Efficient Isosurface Extraction for Large Scale Time-Varying Data Using the Persistent Hyperoctree (PHOT)

(CS-TR-4776 & UMIACS-TR-2006-01)

Qingmin Shi and Joseph JaJa
Institute for Advanced Computer Studies,
Department of Electrical and Computer Engineering,
University of Maryland, College Park, Maryland

Abstract

We introduce the Persistent HyperOcTree (PHOT) to handle the 4D isocontouring problem for large scale time-varying data sets. This novel data structure is provably space efficient and optimal in retrieving active cells. More importantly, the set of active cells for any possible isovalue are already organized in a Compact Hyperoctree, which enables very efficient slicing of the isocontour along spatial and temporal dimensions. Experimental results based on the very large Richtmyer-Meshkov instability data set demonstrate the effectiveness of our approach. This technique can also be used for other isosurfacing schemes such as view-dependent isosurfacing and ray-tracing, which will benefit from the inherent hierarchical structure associated with the active cells.

1 Introduction

A great amount of time-varying data has been generated in recent years, with a very significant contribution from large scale scientific simulations. Most of this data comes as a time-series of volume data. Efficient visualization tools are needed to reveal not only the structures in the 3D scalar field at individual timesteps, but also the evolution of these structures across timesteps. The problem is made even more challenging because of the drastically increased data set sizes due to more complex and finer scale simulation models. For example, the complete Richtmyer-Meshkov instability data set [MPW$^+$99] as a result of a simulation of a shock tube experiment consists of 274 timesteps with a $2048 \times 2048 \times 1920$ grid of scalar field for each timestep, resulting in a total of about 2.1 terabytes of data. Effective analysis of such large scale time-varying data requires new techniques for data access and visualization.

In this paper, we focus on one of the most commonly used and effective techniques for visualizing scalar fields called *isocontouring*. An isocontour in a d-dimensional scalar field is a $d-1$ dimensional manifold defined by a specified scalar value, called *isovalue*. In a 3D scalar field, an *isocontour* is a 2D surface referred to as an *isosurface*.

A common way to visualize time-varying data is to generate a sequence of isosurfaces for individual timesteps [She98, SCM99, SH99, GSDJ04]. A high degree of interactivity is needed in such an approach in order to provide visual cues to the discovery of correspondences between timesteps. Even though significant progress has been made in accelerating the isosurface extraction process by exploring the spatial and temporal correlation between time steps [She98, SCM99, SH99, Chi03, GSDJ04], such a requirement is still hard to meet especially for very high resolution data. Few recent systems (for example [GSG01, DPH$^+$03]) that achieve limited interactivity all require significant computational resource such as computer clusters or shared-memory multiprocessors, which may not be easily accessible to many researchers. Even if such interactive capability is available, studying temporal structures of isosurfaces can be difficult as associations between spatial patterns in different timesteps may be hard to capture and analyze.

Another approach is to treat a time-varying data set as a 4D scalar field, with time as the fourth dimension [WB96, WB98, BWC04]. A 4D isocontour is generated, which can be visualized by slicing using a

4D hyperplane (the *cutting hyperplane*). This approach provides a straightforward description of the evolving isosurface over time without having to generate a complete isosurface for each timestep.

Time-varying data from scientific simulations typically come in the form of a series of regular grid of scalar values, which can be viewed as a regular grid of 4D hypecubes (*cells*). An isocontour in this case is usually generated by first identifying the *active cells* whose value ranges include the specified isovalue, and then performing piece-wise triangulation for each active cell. The problem of identifying active cells can be viewed as the *range stabbing query* of computational geometry.

A major bottleneck in visualizing 4D isocontours is the identification of cells that are relevant to the isocontour slice to be rendered. Unlike the 3D case, not all the active cells need to be accessed since they may not intersect the cutting hyperplane. We call a cell that is intersected by the cutting hyperplane a *relevant cell*. Thus our task is to avoid accessing cells that are not both active and relevant. A possible solution is to filter out irrelevant but active cells after all the active cells are identified. Efficient solutions have been developed for the latter task [CMM+97, LSJ96, SHLJ96]), but they do not provide a structured representation of the output active cells, making the filtering step time-consuming. On the other hand, a 4D hyperoctree augmented with extreme values [WvG92] allows relevant cells to be identified quickly but does not guarantee the efficiency in determining active cells.

In this paper, we provide a data structure that enables the identification of active and relevant cells using a techniques called *persistent data structures* [DSST89]. Like the interval tree [SHLJ96], our data structure requires only linear space and is able to report active cells in optimal $O(\log n + k)$ time, where n is the number of cells in the data set and k is the number of active cells. Furthermore, for each particular isovalue, the corresponding active cells are already organized as a 4D hyperoctree and thus can efficiently be sliced using the cutting hyperplane. We call such a tree a Persistent HyperOctree (or a *PHOT*). Because the active cells are naturally organized in a hierarchical spatial representation, this technique can also be used for other isosurfacing schemes, such as view-dependent isosurfacing, isosurfacing by ray tracing, and adaptive resolution isosurface construction.

We have used PHOT to develop a very efficient out-of-core isosurfacing algorithm and applied it to a significant subset of the Richtmyer-Meshkov instability data set [MPW+99] that consists of 35 GB of data. This algorithm allows the slicing of the isocontour along any dimension. An additional feature of our algorithm is that it preserves the initial data layout, making the data also available to other visualization and analysis tasks. Our experiments show that this algorithm was able to extract and render isosurfaces very fast (for example less than 15 seconds along the spatial dimensions).

