Speeding Up Isosurface Extraction using Interval Trees

Paolo Cignoni, Paola Marino, Claudio Montani, *Member, IEEE*,
Enrico Puppo, *Member, IEEE*, Roberto Scopigno, *Member, IEEE*

P. Cignoni, P. Marino, and C. Montani are with the Istituto di Elaborazione dell'Informazione - CNR, Via S. Maria 46, Pisa, Italy. E-mail: {cignoni,marino,montani}@iei.pi.cnr.it.

E. Puppo is with the Istituto per la Matematica Applicata - CNR, Via De Marini 6, Genova, Italy. E-mail: puppo@ima.ge.cnr.it.

R. Scopigno is with the Istituto CNUCE - CNR, Via S. Maria 36, Pisa, Italy. E-mail: r.scopigno@cnuce.cnr.it.

Abstract

The interval tree is an optimally efficient search structure proposed by Edelsbrunner [5] to retrieve intervals of the real line that contain a given query value. We propose the application of such a data structure to the fast location of cells intersected by an isosurface in a volume dataset. The resulting search method can be applied to both structured and unstructured volume datasets, and it can be applied incrementally to exploit coherence between isosurfaces. We also address issues about storage requirements, and operations other than the location of cells, whose impact is relevant in the whole isosurface extraction task.

In the case of unstructured grids, the overhead due to the search structure is compatible with the storage cost of the dataset, and local coherence in the computation of isosurface patches is exploited through a hash table. In the case of a structured dataset, a new conceptual organization is adopted, called the *chess-board approach*, wich exploits the regular structure of the dataset to reduce memory usage and to exploit local coherence. In both cases, efficiency in the computation of surface normals on the isosurface is obtained by a pre-computation of the gradients at the vertices of the mesh.

Experiments on different kinds of input show that the practical performance of the method reflects its theoretical optimality.

Keywords

Volume Visualization, Isosurface Extraction, Marching Cubes, Interval Tree.

I. Introduction

A *scalar volume dataset* is a pair (V, W), where $V = \{v_i \in \Re^3,\ i = 1, \ldots, n\}$ is a finite set of points spanning a domain $\Omega \subset \Re^3$, and $W = \{w_i \in \Re,\ i = 1, \ldots, n\}$ is a corresponding set of values of a scalar field $f(x, y, z)$, sampled at the points of V, i.e., $w_i = f(v_i)$. A mesh Σ subdividing Ω into polyhedral cells having their vertices at the points of V is also given (or computed from V, if the dataset is scattered): Σ can be made of hexahedra, or tetrahedra, or it can be hybrid, i.e., made of tetrahedra, hexahedra, triangular prisms, and pyramids.

Given an *isovalue* $q \in \Re$, the set $S(q) = \{p \in \Omega \mid f(p) = q\}$ is called an *isosurface* of field f at value q. For the purpose of data visualization, an isosurface $S(q)$ is approximated by a triangular mesh, defined piecewise on the cells of Σ: a cell $\sigma_j \in \Sigma$ with vertices $v_{j_1}, \ldots v_{j_h}$ is called *active* at q if $\min_i w_{j_i} \leq q \leq \max_i w_{j_i}$. An active cell contributes to the approximated isosurface for a patch made of triangles: patches are obtained by joining points on the active cells' edges that intersect the isosurface (*active edges*), by assuming linear interpolation of the field along each edge of the mesh. Such intersection points are called *isosurface vertices*. In order to use smooth shading to render the isosurface, the surface normal at each surface vertex must also be estimated.

Therefore, the *isosurface extraction problem* consists of four main subproblems:

1. *Cell selection:* finding all active cells in the mesh Σ.
2. *Cell classification:* for each active cell, determining its active edges, and how corresponding isosurface vertices must be connected to form triangles.
3. *Vertex computation:* for each active edge, computing the 3D coordinates of its surface vertex by linear interpolation.
4. *Surface normal computation:* for each vertex of the isosurface, computing its corresponding surface normal.

In terms of computation costs, the impact of cell selection may be relevant on the whole isosurface extraction process, in spite of the simplicity of operations involved at each cell, because it involves searching the whole set of cells of Σ. Cell classification has a negligible cost, because it is performed only on active cells, and it involves only comparisons of values. Although also vertex and normal computations are performed only on active cells, they have a relevant impact, because they involve floating point operations. Besides, such operations can also be redundant, if the dataset is processed on a per-cell basis, because each active edge is shared by different cells.

In order to speedup such tasks, it can be worth using search structures, and techniques that permit to traverse as less non-active cells as possible during cell selection, and to avoid redundant vertex and normal computations. Speedup techniques can be classified according to the following criteria:

- *search modality* adopted in selecting active cells; there are three main approaches: in *space-based* methods, the domain spanned by the dataset is searched for portions intersected by the isosurface; in *range-based* methods, each cell is identified with the interval it spans in the range of the scalar field, and the range space is searched for intervals containing the isovalue; in *surface-based* methods, some facets of the isosurface are detected first, and the isosurface is traversed starting at such faces and moving through face/cell adjacencies;

- *local coherence* (or coherence between cells), refers to the ability of a method to avoid redundancy in geometric computations, by reusing the results obtained for an active face or edge at all its incident cells.

Since additional data structures may involve non-negligible storage requirements, it is important to look for methods and structures that warrant a good tradeoff between time efficiency and memory requirements. The overhead due to auxiliary structures must be compared to the cost of storing a minimal amount of information necessary to support isosurface extraction, disregarding the computational complexity: this can be highly variable depending on whether the dataset considered is *structured* or *unstructured* (i.e., its connectivity is implicitly given, or it must be stored explicitly, respectively [17]). Therefore, evaluation criteria for a speedup method must take into account: its *range of applicability*, i.e., the types of dataset (structured, or unstructured, or both) for which the method is suitable; its *efficiency*, i.e., the speedup it achieves with respect to a non-optimized reference method; its *overhead*, i.e., the storage cost due to auxiliary structures.

On the basis of our preliminary work on unstructured data [3], in this paper we address the application of speedup techniques in the various phases of isosurface extraction from both *structured and unstructured* datasets. A highly efficient technique for cell selection that is based on the *interval tree* [5] is adopted. In the unstructured case, this technique is associated to the use of a hash table in order to exploit local coherence to avoid redundant vertex computations. In the structured case, a new conceptual organization of the dataset, called the *chess-board approach*, is adopted in order to reduce the memory requirements of the interval tree, and to exploit local coherence intrinsic in the implicit structure of the dataset. In both cases, we adopt a pre-computation of field gradients at data points in order to speedup the computation of surface normals. Moreover, we describe how the interval tree can be efficiently used to develop an incremental technique that exploits coherence between isosurfaces.

Time and space analyses of our isosurface extraction technique are given, and compared with those of other methods. Theoretical results are borne out by experimental results that show the performance of the method on different test datasets.

II. Related Work

The simplest speedup methods for cell selection are based on space partitions, and they are only suitable for structured data. Wilhelms and Van Gelder [19] use of a branch-on-need *octree* to purge sub-volumes while fitting isosurfaces, based on the range interval spanned by each sub-volume. This method achieves a worst case time efficiency $O(k + k \ \log(n/k))$ (where n is the total number of cells, and k is the number of active cells) [11], with small overhead (the octree increases storage occupancy only for about 16%). An alternative approach for structured data is also proposed by Criscione et al. [4], which is based on a *pyramid* data structure. This approach has similar efficiency and overhead, while it is easier to implement than the previous one. Space-based techniques cannot be generalized easily to unstructured data, because spatial indexes rely on the regular structure of the underlying dataset.

