create at most two new internal nodes (one in each tree). Similarly, deleting an interval can remove at most two internal nodes, one in each tree (see Fig. 4(c)). In this way, each interval insertion/deletion only causes $O(1)$ structural changes, and thus the space of our persistent time-octree becomes *linear* ($O(NT)$), which is asymptotically optimal. The query time bounds stay the same, and the preprocessing time is the same as the sorting bound on the NT intervals stored, which is also asymptotically optimal.

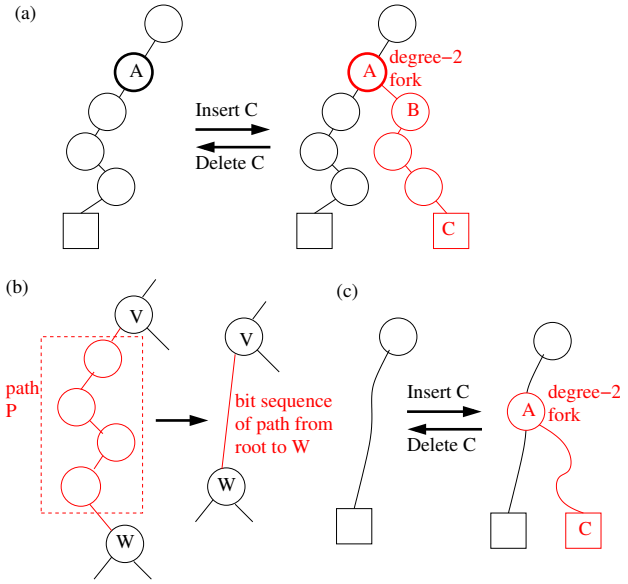

Fig. 4. (a) Standard representation. Inserting/deleting a leaf C can create/remove many nodes (shown in red excluding A). (b) The compact representation. (c) With the compact representation, inserting/deleting a leaf C (as the event in (a)) can create/remove at most one internal node, the *degree-2 fork* node A.

3.1.3 Comparisons with Persistent Octree (POT) and 4D-POT

When applying to a steady-state dataset, the time tree of our PTOT becomes a single leaf and can be removed, in which case our PTOT essentially becomes the POT [23]. However, the original POT in [23] classifies all tree nodes into white, black and gray and changes the classifications during operations, which we found is more complicated than needed. Also, for an octree internal node u, if an insertion makes the subtree rooted at u a complete octree, then all 8 children of u are merged into u bottom-up [23]. This makes it less refined in view-dependent filtering, and can cause a ripple effect to merge many levels bottom-up (and similarly for deletions). The latter makes an insert/delete more complex and costly, with possibly more than $O(1)$ structural changes (where one needs an amortized argument to achieve the linear space of POT [23]). As for time-varying data, the 4D-POT was proposed in [23], which treats the time domain as the 4th spatial dimension and the octree becomes a "4D octree" where each internal node has 16 children. As mentioned, 4D-POT can support 4D isocontour slicing on the x-, y-, z-, or time-domain; in particular, querying (q, t) amounts to 4D isocontour slicing on the time domain at t. However, such isocontour slicing is not output-sensitive and thus is suboptimal. Moreover, it is not known how to perform view-dependent filtering in addition to isosurface extraction with query (q, t).

3.2 Out-of-Core Scheme and View-Dependent Filtering

3.2.1 The Overall Scheme in the Out-of-Core Setting

In order to handle time-varying datasets larger than main memory, we use a simple and effective out-of-core scheme. First, we partition the space domain into meta-cells which are subvolumes of $k \times k \times k$ cells, and use our PTOT to index the meta-cells so that the PTOT can entirely fit in main memory; the meta-cells are kept on disk and read to main memory when needed. The parameter k decides the octree depth in our PTOT and hence the size of PTOT.

In the preprocessing phase, we perform the following tasks:

1. For each time step t, construct meta-cells for t and store them on disk. The grid data points at t are organized in x-y slices of increasing z; we only read to main memory k slices at a time to produce one slice of meta-cells, so that the memory footprint is small.
2. In the process of task 1, for each meta-cell c at t, produce the interval $I_{c,t} = [\min, \max]$ to be indexed by PTOT. Interval $I_{c,t}$ contains

the min, max values, the time step t, the leaf cell c (the leaf ID in the octree), and the meta-cell ID (the position of the meta-cell stored on disk, for reading the meta-cell from disk).
3. In the process of task 1, after the current time step t is processed, construct the *octree skeleton* for t and store it on disk, to be used for view-dependent filtering (discussed later).
4. After all time steps are processed, use all intervals $I_{c,t}$ generated in task 2 to build the PTOT, and store the PTOT on disk.