We note that, concurrently with this work, a different scheme was developed in [KJ05] for isosurface extraction of time-varying data. While the data structure in [KJ05] is smaller than PHOT and performs about the same along the temporal axis, PHOT achieves superior performance for cuts along the spatial dimensions and can be used to handle other isosurfacing techniques as mentioned above.

The remainder of this paper is organized as follows. In Section 2 we give a brief review of related techniques. We describe the main idea of the persistent data structure and its application in interval stabbing queries in Section 3. The PHOT structure and its search algorithm are presented in Sections 4 and 5 respectively. Implementation details of our isosurfacing algorithm are provided in Section 6. We discuss our experimental results in Section 7 and conclude in Section 8.

2 Related Previous Work

The well known Marching Cubes algorithm [LC87] generates an isosufrace by sequentially searching for the active cells and performing triangulation within each active cell. Subsequent techniques have been proposed to accelerate the process of finding active cells. Wilhelms and van Gelder [WvG92] built an Branch-on-Need Octree (BONO) on a grid of cells. Each node of BONO stores the extreme values in its subtree to guide the search algorithm. The effectiveness of this technique is generally sensitive to the value distribution in the scalar field. Subsequent algorithms view the problem of identifying active cells as a *range stabbing query*. They either solve the problem by viewing the value ranges as 2D points in the so-called *span space* [LSJ96, SHLJ96, BS03, WCJ05], or handle the value ranges directly [CMM+97, CSS98, MIPS04]. Some of them [CMM+97, MIPS04] achieve optimal search complexity. However, all these methods require

rearrangement of the data, thus limiting its usage for other applications. More importantly, the set of active cells reported by the search algorithm is not structured, which makes further filtering steps expensive.

A number of previous papers have addressed the isosurfacing problem concerning time-varying data. Shen's *Temporal Hierarchical Index Tree (THIT)* [She98] puts cells, whose values do not change much over time in high-level nodes of a *time tree*, thus reducing the storage cost. Sutton and Hansen extend the BONO [WvG92] to derive an I/O efficient Temporal Branch-On-Need Octree (T-BON) [SH99]. This algorithm basically uses one BONO for each timestep, but allows all the trees to share the same structural information. Shen, Chiang, and Ma's Time-Space Partition Tree [SCM99] can be viewed as a further modification of the T-BON algorithm in that the summary information stored at an octree node is organized as a binary tree built on the timesteps. Recently, Gregorski et al. [GDL+02] proposed using hierarchical tetrahedral mesh to extract time-varying isosurfaces, which enables fast generation of isosurfaces through localized mesh refinement and coarsening. All the above mentioned techniques only focus on generating isosurfaces for separate timesteps.

Viewing time-varying 3D volume data as 4D data was first suggested by Weigle and Banks [WB98]. They proposed using recursive contour meshing [WB96] to split cells into simplices and then contour them using the isovalue. The resulting isocontour is then sliced for visualization. Bhaniramka, Wenger, and Crawfis [BWC04] provided a general algorithm using convex hulls to generate the look-up table for piece-wise triangulation within each cell, thus avoiding the split operations. These attempts do not address the problem of determining active cells using both the isovalue and the cutting hyperplane simultaneously.

Persistent Data Structure was first introduced by Driscoll et al. [DSST89] 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 [ST86, BM95, GJS95, SJ05]), including the range stabbing queries. The PHOT described in this paper solves a problem harder than the standard range stabbing query.

3 Persistent Data Structures

3.1 Properties

Suppose we have an ephemeral data structure, which may be modified due to insertion or deletion of data elements. Each such modification is associated with a time stamp v referred to as 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. An obvious solution would be to make a separate copy for each version whenever a change is made. However, the storage cost of doing so would be prohibitive.

Persistent data structure is an elegant technique that provides a compact representation of all the versions of a so-called *linked data structure*. A linked data structure consists of a set of nodes with a fixed number of pointers. Let an *update operation* be the addition/deletion of a data item to/from the data structure and let an *update step* be an atomic modification operation that creates a new node or changes a pointer. Driscoll et al. [DSST89] showed that, if an ephemeral linked data structure D has a constant in-degree, then it can be made persistent such that each update step only contributes $O(1)$ amortized space to the persistent data structure and that each version of the persistent data structure can be queried with the same asymptotic time bound. This means that, if an update operation requires only $O(1)$ update steps, then the persistent data structure obtained as a result of n update operations requires only $O(n)$ space.

The basic idea of making a data structure persistent is to augment its nodes with additional pointers so that a node can have pointers to different versions of sub-structures. As a result, a node does not need to be copied until enough changes have been made to its successors. The cost of such copying operation is thus amortized over a number of update operations. A proof of this property can be found in [DSST89].

3.2 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. We insert a cell into D when its minimum value is encountered and remove it 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.

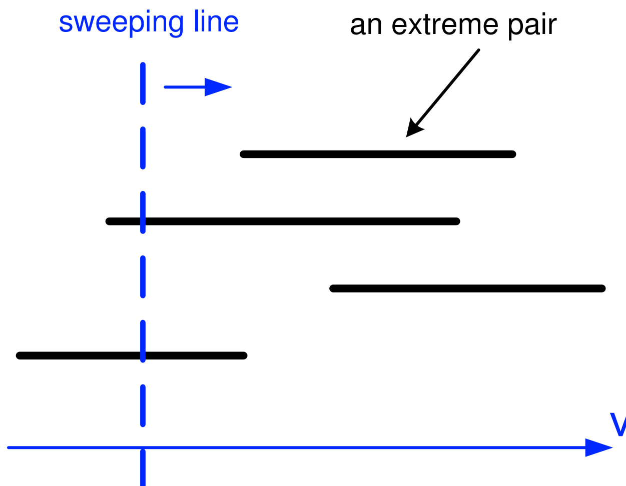


Figure 1: Handling range stabbing queries using value sweeping.

