Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

ISOSLIDER: A System for Interactive Exploration of Isosurfaces

Jatin Chhugani, Sudhir Vishwanath, Jonathan Cohen and Subodh Kumar

Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, USA

Abstract

We present ISOSLIDER, a system for interactive exploration of isosurfaces of a scalar field. Our algorithm focuses on fast update of isosurfaces for interactive display as a user makes small changes to the isovalue of the desired surface. We exploit the coherence of this update. Larger changes are supported as well. The update to the isosurface is made at a correct level of detail so that not too many operations need be performed nor too many triangles need be rendered. ISOSLIDER does not need to retain the entire volume in the main memory and stores most data out of core. The central idea of the ISOSLIDER algorithm is to determine salient isovalues where surface topology changes and pre-encode these changes so as to facilitate fast updates to the triangulation.

Keywords: Volume visualization, Isosurface extraction

1. Introduction

Spatial distribution of scalar data like 'bone density', 'wind speed' and 'fluid pressure' often needs to be visualized in medical and scientific applications (Figure 8(a)). Many such applications involve knowledge discovery, in which scientists explore a large field looking for "interesting" characteristics. Two common ways of visualizing these fields are direct volume rendering [4] and isosurface computation [12, 15]. Direct volume rendering produces a color at each pixel that is the composition of the scalar values at points intersected by a ray through that pixel. Isosurface visualization requires computation and display of all points (i.e., surfaces in a three-dimensional field) in the data with a given scalar value.

One common *modus operandi* for data exploration is to continuously vary the desired isovalue, generate the resulting isosurface and see how the surface changes as it *slides* from value to value. Our algorithm addresses the need for such isosurface exploration and is designed to take full advantage of the resulting coherence. In addition, our algorithm works well with large data sizes by allowing most of the data to reside on the disk at the exploration time. Its focus is on interactive isosurface update so that the scientist may easily maintain the context as the isovalue changes.

Formally, a volumetric scalar field is represented by the set of tuples $<(\mathcal{S}_i, p_i)>, i = 0..N$, such that $\mathcal{S}_i$ is the scalar value at point $p_i \in \Re^3$. The points p_i are usually selected to lie on a grid (structured or unstructured). Adjacent points on the grid are connected to form a cell. We sometimes denote $\mathcal{S}_i$ as $\mathcal{S}(p_i)$ to be explicit. Furthermore, $\mathcal{S}(p)$ at points p not in the tuple set are computed from cell's $\mathcal{S}_i$s (usually by tri-linear interpolation). Isosurface of a field, Ψ_λ, at scalar value λ is the set of all points p, such that $\mathcal{S}(p) = \lambda$. Variants of the Marching Cubes algorithm [15] are commonly used to compute Ψ_λ.

The Marching Cubes algorithm finds the overall surface by processing every cell in the input. The computation per cell is O(1). Recent isosurface algorithms reduce the number of cells considered by predicting the *active* cells: the cells, which actually contain the desired isovalue. The search for active cells takes one of three forms: spatial search [10], range search [3] or surface growth [1]. Spatial search techniques subdivide space and hierarchically eliminate partitions that do not contain an active cell. Range search methods associate each cell with the range of values it contains. These then search for the ranges that contain the desired isovalue. Surface growing methods start with some active cell(s) and find other active cells by tracking surface adjacencies. Range based methods are the most general and can handle unstructured cells, even if they do not directly benefit from spatial coherence of computations. Our algorithm is range based,

appropriately modified to exploit spatial coherence. While the framework of our algorithm can trade off time and disk requirements, in our current implementation we have chosen to favor fast computation over disk requirement. Also, we currently handle uniform (voxel) grid input only. While one may handle curvilinear and unstructured grids by converting them to a uniform grid [11] first, our algorithm can be easily extended to directly handle such data.

The main idea of our algorithm is to realize that small changes in isovalues require small changes to active cells. By simply pre-computing these changes and storing them out-of-core we can find the active cells quickly. Note that the isovalues at which a topological change happens are all in $\mathcal{S}_i$ (In case of trilinear interpolation, topology changes can occur inside a cell as well, but are directly handled by the render-time algorithm). Hence, in principle, we merely need to store $<\mathcal{S}_i, p_i>$ sorted by their values $\mathcal{S}_i$. Every time we slide past an isovalue, $\mathcal{S}_i$, the cells adjacent to p_i are inactivated or activated or sometimes simply re-triangulated. We augment this simple scheme to allow efficient sliding of isosurfaces across small as well as large values. In addition, we incorporate multi-resolution isosurface construction into our scheme. Our in-memory data structure also allows re-use of most topological information from frame to frame. Only interpolations of values need be computed afresh.