In the run-time phase, we first read and keep the PTOT into main memory. To perform an isosurface query (q,t), we perform the following tasks:
1. Query the PTOT to find the active meta-cells.
2. Read the active meta-cells from disk to main memory under view-dependent filtering. To perform the filtering, read the octree skeleton for time step t from disk, and use it together with the PTOT. The meta-cell reading (from disk) and the view-dependent filtering are performed in an integrated fashion, discussed later.
3. For the meta-cells read into main memory, send them in batches to CUDA for extracting and rendering isosurface triangles using the programmable GPU.

In the rest of this section, we discuss the remaining technical components, namely view-dependent filtering and batching the CUDA computation.

3.2.2 View-Dependent Filtering

Recall from Section 3.1 that our PTOT already provides a method for view-dependent filtering on the active meta-cells of the query (q,t): after finding the correct version number i for q, we access version i of our PTOT, where the octree of time t in this version contains only the active meta-cells (and their ancestors) of (q,t). Therefore we can use this octree to perform view-dependent filtering. For simplicity, we call this octree the *active octree*.

A basic approach is to traverse the active octree hierarchically in the front-to-back visibility order. In the process, we use the portions of the isosurface already rendered as occluders. To reduce the latency in hardware occlusion queries, starting from the root, for each current node u visited, we perform *batched* occlusion queries on all available children of u at the same time; if any child's bounding box is entirely occluded, we skip the entire subtree rooted at that child. When we reach a non-occluded leaf, we read its meta-cell from disk, and send the meta-cell to CUDA for extracting and rendering isosurface triangles. In addition, to increase the amount of data being concurrently computed in GPU, we send meta-cells to CUDA *in batches* as well. Namely, we read meta-cells one by one from disk in the above process; each time when there are already B meta-cells available, we send these B meta-cells altogether to CUDA, where B is a parameter decided by the GPU memory.

Further Optimization with Implicit Occluders

In order to reduce the I/O cost in the out-of-core setting, it is equally important to reduce the number of disk reads and to reduce the disk seek time. Therefore we have two objectives: (1) occlude meta-cells as many as possible to reduce the number of meta-cells read; (2) read the necessary meta-cells as *sequential* as possible (in terms of the *order the meta-cells are stored on disk*, indicated by the meta-cell IDs (positions in file); see the preprocessing task 2 in Section 3.2.1) to reduce the disk seek time. However, the above approach is not very good for these objectives. First, the initial occluders are small and thus not good for (1). Secondly, the visibility traversal order can be far from sequential and thus not good for (2) either.

We propose an approach that can strike a balance between (1) and (2), by integrating the technique of *implicit occluders* [19]. This technique builds implicit occluders without the need to actually render the isosurface, and thus the initial occluders are much larger. Moreover, since rendering isosurface and constructing occluders are *de-coupled*, we can sort the meta-cell IDs, read and render them in this sorted order of disk positions to get more sequential I/Os.

We briefly summarize the technique [19]. Consider casting a ray from the eye through a pixel; if the ray hits two points v,w in the volume, with one point's scalar value smaller than the isovalue q and the other's larger, where their depth values are $z(v) < z(w)$, then an isosurface fragment s exists for this pixel within the depth range $(z(v), z(w))$, and any fragment behind w will surely be occluded by s. Therefore, we can set the z-value of this pixel in the z-buffer to $z(w)$, which serves as an "implicit" occluder to mask out anything behind w. The technique uses octree-node bounding boxes to identify such pair v,w as close to the eye as possible and set up the z-buffer appropriately, whose z-values form an *occlusion map*, to be used to mask out anything behind.

Specifically, the method needs an octree covering the entire volume, where each octree node stores the $[\min, \max]$ scalar value range of its subvolume. Given an isovalue q and the viewpoint, an occlusion map is built by rendering the octree-node bounding boxes into the z-buffer in two passes, first those nodes with scalar values all *below* q, with the smallest z-values kept in each pixel, and second those nodes with scalar values all *above* q, rendered in front-to-back order, with the z-values updated appropriately [19].

