Isosurface Extraction and Spatial Filtering Using Persistent Octree (POT)

Qingmin Shi, *Member, IEEE* and Joseph JaJa, *Fellow, IEEE*

Abstract— We propose a novel Persistent OcTree (POT) indexing structure for accelerating isosurface extraction and spatial filtering from volumetric data. This data structure efficiently handles a wide range of visualization problems such as the generation of view-dependent isosurfaces, ray tracing, and isocontour slicing for high dimensional data. POT can be viewed as a hybrid data structure between the interval tree and the Branch-On-Need Octree (BONO) in the sense that it achieves the asymptotic bound of the interval tree for identifying the active cells corresponding to an isosurface and is more efficient than BONO for handling spatial queries. We encode a compact octree for each isovalue. Each such octree contains only the corresponding active cells, in such a way that the combined structure has linear space. The inherent hierarchical structure associated with the active cells enables very fast filtering of the active cells based on spatial constraints. We demonstrate the effectiveness of our approach by performing view-dependent isosurfacing on a wide variety of volumetric data sets and 4D isocontour slicing on the time-varying Richtmyer-Meshkov instability dataset.

Index Terms—scientific visualization, isosurface extraction, indexing.

1 Introduction

Isosurface extraction is an important tool for visualizing multi-dimensional scalar fields. It exposes contours of a constant scale value, thus providing an effective way to discover the embedded structures such as the boundaries between different types of tissues, the shock-wave in a fluid dynamics experiment, or the changing of 3D contours in time-varying data sets obtained from physical simulations.

Since the introduction of the Marching cubes algorithm [15], much of the research effort has been put on reducing the amount of data touched for the extraction of the isosurface. A number of efficient techniques have been developed such that the cost of the extraction is more sensitive to the size of the isosurface than to the size of the complete data set [7, 30, 13, 23, 1, 28, 4, 3, 16].

More aggressive approaches have been proposed in recent years to only extract the *relevant* portion of the isosurface needed for visualization. For example, in 3-D isosurface visualization, the view-dependent approach [11, 19] and the ray-tracing approach [18] extract only the visible part of the isosurface. For multi-dimensional data, visualization is possible only for its 3D slices [29]. Thus only isosurfaces in such slices need to be extracted. We call the extraction of relevant portion of the isosurface *spatial filtering*.

In the context of scalar fields sampled on a structured grid, the data set consists of a set of cubes (*cells*) with sampled scalar values associated with their vertices. We call a cell *active* if its value range covers the specified *isovalue*. We call a cell *relevant* if it contains the isosurface patches that need to be rendered. The problem of identifying active cells can be viewed as the *range stabbing query* of computational geometry. The type of problem involved in identifying relevant cells depends on the type of the spatial filtering operation. For example, in view-dependent isosurfacing, it is to single out only the visible cells for triangulation and rendering. In ray tracing, it is to find the cells that the rays shooting from the view point first encounter. And in 4D isocontour slicing, it is to find the cells that are cut by a 4D hyperplane.

- *Qingmin Shi and Joseph JaJa are with the Institute for Advanced Computer Studies and the Department of Electrical and Computer Engineering at the University of Maryland, College Park, E-mail: {qshi,joseph}@umiacs.umd.edu*

Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

In isosurface extraction with spatial filtering, the extraction part is basically a query in the scalar value space and the filtering part is a query in the spatial grid space. The problem is to develop a structure that efficiently supports simultaneous queries both in the value space and in the spatial space.

A straightforward approach to handle the space filtering is to first identify the active cells using existing algorithms (e.g. [13, 4]) and then pick among these active cells the relevant ones. However, this approach may waste a significant amount of time on identifying and then discarding irrelevant cells, as these algorithms produce no particular spatial ordering among the active cells initially identified. Another approach is to use data structures based on spatial partitioning such as the well known min-max octree [30]. This approach is not efficient either since such data structures are not optimal in terms of performing value based queries.

In this paper, we provide a data structure called the *Persistent Octree (POT)* that enables very efficient identification of cells that are simultaneously relevant and active. It possesses the properties of both the interval-tree based isosurface extraction schemes, which are known to be efficient in extracting active cells, and the octree-based schemes (notably the Branch-On-Need Octree (BONO) [30]), which are well suited for spatial filtering. In fact, POT is provably optimal in terms of asymptotic bounds for both space and query time for identifying active cells. It achieves the worst case time complexity of $O(\log N + K)$ for active cell identification, where N is the number of cells in the data set and K is the number of active cells, and requires $O(N)$ space. At the same time, for each possible isovalue, the corresponding active cells are already encoded a priori in a compact octree. Such an inherent hierarchical structure enables very efficient spatial filtering for identifying active cells that are also relevant.

We demonstrate the effectiveness of POT by applying it to view-independent and view-dependent isosurface extraction and 4D isocontour slicing. Our algorithm for view-dependent isosurface extraction follows the general approach of Livnat and Hansen [11, 12], but improves upon theirs in that we visit in a view-dependent way a compact octree consisting ONLY active cells. Our techniques can also be combined with the idea of *implicit occluders* [19] to only traverse the visible part of the compact octree without actually rendering any isosurface. Our 4D isocontour slicing algorithm deals with time-varying data and handles a more general problem than those discussed in [21, 30, 27, 22, 9] in that it allows visualizing not only the isosurface at individual timesteps but also the change of the isosurfaces across timesteps.

We conduct experimental comparison of our approach for view-

dependent isosurface extraction with BONO, which is considered one of the most efficient data structures for isosurface extraction [26]. Results show that our data structure consistently performs better than BONO. In particular, we measure the number of tree nodes visited by POT and BONO for the same view-dependent isosurface extraction queries and show that POT saves a significant amount (as much as 70%) of node visits compared with BONO. We also test our 4D isocontour slicing algorithm on a subset of the Richtmyer-Meshkov instability data set [17] that consists of 35 GB of time-varying data and are able to perform the isosurface extraction, slicing and rendering very fast (for example less than 15 seconds along the spatial dimensions).