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 update steps. In the next section, we introduce the Persistent Hyperoctree, which satisfies all the above requirements.

4 Persistent Hyperoctrees (PHOTs)

4.1 Compact Hyperoctrees

In this section, we describe an ephemeral data structure called *Compact Hyperoctree* to index the active cells for a particular isovalue, which allows efficient search and update operations. We will call the entire time-varying data set a *volume*. A standard hyperoctree is based on hierarchical regular partitioning. The root node represents the entire volume. A node u has 16 children, each getting one hyperoctant of the subvolume of u. Here we assume the resolution of the data set along any dimension to be the same and be a power of two. If it is not the case, we adopt the strategy of BONO [WvG92] by viewing the hyperoctree as a complete one but avoiding allocating nodes for empty subtrees.

A node at the lowest level of the hypertree represents a cell and is colored black if the cell is active or 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 an example of the hyperoctree.

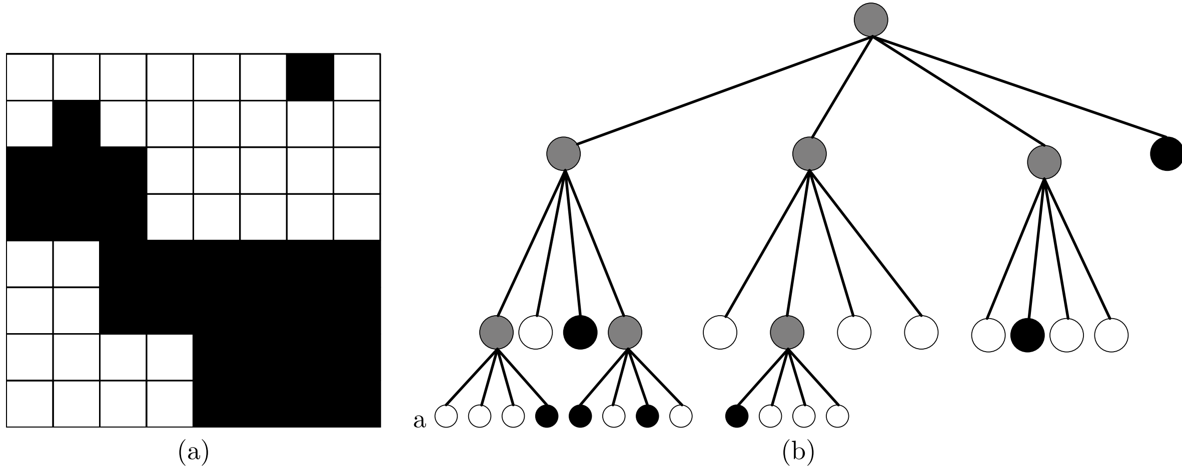


Figure 2: A 2D illustration of a hyperoctree. (a) The set of active cells. (b) The corresponding hyperoctree.

One problem with such a standard hyperoctree is that it does not support efficient update operations and therefore cannot be made persistent in a space-efficient way. In fact, consider the case when no value ranges of any two cells overlap. When we perform the value sweeping, all the cells will be inserted into the octree one by one, and each will have been removed before the next one is inserted. As a result, each insertion or deletion operation incurs $O(\log n)$ update steps, since $O(\log n)$ white nodes and $O(\log n)$ gray nodes will have to be created for an insertion and the same number of nodes be removed for a deletion.

Compact Hyperoctree is our solution to this problem. In a Compact Hyperoctree, white nodes are removed and pointers are replaced by *jumpers*. Let $u_1, u_2, \ldots u_l$ be a maximal chain of nodes in the original hyperoctree such that i) u_1 has at least one gray or black sibling; ii) u_i is the only gray or black child of u_{i-1} for $i = 2, \ldots l$; and iii) u_l is either a black node or has at least two gray or black children. We store a jumper to u_l in u_1. This jumper is associated with the path from u_1 to u_l, which is represented by a list of $(l-1)$ 4-bit binary strings, each indicating which branch the path takes at a particular level of the original hyperoctree. Notice that the subvolume of a node w is uniquely determined by the path from the root to w. We call a child w of u a *immediate child* if the path from u to w has a length of 1. Otherwise, we call w a *distant child* of u. Notice that there is no white node in a Compact Hyperoctree. Figure 3 shows the Compact Hyperoctree derived from the one shown in Figure 2(b).