Potentially there are two issues for us to use implicit occluders: (a) An octree covering the entire volume is needed; in particular, we need octree nodes with scalar values completely above or below q, but in our PTOT the active octree only has *active* leaves/meta-cells and their ancestors. (b) The implicit occluders could be too conservative and thus not large enough.

We address issue (a) easily by providing, for each time step t, a *separate* octree skeleton, which is a complete octree storing the $[\min, \max]$ scalar value range in each node. The leaves of the octree skeleton can be at meta-cells, or we can subdivide one or more levels further to get more refined occlusion maps. When we construct meta-cells for t in preprocessing, we can obtain such leaves and take the $[\min, \max]$ ranges; we can then compute the $[\min, \max]$ ranges for all octree nodes in a bottom-up fashion.

Now the view-dependent filtering goes as follows. First, we follow the two-pass process of [19], using the octree skeleton at time t to build the occlusion map M in the z-buffer. Then we will traverse the active octree of our PTOT in front-to-back order and perform subtree skipping (with batched occlusion queries) as before, but now the occlusion filtering is done against the occlusion map M. Note that the filtering can be performed completely without any I/O. To one extreme, we can finish the entire filtering against the same M, and for the resulting meta-cells that need to be rendered, we sort them globally by the meta-cell IDs and then perform the I/Os. This makes the disk reads as sequential as possible, in favor of objective (2).

For issue (b), which is related to objective (1), we would like to *grow* the occlusion map M so that we can reduce the number of I/Os even further. That is, for the isosurface portions rendered, we also let them become occluders by rendering them to the z-buffer, which effectively updates/grows M. To this end, we use an interleaving process. Initially, we perform occlusion filtering against M as before, until we accumulate L meta-cell IDs, where L is a parameter set to a large enough number. We then sort these L IDs, perform I/Os to read meta-cells in this sorted order one at a time, and send them to CUDA in batches of B meta-cells as before—in a total of L/B batches. (Note that we only need to keep up to B meta-cells in main memory at any time, since they are no longer needed once sent to CUDA.) Now the occlusion map M is *grown* with L additional meta-cells rendered. We repeat this process for another L meta-cells (where the filtering is against the new M), and so on, until we finish traversing the active octree of our PTOT.

This approach allows us to obtain a best combination of objectives (1) and (2). Ideally, there is a good, balanced value of L: L is large enough so that the I/Os are sequential enough, and also L is small enough so that the occlusion map M grows sufficiently often (to filter out more meta-cells to be read). In our experiments we found such good value for L (see Section 4).

3.2.3 Batching the CUDA Computation

Recall that we send to CUDA a batch of B meta-cells at a time for extracting isosurface triangles, in order to increase the amount of data being concurrently computed in GPU. The original CUDA code for hardware marching cubes [17] takes a 1D array of scalar values at grid

Data	# time steps	Dimensions	Size
Jets	16	1024x1024x1024	64GB
Syn	16	1024x1024x1024	64GB
Turb	32	1024x1024x1024	128GB
Vort	48	1024x1024x1024	192GB

Table 1. Statistics of our test datasets.

Data	Jets (64GB)	Syn (64GB)	Turb (128GB)	Vort (192GB)
PTOT	30MB	30MB	59MB	88MB
Octree skel.	54MB	54MB	108MB	162MB
Meta-cell	70GB	70GB	140GB	210GB
Total	70.08GB	70.08GB	140.17GB	210.25GB
Increase	9.5%	9.5%	9.5%	9.5%
Footprint	290MB	290MB	600MB	870MB
PTOT	25s	25s	58s	100s
Meta-cell	2672s	2672s	5344s	8016s
Total	2697s	2697s	5402s	8116s

Table 2. Preprocessing results. The upper table shows the space statistics of the resulting data structure on disk. The lower table shows the execution performance of the preprocessing. Not including the root, the octrees in PTOT had 5 levels, and the octree skeletons had 6 levels.

points, and uses the provided volume-grid dimension to get, for each point, its (i,j,k) index in the volume. Then for each point (i,j,k), its 8 neighboring points forming the cell with base point at (i,j,k) are collected to form a cell and the isosurface triangles are computed, with all points/cells done in parallel.