1.1. Previous Work

Span-space computation of Livnat et al.[14] achieved $O(\sqrt{N})$ search time for active cells. The interval-tree based search of Cignoni et al.[3] reduced that further to $O(\log N)$, which was extended for out-of-core searches by Chiang et al.[2]. Interval tree has an optimal worst case complexity for range search but is not well suited for sequential out-of-core access of successive intervals. While it is possible to augment the interval tree with 'next-in-sequence' links, we have opted for a simpler data structure based on skip-lists [17] that works well for isosurface sliding. The idea of using such local updates has been applied before. Giles and Haimes[7] maintained two lists: *minlist*, sorted by the minimum values of each cell and *maxlist*, sorted by the maximum value of each cell. If the isovalue is changed by a small value from λ_0 to λ_1, all intervals beginning in the range $[\lambda_0$-$\lambda_1]$ are added as potential active cells. The list is then purged of inactive cells. The performance for large changes in λ is $O(N)$. Shen and Johnson[19] improve the average performance by transforming the maxlist into *sweeping list* by adding a flag for each entry marking whether the minimum value of the corresponding cell is less than the current isovalue. The worst case performance remains $O(N+K)$, K being the size of the resulting triangulation [19]. Our algorithm performs work independent of N and is proportional only to δK, the change in triangulation at each frame for small changes in isovalue. For large changes in the isovalue, the total work is still $O(\log N + K)$ on average. The main advantage of our work is its out-of-core operation.

One problem with the Marching Cubes algorithm is that it generates a large number of polygons, up to five per cell. For large volumes, especially voxel grids, this is usually too many. Ramachandran et al.[20] present a non-interactive view-dependent algorithm for isosurface extraction on a cluster of machines. Dynamic simplification [9, 5] of isosurfaces is too slow to be interactive, however, and direct cell simplification schemes are popular [10, 18]. Most focus on using hierarchically larger cells composed of input cells when the resulting interpolation error is small. Some allow controlled simplification of topology [6]. Most are able to produce drastic simplification at high error allowance. Gregorski et al.[8], in particular, reports near interactive performance by enforcing a limit on the number of operations performed per frame. Their algorithm is designed for view-dependent updates to a given isosurface. Isosurface updates are slower, unless the error threshold is turned very high. Our algorithm performs conservative and static simplification only. The resulting triangles can be further simplified in a view-dependent fashion, but that is not currently done in our system.

2. ISOSLIDER algorithm

The main steps of isosurface visualization are as follows:

1. Find the active cells
2. Find the active edges for each cell
3. Classify each cell and compute the adjacencies of resulting triangles
4. Find the vertices on the active edges (location and normal)
5. Send the resulting triangles to the graphics card

A note about Step 5: it must be performed every frame that the isovalue changes. As the resulting isosurface slides, the vertex positions change. One might consider performing interpolations on the graphics card. However, many applications require the surface to be on the CPU for geometric analysis. We have chosen to perform the interpolations on the CPU. Note that for small changes in λ, most cells remain either active or inactive during the slide. Furthermore, the same edges of the active cells usually remain active implying that the cell maintains the same classification and uses the same triangle adjacencies. We exploit coherence at all these levels.

To find the active cells (Step 1), we preprocess the data. Consider a cell, C, with points $p_j^C, j = 1..k$, sorted by their scalar values, i.e., $\mathcal{S}_j^C < \mathcal{S}_{j+1}^C$. Recall that C is active only for isovalues $\mathcal{S}_1^C \leq \lambda \leq \mathcal{S}_k^C$. Thus, as we slide the isosurface from λ_1 to λ_2, cell C becomes active or inactive when we cross $\mathcal{S}_1^C$ or $\mathcal{S}_k^C$. We could then recompute the triangulation for those cells that change their status. Note also that the topology of triangles generated by C changes only at

$\lambda = \mathcal{S}_j^C, j = 1..k$. If we maintain a sorted list of all scalar values in the field, $\mathcal{S}_j, j = 1..N$, we know that the topology of some cell changes each time λ crosses an $\mathcal{S}_j$. If we store this information, we obtain an algorithm that does not need to touch any cell that does not undergo a change in its triangulation. We nominally store the following information block B_j with each $\mathcal{S}_j$ in our sorted list:

1. p_j
2. E^+: the list of indices of the edges that become active
3. E^-: the list of indices of the edges that become inactive
4. S^+: the list of scalar values $\mathcal{S}(p_l)$ for each point p_l such that edge $p_j p_l \in E^+$
5. S^-: the list of scalar values for edges in E^-

For high quality shading, we also retain the normals at p_j and all p_l. This naive approach replicates each input point for each adjacent edge. This can be avoided at the cost of additional disk I/O by using a point layout similar to that used by Gregorski et al.[8].

Note that the main drawback of this approach is that large change in λ requires several blocks to be read from the disk resulting in slow isosurface update even if the blocks are stored contiguously on the disk. We solve this problem by using a skip list of blocks.

2.1. Skip-list

Block B_j provides all changes to the active list when the isovalue changes from a value $\lambda_1 \in [\mathcal{S}_{j-1}, \mathcal{S}_j]$ to $\lambda_2 \in [\mathcal{S}_j, \mathcal{S}_{j+1}]$. In general, if the isovalue $\lambda_1 \in [\mathcal{S}_{m-1}, \mathcal{S}_m]$ goes to $\lambda_2 \in [\mathcal{S}_n, \mathcal{S}_{n+1}]$, all blocks $B_m..B_n$ are required. Unfortunately, this larger isovalue change could require $O(N)$ work in reading and applying the blocks. If the maximum number of active edges involved is $K = E(\lambda_1) + E(\lambda_2)$, we would like to limit the work performed to more like $O(K)$. Using a skip-list approach [17], we can reduce the actual work to $O(\log N + K)$.