Range-based techniques apply to both structured and unstructured datasets, but they are generally more suitable for unstructured datasets, because they cannot exploit the implicit spatial information contained in structured datasets, and they have higher memory requirements. In the unstructured case, there is no implicit spatial information to exploit, and the higher storage cost for the input mesh hughly reduces the overhead factor of auxiliary structures.

Gallagher [6] proposes a method based on a subdivision of the range domain into buckets, and on a classification of intervals based on the buckets they intersect. The tradeoff between efficiency and memory requirements is highly dependent on the resolution of the bucketing structure. Giles and Haimes [7] report an approach in which two sorted lists of intervals are constructed in a pre-processing phase by sorting the cells according to their minimum and maximum values, respectively. This method is addressing the specific problem of *global coherence* (or coherence between isosurfaces), which is aimed to exploit part of the information derived from the extraction of a given isosurface to speedup the selection of active cells for another isosurface, corresponding to a near isovalue. This feature is useful in applications that change the isovalue continuously and smoothly, while it gives small improvement over a non-optimized method in the generic extraction of isosurfaces at arbitrary isovalues. In a more recent paper, Shen and Johnson [16] try

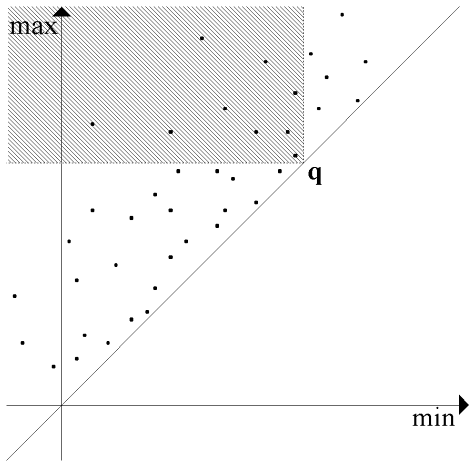

Fig. 1. The span space. Each interval $I = [a, b]$ is represented as a point of coordinates (a, b). To detect the intervals that contain the query value q we have to find the points which lye to the left of the line $min = q$ and above the line $max = q$.

to overcome some limitations of [6], and [7] by adopting similar auxiliary structures to address global coherence. However, a worst case computational complexity of $O(n)$ has been estimated for all three methods outlined above [11].

Livnat *et al.* [11] introduce the *span space* (see Figure 1), which is a two-dimensional space where each point corresponds to an interval in the range domain. The span space is very useful to geometrically understand range-based methods, therefore we will refer to this representation also in the next sections. A *kd*-tree is used to locate the active intervals in this space, achieving an $O(\sqrt{n} + k)$ time complexity in the worst case. The possibility of exploiting global coherence is also outlined. In a more recent paper, Shen *et al.* [15] propose the use of a uniform grid to locate the active intervals in the span space. Such an approach is suitable to parallel implementation.

The data structure we adopt in this paper, i.e., the *interval tree*, was proposed by Edelsbrunner [5] to support queries on a set of intervals. It is optimally efficient, i.e., it warrants a worst case time complexity of $\theta(k + \log n)$, while its memory overhead is comparable with those of the other range-based methods. It is worth mentioning that, although our proposal is the first application of such a data structure to speedup isosurface extraction, other authors have used it to address related problems: Laszlo [10] considers the extraction of wireframes from a grid of generic polyhedra, by using an interval tree, where each interval corresponds to an edge of the input grid; van Kreveld [18] extracts isolines from triangulated terrain data, by associating each triangle with the interval of altitudes it spans.

Surface-based approaches rely essentially on two requirements: the ability to find an active cell *(seed)* for each connected component of the isosurface; and the ability to propagate the surface by traversing the mesh from cell to cell through adjacencies [17]. Adjacencies are implicit in structured datasets, while they need to be stored explicitly in unstructured datasets. Storing adjacencies explicitly roughly doubles the memory requirement of the dataset, hence making the overhead of surface-based methods in the unstructured case either comparable to, or even higher than the overhead of range-based methods. Moreover, further auxiliary structures are needed in order to find seeds.

Itoh et al. [9], [8] base the search of seeds on a graph, whose nodes are the cells holding local minima or maxima data values: therefore, an arc of the graph spans an interval in the range domain. Each arc supports a list of cells connecting its two end nodes. Given an isovalue, the graph is searched for an active arc, and the cells connected to this arc are sequentially scanned until a seed is found. A propagation method is activated on this seed. Since the isosurface can be made of many connected components, seed search must be repeated until all active arcs have been visited. This can take $O(n)$ time in the worst case [11]. A more efficient method to find active seeds is proposed by Bajaj et al. [1]: a minimally sufficient *seed set* is found in a pre-processing phase, such that any connected component of any arbitrary isosurface is guaranteed to traverse at least one cell in the seed set. The seed set can be encoded in a range-based search structure, in order to efficiently locate active seeds for a given isovalue: optimal time efficiency can be achieved by using an interval tree. The seed set is very small on average, hence causing a small overhead, but it can be as big as $O(n)$ in the worst case (e.g., if the underlying field has a sinusoidal shape). The algorithm for finding the seed set is complicated, and its time complexity is high.

Local coherence is exploited only by some of the reviewed methods. Spatial indexes adopted by space-based methods destroy the locality of computation, which supports local coherence in the original Marching Cubes [12]. For this reason, following Wyvill et al. [20], Wilhelms and Van Gelder [19] adopt a hash-based caching strategy to save and reuse local computations. The tradeoff between efficiency and overhead depends also on the size of the hash table: Wilhelms and Van Gelder empirically define statically a size eight times the square root of the size of the dataset.

Surface-based methods can partially exploit local coherence during the propagation phase: since each new cell is accessed through an adjacent cell, having a common face intersected by the isosurface, it is easy to reuse vertices related to such a common face without computing them again. This implies no further overhead with respect to the structure that supports dataset encoding and traversal. However, this fact alone cannot warrant that no redundant computations will be performed, unless explicit links from cells and edges of the input mesh to the list of extracted cells are maintained, with a further relevant overhead.

III. Selecting Cells Through Interval Trees

The technique we propose for active cell selection is in the class of range-based methods, and therefore it can be used both for structured and unstructured datasets. Let Σ be the input mesh. Each cell $\sigma_j \in \Sigma$ is associated with an interval I_j, whose extremes a_j and b_j are the minimum and maximum field values at the vertices of σ_j, respectively. Since σ_j is active for an isovalue q if and only if its corresponding interval I_j contains q, the following general query problem is resolved:

> "given a set $\mathcal{I} = \{I_1, \ldots, I_m\}$ of intervals of the form $[a_i, b_i]$, with $a_i \leq b_i$ on the real line, and a query value q, report all intervals of $\mathcal{I}$ that contain q".

The problem is effectively visualized using the *span space* introduced by Livnat et al. [11] (see Figure 1): each interval $Ii = [a_i, b_i]$ is represented as a point in a 2D Cartesian space using the extremes a_i and b_i as the x and y coordinates of the point, respectively. From a geometrical point of view, the problem of reporting all intervals that contain the query value q reduces to collecting the points in the span space lying in the intersection of the two half–spaces $min \leq q$ and $max \geq q$.