We need to modify the above CUDA code when sending B meta-cells in a batch. Our meta-cells are $k \times k \times k$ grids; sending B meta-cells in a batch means that the grid points in the CUDA 1D array have global IDs from 0 to k^3B-1. We also provide for each meta-cell the (x,y,z) coordinates of its base point. For each grid point, from its global ID and the meta-cell size we know which of the B meta-cells it belongs to, and its offset from the base point of this meta-cell. Using its offset and the meta-cell grid dimension, we get its index $< i', j', k' >$ *within* this meta-cell. Therefore, we can map between the global ID and the index $(m, < i', j', k' >)$, indicating that it has index $< i', j', k' >$ within the m-th meta-cell sent. For $(m, < i', j', k' >)$, we obtain its real coordinates by adding the meta-cell base-point coordinates, and we can also collect its 8 neighboring points within the same meta-cell to form the correct cell. With this modification, we can again make both cell formation (together with real coordinates) and isosurface-triangle computation done for all points/cells in parallel.

4 RESULTS

We have implemented our technique in C/C++[3] and ran our experiments on a Dell Precision PC with the following configuration: 1GB of RAM, two 3GHz Intel Xeon CPUs, Nvidia GeForce 9800 GTX graphics (512MB graphics memory), and 300GB SCSI 10K rpm disk, running under Fedora-9 64bit Linux OS. The datasets used are listed in Table 1; Jets, Turb and Vort are real-world datasets from scientific applications, and Syn is a synthetic dataset generated with scalar value function $f(x,y,z,t) = \sin(xyz/(0.1 \cdot t + 1)) + \cos((x-2)(y-2)(z-2)/(0.1 \cdot t + 1))$ over the spatial domain $[-5,5] \times [-5,5] \times [-5,5]$ with 16 time steps in the time domain. Each dataset has 4GB in each time step.

Preprocessing

In Table 2 we show our preprocessing results. We chose the meta-cell grid dimension to be $k \times k \times k$ with $k = 32$, and thus the octrees in the PTOT had 5 levels (not including the root); also, we chose the octree skeletons to have 6 levels. We can see that both PTOT and octree skeletons were relatively small, and certainly they could fit in main memory. Each data structure size was basically proportional to the dataset sizes; in particular, for PTOT this was due to the fact that each interval was inserted and deleted exactly once during line sweep, with all history of updates recorded. In meta-cell construction, our approach of reading only k slices of data at a time is effective—although each time step had 4GB, our memory footprint was only 140MB, independent of the number of time steps. The largest memory footprint occurred when building the PTOT, where all intervals from all time steps were kept in main memory; such memory footprint was no more than 870MB for Vort with 192GB. Our overall data structure was quite space efficient; the size overhead was only 9.5%, mainly due to meta-cells.

[3] We used the CUDA code from [17].

	Size	# M-cells	Search	I/O	Total
Jets (16)					
PTOT	30MB	61703	0.12s	284s	308s
4D-POT	32MB	61703	0.18s	284s	308.1s
THI (50%)	6.3MB	67272	0.01s	294s	319s
Syn (16)					
PTOT	30MB	448289	0.5s	1103s	1292s
4D-POT	32MB	448289	1.15s	1103s	1292.7s
THI (49%)	6.2MB	504726	0.08s	1124s	1328s
Turb (32)					
PTOT	59MB	133925	0.21s	551s	603s
4D-POT	63MB	133925	0.49s	551s	603.3s
THI (50%)	12.5MB	140174	0.01s	578s	631s
Vort (48)					
PTOT	88MB	403694	0.89s	1678s	1839s
4D-POT	95MB	403694	2.10s	1678s	1840.2s
THI (49%)	19MB	447128	0.02s	1826s	1999s

Table 3. **View-independent** isosurface extraction over all time steps (Jets and Syn: 16 steps; Turb: 32 steps; Vort: 48 steps). "Size" is the tree size and "# M-cells" is the number of meta-cells reported from index searching. "Total" is the total running time, including the CUDA time for isosurface generation and rendering (not shown separately).

Run-Time Query: View-Independent Isosurfacing

In run-time queries, we first considered view-independent isosurface extraction. Recall that we batch the CUDA computation by sending B meta-cells to CUDA at a time. To see how much the batching effect was, we tested our method with $B = 1$ (i.e., no batching) against $B = 32$, and found that for Vort with $q = 4, t = 15$, the total time (including query, extraction and rendering, but excluding I/Os) difference was between 13.89s and 3.89s, about a factor of 3. Certainly, batching the CUDA computation is effective. In the following, we always set B to 32.