The remainder of this paper is organized as follows. We describe the concept of the persistent data structure and its application in range stabbing queries in Section 2. The POT structure and its traversal are presented in Sections 3. In Section 4, we show how POT can accelerate spatial filtering in the cases of view-dependent isosurface extraction and 4D contour slicing. Implementation issues are discussed in Section 5. Our experimental results are reported in Section 6. We discuss limitations of our approach in Section 7 and conclude in Section 8.

2 Persistent Data Structures

The notion of the *persistent data structure* was first introduced by Driscoll et al. [5] as a space-efficient mechanism for maintaining the evolution history of dynamic data structures. It has been used to provide optimal solutions to a number of intersection problems in computational geometry (see for example [20, 2, 10, 24]). A recent paper by Edelsbrunner et al. [6] applied this technique to compute the *Reeb graph*. In this section, we give its properties and explain its application in handling range stabbing queries.

2.1 Properties

Suppose we have an ephemeral data structure, which may be modified due to the insertion or deletion of data elements. Each such modification is associated with a *version number*. The problem is to maintain all the *versions* of the ephemeral data structure such that, given a version number, the corresponding version can be easily accessed.

Persistent data structure is an elegant technique that provides a compact representation of all the versions of a so-called *linked structure*, which consists of a set of nodes with a fixed number of pointers. In particular, Driscoll et al. [5] showed that, if an ephemeral tree structure requires only a constant number of node changes for each insertion or deletion, then it can be made persistent such that each version of the tree can be queried with the same asymptotic time bound as the ephemeral version and a persistent tree structure obtained as a result of N insertions/deletions requires only $O(N)$ space.

The basic idea of making a tree structure persistent is to augment its nodes with additional pointers so that a node can have pointers to different versions of sub-trees. As a result, a new version of a node does not need to be created until enough changes have been made to its successors. The cost of an insertion or a deletion, which possibly includes the creation of new versions of several nodes, thus can be amortized over a long sequence of update operations.

2.2 Handling Range Stabbing Queries

In the context of isocontour generation, assume that we have already computed the minimum and maximum values for each cell. We sort the extreme values in increasing order and employ a value sweep from the smallest extreme value to the largest. In the process, we maintain a data structure D to store the cells whose value ranges are "stabbed" by the current sweeping value. A cell is inserted into D when its minimum value is encountered and removed when its maximum value is reached. Each such update operation creates a new version of D with the version number being the current sweeping value. Figure 1 illustrates the sweeping process. It is easy to see that, given a particular isovalue v, the most recent version of D no later than v stores the exact set of active cells corresponding to v, which means that determining active cells is as simple as traversing this particular version of D.

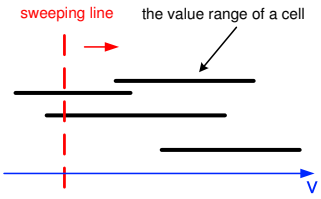


Fig. 1. Handling range stabbing queries using value sweeping.

Of course, our task is not simply to find the active cells. We need to determine the cells that are both active and relevant. To this end, we want D to be a data structure that can efficiently filter out irrelevant cells. To achieve optimality in terms of reporting active cells, we want the size of D to be linear in the number of active cells it stores. And finally, to make it persistent without introducing additional space in an asymptotic sense, addition or deletion operations on D are allowed to incur only a constant number of node changes. In the next section, we introduce the Persistent OcTree (POT), which satisfies all the above requirements.

We shall note that, while the fact that a dynamic binary search tree can be made persistent in a space efficient way has long been established (see [5]), making a octree persistent is not trivial and generally is not possible without introducing non-linear space (we will give such a "bad" case in Section 3.1).

3 Persistent Octrees (POTs)

In this section, we will first describe an ephemeral data structure called the *compact octree* to index the active cells for a particular isovalue, which allows efficient search and update operations. We then show how it can be made persistent to yield an efficient data structure to handle the isosurface extraction with spatial filtering. Our description assumes that the dimension of the data set is three. But exactly the same technique can be applied to higher dimensional data sets as well. In the rest of the paper, we will call the the space occupied by the entire data set a *volume*.

3.1 Compact Octrees

A classic octree is based on hierarchical regular partitioning. The root represents the entire volume. A node u has 8 children, each getting one octant of the subvolume of u. For data resolutions other than powers of two, we adopt the strategy of BONO [30] by viewing the hyperoctree as a complete one but avoiding allocating nodes for empty subtrees. A node at the lowest level of the tree represents a cell and is colored black if the cell is active and white otherwise. A node at a higher level is assigned one of the three colors: black, white, and gray. A black (resp. white) node indicates that the entire subvolume it represents consists of only black (resp. white) cells. A gray node corresponds to a subvolume that contains both black cells and white cells. Figure 2 gives a 2D illustration of the octree.

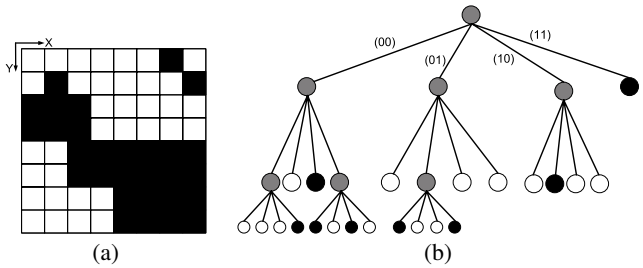


Fig. 2. A 2D illustration of the octree. (a) A set of active cells. (b) The corresponding octree.