In the skip-list structure, the original list of blocks B_i becomes level 0, or B_i^0. Each successive level B^L of the skip list contains fewer scalar values, and thus fewer blocks (Figure 1). All the edge changes from the associated lower-level blocks are consolidated into a single pair of E^+ and E^- lists for the higher-level block. As part of this consolidation, *redundant edges* are eliminated. These redundant edges were both activated and deactivated within the span of the higher-level block, so they represent unnecessary work.

We build level L from level $L-1$ as follows. We traverse level $L-1$, consolidating E^+ and E^- lists as we proceed, removing redundant edges. When the number of redundant edges as a ratio of the number of active edges, k, at the current isovalue becomes greater than a user-specified threshold, these consolidated lists become a single block at level L. Each block B_i^L contains the starting scalar value as well as the consolidated edges lists and their associated data. This level-by-level streaming approach allows us to generate a new level from the previous one when the data is stored out of core memory. Because each block at level L is associated with at least two blocks at level $L-1$, there can be no more than $O(\log N)$ levels.

Now let us reconsider the algorithm for changing an isovalue from λ_1 to λ_2. We start at block B_{m-1}^0 and wish to reach block B_n^0. Rather than reading and applying all the individual blocks on level 0 from $m-1$ to n, we take any available opportunities along the way to step up to a higher level, walking as far as possible at the higher levels to avoid redundant work. When we can walk no further at a higher level without passing λ_2, we step back down to a lower level. This is similar to the standard skip list algorithm, except we can start from some arbitrary position in the 0-level list. The total work performed is reduced by this algorithm to $O(\log N)$ walking up and down the lists plus $O(K)$ time traversing the levels.

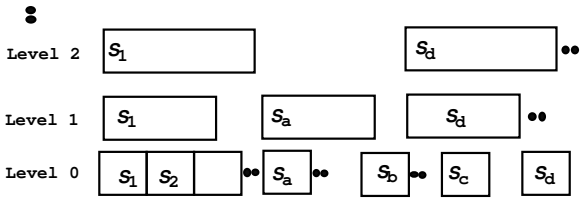

Figure 1: *Skip List*

Asymptotically, the skip list described above has $O(\log n)$ levels and the search time for an isovalue is $O(\log N)$. An edge can appear at most twice at any level. Thus a naive counting bounds the total space by $O(N \log N)$, given that there are $O(\log N)$ levels in the skip list. However, not all edges appear on levels above zero. In fact, by our construction, at least fraction k of the edges of a level do not appear in the level above, for a user specified constant k. Thus the total space requirement is only $\frac{N}{k} = O(N)$.

Even this space requirement can be reduced by simple compression. In particular, the normals for the vertices can be quantized to 16 bits. The data for each vertex adjacent along the edges in E^+ and E^- can be predictively encoded. In case of voxel data, for example, we store 10 bits for normal and 10 bits for isovalue residuals for each edge. The edge itself is identified by a three bit number. The vertex's ID takes 32 bits, isovalue another 32 and normal 16. This totals to $6E + 10N$ bytes, for N vertices and E edges, less that 400 megabytes for a 256^3 model at the lowest level and about a gigabyte for the entire skip list.

At the higher levels, the algorithm degenerates to simply keeping the list of active cells for each chosen isovalue because of the lack in coherence, in a way similar to Bajaj et al.[1]. For application where large jumps in the isovalue are not required, the higher levels of the skip list need not even

be precomputed and computed online when necessary. We usually only precompute 4-5 levels and leave the rest ungenerated, which may be computed in a lazy fashion.

2.2. Level of Detail

Since the naive Marching Cubes algorithm often produces too many triangles, we reduce this geometric complexity for better interactive performance. Our goal in this algorithm is to maintain the topological structure of the isosurface with minimal error in the position of triangle vertices. We currently do not perform view-dependent simplification [8] or visibility culling [13]. Our main criterion is to replace groups of small triangles by larger ones, if the resulting error is small. We perform this simplification by merging cells into larger cells. For example, in Figure 2 cells $C_{00}..C_{03}$ of level zero have been merged to form cell C_{10} of level one, their ancestor. The neighboring cells $C_{04}..C_{07}$ are not merged in this example. We ensure that the triangulation at the common boundary *AC* matches as explained below.

A group of cells is 'mergeable' if the common edges may be deleted subject to topological constraints. An edge p_ip_j may be deleted if both values $\mathcal{S}(p_i)$ and $\mathcal{S}(p_j)$ may be reliably interpolated from the remaining points of the parent (ancestor) cell after merger. In Figure 3, after edge deletions, active edge *AB* may be replaced by edge *AC* and *HI* may be replaced by *GI*. If linear interpolation of *ac* and *gi* does not differ much from the edges formed by the sub cells, cell C_{10} may be used. We ensure this by restricting the difference between the deleted isovalues and the isovalue interpolated along each retained edge crossing the sample. For example in Figure 3, edge *HB* may be used if $(\mathcal{I}(\mathcal{S}(H),\mathcal{S}(B)) - \mathcal{S}(E)) < \Delta$ for a small Δ, where $\mathcal{I}$ denotes linear interpolation. If all edges of a cell (HB, DF, AC, AG, GI and CI in this case) are usable, the cell is usable. The topological correctness is guaranteed by the following restrictions on merger of edges *AB* and *BC*:

1. Only one of *AB* and *BC* may be active at a time, i.e., values $\mathcal{S}(A), \mathcal{S}(B)$, and $\mathcal{S}(C)$ monotonically increase or decrease. Since the merged edge may not have two intersections with the isosurface, so must not the original edges.
2. An active edge may not be deleted.