Next, we compared our PTOT against the 4D-POT [23] and the THI tree [21] (a well-known value-based indexing structure for time-varying data) under the same out-of-core scheme—they performed exactly the same steps except for searching on different indexing structures. For each dataset we queried the same isovalue over all time steps, where at each time step we first sorted all reported meta-cell IDs before reading them from disk; the results are shown in Table 3. (For THI, we set the lattice-partition parameter so that after interval merging there were about 50% intervals remaining and indexed, where "THI (a%)" in Table 3 means there were a% remaining.) Comparing PTOT against 4D-POT, we see that the tree size was a bit better, and the index searching time was about twice as fast, but the difference was small. The dominating cost was the I/O time, which was the same since the same active meta-cells were first sorted by IDs and then read from disk. (Without ID sorting, the I/O time was much worse, e.g., 2169s vs. 1236s for Vort when querying over the first 32 time steps.)

Comparing our PTOT against THI, we see that the tree size was 4.63-4.84 times as big. However, under our out-of-core scheme, the PTOT tree size was insignificant compared to meta-cells (see Table 2)

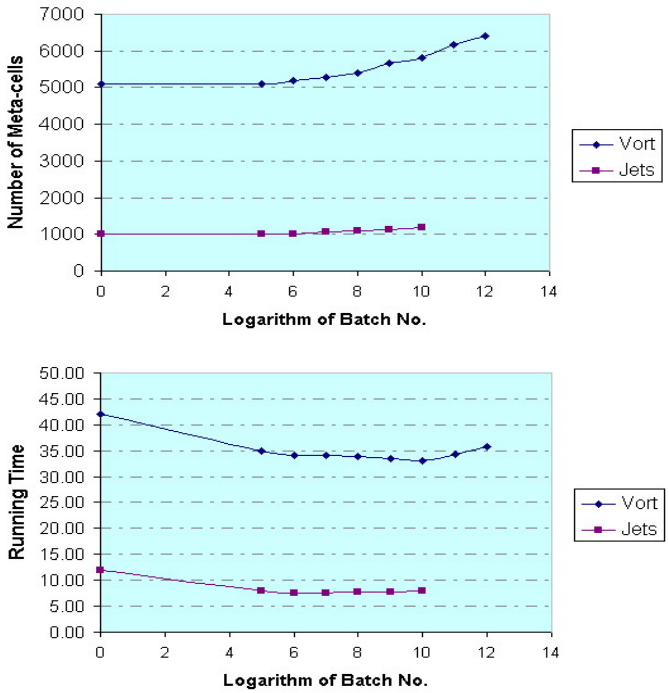

Fig. 5. Results of View-dependent filtering with various values of L. Top: number of meta-cells read from disk against L. Bottom: running time in seconds against L. The results are for isosurfaces of one time step. Note that the axis of L is in a logarithmic ($\log_2(\cdot)$) scale.

and we can always make it fit to main memory by choosing a suitable meta-cell dimension. The index searching time was also worse but again the difference was small. However, due to the nature of interval merging, THI typically reported a *super-set* of active meta-cells (see Table 3: the number of reported meta-cells was always larger), which incurred a large I/O penalty and thus a much worse total time.

The major advantage of our PTOT over 4D-POT (and value-based indexing structures such as THI as well), under the out-of-core setting, lies in its ability to perform view-dependent filtering, discussed next.

Run-Time Query: View-Dependent Filtering

Our view-dependent filtering approach provides a scheme to balance between the number of I/Os and the disk seek time, where we can adjust the parameter L (recall from Section 3.2.2). To see the effect of L, we ran our approach on both Jets and Vort with various values of L; the results are shown in Fig. 5. Note that a larger value of L means more sequential I/Os (in favor of reducing the seek time) while a smaller value of L means we grew the occlusion map more often and could reduce more meta-cells to be read. It is quite interesting to see that for a very small value of L, although the number of I/Os was small, the running time was quite high, showing that sequential (sorted) disk reads are important. However, as long as L got bigger, even if there was just a small amount of sorting on the meta-cell IDs, the running time improved quickly. And after L got large enough, making L even larger did not help much, since now growing the occlusion map more often would have a bigger effect. Therefore, we chose L to be some big enough number but not too big, so that we got the sorting advantage and also we could grow the occlusion map sufficiently often. In the rest of the experiments, we set $L = 128$.