An optimally efficient solution for the query problem above can be obtained by organizing the intervals of $\mathcal{I}$ into an *interval tree*, a data structure originally proposed by Edelsbrunner [5] (see also [14]), which is reviewed in the following. For each $i = 1, \ldots, m$, let us consider the sorted sequence of values $X = (x_1, \ldots, x_h)$ corresponding to distinct extremes of intervals (i.e., each extreme a_i, b_i is equal to some x_j). The interval tree for $\mathcal{I}$ consists of a balanced binary search tree $\mathcal{T}$ whose nodes correspond to values of X, plus a structure of lists of intervals appended to non-leaf nodes of $\mathcal{T}$. The interval tree is defined recursively as follows. The root of $\mathcal{T}$ has a discriminant $\delta_r = x_r = x_{\lceil \frac{h}{2} \rceil}$, and $\mathcal{I}$ is partitioned into three subsets as follows:

- $\mathcal{I}_l = \{I_i \in \mathcal{I} \mid b_i < \delta_r\}$;
- $\mathcal{I}_r = \{I_i \in \mathcal{I} \mid a_i > \delta_r\}$;
- $\mathcal{I}_{\delta_r} = \{I_i \in \mathcal{I} \mid a_i \leq \delta_r \leq b_i\}$.

The intervals of $\mathcal{I}_{\delta_r}$ are arranged into two sorted lists $\mathcal{AL}$ and $\mathcal{DR}$ as follows:

- $\mathcal{AL}$ contains all elements of $\mathcal{I}_{\delta_r}$ sorted in $\mathcal{A}$scending order according to their $\mathcal{L}$eft extremes a_i;
- $\mathcal{DR}$ contains all elements of $\mathcal{I}_{\delta_r}$ sorted in $\mathcal{D}$escending order according to their $\mathcal{R}$ight extremes b_i.

The left and the right subtrees are defined recursively by considering interval sets $\mathcal{I}_l$ and $\mathcal{I}_r$, and extreme sets $(x_1, \ldots, x_{\lceil \frac{h}{2} \rceil - 1})$ and $(x_{\lceil \frac{h}{2} \rceil + 1}, \ldots, x_h)$, respectively. The interval tree can be constructed in $O(m \log m)$ time by a direct implementation of its recursive definition. The resulting structure is a binary balanced tree with h nodes, and a height of $\lceil \log h \rceil$, plus a collection of lists of type $\mathcal{AL}$ and $\mathcal{DR}$, each attached to a node of the tree, for a total of $2m$ list elements.

An example of a simple interval tree, built on few intervals, is shown in Figure 2; a representation of the same structure in the span space is given by the subdivision in Figure 3 (solid lines) It is noteworthy that, by construction, the last level of the tree is generally empty. The intervals of this level, if they exist, have to be null intervals (in our case, such intervals are in fact associated with cells having the same values at all vertices). Given a query value q, tree $\mathcal{T}$ is visited recursively starting at its root:

- if $q < \delta_r$ then list $\mathcal{AL}$ is scanned until an interval I_i is found such that $a_i > q$; all scanned intervals are reported; the left subtree is visited recursively;
- if $q > \delta_r$ then list $\mathcal{DR}$ is scanned until an interval I_i is found such that $b_i < q$; all scanned intervals are reported; the right subtree is visited recursively;
- if $q = \delta_r$ then the whole list $\mathcal{AL}$ is reported.

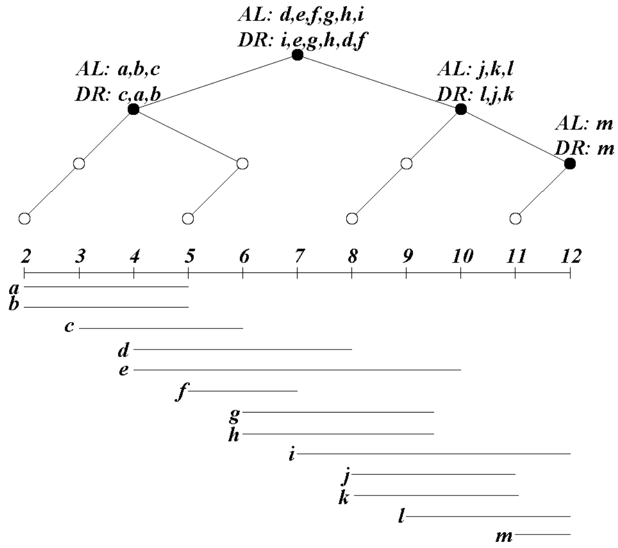

Fig. 2. An example of an interval tree built over a simple dataset (13 cells). The white dots represent nodes with empty $\mathcal{AL}$ and $\mathcal{DR}$ lists.

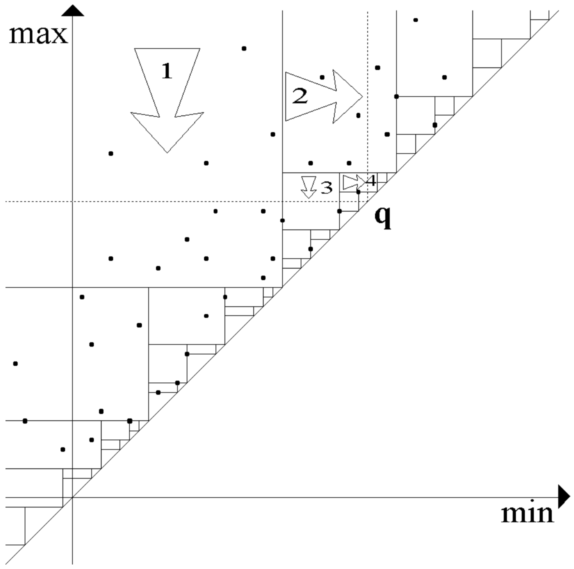

Fig. 3. A graphical representation of the interval tree of Figure 1 in the span space. By definition, the intervals lying on subdivision lines belong to the upper level of the tree. The tree search for a value q: sectors with $\delta_r < q$ (intersected by the horizontal line $max = q$) are visited top-down; sectors with $\delta_r > q$ (intersected by the vertical line $min = q$) are visited left to right.

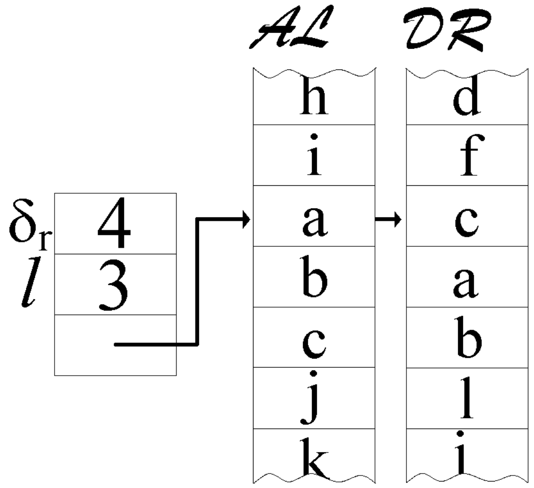

Fig. 4. In our implementation of the Interval Tree data structure, a generic node contains the discriminat value (δ_r), the length of the $\mathcal{AL}$ and $\mathcal{DR}$ lists (l) and the starting position in the *big* $\mathcal{AL}$ and *big* $\mathcal{DR}$ arrays; in the example, the values stored in the left-most, not empty node of the interval tree of previous Figure 2 are reported.