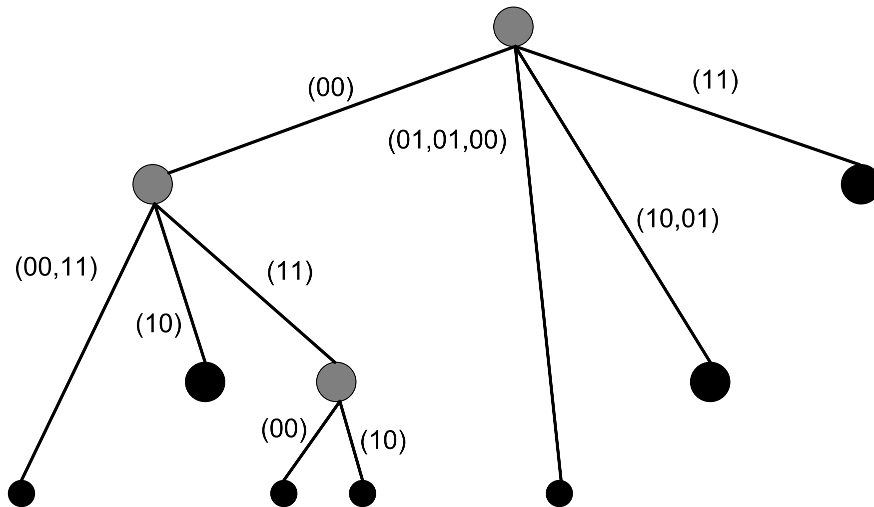


Figure 3: A 2D illustration of the Compact Hyperoctree. Each jumper is associated with a path, which is represented by a list of 2-bit strings. Null pointers are omitted.

Since each node in the Compact Hyperoctree, 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 linear in the number active cells it stores, so is the time it takes to traverse the tree to report them.

We now argue that a Compact Hyperoctree T requires only a constant number of update steps for each insertion or deletion. First consider the insertion of a black cell c. Let u be the lowest node whose corresponding subvolume contains c. Obviously u can only be a gray node, which by definition cannot be a leaf. We first create a leaf node which corresponds to c. If u is not the immediate parent of w, then we simply add a jumper to u pointing to w and store at u the path from u to w. On the other hand, if u is the immediate parent of w, there are two cases we need to consider. In the first case, u has less than 15 immediate black children. We simply record w as the child of u. In the second case, u already has 15 immediate black children. Adding c means that the entire region represented by u becomes black. Hence we need to recolor node u itself as black. This recoloring could be propagated upwards toward the root. To be more precise, let $u_1, u_2, \ldots, u_l = u$ be the longest path such that, for each node u_i on this path, all its children except u_{i+1} ($u_{l+1} = w$) are immediate and black. We then recolor u_1 as black and let it become a leaf node.

Now consider the deletion of a black cell c. We first identify the black leaf node w in the T whose corresponding subvolume contains c. If w is at the lowest possible level of T, i.e. its corresponding volume is c, then w is removed. Let u be the parent of w. If u has more than two children, then we simply remove the jumper from u to w. Otherwise, we add a jumper to the parent of u pointing to the child of u other than w. Note that the parent of u will still be gray after the deletion.

A trickier case is when w corresponds to a subvolume that is larger than c. In this case, removing c will introduce a sequence nodes to T, each of which has 15 black children and one gray child. In the worst case, up to $16 log_{16} n$ nodes need to be either created or updated. Fortunately, we can show that, on average, $O(1)$ update steps are needed in this case.

Consider the subtree T_u rooted at u and let m be the number of cells in the corresponding subvolume. Notice that subsequent operations on T_u after the deletion of c are all deletions, each of which will involve the creation of some black nodes and the removal of a black node. Notice that a node corresponding to a particular subvolume will be created, recolored to gray, and removed at most once. Also notice that the number of different subvolumes covered by T_u does not exceed $16n/15$. Thus on average only $O(1)$ update steps are needed for any deletion operation. We should notice that in general, a Compact Hyperoctree cannot be made persistent without asymptotically increasing its size. The reason our argument is valid here is that during the sweeping process, each cell will be inserted and removed exactly once.

Figure 4 gives the Compact Hyperoctree after three insertion and deletion operations are performed on the tree in Figure 3.

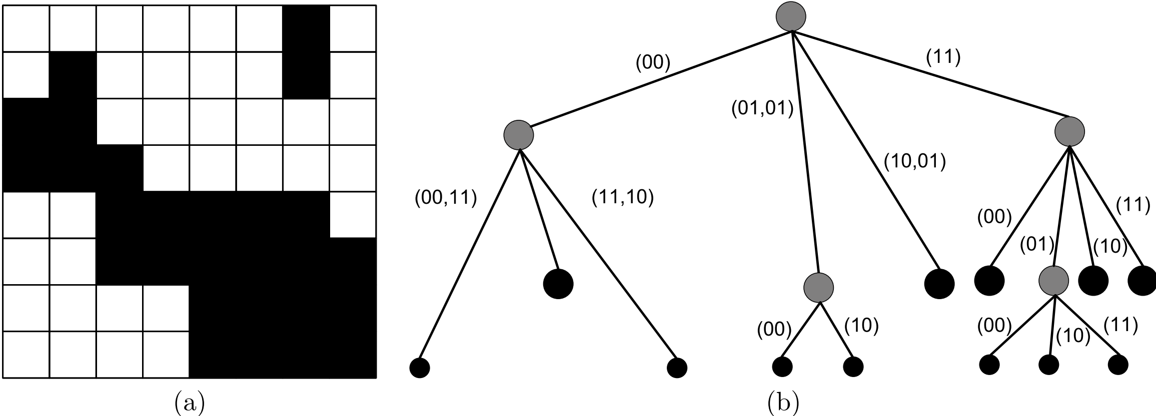


Figure 4: The new Compact Hyperoctree after the deletion of cells $(00, 11, 00)$ and $(11, 01, 01)$, and the insertion of cell $(01, 01, 10)$. (a) The set of active cells after the update operations. (b) The resulting Compact Hyperoctree.

4.2 Making a Compact Hyperoctree Persistent

In a Compact Hyperoctree, each internal node has exactly 16 (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 $16+k$ 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 PHOT. 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 16 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 5 illustrates a PHOT 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 5(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 extents of the shaded rectangles represent the value ranges of the corresponding segments. For a particular isovalue, the active segments are indexed by a binary tree. Figure 5(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 other nodes indicates the subvolume it represents. Notice that such a binary string is actually a concatenation of the legs of the path from the root to the corresponding node.