Finally, we compared our approach (using implicit occluders, with $B = 32$ and $L = 128$), called Implicit, with two other methods: (1) Explicit — a standard approach where we used the rendered isosurface as the occluder. Both Implicit and Explicit traversed the active octree of our PTOT with batched occlusion queries (as described in Section 3.2.2). (2) No-Occ — no occlusion filtering, where we accumulated all active meta-cell IDs, sorted them, and read them from disk

	Implicit	Explicit	No-Occ
Jets			
Ave. # Triangles	2,305,507	2,300,439	3,815,320
Ave. # Meta-cells	1070	1022	1718
Ave. Time (s)	7.74	9.61	9.73
Total Time (s)	77.4	96.1	97.3
Syn			
Ave. # Triangles	30,758,933	25,372,828	149,645,961
Ave. # Meta-cells	5993	5040	29285
Ave. Time (s)	34.28	46.72	86.52
Total Time (s)	342.75	467.22	865.15
Turb			
Ave. # Triangles	3,582,239	3,291,567	6,498,112
Ave. # Meta-cells	1469	1347	2621
Ave. Time (s)	8.25	10.9	10
Total Time (s)	82.5	109.1	100.3
Vort			
Ave. # Triangles	11,399,060	11,083,529	17,744,784
Ave. # Meta-cells	5279	5123	8320
Ave. Time (s)	33.3	43.54	40.43
Total Time (s)	333.02	435.42	404.29

Table 4. **View-dependent filtering** for isosurface extraction on querying 10 time steps. All three methods used our PTOT. The memory footprint was at most 230MB. (The results of running 4D-POT without occlusion filtering were essentially the same as No-Occ and thus are not shown.)

in that order, as we did in view-independent isosurfacing. All three methods used our PTOT and batched CUDA with $B = 32$, over 10 time steps for each dataset. The results are shown in Table 4. (We also ran 4D-POT without occlusion filtering and got essentially the same results as No-Occ, and thus the results are not shown here.) Note that "Ave. # Meta-cells" means how many meta-cells were read per time step on an average. Comparing these numbers among Implicit, Explicit, and No-Occ we can see how much saving in I/O reads the occlusion filtering achieved. We see that typically Implicit was the fastest, and No-Occ was the slowest. However, for Turb and Vort, Explicit had a much smaller number of I/Os than No-Occ, and yet Explicit was still slower, due to the sorting effect. It is also very interesting that Implicit always had more I/Os than Explicit, and yet was always faster. In fact, Implicit was always the fastest with a large margin, showing the big advantages of our new technique.

Representative isosurfaces resulting from running our Implicit are shown in Fig. 6. In the supplemental material we show a short movie with two video clips side by side, one from running our Implicit (left) and the other from running Explicit (right), on the Vort dataset, where the same isosurface at the same time step was rendered progressively, with our Implicit finished in about 32 seconds and Explicit in about 44 seconds.

5 Conclusions

We have presented a novel technique for isosurface extraction with view-dependent filtering under the out-of-core setting. Our new PTOT data structure achieves optimal searching for active cells in time-varying fields, and in addition supports view-dependent filtering. Our view-dependent filtering approach using implicit occluders can strike a balance between reducing the number of I/Os and reducing the disk seek time, which is simple and effective. In addition, we show how to batch CUDA computations to increase the amount of data being concurrently computed in GPU.

The sorting effect on the I/O cost seems quite interesting, which might deserve further investigation in the out-of-core research.

Acknowledgments

We thank Nvidia for the CUDA code [17].