Let us assume that the cell is a cube. In three dimensions an active edge must activate four cells adjacent to it as shown for edge *AB* in Figure 4. If one of the adjacent cells is already active, its topology changes. For multi-resolution cubes the adjacency is not as simple. As demonstrated in two dimensions in Figure 2, cells $C_{00}..C_{03}$ merge to form cell C_{10}, whereas cells $C_{04}..C_{07}$ remain. Edge *AB* has adjacent cell C_{00} inactive. Technically, edges *AB* and *AC* are both parts of active cells, but to avoid replication we use only the smallest edge as the active edge, *AB* in this case. This also prevents cracks because cell C_{10} is forced to use the edge *AB* to compute its vertex. Note that adjacent edges *AB* and *BC* cannot

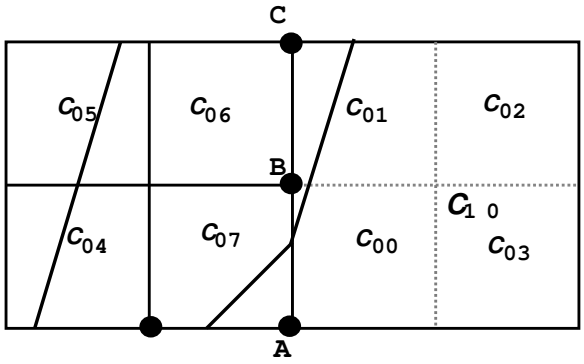

Figure 2: *Cell Merging*

both be active if cell C_{10} is used. Consequently, when *AB* becomes active, we must infer from *AB* (and the other active edges) the active cells: C_{10} in this case as opposed to C_{00}. To enable this inference we keep a flag for all potentially active ancestors of an active child cell.

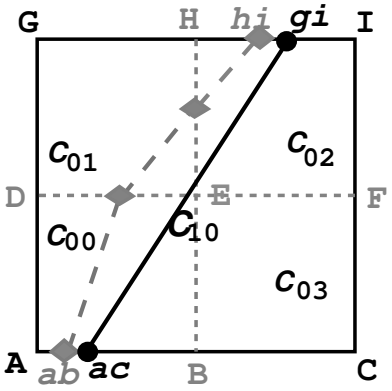

Figure 3: *Simplified isosurface*

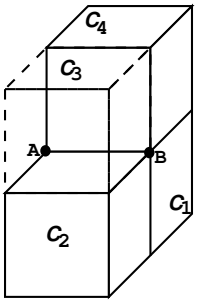

Figure 4: *An edge activates adjacent cells*

1. An active edge marks all its adjacent cells active. In the example in Figure 2, *AB* is active due to cell C_{07} but

marks C_{07} as well as C_{00}. C_{00} is marked erroneously in this case.

2. If a cell is marked erroneously, it must be the child of an appropriately active cell. In the example in Figure 5, that active ancestor is the large cell C. The erroneously active child is either in the middle of its ancestor's edge as is cell C_{03} in Figure 5(a) or at the corner of the edge as is cell C_{00} in Figure 5(b).
3. In the first case, C_{03} may not have any other edges marked active, a clear indication that the edge AB needs to be contributed to an ancestor.
4. In the second case, C_{00} may have enough edges of the ancestor (two in 2D, three in 3D) active to be confused for a real Marching Cubes case. However, even if we 'assign' edges AB and AD to cell C_{00} and triangulate it, the final result is correct. This happens because the edges containing AB and AD, AB' and AD', respectively of cell C will not be marked active as C_{00} never reports it to C. That is still correct as there is no Marching Cubes case where the mis-assigned edges AB' and AD' are used for any *other* triangle of C.

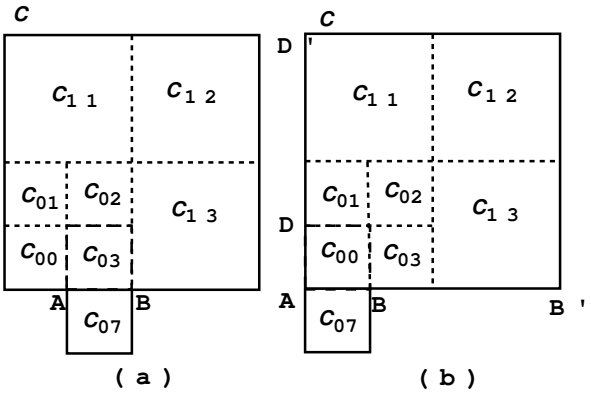

Figure 5: *Parent Location from Children of Cell C*

Thus we are able to precisely recognize the case when a child cell is mis-activated and must now look for its proper active ancestor. To enable such a search for the ancestor, we simply ensure that an ancestor is activated before any children: the highest level edges in E^+ are added first.