One problem with such an octree is that an update operation may require up to $O(\log N)$ node changes, where N is the size of the current

octree. This prevents the octree from being made persistent in a space-efficient way. In fact, consider the case when no value ranges of any two cells overlap. Each insertion or deletion operation would require O(log N) nodes to be modified, resulting in a persistent data structure of size $O(N \log N)$.

In a compact octree, we store only black and gray nodes, and replace pointers to white nodes with NULL pointers. In addition, we "collapse" subtrees where each node has only one child. Formally, consider a path Π from the root to a leaf node and let $u_1, u_2, \ldots, u_k$ be a subpath of Π such that i) u_1 is not the root and thus has parent u_0; ii) u_0 either is a root node or has more than two gray or black children; iii) u_i is the only gray or black child of u_{i-1} for $i = 2, \ldots k$; and iv) u_k is either a black node or has at least two gray or black children. We then replace the pointer in u_0 which points to u_1 with a jumper, which points to u_k and stores the path from u_0 to u_k. The subvolume of a node w is uniquely determined by the path from the root to w. Figure 3 shows the compact octree derived from the one shown in Figure 2(b).

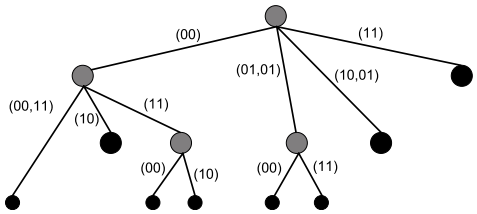


Fig. 3. A 2D illustration of the compact octree. Each jumper is associated with a path, which is represented by a list of 2-bit strings. NULL pointers are omitted. Each 2-bit string gives the YX encoding of the branch index of the corresponding edge in the original octree.

Since each node in the compact octree, except for the root, has at least two children and each leaf node represents at least one active cell, the size of the tree structure is obviously linear in the number of active cells it stores and so is the time it takes to traverse the tree to report these active cells. Furthermore, we can show that the compact octree requires only an amortized constant number of node changes for each insertion or deletion when applied to our particular problem (see Section 4.1 of [25] for details). This is due to the fact that, during the value sweeping process, each cell will be inserted into and removed from the compact octree exactly once.

3.2 Making a Compact Octree Persistent

In a compact octree, each internal node has exactly 8 (possibly NULL) jumpers. To make such a tree persistent, we allow a node to hold k additional jumpers, where k is a small constant. Each of the $8+k$ jumpers is associated with a version number and a path.

An update operation with a new version number v is always performed on the latest version of the POT. New nodes are created as necessary. When we need to change a jumper in a node u, we first try to find an empty slot in u. If there is one, then the new jumper is added to u along with the version number of the update operation. Otherwise, we create a copy u' of u. The initial 8 jumpers of u' are set to be their latest values in u and are assigned the version number v. Note that we also need to add a jumper to u' to the latest parent of u'. Thus, this jumper copying step will be propagated towards the root until a node with a free slot is reached or the root itself is copied. An update operation with the latest existing version number will replace jumpers rather than copying them.

Figure 4 illustrates a POT using a 1D example. We have a grid of 8 1D "cubes" (segments) each of which is labeled by a 3-bit string. In Figure 4(a), these segments are represented by vertical stripes separated by vertical lines with their labels shown at the bottom of the respective stripes. The numbers on the left are the scalar values. The vertical extent of the shaded rectangles represents the value ranges of the corresponding segments. For a particular isovalue, the active segments are indexed by a compact binary tree. Figure 4(b) gives the persistent binary tree, in which each node has three available slots for jumpers. The label Ln or Rn tells whether a jumper corresponds to a left branch or a right branch as well as the associated version number. The numbers above the root nodes are their version numbers and the binary string beside each of the remaining nodes indicates the subvolume it represents.

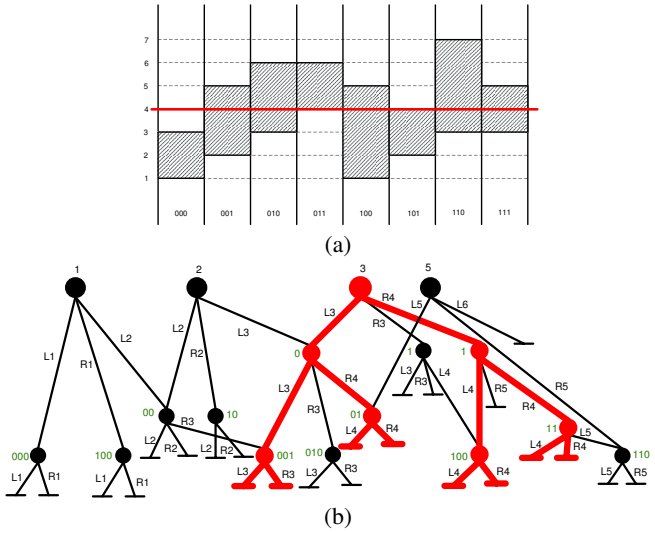


Fig. 4. A 1D illustration of the POT. Each node has three slots for jumpers. (a) The evolution of active segments for different isovalues. (b) The resulting POT. An edge with a bar underneath it represents a NULL jumper.

3.3 Constructing a POT

To construct POT, we first collect all the cells, and temporarily keep two copies for each of them. One copy uses the minimum value as its key and the other uses the maximum value. We sort the cells using their keys in increasing order and then scan through the sorted list. For each cell c we encounter, if its key is its minimum value, we insert c into the current version of the compact octree. Otherwise we delete c from the current version. If this value is larger than the most recent version number, a new version is created. Since an insertion or a deletion on a compact octree requires only $O(1)$ node changes, the POT is linear in the total number of cells in the data set. To further reduce the size of the POT, we do not actually store a node w if it is a leaf for all the versions of the POT it belongs to. Such omission is recorded by repointing any jumper to w to the node where that jumper is stored.