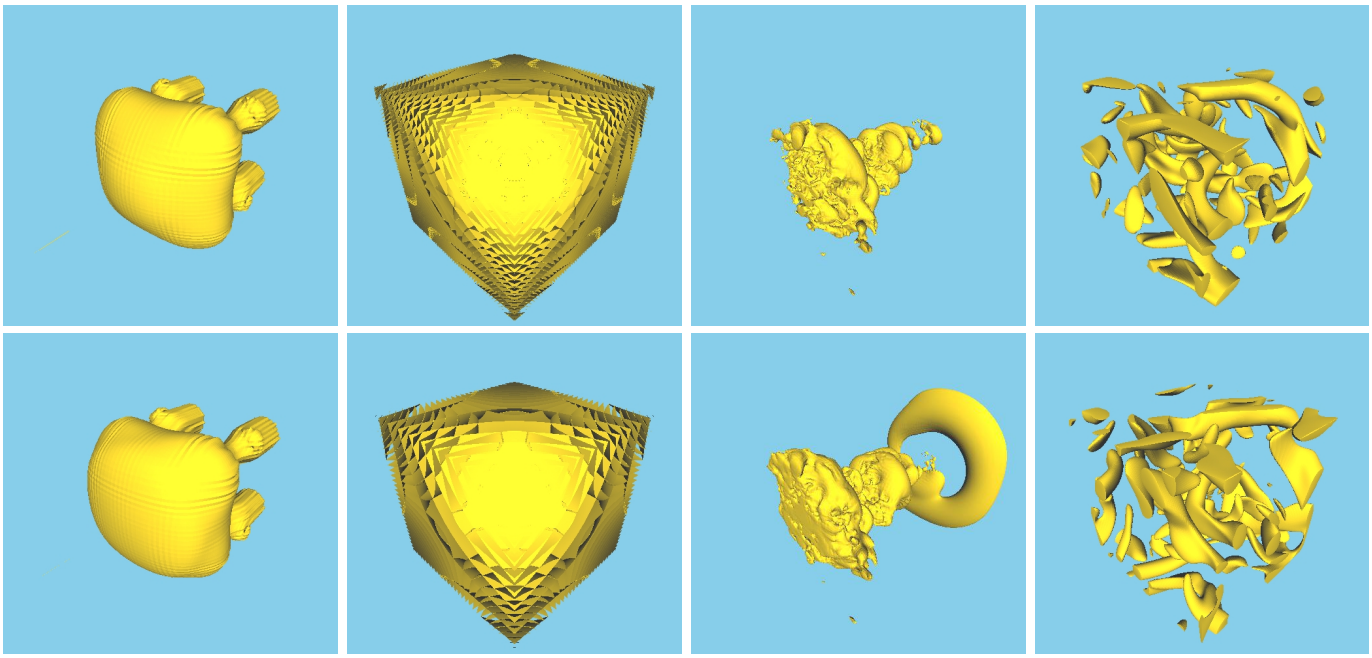

Fig. 6. Representative isosurfaces. Each column shows isosurfaces from the same dataset with two different time steps of the same isovalue. Datasets from left to right: Jets, Syn, Turb, and Vort.

REFERENCES

[1] L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, pages 560–569, 1996.

[2] 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.

[3] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, pages 39–46, Oct. 1996.

[4] U. Bordoloi and H.-W. Shen. Space efficient fast isosurface extraction for large datasets. In *Proc. IEEE Visualization*, 2003.

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

[6] Y.-J. Chiang, R. Farias, C. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc. Sympos. Parallel and Large-Data Visualization and Graphics*, pages 59–66, 2001.

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

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

[9] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[10] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[11] J. Gao and H.-W. Shen. Hardware-assisted view-dependent isosurface extraction using spherical partition. In *Proc. Symposium on Visualization*, 2003.

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

[13] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, Dec. 1995.

[14] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Proc. IEEE Visualization*, pages 175–180, 1998.

[15] Y. Livnat, H.-W. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, Mar. 1996.

[16] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[17] NVIDIA. CUDA SDK. http://www.nvidia.com/content/cudazone/cuda_sdk/Physically-Based_Simulation.html.

[18] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proc. IEEE Visualization*, pages 233–238, 1998.

[19] S. Pesco, P. Lindstrom, V. Pascucci, and C. Silva. Implicit occluders. In *Proc. IEEE Symposium on Volume Visualization and Graphics*, pages 47–54, 2004.

[20] H. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, Oct. 1996.

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

[22] H.-W. Shen and C. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *IEEE Visualization '95*, pages 143–150, Oct. 1995.

[23] Q. Shi and J. JáJá. Isosurface extraction and spatial filtering using persistent octree (POT). *IEEE Trans. Vis. Comput. Graph. (Vis'06)*, 12(5):1283–1290, 2006.

[24] 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.

[25] K. Stockinger, J. Shalf, K. Wu, and E. Bethel. Query-driven visualization of large data sets. In *Proc. IEEE Visualization*, pages 167–174, 2005.

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

[27] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Symp. on Comput. Geom.*, pages 212–220, 1997.

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

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

[30] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.