The geometric interpretation of the search in the span space is also given in Figure 3. The regions containing the active intervals is that left and above dotted lines from q. Each sector of space (node of tree) which contains the horizontal dotted line (i.e., such that $\delta_r \geq q$) is visited top-down (scanning the $\mathcal{AL}$ list) until such a line is reached; each sector containing the vertical dotted line is visited left to right (scanning the $\mathcal{DR}$ list) until such a line is reached. Therefore, $\lceil \log h \rceil$ nodes of the tree are visited, and for each node only the intervals reported in output, plus one, are visited. Hence, if k is the output size, then the computational complexity of the search is $O(k + \log h)$. Since $\log h$ is the minimum number of bits needed to discriminate between two different extreme values, no query technique could have a computational complexity smaller than $\Omega(\log h)$, hence the computational complexity of querying with the interval tree is output-sensitive optimal. It is interesting to note that the time complexity is independent of the total number m of intervals, i.e., on the input size: indeed it only depends on the output size, and on the number of distinct extremes.

A. A General Data Structure

A general data structure for the interval tree can be devised by assuming that the set of input intervals is stored independently from the search structure, while each interval in the set can be accessed through a pointer. Therefore, for each element of $\mathcal{AL}$ and $\mathcal{DR}$ lists we store only a pointer to its related interval. All such lists are packed into two arrays, one for lists of type $\mathcal{AL}$, and one for lists of type $\mathcal{DR}$, which will be called the *big* $\mathcal{AL}$, and *big* $\mathcal{DR}$ arrays, respectively. Lists are packed in a consistent order (e.g., by following a depth-first visit of the tree), in such a way that the $\mathcal{AL}$ and $\mathcal{DR}$ lists attached to a given node of the tree start at the same location in the two big arrays, respectively. For each node r of the tree (see Figure 4) we store the discriminant value δ_r, an index referring to the starting element of its lists in the two arrays described above, and the length of such lists (recall that both lists have the same length). Since the tree is binary balanced, it can be also stored implicitly by using an array of nodes.

Therefore, if we assume a cost of a word for both integers, pointers, and floating point values, we will have that the bare tree requires $3h$ words, while the lists will require $2m$ words, for a total of $3h + 2m$. It shoud be taken into account that the cost of encoding the bare tree is expected to be small, at least in our application. Indeed, although in general we have $h \leq 2m$, in practice the intervals of $\mathcal{I}$ can only have extremes at a predefined, and relatively small, set of values: for instance, if data values are encoded by 16 bits, h is at most 65,536, while m can be several millions.

As for all other range-based methods, the storage cost of the interval tree is a crucial issue. In Section IV, we will address separately its application to unstructured and structured datasets, respectively, and we will discuss how storage cost can be optimized by exploiting special characteristics of the two kinds of datasets, respectively.

B. Exploiting Global Coherence

The interval tree can be used as an effective structure also to address coherence between isosurfaces: active cells for a given isovalue q', sufficiently close to another isovalue q, can be extracted efficiently by exploiting partial information from the set of cells active at isovalue q. Following Livnat et al. [11], this problem can be visualized in the span space as in Figure 5a: assuming that active cells at isovalue q are known, the list of active cells at isovalues q' are obtained by eliminating all points lying in the right rectangular strip (dashed), and by adding all points lying in the bottom rectangular strip (gridded).

In order to perform this task, active cells at q must be stored into an *active list*, which is updated next to obtain the corresponding active list for isovalue q'. By using an interval tree, the active list can be maintained in a compressed form, as a path on the tree, namely the path that is traversed when extracting active cells for isovalue q through the query algorithm described in Section III. The path starts from the root node, and has a length of $\log h$. For each node in the path we just need to maintain one flag (1 bit) to discriminate whether the $\mathcal{AL}$ or the $\mathcal{DR}$ list was used, one index addressing the first interval that was *not* active in such a list, and one flag (1 bit) to denote whether the next node is the left or the right child of the current node. In the example of Figure 3, the path is encoded as follows (by assuming that list locations are addressed starting at 0): ($\mathcal{DR}$,4,right),($\mathcal{AL}$,4,left),($\mathcal{DR}$,0,right), ($\mathcal{AL}$,1,null). It is evident that with a real dataset, the length of such a path is (on average) extremely smaller than the actual number of active cells.

The algorithm for computing active cells at isovalue q' scans the tree path, and updates it by either adjusting the index associated to each node, or recomputing the node completely. The traversal algorithm is described in detail by the pseudo-code in Figure 6. The main principle of the algorithm is the following. As long as both q and q' lie on the same side of the discriminant of the current node, then the same list is used, and the same child will follow, while it is sufficient to adjust the interval index by moving it either backward or forward, depending on whether $q > q'$ or $q < q'$, and whether $\mathcal{AL}$ or $\mathcal{DR}$ list is used. In the example of Figure 5b, this happens for nodes 1 and 2 in the path: in this case, all intervals in the gridded part of the horizontal stripe are included simply by advancing the index in the first triple from 4 to 8, while all intervals in the dashed part of the vertical stripe are included simply by backtracking the index in the second triple from 4 to 1. As soon as a *branching* node is found, i.e., a node such that its discriminant lies between q and q', the search is continued independently of the rest of the active path at q. Indeed, in this case, the other list for the current node must be used, while the rest of the path

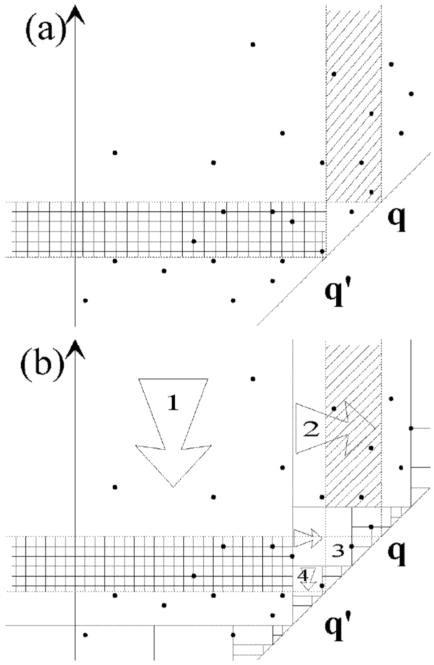

Fig. 5. The active intervals at q' are obtained by taking active intervals at q, subtracting those in the dashed strip, and adding those in the gridded strip (a). Active list update: node 1 is updated by moving the index forward, in order to include points in the gridded strip; node 2 is updated by moving the index backward, in order to remove points in the dashed strip; tree traversal is repeated for nodes 3 and 4 (b).

will be certainly not active at q'. This happens at node 3 in the example (compare it with Figure 3), when the $\mathcal{DR}$ list was traversed for q, while the $\mathcal{AL}$ list must be traversed for q'. Note that after visiting such a node,the opposite branch of the tree (in the example just the new node 4) must be visited.