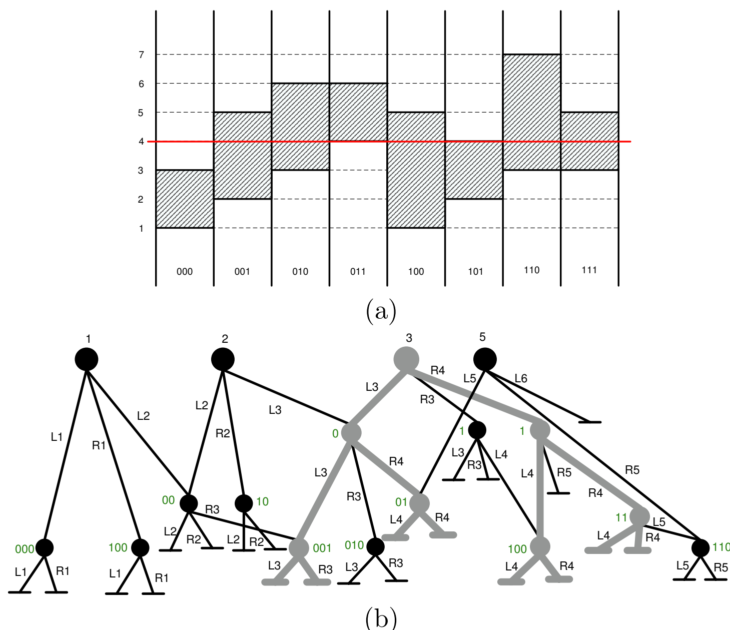


Figure 5: A 1D illustration of the PHOT. Each node has three slots for jumpers (a) The evolving of active segments for different isovalues. (b) The resulting PHOT.

Notice that a PHOT is a DAG rather than a tree. A node in a PHOT may have multiple predecessors.

4.3 Constructing, Exporting, and Importing a PHOT

To construct a PHOT, 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 PHOT. Otherwise we delete c from the PHOT. Since an insertion or a deletion on a Compact Hyperoctree requires only $O(1)$ update steps, the PHOT is linear in the total number of cells in the data set.

To further reduce the size of the PHOT, we do not actually store a node w if it is a leaf for all the versions of the PHOT it belongs to. Such omission is recorded by repointing any jumper to w to the node where that jumper is stored. For example, in Figure 5(b), the nodes 000, 100, 10, 001, 010, 01, 100, and 110 do not need to be stored while the nodes 00, 1, and 11 are still needed even though they are leaf nodes for some versions of the persistent binary tree. To indicate that a node w has been omitted, for each node u that has a jumper pointing to w, we replace that jumper with a jumper pointing to u itself. In this way, whenever we encounter a jumper at a node u that points to itself, we know it actually has a black child which is uniquely identified by the path associated with the jumper.

Once a PHOT is built, we need to serialize it so that it can be stored on disk and later be imported from the disk in a efficient manner. Unlike a tree, which can be easily serialized using postorder traversal, serializing a DAG like the PHOT is not a trivial task, because the location of a node on the disk has to be correctly recorded in all its predecessors.

To deal with this problem, we first move all the nodes to an array and then write the entire array to the disk. The movement of the nodes is done in a version-by-version. We start from each root node and perform a preorder traversal using the root's version number. To prevent a node from being moved multiple times, we allocate a counter for each node which is initialized as the number of its predecessors. This counter decrements each time the corresponding node is visited. When a node is visited for the last time, signaled by the counter becoming zero, it is copied to the next available slot in the array. To ensure that the jumpers remain valid after the movement, we allocate for each PHOT node a small array to store back-pointers to its predecessors. After a PHOT node is moved, its new location is represented by its index in the array and this new location is used to update the back-pointers of its children as well as the jumpers of its predecessors, which can be located using its own back-pointers.

To import the PHOT into memory, we simply load the entire array. The jumpers at each PHOT node can easily be recovered based on its location relative to the beginning of the array.

5 Searching a PHOT

Searching a PHOT is easy. We first identify the root of the PHOT with the largest version number smaller than or equal to the isovalue v. If we want to determine all the active (but not necessarily relevant) cells, then we simply traverse an appropriate version of the PHOT by following the latest jumpers no later than v. For example, in Figure 5(b), the portion of the PHOT colored in gray and connected by thick edges is the version traversed for 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 $O(\log n + k)$, where n is the size of the data set and k is the number of active cells corresponding to v. The $O(\log n)$ term is due to the fact that we may have to find the appropriate root using binary search. In practice, the number of roots is typically small enough (35 in our experiment) to be considered a constant.

An attractive feature about the PHOT is that, we can easily identify active and relevant cells by pruning an appropriate version of the PHOT using the cutting hyperplane. At each node u of this version, we check each of its jumpers to see if the subvolume of the node w it points to is cut by the cutting hyperplane. If this is not the case, then the subtree rooted at w will not be visited. An additional benefit of using jumpers is that we can very quickly get to a node representing a small active subvolume without having to access a long list of nested subvolumes at higher levels.