Sorting the $2N$ cells requires $O(N \log N)$ time. The construction of the POT itself is also $O(N \log N)$ because each insertion or deletion may touch a single path in the POT from the root to a leaf whose length is at most $O(\log N)$.

3.4 Traversing A POT

To traverse the version of a POT corresponding to a particular isovalue v, we first identify the root with the largest version number smaller than or equal to v. To report all the active (but not necessarily relevant) cells, we simply traverse an appropriate version of the POT by following the latest jumpers no later than v. For example, in Figure 4(b), the portion of the POT colored in red and connected by bold edges is the active version corresponding to isovalue 4. Once we reach a leaf node, we report all the cells within its corresponding subvolume. It is easy to see that the complexity of reporting all the active cells is $O(\log M + K)$, where M is the number of roots in the POT and K is the number of active cells. The $O(\log M)$ term reflects the fact that we may have to find the appropriate version of the root using a binary search. $M \leq N$ and typically is small enough (in the order of tens in all of our experiments) to be considered a constant. Consequently, the complexity of reporting active cells using a POT depends solely and linearly on the number of active cells.

4 Spatial Filtering of Active Cells using POT

A more attractive feature of the POT is that each version of the POT provides a spatial partitioning of the active cells, which enables very efficient spatial filtering. we now discuss it in the context of view-dependent isosurface extraction and 4D contour slicing.

4.1 View-dependent isosurface extraction

A view-dependent isosurface extraction avoids triangulating and rendering invisible portions of the isosurface. The most commonly used data structures for the occlusion test are based on the octree. The nodes in the octree are visited in a front-to-back order based on the view point. An occlusion mask is maintained against which the visibility of the octree nodes are tested. Only visible nodes and their descendants are examined further.

Using a standard octree (or octree for short), as long as there is an active cell v in the corresponding subvolume, a node will be active. This means a long list of nested nodes containing v may have to be examined, since nodes at higher level (thus corresponding to larger subvolumes) are more likely to be determined as visible even if the patch of the isosurface in it only occupies a tiny portion of the subvolume. Worse yet, that patch may not even be visible. On the other hand, in POT we can follow the jumpers to get to a small active subvolume v very quickly and determine if the isosurface patch inside it is visible. Figure 5 illustrates such a case. In Figure 5, the active cells are colored in gray. There are three sections of the isosurface: A, B, and C. Part of A and B are visible and the entire section of *C* are occluded by *A*. In order to determine the visibility of the cell that contains *C*, an octree will have to examine eight nodes in the upper-left quarter of the data space (four children of the node *w* corresponding to the left-top quarter and four grand children of *w*, including the one containing *C*). Using POT, only the node containing *C* needs to be checked against the occlusion mask.

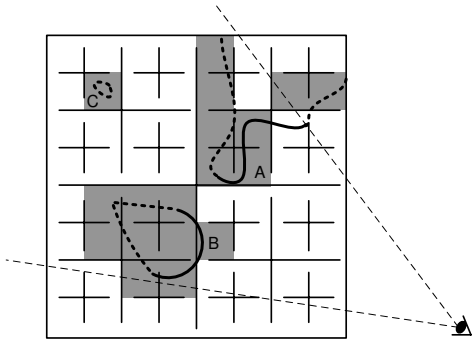


Fig. 5. A 2D illustration of the pruning of the POT for view-dependent isosurface extraction.

4.2 4D isocontour slicing

In 4D isocontour visualization, we want to slice the 4D isocontour using a 4D hyperplane to generate a 3D isosurface. POT provides a simple and efficient way to identify the active cells that are also cut by the hyperplane. At each node *u* of the version of POT that corresponds to the isovalue, we check each jumper to see if the subvolume of the node *w* it points to is cut by the hyperplane. If this is not the case, then the subtree rooted at *w* will not be visited. Notice again that using jumpers allows us to quickly get to a node representing a small active subvolume without having to access a long list of nested subvolumes at higher levels.

5 Implementation Issues

We implemented the internal-memory view-dependent isosurface extraction algorithm using POT as the underline indexing structure. The overall scheme is similar to the one described in [11]. We use a hierarchical set of occlusion masks to keep track of the screen pixels that are covered by previously extracted isosurface patches. Each pixel of an occlusion mask at a certain level corresponds to an 8×8 array of pixels at the next level. Precomputed binary coverage masks are used to quickly determine the pixels covered by the newly generated isopatches and to update the occlusion masks at the lowest level. Triage coverage masks [8] are used to speedup the update of the occlusion masks as well as the occlusion tests at higher levels. Like [12], the update of the occlusion mask is performed from bottom up, thus restricting the propagation of the change to a limited number of levels. The bounding rectangle of the projection of a node is used to conservatively test its visibility.

As suggested in [14], our POT is built not on individual cells, but on a small cube of cells ($4 \times 4 \times 4$ in our implementation) called *supercells*. Using supercells not only reduces the size of the indexing structure but also avoids excessive visibility tests, which usually are not justifiable at very low levels of the structure. Once we have identified an active and visible supercell, we search it using the standard Marching Cubes algorithm to identify the active cells and perform triangulation on them without performing the visibility test.

Figure 6 shows a view dependent rendering of the MRbrain data set, a side view of the visible portion of the isosurface and the final occlusion mask.

For 4D isocontour slicing, the size of the data set is often too big to fit in memory and therefore has to reside on disk. We choose the size of a supercell to be approximately the same as a disk block size ($16 \times 16 \times 16$ in our experiment) so that it can be loaded into memory using one I/O access. When searching the POT, we maintain the path from the root to every leaf node we reach. This path provides sufficient information for locating the corresponding supercell on disk and loading it into memory for further processing.