In conclusion, we have that the update has a small overhead for encoding the list of active intervals, while it involves only traversing the intervals that make the difference between q and q', plus all the intervals appended to the branching node (in the example, node 3). In the worst case (i.e., when q and q' lie on opposite sides of the discriminant of the root node, this algorithm is totally equivalent to performing the query from scratch on the interval tree. An average case analysis depends on the distribution of intervals, and it involves evaluating the probability that a branching node is more or less deep in the path, and the probability to have more or less intervals inside that node. Due to its complexity, such an analysis is omitted.

```
UpdatePath(IntervalTree 𝒯, Path P, Real q,q')
begin
    r = root(𝒯);
    (ℒ,i,c) = first(P);
    δ_r = discriminant(r);
    while (q and q' are on the same side of δ_r)
        if (q > q') then
            if (ℒ = 𝒜ℒ) then
                while intervals not active at q' are found move i backward
            else
                while intervals active at q' are found move i forward
        else
            if (ℒ = 𝒜ℒ) then
                while intervals active at q' are found move i forward
            else
                while intervals not active at q' are found move i backward;
        r = child(r,c);
        δ_r = discriminant(r);
        (ℒ,i,c) = next(P);
    end while;
    if r not empty then
        set flag ℒ to the other list;
        set flag c to the other child;
        traverse list ℒ to set value of i;
        discard the rest of the path;
        traverse 𝒯 starting at child(r,c);
end;
```

Fig. 6. Pseudo-code of the algorithm for active list update

IV. Extraction of Isosurfaces from Structured and Unstructured Grids

As stated in Section I, the isosurface extraction problem is not limited to the selection of active cells. Other important aspects (cell classification, vertex and normal computation) must be taken into account in order to ensure the efficiency of the whole extraction process. Moreover, the memory overhead of the auxiliary data structure used for cell selection has to be considered in order to get a good tradeoff between time efficiency and memory requirements. While referring to the general method described in the previous section, we stress these aspects in the next subsections, by distinguishing between *un-*

structured datasets, whose cells can be tetrahedra, hexahedra, prisms, or pyramids, whose connectivity must be encoded explicitly, and *structured* datasets (i.e., cartesian, regular, rectilinear, curvilinear, and block structured grids), in which the connectivity among the hexahedral cells is implicit [17].

A. The case of Unstructured Grids

In the case of unstructured datasets, the input mesh is encoded by an array of vertices, where for each vertex we maintain its three coordinates, and its field value; and by a list of cells, where for each cell we maintain its connectivity list, made of four, five, six, or eight indices addressing its vertices in the vertex array, depending on whether the cell is a tetrahedron, a pyramid, a prism, or a hexahedron, respectively. The indices in the connectiviy list of each cell are sorted in ascending order, according to the field value of their corresponding vertices, in such a way that the minimum and maximum of the interval spanned by each cell will be given by the field values of the first and last vertex in the connectivity list, respectively. For a hybrid dataset, the list of cells can be encoded by using up to four different arrays, one for each type of cells. However, the list can be addressed as a single array, by assuming a conventional order (e.g., tetrahedra come first, next pyramids, next prisms, and last hexahedra), and by using the length of each list as an offset.

Given a dataset composed of n points, t tetrahedra, p pyramids, s prisms, and k hexahedra, we have a storage cost of $4n + 4t + 5p + 6s + 8k$ for the whole dataset. Recall that $t + p + s + k = m$ is the total number of cells, that $3h + 2m$ is the cost of the interval tree, and that $h \leq n$. Therefore, we have a memory overhead for the interval tree variable between 25% and 50%, depending on the number of cells of the different types, 25% being obtained for a dataset made only of hexahedra, and 50% for a dataset made only of tetrahedra: extreme values are not significative, however, since in the first case the dataset would probably be a structured one, while in the second case further optimization can be adopted, as discussed next.

If the input mesh is a tetrahedralization, the cost of storing the input mesh is $4n + 4m$. Since all cells are of the same type, we can sort the whole array of tetrahedra according to the order of their corresponding intervals in the big $\mathcal{AL}$ array, described in the Section

III-A. In this case, we can avoid storing the big $\mathcal{AL}$ array explicitly, since this comes free from the list of tetrahedra. In this case, we only need to maintain the big $\mathcal{DR}$ array, with a total cost of $3h + m$, hence less than 25% overhead.

After active cells have been extracted, cell classification consists in testing the values of the cell's vertices with respect the user-selected threshold, in order to devise the topology of the isosurface patch inside the active cell. Cell classification is generally not a critical task in the isosurface extraction process. However, in the case of tetrahedral meshes, this step can be slightly improved by exploiting the fact that indices of vertices in the connectivity list of each cell are stored in ascending value of field [16]. This implies that cell classification can be performed with at most two tests by using bisection.

A more critical task is the computation of vertices and normals. Due to the computational cost of this task, it is important to exploit the local coherence in order to avoid redundant computations. In the case of unstructured datasets, we adopt a dynamic hash indexing technique. For each isosurface extraction a hash table is built, and it is used to store and retrieve efficiently isosurface vertices and normals. In our implementation, the extracted surfaces are represented by adopting an indexed representation: an array of isosurface vertices is maintained, storing coordinates and normal vectors for each vertex, and an array of isosurface faces, each storing a connectivity list of three indices to the vertex array. Each isosurface vertex is identified by the active edge $v_1 v_2$ of the cell where it lies by using the two data points indexes v_1 and v_2 to build the hash key:

$$key(v_1, v_2) = \mid XOR(v_1, v_2 * n_{prim}) \mid_{hashsize}$$

where n_{prim} is a sufficiently large prime number. The computational overhead due to the computation of hash indexes is therefore small. When processing an edge during vertex computation, the hash table is inquired to know whether such computation has been done before and, if so, to retrieve the index of the interpolated vertex and normal in the corresponding array. Isosurface vertices and normals are computed explicitly, and stored only if the hash search fails. In this way each interpolation is done exactly once.

A common problem in the use of hashing is the definition of a suitable size for the hash table. In our case, all vertex and normal computations are performed after cell selection is completed, hence the hash table can be sized up dynamically by using the number k of active cells. Other approaches based on hashing define statically the hash table size [19]; using an hash table much larger than the current number of active cells may result in a degradation of efficiency due to more frequent chache miss. The number of intersected cells gives us a good estimate of the number of vertices in the resulting surface, and therefore of the number of entries in the hash table. Given k active cells, the number of vertices produced is lower than $3k$ in the case of hexahedral cells (redundancy factor 4) and $\frac{4}{3}k$ for tetrahedral cells (redundancy factor not less than 3). The effective redundancy measured in our experiments on tetrahedral meshes (the ratio between the number of access to the hash table and the number of vertices interpolated) is approximately equal to 5. In our implementation, we obtained a low rate of hash collisions by setting the hash table size equal to a prime slightly larger than $2.5k$. We detected a collision approximately every 100 accesses, and the maximum number of subsequent collisions detected in different extractions was in the range 3:7. Collisions are managed by adopting a linear scan strategy.

In order to speedup the computation of surface normals during isosurface extraction, we compute as a preprocessing step all field gradients at mesh vertices. Therefore, the surface normal at an isosurface vertex v can be simply obtained by linear interpolation from the normalized gradients at the endpoints of cell edge where v lies. In the case of tetrahedral meshes, the gradient of the scalar field within each cell σ of the mesh is assumed constant, i.e., it is the gradient of the linear function interpolating the field at the four vertices of σ. Similar interpolating functions can be adopted in order to estimate the gradient within a single cell of the other types. Then, the gradient at each mesh vertex v is computed as the weighted average of normalized gradients at all cells incident at v, where the weight for the contribution of a cell σ is given by the solid angle of σ at v. Note that this optimization on surface normal computation involves a further $3n$ storage cost, due to the need of maintaining gradients for all data points. The corresponding overhead is highly dependent on the ratio between the number of points n and the number of cells m. For a tetrahedral mesh, we have that on average $m \approx 6n$, and, therefore, the overhead would be less than 12.5%.