6 Data Description and Implementation Issues

6.1 Data Description

We have developed an algorithm based on PHOT to perform isocontouring on the Richtmyer-Meshkov instability data set [MPW+99]. This data set is the result of a simulation in which two gases of different density are initially separated by a membrane pushed against a wire mesh. The system is then perturbed by a superposition of a long wavelength and a short wavelength disturbance. This simulation produces a data set with 274 timesteps, each consisting of a 3D grid of $2048 \times 2048 \times 1920$ 8-bit scalar values. The data for a timestep is divided into 960 non-overlapping *brick* files each of size $256 \times 256 \times 128$, forming a brick grid of size $8 \times 8 \times 15$.

We downsample each brick by a factor of 2 in each dimension and use every 8th timestep starting from timestep 0. The resulting data set consists of 35 timesteps, each with a resolution of $1024 \times 1024 \times 960$. The overall size of the data set is about 35 GB. Each brick is of size 1 MB.

6.2 Implementation Issues

In our implementation, we allow each PHOT node to have 4 additional jumper ($k = 4$) slots in addition to the initial 16. A PHOT node thus has 20 jumper slots, the first 16 of which are always used.

To reduce the I/O cost, we partition each timestep into a grid of 3D *supercells*. Each 3D supercell consists of a block of 3D cells and a brick typically consists of a grid of 3D supercells. Each pair of 3D supercells which come from two consecutive timesteps and occupy the same 3D subvolume form a 4D supercell. We compute the value ranges of these 4D supercells and index them using a PHOT. Given an isovalue and a cutting hyperplane, determining the active and relevant cells thus involves locating active and relevant 4D supercells and then scanning these supercells to find the cells that are both active and relevant.

Loading a 4D supercell involves loading two 3D supercells from two consecutive timesteps. To improve the loading speed of 3D supercells while keeping the raw data layout intact, we load an entire brick each time one of its supercells is needed, using the fact that an isosurface typically is continuous in space. To avoid loading the same brick multiple times, we search the supercells in Z-order of their spatial positions. The size of the supercells provides a trade-off between the size of the indexing structure and the amount of cells to be loaded. We tested several 3D supercell sizes and ended up using $16 \times 16 \times 16$ in our experiment.

Once we have identified the active and relevant cells, we perform piece-wise triangulation for each of them using the isovalue and the cutting hyperplane. The lookup tables we used are generated using the program developed by Wenger and Bhaniramka [BWC04].

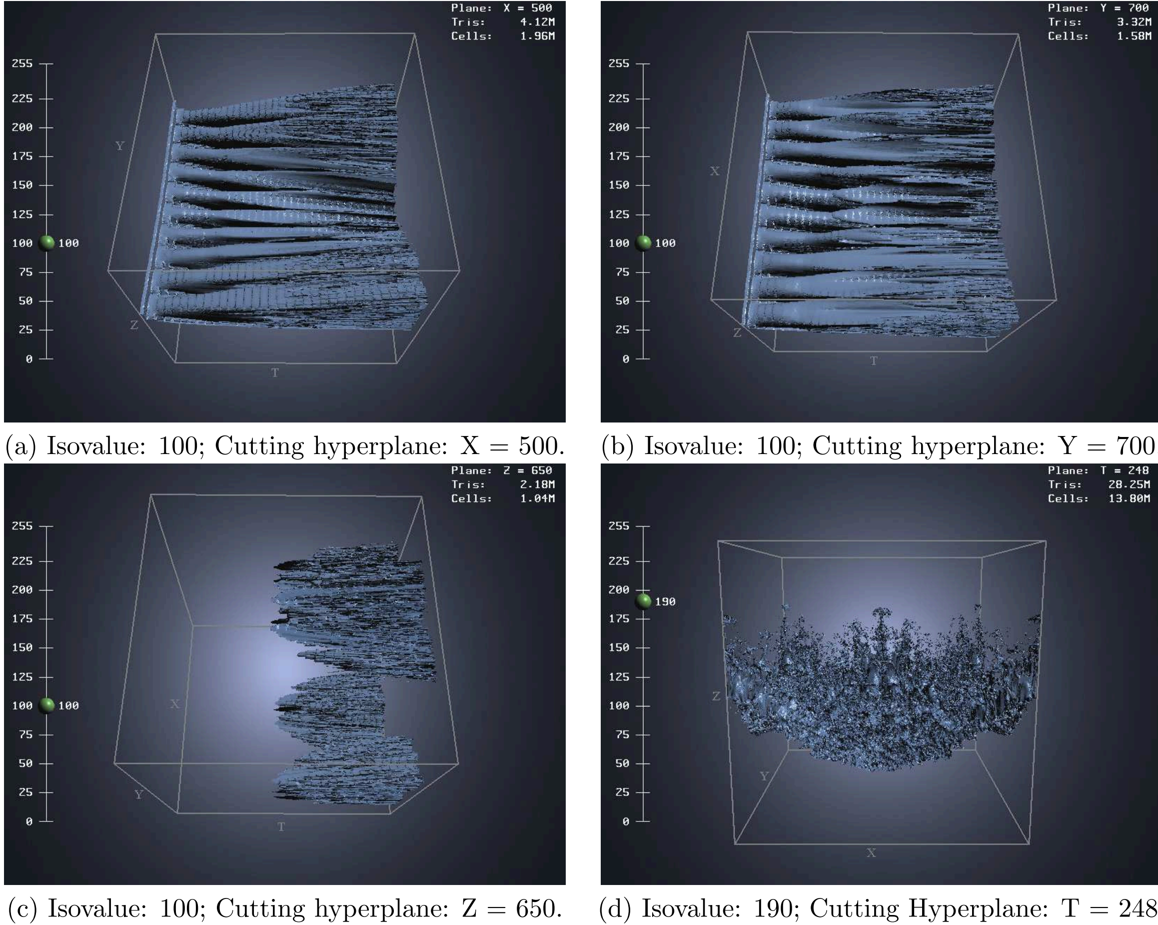
7 Experimental Results

The PHOT was constructed on a Sun-Fire-280R server with two 1200 MHz UltraSparc III processors and 8 GB main memory running Solaris 9. We used only one processor. The preprocessing involves three main steps: 1) padding each brick so that it contains a complete set of scalar values for the boundary cells; 2) computing the value ranges for the supercells; and 3) constructing the PHOT. The first two steps take about 2 hours for the entire 35 GB of data and the construction of the PHOT took only about 5 minutes. The PHOT uses about 80 megabytes of disk space, which is quite small compared to the size of the overall data set.