6 Experimental Results

We conducted two sets of experiments to evaluate the performance of POT. In the first set, we compared the POT to the classic BONO structure in the case of view-independent and view-dependent isosurface extraction for 3D volumetric data. In the second set of experiments, we studied the performance of POT for 4D isocontour slicing of time-varying data.

System setup Our experiments were performed on a PC with dual 3.0 GHz Xeon processors, 8 GB main memory, 140 GB local disk with around 50 MB/sec I/O peak transfer rate, and one NVidia6800 Ultra GPU card with a bi-directional 1 Gbps data transfer rate to memory via PCI-Express (x16) Bus. It runs Linux 2.4.21-27.ELsmp. Only one processor was used.

Data description Four sets of data were used in our test. The complete Richtmyer-Meshkov instability (R-M) data set consists of 274 timesteps, each consisting of a 3D grid of $2048 \times 2048 \times 1920$ 8-bit scalar values. We downsampled each timestep by a factor of 2 in each dimension. For 4D isocontour slicing we used every 8th timestep starting from timestep 0. The resulting data set consisted of 35 timesteps and the overall size of the data set was about 35 GB. For view-dependent isosurface extraction we only used timestep 248. A summary the other three data sets along with a single timestep in the Richtmyer-Meshknov data set is given in Table 1. The construction times of the POTs for these data sets are reported in Table 2. Figure 8 gives some views of these data sets.

data set	data type	resolution
R-M	byte	$960 \times 1024 \times 1024$
Stanford bunny	short	$360 \times 512 \times 512$
MRbrain	short	$109 \times 256 \times 256$
Head Aneuyrism	byte	$512 \times 512 \times 512$

Table 1. Description of the data sets used in our experiments.

As a base for comparison, we also implemented the BONO algorithm. At each internal node of the BONO, we stored a single pointer to its first child and a bit mask stating which children actually exist. As in our POT implementation, the BONO was also built on $4 \times 4 \times 4$

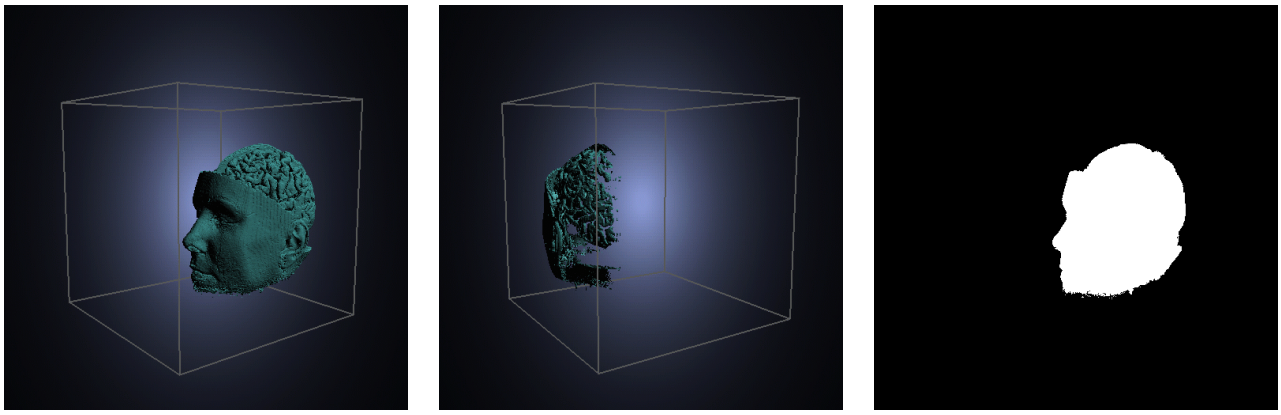


Fig. 6. View dependent rendering of the MRbrain data set. Left: the normal view of the MRbrain data set; Center: a side view of the visible part of the isosurface; Right: the final occlusion mask at the lowest level.

data set	R-M	Stanford Bunny	MRbrain	Head Aneuyrism
const. time (sec.)	356	35	3	38

Table 2. Construction time of the POTs.

supercells. Table 3 compares the sizes of the POTs and BONOs. POT requires more space than BONO by a factor of 1.6 to 4, which is not surprising, considering the fact that our BONO implementation is almost pointerless and POT stores a number of pointers at each node.

6.1 View-independent and view-independent isosurface extraction

We compare the performance of BONO and POT in terms of search efficiency for view-independent and view-dependent isosurface extraction. The comparison of index searching time for view-independent isosurfacing is reported in Table 4. The screen size was 512×512. We note that the actual execution time also includes the time to search the supercells, perform triangulation, and render the isosurface, which is identical for both BONO and POT.

Except for the Stanford bunny data set, POT was able to achieve a significant speedup relative to BONO. Its search time is 34% less than that of BONO for the R-M data set, 18% for the MRbrain set, and 16% for the Head Aneuyrism set.

The performance comparison of these two data structures for view-dependent isosurface extraction is given in Table 5. Unlike view-independent isosurface extraction where all active cells are extracted before the other steps take place, the surface extraction, triangulation, and rendering steps are intertwined in view-dependent isosurface extraction. It is very difficult to accurately measure the index search time. Therefore, we instead report the number of nodes visited and the overall execution time. However, we must point out that, as explained before, the latter is not an appropriate measurement of the effectiveness of the index structure. The number of tree nodes visited is a more accurate comparison benchmark for two reasons. First, it is implementation independent. Second, since for the same isovalue and view point, the number of active and relevant active cells for both POT and BONO are exactly the same, this benchmark more accurately reflects how efficiently each data structure is searched.