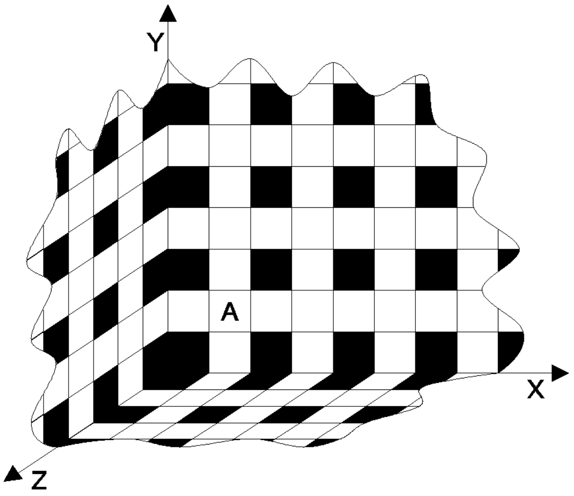

Fig. 7. The chess-board arrangement: in the case of regular grids, the data structure used to speedup the isosurface extraction does not need to store the *min-max* intervals of all the cells of the volume. Because each internal edge belongs to four cells, only the intervals corresponding to the *black* cells (as in a 3D chess-board) have to be maintained.

B. The case of Structured Grids

In the case of structured datasets, i.e. grids based on a hexahedral decomposition in which the connectivity information is implicit, we propose the use of a new conceptual organization of the dataset which both reduces the number of interval to be stored into the interval tree, and permits to devise a dataset traversal strategy that can efficiently exploit the local coherence. The resulting technique is in practice a compromise between a space-based approach, and a range-based approach, which tries to exploit the advantages of both. Though our proposal applies to every structured dataset, we will refer to regular ones in our discussion for sake of semplicity.

The number of intervals stored can be reduced on the basis of a simple but effective observation: in a Marching Cubes-like algorithm, the vertices of the triangular patches that form the extracted isosurface lie on the edges of the cells and, in a regular dataset, each internal edge is shared by four cells. Therefore, in order to be sure that every isosurface parcel will be detected, we just need to store the intervals of a minimal subset of cells that hold all the edges of the dataset.

Such a subset can be devised easily if we think to the volume dataset as a 3D chess-

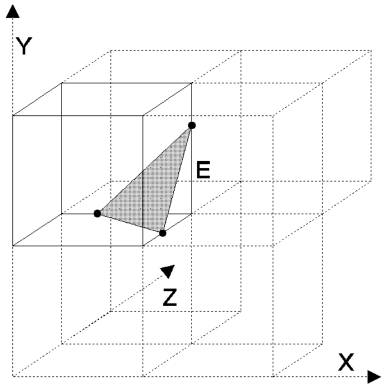

Fig. 8. Isosurface extraction is propagated from each active *black* cell to the adjacent *white* cells which share one or more active edges.

board in which the *black* cells (Figure 7) are those we are interested in. In other words, if $c[i,j,k]$ is a black cell, then its adjacent black cells are those which share a single vertex with $c[i,j,k]$. This conceptual arrangement presents some advantages:

- given a regular $I \times J \times K$ dataset (i.e. a volume of $(I-1) \times (J-1) \times (K-1)$ cells), the black cells can be easily indexed as follows:

$$[2i + |k|_2,\ 2j + |k|_2,\ k],$$

  with $i \in \{0, 1, \cdots, \lfloor \frac{I-2-|k|_2}{2} \rfloor\}$, $j \in \{0, 1, \cdots, \lfloor \frac{J-2-|k|_2}{2} \rfloor\}$, and $k \in \{0, 1, \cdots, K-2\}$, where terms $|k|_2$ (k modulo 2) make it possible to compute the indices for the even and odd layers of cells.
- the number of black cells in the dataset is 1/4 of the total number of cells, hence the number of intervals to be stored in the interval tree data structure is 1/4 of the total. This not only implies lower memory occupancy but also shorter construction and traversal times for the auxiliary data structure.

Each black cell has (at most) 18 edge-connected white cells. For each active black cell, the adjacent white cells that are also active (because of isosurface intersections occurring at the edges of the black cell) are determined easily on the basis of the configuration of the current black cell (Figure 8). Conversely, if a white cell is active, there must exist at least a black cell adjacent to it that is also active (special cases of white cells lying on the boundary of the dataset are discussed later). Therefore, once all active black cells have been located efficiently with an interval tree, all active white cells can be located by searching, in constant time, the neighbourhood of each active black cell.

The chess-board reduces the number of intervals to be stored, but it does not help with the local coherence: this can be managed by maintaining in a compact and easy-to-access data structure the information already computed for vertices and normals of the isosurface. Such an auxiliary structure would require a relevant memory overhead, unless we maintain a sort of locality of the computations. This simple observation gives the key to a compromise between a space-based and a range-based approach: we need to visit the black cells not only on the basis of the intervals arrangement, but also taking into account the topology of the grid.

In order to achieve this objective, we build an interval tree for each layer of cells (i.e., the cells formed by two adjacent slices of data), rather than building a single interval tree for the whole dataset. The interval tree for each layer stores the min-max intervals of the black cells in that layer. Each tree is then labelled with the $Tmin$-$Tmax$ interval, where $Tmin$ [$Tmax$] represents the minimum [maximum] of the min [max] values in the corresponding layer. Therefore, we have a forest of interval trees, and, for each tree, we can know in constant time whether the tree contains active cells or not.

If the interval trees in the forest are visited according to the sequence of layers in the dataset, then, during the traversal of the k-th tree, we only need to maintain a compact auxiliary data structure (called a *Vertex&Normal* data structure) for active cells of the three layers indexed by $k-1$, k, and $k+1$. The *Vertex&Normal* data structure stores information (vertices, normals, visited cells, etc.) that is being computed at each active cell, and it avoids redundant geometrical computations. Advancing to the $(k+1)$-th interval tree simply implies a circular shift of the indices of the layers in the Vertex&Normal data structure. The extraction strategy and the exploitation of the local coherence (i.e., the runtime part of the method) can be now summarized as follows:

- *Interval tree selection:* given an isovalue q, the trees in the forest are tested in sequence in order to individuate the active trees, i.e. the trees for which $Tmin \leq q \leq Tmax$. Each active interval tree, say the k-th, is visited using the algorithm presented in Section III;

- *Black cell processing:* for each active black cell, the Marching Cubes [12] algorithm is applied: on the basis of the configuration of the cell (determined with respect to q) we access the Marching Cubes lookup table, and we find the active edges of the current cell. By exploiting the *Vertex&Normal* data structure, we compute (and save) only the vertices and the normals not already computed in the processing of an adjacent white cell. On the basis of the configuration of the cell, we also select the adjacent active white cells where the isosurface extraction must be propagated. For example, if a vertex of the isosurface has been found on the edge E of the black cell $c[i,j,k]$ of the example in Figure 8, then the edge-connected white cells $c[i+1,j,k]$, $c[i+1,j,k+1]$, and $c[i,j,k+1]$ will be examined;
- *Active white cells processing:* once a black cell has been processed, the algorithm examines the connected active white cells that have not been processed yet. For each of them, the Marching Cubes algorithm is applied as in the previous case. White cells already examined are individuated by means of simple flags in the Vertex&Normal data structures. Note that a *propagation* list for the white cells is not necessary, because we individuate all the active white cells starting from one of the adjacent black cells;
- *Advancing:* the algorithm iterates on the next $k+1$-th interval tree (if it is active) by a simple circular shift of the layers in the Vertex&Normal data structure: information for the $k-1$-th layer is no longer necessary, and it is therefore rewritten by information on the $k+2$-th layer.