We have tested out isocontouring algorithm on a PC with dual 3.0 GHz Opteron processors, 8GB main memory, 60 GB local disk with around 50 MB/sec peak I/O transfer rate, and one NVidia6800 GPU card with a bi-directional 4Gbps data transfer rate to memory via PCI-Express (x16) Bus. It runs Linux 2.4.21-27.ELsmp. Only one processor was used.

We have done extensive testing of our algorithm on a wide variety of isovalues and different cutting hyperplanes. Figure 6 gives a few examples of isocontours sliced by different cutting hyperplanes.

We decompose an execution of the algorithm into 5 major steps: searching index, loading supercells, searching supercells, triangulation, and rendering, and measure the contribution of each step to the overall execution time. Each experiment was run 10 times. The measurements were averaged after the extreme numbers were discarded.



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

(b) Isovalue: 100; Cutting hyperplane: Y = 700.

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

(d) Isovalue: 190; Cutting Hyperplane: T = 248.

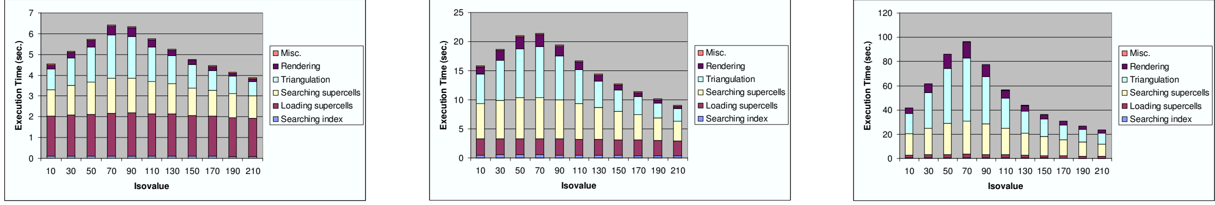
Figure 6: Slices of 4D isocontours.

In the first set of experiments, we changed the isovalue while fixing the cutting hyperplanes. Figure 7 shows the execution time of our program as a function of the isovalue for three different cutting hyperplanes, and Figure 8 reports the number of triangles generated in each case. Since the temporal resolution of our data set is much lower than the spatial resolution, slicing along spatial axes generates much less number of triangles compared to slicing along the temporal axis, and therefore results in very fast execution time (on average about 5 seconds when slicing along the X-axis and 15 seconds when slicing along the Z-axis).

Slicing along the T-axis is equivalent to generating isosurface at a particular timestep. The timestep 264 we used is close to the end of the simulation and has one of the most complex structures. The number of triangles varies from 26 million for isovalue 210 to 146 million for isovalue 70. Even for the most complex isosurface, we were able to render it in less than 100 seconds. On average, our program was able to generate about 1.2 million triangles per second. Searching the PHOT is extremely fast in all the cases. Even for the most time-consuming slicing along the T-axis, it only took 0.26 seconds on average.

Figure 7 shows that, when slicing along the T-axis, searching supercells and triangulation are the most time-consuming steps while loading supercells takes only a very smaller portion of the overall execution time. In the other two cases, the supercell loading time becomes more significant. This is partly due to the fact that, among the supercells that are loaded, a relatively smaller percentage of them are actually searched.

In the second set of experiments, we fix the isovalue to 100 and change the cutting hyperplanes. Figures 9 reports the corresponding execution times. Notice the different characteristics of slices along different axes. The numbers of triangles in slices along the Y-axis remains relatively stable across different Y values, while such numbers change dramatically for slices along the Z-axis. This is because the isosurface in a single timestep is generally parallel to the XY-plane, and the most active spatial area in the simulation is along the initial border between the two gases. Also, notice that the isosurface for a single timestep becomes more



(a) Cutting hyperplane: X = 500. (b) Cutting hyperplane: Z = 500. (c) Cutting hyperplane: T = 264.

Figure 7: Execution time of our program as a function of the isovalue.

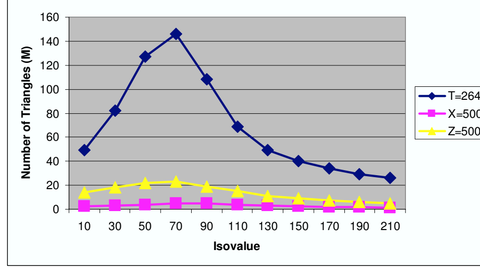
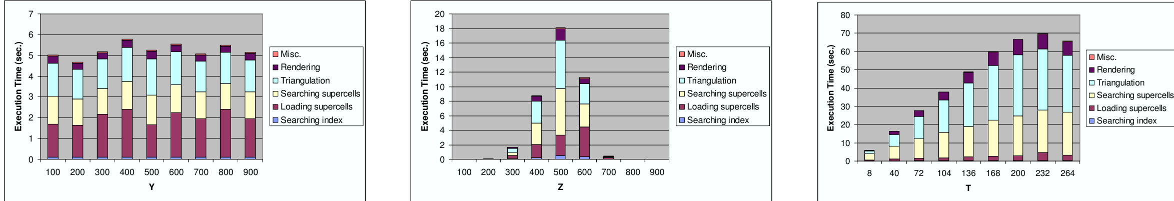


Figure 8: Number of triangles generated for three different cutting hyperplanes.

complex, thus requiring longer extraction time, as the system departs further away from initial equilibrium state.