It can be seen that POT performs much better than BONO for all the data sets in terms of the number of nodes visited. The number for POT is only 30% to 73% that of BONO. This is consistent with our argument that POT locates active and relevant cells more efficiently. We also observe that the overall execution time of POT is also consistently better than that of BONO, albeit in a lesser degree.

6.2 4D isocontour slicing

In this experiment, we demonstrate the efficiency of our scheme in computing the slices of the 4D isosurfaces along different axes, especially the X, Y, and Z axes, in which case the slices reflects the change of the isosurface across time steps.

Constructing the 4D POT for the 35 time steps of the R-M data set took about 68 minutes, a majority of which was spent on collecting the extreme values. The construction of the POT itself took only 2 minutes. The resulting indexing structure occupies 106 MB of memory.

We measure the performance of our algorithm using different combinations of isovalues and cutting planes. Figures 7(a), (b) and (c) show the execution time of our program as a function of the isovalue for three different cutting hyperplanes as well as its decomposition into five computation steps. We also give the number of triangles generated in each case in Figure 7(d).

It can be seen that the number of triangles varies from 26 million for isovalue 210 to 146 million for isovalue 70. We were able to extract and render the most complicated cuts (along the T-axis) in less than 100 seconds, and much faster for other cuts. Searching the POT is very fast in all the cases. Even for the most time-consuming slicing along the T-axis, it only took 0.26 seconds on average.

7 Discussion

We now briefly discuss the limitations of our technique. First, the data sets we used are from discrete fields, and hence the number of possible versions is limited. This presents a unique opportunity to avoid storing the cells with constant values, since they are inserted and removed during the update of a single version of the POT. This may not be the case if the data sets are from continuous fields, which could lead to larger indexing structure size. However, we do not expect the increase in storage to be significant since changes of the isosurface between versions would be smaller, which allows better node sharing between versions.

A second possible limitation is that the construction time of a POT is typically longer than that of a BONO. However, we intend this technique to ultimately be used for interactive exploration of the data sets and hence the preprocessing time is less important.

Finally, for view-dependent rendering, our scheme, similar to the well known BONO/Grid Tree based scheme [11], requires the front-to-back generation of the triangles in order to update the coverage mask. This may lead to inefficient I/O operation for large data sets residing on disk. One possible solution is to use the technique of *Implicit Occluders* [19] to first generate the coverage mask and then perform a traversal of the visible part of the octree. This approach requires further exploration.

data set		R-M	Stanford Bunny	MRBrain	Head Aneuyrism
Storage Cost (byte)	BONO	143,805,000	20,231,340	1,524,156	19,173,960
	POT	229,855,564	61,510,700	4,955,424	76,495,280

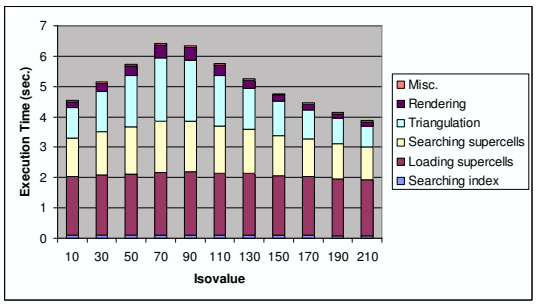
Table 3. Storage requirements for BONO and POT (index only).

data set		R-M	Standford Bunny	MRbrain	Head Aneuyrism
isovalue		190	1750	1750	55
BONO	index search time (ms)	710	64	17	22
POT	index search time (ms)	401	61	14	15
Speedup of index searching time relative to BONO		1.77	1.05	1.21	1.47

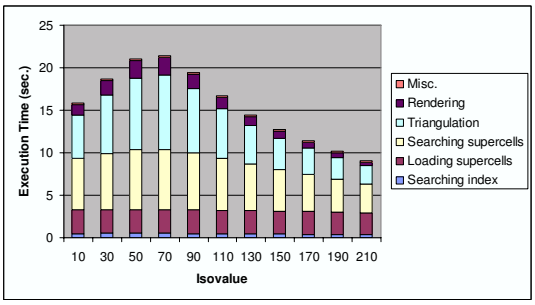
Table 4. Performance comparison of view-independent isosurface extraction.

data set		R-M	Standford Bunny	MRbrain	Head Aneuyrism
BONO	# of Nodes Visited	212,992	19,897	7,318	17,260
	Execution Time (ms)	10,877	1,130	614	938
POT	# of Nodes Visited	142,157	6,076	5,368	6,467
	Execution Time (ms)	9,511	1,040	602	806
Speedup relative to BONO	# of Nodes Visited	1.50	3.27	1.36	2.67
	Execution Time	1.14	1.09	1.02	1.16

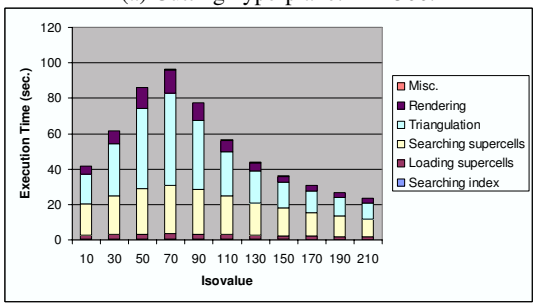
Table 5. Performance comparison of view-dependent isosurface extraction. The same set of isovalues as in the Table 4 are used.



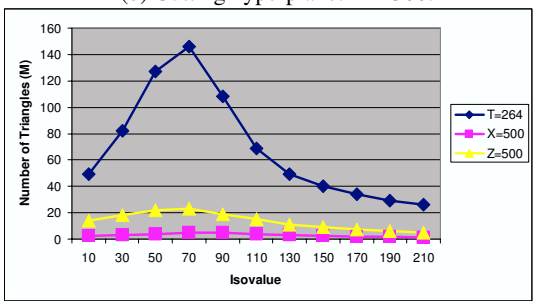
(a) Cutting hyperplane: X = 500.