A further remark is necessary for the white cells that lie on the boundary of the dataset. As shown in Figure 7, some boundary edges of the dataset are not *captured* by black cells (e.g., the external edges of cell labeled A in the figure). However, if all sizes of the dataset are even, no further information is needed for such edges: it is easy to see that if an isosurface cuts one or more edges of a white cell that do not belong to any black cell, the isosurface must also cut some edge of the same cell internal to the volume, hence shared by a black cell.

In case one or more of the sizes I, J, and K of the dataset are odd numbers, then

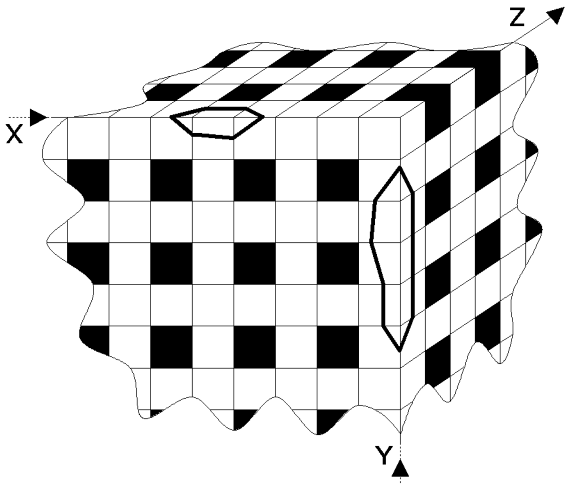

Fig. 9. Some of the cells of a dataset with two odd sizes are not *covered* by the chess-board. Small parts of the two isosurfaces could be lost.

part of the edges of at most $2(I-1)+2(J-1)+2(K-1)$ cells (i.e. cells forming six of the twelve corners of the volume) are not captured (not even indirectly) by the black cells of the chess-board (see Figure 9). As shown in the figure, in these situations small isosurface subsections can be lost. To solve this problem we can add the following step to our algorithm:

- *Unreachable cells test:* once an active tree has been visited and the corresponding active cells processed, the algorithm examines the (still not processed) white cells of the current layer whose edges are not captured by black cells.

An alternative solution to the previous step could be the insertion into the corresponding interval tree of the *black edges* of the unreachable white cells. However small, the number of cells to be tested separately does not justify the effort.

With the *chess-board* approach, the total asymptotic time for a query is, in the worst case, $O(\sqrt[3]{n}\log n + k)$, where k is the output size, by assuming a dataset with $\sqrt[3]{n}$ layers (i.e., $I = J = K$). Note that using a forest rather than a single interval tree adds an extra factor of $\sqrt[3]{n}$ to the optimal query time. Therefore, in this case we trade optimal asymptotic time for space. However, it should be noted that the $\sqrt[3]{n}\log n$ factor is usually negligible in practice, while the advantage that derives from exploiting local coherence is

Dataset			**Interval Tree**			
Name	grid type	nodes (n)	intervals (m)	depth	nodes (h)	creation time
Fighter	Unstr	13,832	70,125	15	12,538	1.50
Bluntfin	Unstr	40,960	224,874	16	28,022	5.34
Bucky	Str	2,097,152	512,191	8	27,404	7.37
CTHead	Str	7,405,568	1,820,728	8	24,524	29.58

TABLE I

DATA ON THE INTERVAL TREES FOR THE TEST DATASETS (TIMES ARE CPU SECONDS). FOR THE STRUCTURED DATASETS, THE COLUMN *nodes* INDICATES THE SUM OF THE NODES OF ALL THE $2D$ INTERVAL TREES, THE COLUMN *depth* INDICATES THE DEPTH OF THE DEEPEST TREE.

relevant.

As stated for the case of unstructured datasets, the complexity in space for the interval tree data structures can be expressed in terms of $3h + 2m$, with h the number of distinct interval endpoints and m the number of intervals to be stored. For a regular dataset with n data values, we have to store the intervals corresponding to the black cells, i.e., $m \cong n/4$ intervals. Since in real applications we usually have $h \ll n$, the requirement for $n/2 + 3h$ storage locations is very close to one half of dataset size.

The ratio between the interval tree memory requirements and the dataset occupancy becomes obviously more propitious in the case of non-regular structured datasets (e.g. the curvilinear ones).

Therefore, the *chess-board approach* helps in solving the problem of the memory occupancy of the interval tree data structure together with the problem of the local coherence.

V. Experimental Results

Our proposals, based on the interval tree data structure, were tested on a number of different datasets. We report here the results for two unstructured datasets:

Fighter, an unstructured dataset built on the Langley Fighter, reporting a wind tunnel model simulation performed at NASA Langley Research Center. The dataset was

Threshold	Facets	IT On	IT Off
Nasa Fighter - 70125 Tetrahedral cells			
2.6534	386	3	142
2.4420	1754	13	154
2.2007	5545	41	185
2.0221	9735	78	220
0.5599	20560	164	312
Bluntfin - 224874 Tetrahedral cells			
4.8722	444	3	255
0.3409	1184	7	238
4.2741	2100	12	263
3.2071	5171	33	279
2.1305	10384	72	304
0.5371	20663	154	357

TABLE II

Isosurface extraction times on tetrahedrized datasets, in milliseconds.

represented by adopting a 3D Delaunay triangulation;

Bluntfin, originally defined as a curvilinear dataset, it has been represented here by adopting a tetrahedral decomposition; Bluntfin represents the air flow over a flat plate with a blunt fin rising from the plate. Courtesy of NASA Ames Research Center;

and for two structured ones:

Bucky, a $128 \times 128 \times 128$ regular dataset representing the electron density map of a C_{60} fullerene molecule. Courtesy of AVS International Centre;

CTHead, a $256 \times 256 \times 113$ CAT scan of a head. Courtesy of the University of North Carolina at Chapel Hill.

The results refer to:

- the use of the interval tree data structure, the hash indexing technique and the precomputation of the gradients of the field in the vertices of the cells in the case of unstructured datasets, *IT On*, compared with a standard Marching Tetrahedra implementation, *IT Off* (see Table II);

Threshold	Facets	IT On	IT Off
Bucky - 501,191 cells			
242.5	5,376	40	2,060
237.5	10,208	60	2,090
230.5	20,936	120	2,140
214.5	50,704	270	2,300
193.5	10,1392	540	2,560
96.5	200,148	1,110	3,090
CTHead - 1,820,728 cells			
229.5	10,556	70	7,550
210.5	54,776	340	7,830
204.5	99,340	610	8,100
62.5	502,722	3,140	10,450
26.5	1,051,114	6,630	13,720
20.5	1,504,368	9,560	16,150
89.5	2,088,976	12,290	19,410

TABLE III

ISOSURFACE EXTRACTION TIMES ON STRUCTURED (REGULAR) DATASETS, IN MILLISECONDS.

- the use of the forest of interval trees and the chess-board approach in the case of structured ones, *IT On*, compared with a standard Marching Cubes implementation, *IT Off* (see Table III).

Numeric results have been obtained on an SGI Indigo2 (200MHz R4400 CPU, 16K instruction and 16K data caches, 1MB secondary cache, 32MB RAM, IRIX 5.3).

Table I reports numeric values on the datasets used and on the associated interval trees: the *type* of the grid (*Unstr* for unstructured or *Str* for structured); the number m of intervals, which is equal to the number of tetrahedral cells for the unstructured datasets and to the number of black cells for the structured ones; the interval tree *depth*, which represents the depth of the deepest tree in the case of structured datasets; the number h of nodes of the interval tree(s) (in the case of structured grids, this field represents the sum of the nodes of all of the trees); the *time* (in seconds) required to build the interval tree data structures.