(a) Slicing along the Y-axis. (b) Slicing along the Z-axis. (c) Slicing along the T-axis.

Figure 9: Slicing an isocontour with different cutting hyperplanes. Isovalue: 100.

8 Conclusions

We have presented the novel PHOT data structure to accelerate the visualization of slices of isocontours for time-varying data. This indexing structure is optimal in terms of space and time to report active cells. The set of active cells for an isovalue are organized as a Compact Hyperoctree that allows the active cells relevant to a cutting hyperplane to be identified very quickly. This technique can easily be generalized to higher dimensional scalar fields. We have developed an isocontouring algorithm based on PHOT to visualize the Richtmyer-Meshkov instability data set. Our algorithm can visualize not only isosurfaces at individual timesteps but also the evolution of isosurfaces across timesteps. Our experiments demonstrated that PHOT can be searched very fast 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, PHOT can be used to improve the performance of other visualization schemes such as view-dependent isosurface extraction [LH98] and isosurface rendering using ray tracing [PSL$^+$98]. These algorithms often require some kind of hierarchical structure to find the portion of active cells that either are visible from a certain view point or are first intersected by the shooting rays. PHOT can greatly accelerate such processes by avoiding

unnecessary value range checking against the isovalue and by quickly "zooming in" to the active subvolume through jumpers. We are currently also studying the use of PHOTs for adaptive resolution isosurface rendering which produces isosurfaces based on user provided level-of-detail requirements.

The PHOT technique can be further explored in two directions. First, PHOT is most suitable for data sets in a regular grid. We plan to apply the concept of persistency to other data structures to allow the handling of irregular grids. Second, we assume in this paper that the entire PHOT can be loaded into main memory. It may be possible to rearrange the nodes in the serialized PHOT using techniques such as the cache oblivious data structures [BDC00] to improve its blocking behavior so that high efficiency still can be achieved even if a large portion of the tree has to reside on disk.

9 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.

References

[BDC00] Michael A. Bender, Erik D. Demaine, and Martin-Farach Colton. Cache oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[BM95] A. Boroujerdi and B.M.E. Moret. Persistency in computational geometry. In *Proc. 7th Canadian Conf. Comp. Geometry (CCCG 95)*, pages 241–246, Québec, Canada, 1995.

[BS03] Udeepta Bordoloi and Han-Wei Shen. Space efficient fast isosurface extraction for large datasets. In *IEEE Visualization '03*, pages 201–208, 2003.

[BWC04] Praveen Bhaniramka, Rephael Wenger, and Roger Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, 2004.

[Chi03] Yi-Jen Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *IEEE Visualization 2003 (VIS'03)*, pages 217–224, Washington, D.C., 2003.

[CMM+97] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[CSS98] Yi-Jen Chiang, Cláudio T. Silva, and Wiliam J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization '98*, pages 167–174, 1998.

[DPH+03] David E DeMarle, Steven Parker, Mark Hartner, Christian Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG'03)*, pages 87–94, 2003.

[DSST89] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[GDL+02] Benjamin Gregorski, Mark Duchaineau, Peter Lindstrom, Valerio Pascucci, and Kenneth I. Joy. Interactive view-dependent rendering of large isosurfaces. In *IEEE Visualization '02*, pages 475–484, 2002.

[GJS95] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19:282–317, 1995.

[GSDJ04] Benjamin Gregorski, Joshua Senecal, Mark A. Duchaineau, and Kenneth I. Joy. Adaptive extraction of time-varying isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):683–694, 2004.

[GSG01] Jinzhu Gao, Han-Wei Shen, and Antonio Garcia. Parallel view dependent isosurface extraction for large scale data visualization. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[KJ05] Jusub Kim and Joseph JaJa. Information-aware HyperOctree for effective isosurface rendering of large scale time-varying data. Unpublished menuscript, 2005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[LH98] Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In *IEEE Visualization '98*, pages 175–180, 1998.

[LSJ96] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[MIPS04] Ajith Mascarenhas, Martin Isenburg, Valerio Pascucci, and Jack 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.

[MPW+99] 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.

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnart, Charles Hansen, and Peter-Parker Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, 1998.

[SCM99] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu 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.

[SH99] Philip Sutton and Charles 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.

[She98] Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *IEEE Visualization '98*, pages 159–166, 1998.

[SHLJ96] Han-Wei Shen, Charles D. Hansen, Yarden Livnart, and Christopher R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.

[SJ05] Qingmin Shi and Joseph JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Information Processing Letters*, 95:382–388, 2005.

[ST86] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[WB96] Chris Weigle and David C. Banks. Complex-valued contour meshing. In *IEEE Visualization '96*, pages 173–180, 1996.

[WB98] Chris Weigle and David 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.

[WCJ05] Kenneth W. Waters, Christopher S. Co, and Kenneth I. Joy. Isosurface extraction using fixed-sized buckets. In *EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, pages 207–214, 2005.

[WvG92] Jane Wilhelms and Allen van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.