The above algorithm ensures that the surface has C^0 continuity along the edges of the voxels. To ensure hole-free merging of adjacent voxels along their common faces, we need to modify the triangulation of a higher level cell sharing face boundaries with lower level cells (Figure 6). For every such cell, we use the following steps:

1. For each face of the higher level cell (C in Figure 6), compute the relevant adjacent lower level cells (D_1, D_2 and D_3). Also compute the intersecting points, namely, P_1, P_2 and P_3.
2. The edge AB is replaced by edges AP_1, P_1P_2, P_2P_3 and P_3B. Unlike Shekhar et al.[18], we do not translate the smaller edges towards the larger edge, but instead reverse the process. This ensures consistent triangulation in the smaller cells. Note that the resulting edge still satisfies the initial error threshold criteria.
3. Each edge of the triangulation of the higher level cell is now replaced by a union of one or more smaller (C^0 continuous) edges. We retriangulate each of the original triangles to obtain the new triangulation (Figure 7).

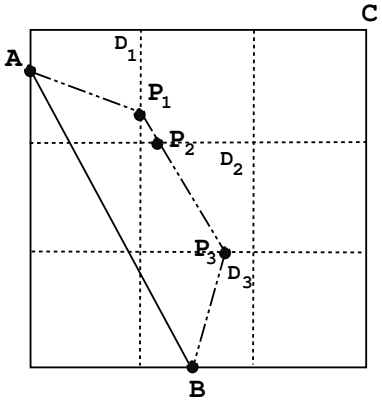

Figure 6: *Edge positions on the common face boundary of neighbors in different levels of the LOD hierarchy*

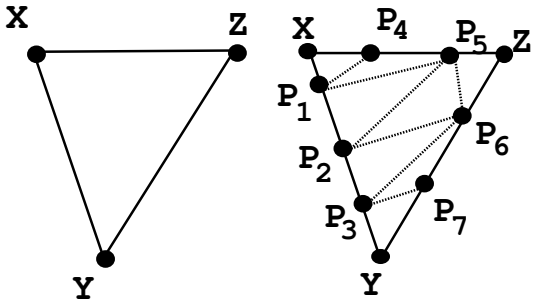

Figure 7: *Retriangulating the original triangle XYZ to get the new triangles*

The above process ensures that the resulting surface is watertight with no holes.

3. Implementation

We have implemented a version of our algorithm on a 1.6 GHz Linux PC with an NVIDIA GeForce3 capable graphics card. Our current implementation of non-uniform cells is incomplete and we report results only on uniform voxel data.

Vertex arrays (`http://www.nvidia.com`) are an efficient tool to display triangles. We use this representation, which requires:

- V: a contiguous list of all vertices used in the triangles and
- T: a contiguous list of all triangles, each triangle specified by three indices into V.

We recompute V every frame the isosurface changes but reuse most of T. In order to reuse the indices of T, it is important to update V coherently. We also retain the active edges and cells in the main memory from frame to frame. We assume the triangle, edge and cell information fit in the main memory. (For very large datasets that do not allow that, it is possible to use the 'metacell' scheme of Chiang et al.[2] to process a subset of the data at a time.)

3.1. Precomputation

The precomputation step sorts (using external memory sorting) the input scalar values and constructs the skip list in a bottom-up traversal, level by level. First, it computes the error of each hierarchically constructed cell from its children's value. We use an error bound of 1% of the total range of isovalues. Any *active* edge with larger error is not merged. The sorted tuples $<\mathcal{S}_i, p_i>$, are next traversed as follows. At each step of level zero, we consider all edges connected to the corresponding point p_i. We find the edges connected to p_i that become active and those that become inactive when crossing the isovalue from below it to above it. The new edges are tested in the order of their size. The largest edge usable is added to E^+. If an added edge causes a new topological constraint violation in any adjacent cell, that cell is subdivided. Deleted edges are added to E^-. Each deleted edge is next tested for topological constraints. If the edge was constraining an adjacent cell, that cell is retested for merger. All information of the affected edges is collected and appended to the end of the file. Merge and Split records are stored as the ID of the edge and the count of how many times it may merges or splits. For example, if an edge AB merges to its grandparent, we store its ID and the count 2.

3.2. In-core data structure

We maintain the following data structures at the rendering time:

- An Edge information array (**E**), which maintains the information related to each active edge. Specifically, we store the vertex coordinates ($\mathbf{V_1}$), isovalue ($\mathcal{S}_1$) at the starting point of the edge, the normal values at the two end-points of the edge ($\mathbf{N_1}$ and $\mathbf{N_2}$), the gradient in the vertex coordinates $\partial\mathbf{V} = \frac{\mathbf{V_2}-\mathbf{V_1}}{\mathcal{S}_2-\mathcal{S}_1}$. In order to expedite the computation of the unit normal at run-time, we also maintain a value $m = 2sin^2(\alpha/2)$, where α is the angle between $\mathbf{N_1}$ and $\mathbf{N_2}$. For the interpolating parameter value t, the norm of the normal is given as $\frac{1}{\sqrt{1-2mt(1-t)}}$, for small $2mt(1-t)$. We use the Taylor's expansion to evaluate the norm, thus avoiding the expensive square-root operation.
- An array of list of indices (**I**) forming the triangles. These indices represent the vertex indices which form each triangle. The marching cubes algorithm can generate up to *five* triangles for each active voxel. We maintain *five* different lists, list i for triangles of voxels with i triangles. The triangles of each cell are stored contiguously in its corresponding list. As a cell undergoes a change in the number of its triangles, this contiguous set is deleted from the old list and transferred to another if needed. This bucketing by triangle set cardinality allows effective memory management. All triangle sets in list i are the same size and a hole created by a deleted set can be easily filled by another set.
- An array of vertex Values (**V**). Each entry in this array corresponds to the interpolated vertex value from the Edge array **E**.
- An array of unit length Normals (**N**). Each entry in this array corresponds to the normal value at the corresponding location in the Vertices Array **V**.
- Hash tables, $\mathbf{H_E}$ and $\mathbf{H_C}$, for the active edges and the active Cells respectively. $\mathbf{H_E}$ stores a tuple $<gid, id>$, where *gid* is the identifier (ID) for the edge, and *id* is its location in **E**. *gid* is formed from the edge's location, direction, and its level in the Level-of-Detail hierarchy. $\mathbf{H_C}$ stores for each voxel, the tuple $<vid$, *Start Address*, *list_number*, *case_number*$>$. Here *vid* is the identifier of the voxel which is formed from its position in space and its level in the LOD hierarchy. *list_number* refers to the number of triangles the voxel has at that instant, *Start Address* refers to its starting index in **I**, and *case_number* refers to the case number of the Marching Cubes table used for this particular voxel. We use a hash function from the family of hash functions $H = \{h_{a,b}|a,b \in Z_p\}$, with $a \neq 0$. $h_{a,b}$ is defined as $h_{a,b}(x)$ = g $(f_{a,b}(x))$, where $g : Z_p \rightarrow \mathcal{N}$, given by $g(x) = x \ mod \ n$, and $f_{a,b}(x) = (ax+b) \ mod \ p$. Here p is a large prime number (greater than the largest entry being hashed to the table), and n is the size of the hash table. The size of the hash table is chosen to be *twice* the average number of edges expected to be active ($O(N^{\frac{2}{3}})$), where N is the total number of sampled points in the data set. This family of hash functions is proven to be strongly 2-universal [16], thereby reducing the probability of collisions. We stress that such a hash function is necessary for attaining real-time rates with our algorithm (Expected O(1) search time). We maintain open chain hashing, i.e. all the entries colliding at one specific location of the hash table are linked together in a link-list.

3.3. Sliding

As the user slides the isovalue, two kinds of updates are required: change of the active edge list and computation of the interpolated vertex and normal coordinates.

Once the active edge list is fixed, only the arrays **V** and **N** need to be changed. A specific entry V_i is changed by ∇V_i =

$\partial\mathcal{S} * \partial\mathbf{V_i}$ (stored in **E**). Similarly, the normal is interpolated from $\hat{\mathbf{N}}_\mathbf{1}$ and $\hat{\mathbf{N}}_\mathbf{2}$, and scaled by its length to normalize it.

If the active edge list changes, these additions and deletions are stored in lists *Add* and *Delete* respectively. We process the *Delete* list before accessing the *Add* list to avoid fragmentation in the various arrays. Each entry of the *Delete* list contains an edge ID, which is then hashed (using $\mathbf{H_E}$) to the Edge information array (**E**), and deleted from both lists. The IDs of cells adjacent to the edge is hashed to obtain their indices in $\mathbf{H_C}$. The case mask of each cell is changed. Also, the *Start_Address* and *list_number* fields of the cell become stale. The corresponding entries in **I** are deleted, fragmenting these arrays temporarily. Next, the *Add* list is processed, and a new entry is inserted for this edge into $\mathbf{H_E}$ and **E** (thereby filling up the holes created by the deletions). Again, $\mathbf{H_C}$ is modified, and the corresponding case masks are updated. In case the number of additions is smaller than the number of deletions, **E** and **I** are compacted. We fill the holes by transferring entries from the end of the list into them. When an edge entry is moved, its cells are re-accessed and their affected triangle's indices are modified to the edge's new location.

Once the topology has been changed, the vertex and normal lists are regenerated from **E**. Finally, **V**, **N** and **I** are sent to the graphics pipeline.

4. Results

We have tested our algorithm on a variety of models. In Table 1, we report the several features of the five major models we used. The number of distinct isovalues in a dataset is an indication of how often we have to compute the change in topology (Figure 9) and store the changes. The Boston Teapot is a 8-bit quantized data set, while the BluntFin (Figure 8(b)), CT Head Scan and the Spherical Shell models are 16-bit quantized data sets, and hence the distinct isovalue set spans almost the entire range, symbolizing greater changes when these values are crossed. The RADMRI dataset has 32-bit floating point values at the voxel end points. This provides for a wider range of values for a given resolution model.

The average number of active edges refers to the voxel edges intersected by the isosurface. Our run-time memory consumption is proportional to the number of intersecting voxels (around $40V$ bytes, where V is the number of intersecting voxels), and not the whole model. This coupled with the level-of-detail aids in rendering high resolution models at near real-time frame rates.