(b) Cutting hyperplane: Z = 500.



(c) Cutting hyperplane: T = 264.



(d) # of triangles generated for different cuttings

Fig. 7. Performance measurements for 4D isocontour slicing.

8 Conclusion

We have presented a novel POT data structure to accelerate the isosurface extraction with spatial filtering. It is asymptotically optimal in terms of space and search time. The set of active cells for an isovalue naturally form a compact octree that allows the active cells to be filtered based on certain spatial criteria very quickly. In particular, we demonstrate the effectiveness of this technique by applying it to view-dependent isosurface extraction and 4D isocontour slicing. Our experiments show that for view-dependent isosurface extraction, our data structure performs consistently better than the widely used BONO structure. For 4D isocontour slicing, our tests on the Richtmyer-Meshkov data set show that POT enables very fast search and that the overall algorithm can perform simultaneous isocontouring and slicing in a very efficient manner.

Because of the inherent hierarchical structure associated with the active cells for any isovalue, POT can be used to improve the performance of other visualization schemes such as ray tracing [18] and multi-resolution isosurface rendering, since they also rely on the hierarchical pruning of the data volume.

Acknowledgements

We would like to thank Mark Duchaineau for making the Richtmyer-Meshkov instability data set available to the University of Maryland through Amitabh Varshney, and Amitabh Varshney for getting us interested in this problem and for his gracious help at various stages of this research. The Standford terra-cotta bunny data set and the MRbrain data set were obtained from the Stanford Volume Data Archive (http://graphics.stanford.edu/data/voldata/). The

Head Aneuyrism data set was obtained from the data archive at the Wilhelm Schickard Institute for Computer Science GRaphical-Interactive Systems, University of Tübingen (http://www.gris.uni-tuebingen.de/areas/scivis/volren/datasets/new.html). This work was supported in part by the NSF Research Infrastructure Grant CNS-04-03313.

REFERENCES

[1] U. Bordoloi and H.-W. Shen. Space efficient fast isosurface extraction for large datasets. In *IEEE Visualization '03*, pages 201–208, 2003.

[2] A. Boroujerdi and B. Moret. Persistency in computational geometry. In *Proc. 7th Canadian Conf. Comp. Geometry (CCCG 95)*, pages 241–246, Québec, Canada, 1995.

[3] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization '98*, pages 167–174, 1998.

[4] 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.

[5] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[6] H. Edelsbrunner, J. Harer, A. Mascarenhas, and V. Pascucci. Time-varying reeb graphs for continuous space-time data. In *SCG'04: Proceedings of the twentieth annual symposium on computational geometry*, pages 366–372, New York, NY, USA, 2004.

[7] R. S. Gallagher. Span filtering: an optimization scheme for volume visualization of large finite element models. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 68–75, 1991.

[8] N. Greene. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1996.

[9] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. I. Joy. Interactive view-dependent rendering of large isosurfaces. In *IEEE Visualization '02*, pages 475–484, 2002.

[10] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19:282–317, 1995.

[11] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *IEEE Visualization '98*, pages 175–180, 1998.

[12] Y. Livnat and C. Hansen. Dynamic view dependent isosurface extraction. Technical Report UUSCI-2003-004, Scientific Computating and Imaging Institute, University of Utah, Salt Lake City, UT 84112, USA, 2002.

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

[14] Y. Livnat and X. Tricoche. Interactive point-based isosurface extraction. In *IEEE Visualization '04*, pages 457–464, 2004.

[15] W. E. Lorensen and H. E. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[16] A. Mascarenhas, M. Isenburg, V. Pascucci, and J. Snoeyink. Encoding volumetric grids for streaming isosurface extraction. In *2nd International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT 2004)*, pages 665–672, 2004.

[17] A. A. Mirin, D. H. Porter, P. R. Woodward, L. J. Shieh, S. W. White, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, and S. E. Anderson. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM)*, 1999.

[18] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, 1998.

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

[20] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[21] H.-W. Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *IEEE Visualization '98*, pages 159–166, 1998.

[22] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partition (TSP) tree. In *IEEE Visualization '99*, pages 371–377, 1999.

[23] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.

[24] Q. Shi and J. JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Information Processing Letters*, 95:382–388, 2005.

[25] Q. Shi and J. JaJa. Efficient isosurface extraction for large scale time-varying data using the persistent hyperoctree (phot). Technical Report CS-TR-4776, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2006.

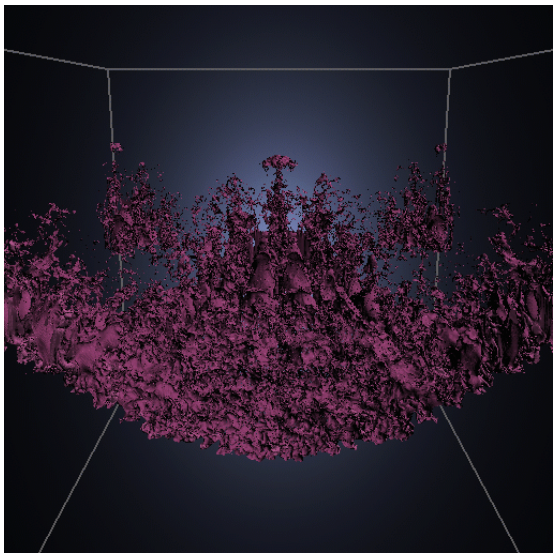
[26] P. Sutton, C. Hansen, H.-W. Shen, and D. Schikore. A case study of isosurface extraction algorithm performance. In *2nd Joint Eurographics-IEEE TCCG Symposium on Visualization*, pages 259–268, 2000.