In Table 2, we give timings and the storage requirements for the preprocessing algorithm. During pre-processing, we store the edge ids for all the changes. In Table 3, we show the time taken when the user makes a big change in the isovalue to be rendered. For very small changes, we can change the isosurface in less than a *millisecond*, without affecting the frame rates. For appreciable changes, we take around 0.1 - 0.27 seconds for changes as large as 1-10% of the total range of isovalues. The average increase in the number of updates reduces as the step size increases, as some of the redundant changes cancel each other. For small changes in the isovalue, we obtain rendering rates of 10-20 frames per second, which drops to around 5-6 frames a second in case of large changes. For small changes, the frame rate is still mostly dominated by the triangle rendering performance of our machine (1.6 GHz PC with GeForce3 graphics card running Linux). During run-time, only a small percentage of time (around 0.04 seconds per frame) is spent in interpolating the vertex and normal values (and normalizing them).

Model	*Resolution*	*Distinct isovalues*	*Avg. No. of Active Edges*
RADMRI	69x261x69	904,216	176,104
BluntFin	256x256x256	54,268	402,918
Spherical Shell	256x256x256	33,743	176,504
Boston Teapot	256x256x256	236	320,382
CT Head Scan	512x512x252	259	803,113

Table 1: *Details of the models used for experimentation*

Model	*Error (%)*	*Pre-proc. Time*	*Change in active edges*	*Disk Space*
RADMRI	0.1	2.9 min.	27,435	21 MB
BluntFin	0.2	5.1 min.	72,196	320 MB
Sph. Shell	0.2	4.5 min.	65,234	300 MB
Bos. Teapot	0.7	16.4 min.	85,556	281 MB
Head Scan	0.8	44.6 min.	203,139	800 MB

Table 2: *Details of the preprocessing algorithm*

5. Conclusion

We have developed an out-of-core coherent algorithm for fast isosurface extraction. The target application is one in which small changes to isovalues are made to try to discover and study the topological changes near some isovalues. The algorithm achieves efficiency by ensuring that most information unchanged from the previous extraction is not even accessed. In our experience this algorithm provides a faster alternative to ones that recompute the isosurface by starting at the top of a hierarchy (like the interval tree).

Part of this fast performance comes at the cost of replicating each piece of data with all its adjacent data. While disk

Model	*Change in isovalue (%)*	*Average Updates per frame*	*Update Time (sec)*
RADMRI	0.1	3,500	0.008
RADMRI	1	29,316	0.07
RADMRI	5	88,481	0.21
RADMRI	10	116,296	0.27
BluntFin	1	64,185	0.14
BluntFin	2	80,068	0.2
BluntFin	5	76,877	0.19
BluntFin	10	71,453	0.17

Table 3: *Run-time behavior of our algorithm*

space is cheap, this still incurs a four to six fold increase over the original data. We are investigating smooth tradeoffs in replicating only some data and exploiting disk-cache coherence to fetch neighboring data.

Acknowledgments

The CT scan of the head was provided by the Department of Computer Science of Johns Hopkins University. The RADMRI dataset was provided by Julian Krolik of the Department of Physics and Astronomy of Johns Hopkins University. The Boston Teapot model was downloaded from http://www.volvis.org. We would like to thank the anonymous reviewers for their useful comments.

References

1. C. L. Bajaj, Valerio Pascucci, and Daniel Schikore. Fast isocontouring for improved interactivity. In *Symposium on Volume Visualization*, pages 39–47, 1996.
2. Yi-Jen Chiang, Cláudio T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proc. IEEE Visualization*, pages 167–174, 1998.
3. 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.
4. R. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, 1988. Proceedings of SIGGRAPH '88.
5. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proc. ACM SIGGRAPH*, pages 209–216, 1997.
6. T. Gerstner and R. Pajarola. Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proc. IEEE Visualization*, pages 259–266, 2000.
7. M. Giles and R. Haimes. Advanced interactive visualization for cfd. *Computing Systems in Engineering*, 1(10):51–62, 1990.
8. B. Gregorski, M. Duchaineau, P. Lindstrom, and V. Pascucci. Interactive view-dependent rendering of large isosurfaces. In *Proc. IEEE Visualization*, 2002.
9. H. Hoppe. Progressive meshes. In *Proc. ACM SIGGRAPH*, pages 99–108, 1996.
10. J.Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions of Graphics*, 11(3):201–227, 1992.
11. J. Leven, J. Corso, J. D. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics 2002*, 2002.
12. M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graphics Appl.*, 8(3):29–37, 1988.
13. Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proc. IEEE Visualization*, pages 175–180, 1998.
14. 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.
15. W. Lorensen and H. Cline. Marching cubes: A high-resolution 3d surface construction algorithm. In *ACM SIGGRAPH'87*, pages 163–169, 1987.
16. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
17. William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.
18. R. Shekhar, E. Fayyad, R. Yagel, and F. Cornhill. Octree-based decimation of marching cubes surfaces. In *Proc. IEEE Visualization*, pages 335–342, 1996.
19. Han-Wei Shen and Christopher R. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Proc. IEEE Visualization*, pages 143–150, 1995.
20. V. Ramachandran X. Zhang, C. Bajaj. Parallel and out-of-core view-dependent isocontour visualization using random data distribution. In *Proc. of the Joint Eurographics IEEE TVCG Symposium on Visualization*, 2002.

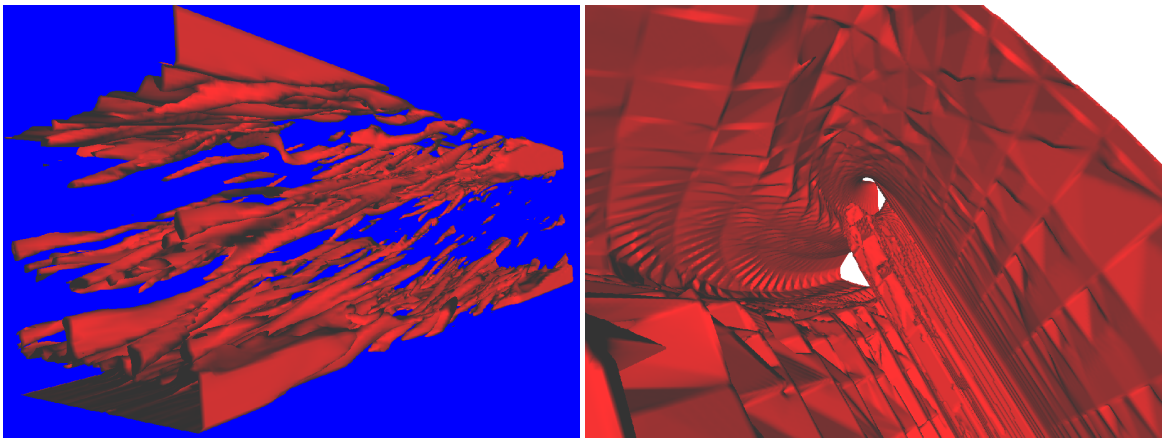

Figure 8: *(a) A RADMRI isosurface (b) A Bluntfin isosurface*

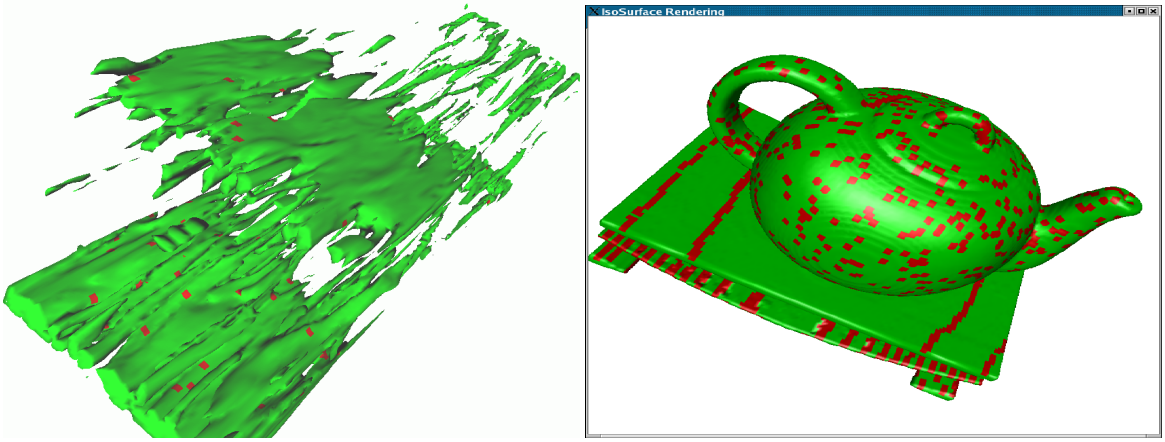

Figure 9: *Red marks the places where the isosurface changes in topology after a small change to the isovalue for (a) A RADMRI isosurface and (b) Boston Teapot isosurface*

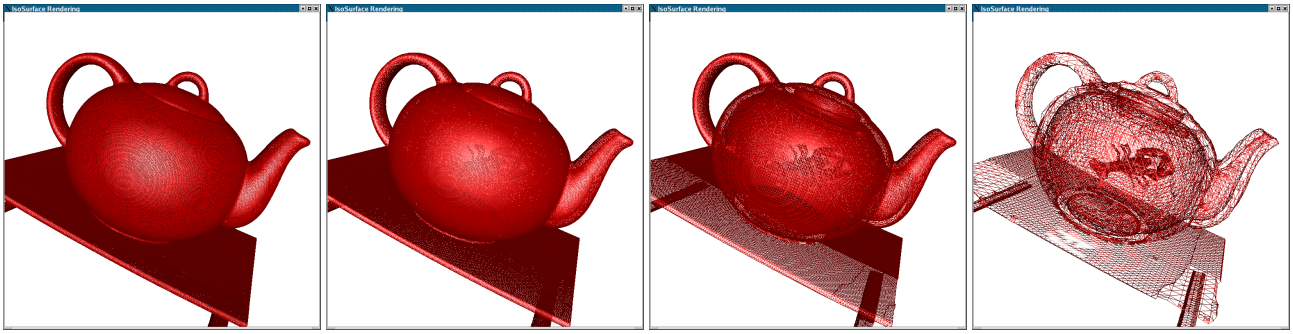

Figure 10: *Change in the number of triangles as the error threshold changes*
(a) No error: 740,249 triangles (b) 0.3% error: 645,153 triangles (c) 0.8% error: 536,856 triangles (d) 1.5% error: 80,197 triangles