[27] P. Sutton and C. D. Hansen. Isosurface extraction in time-varying fields using a Temporal Branch-on-Need Tree (T-BON). In *IEEE Visualization '99*, pages 147–153, 1999.

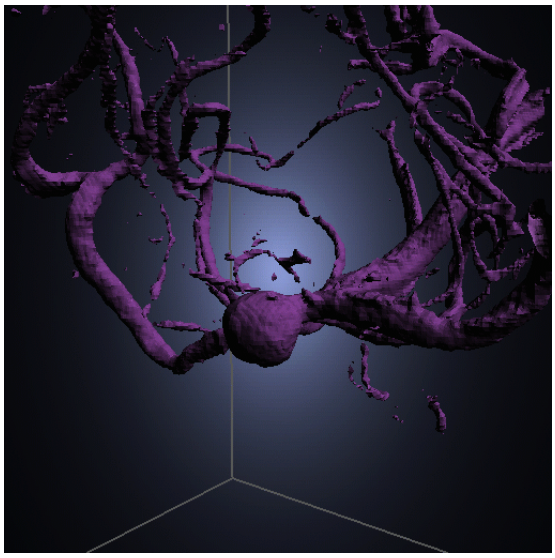
[28] K. W. Waters, C. S. Co, and K. I. Joy. Isosurface extraction using fixed-sized buckets. In *EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, pages 207–214, 2005.

[29] C. Weigle and D. C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, pages 103–110, 1998.

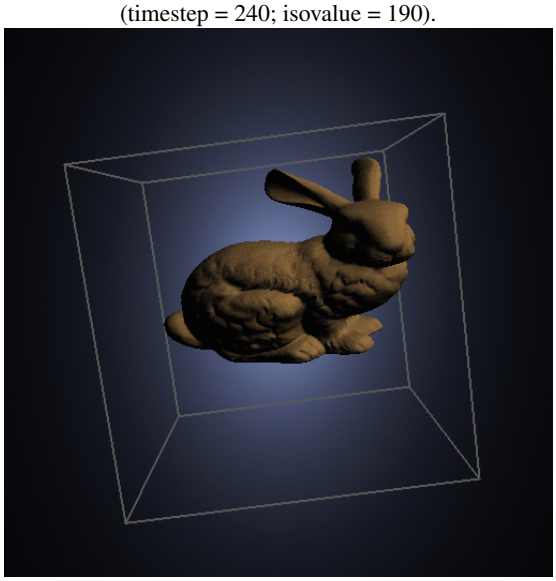
[30] J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.



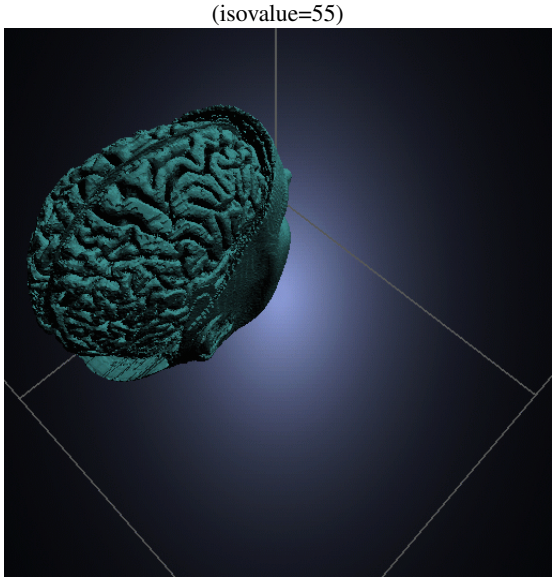
(a) A normal view of the Richtyer-Meshkov instability data set (timestep = 240; isovalue = 190).



(b) A close view of the Head Aneuyrism data set. (isovalue=55)

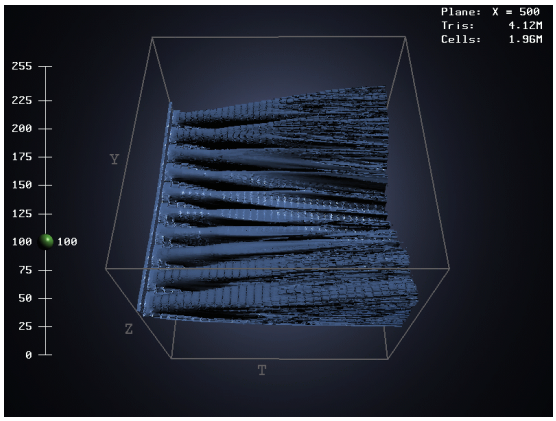


(a) A normal view of the Stanford terra-cotta bunny. (isovalue = 1750)

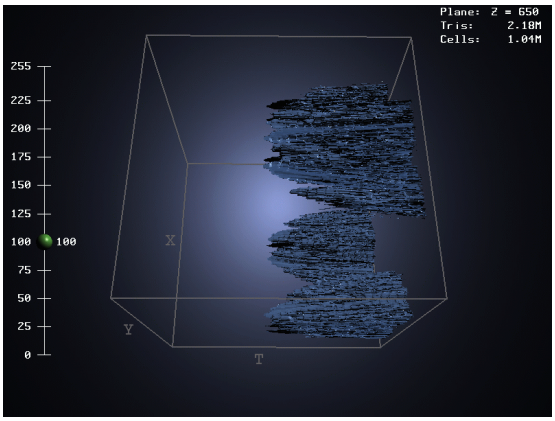


(b) A top view of the MRbrain data. (isovalue = 1750)

Fig. 8. Views of the data set used.



(a) Isovalue: 100; Cutting hyperplane: X = 500.



(b) Isovalue: 100; Cutting hyperplane: Z = 650.

Fig. 9. Slices of 4D isocontours.