Figures 10 and 11 refer to the unstructured grids and show the isosurface fitting times as a function of the number of triangular facets extracted. The reference algorithm is the Marching Tetrahedra with the use of a hash indexing technique and the pre-computation of the gradients of the field in the vertices of the grid (see Subsection IV-A).

Similarly, figures 12 and 13 refer to the regular grids. In this case the reference algorithm is the Marching Cubes with the use of the chess-board approach (see Subsection IV-B).

Figure 14 shows the speedup obtained with the use of the interval tree (the ratio of the times obtained with and without the use of the interval tree data structures) as a function of the fraction of volume data that contains the isosurface (ratio between the number of fitted facets an the number of cells); obviously, greatest speedups are obtained on isosurfaces of small size, i.e. when the traversal phase dominates the whole extraction process.

The space complexity can be simply calculated from the unstructured datasets using the space complexity previously defined, $3h + 2m$ memory words. The space required to store the interval tree data structures results therefore 174K memory words for the Fighter dataset and 522K memory words for the Bluntfin.
The structured datasets Bucky and CTHead required 1,080K and 3,628K memory words, respectively.

VI. CONCLUSIONS

We have presented and tested a speedup method for isosurface extraction based on the use of the *interval tree* data structure. The method considerably improves the performance of the traversal phase with respect to the standard Marching Tetrahedra and Marching Cubes algorithms. Optimal output-sensitive time complexity in extracting active cells is achieved. Memory overhead, according to the general interval tree representation proposed, is $3h + 2m$ words, with h the number of distinct extremes of intervals and m the number of cells.
With reference to the alternative proposals in literature, it should be noted that other range-based methods have comparable (or higher) overheads, and worst computational efficiency. On the other hand, the methods based on the interval modality present either memory costs wich are comparable to the cost of the interval tree [7] or lower expected

performance [11]. Surface-oriented methods give in general a more compact representation of the list of seeds intervals, but in the case of unstructured datasets they require encoding adjacency information which involves a 100% overhead over the minimal volume datasets representation.

To reduce space occupancy, which becomes a critical factor in the case of high resolution datasets, we proposed two different strategies, oriented to the two different data classes. An optimized interval tree representation for *unstructured* datasets (tetrahedra-based) was presented; it allows to reduce space occupancy to $3h + m$, and therefore enabling less than 25% overhead.

A partial representation of the cell intervals, based on the *chess-board* approach, was devised to reduce the number of intervals stored in the interval trees in the case of *structured* datasets. All of the active cells not represented directly are here detected by propagation. Although the reduced number of intervals encoded, the speedups obtained were very similar to those obtained with the naive interval tree implementation which encodes all of the intervals, as demonstrated empirically in the graph of Figure 14. It is noteworthy that our chess-board approach could also be efficiently used together with alternative space-based speedup approaches based on octrees and pyramids, which exploit the implicit addressing of regular datasets.

The other phases of the surface fitting process (cell classification, non redundant interpolation of vertices and normals) were also tackled. In particular, local coherence control is supported: in the case of unstructured datasets, with the use of dynamically-sized hash tables; in the case of structured datasets, representing intervals with a forest of interval trees and using a slice-based order in processing the data, which allows the definition of a compact auxiliary data structure for maintaining interpolated vertices and normals.

Moreover, a general approach for addressing global coherence has been proposed, which updates the active interval list produced in the previous isosurface extraction and results totally equivalent to performing the search by scratch in the worst case.

ACKNOWLEDGEMENTS

This work was partially financed by the Progetto Coordinato "*Modelli multirisoluzione per la visualizzazione di campi scalari multidimensionali*" of the Italian National Research

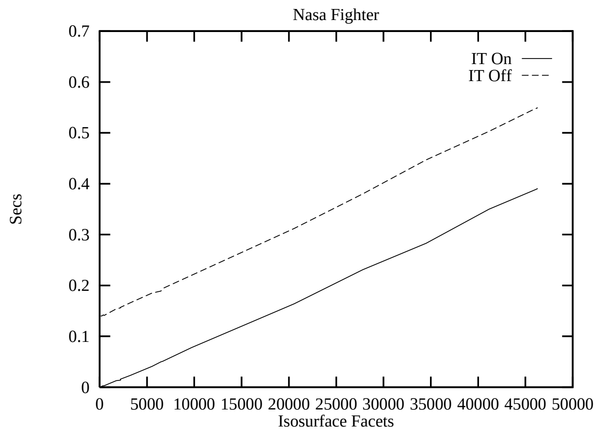

Fig. 10. Timing for the Fighter dataset

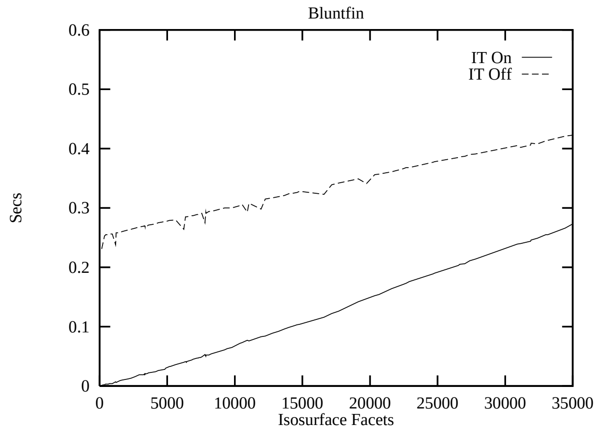

Fig. 11. Timing for the Bluntfin dataset

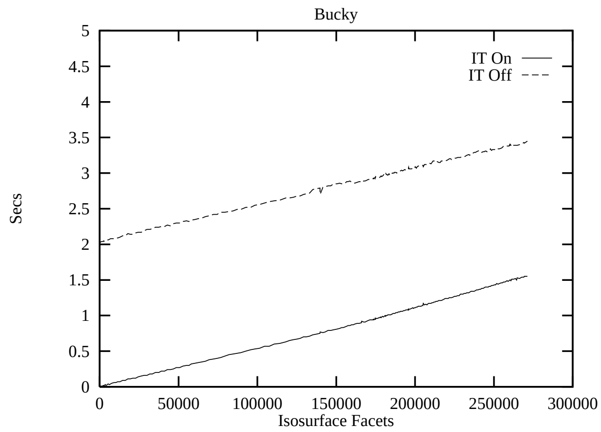

Fig. 12. Timing for the Bucky dataset

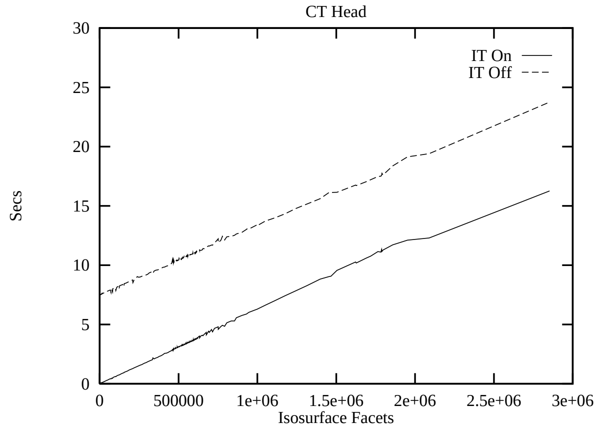

Fig. 13. Timing for the CTHead dataset

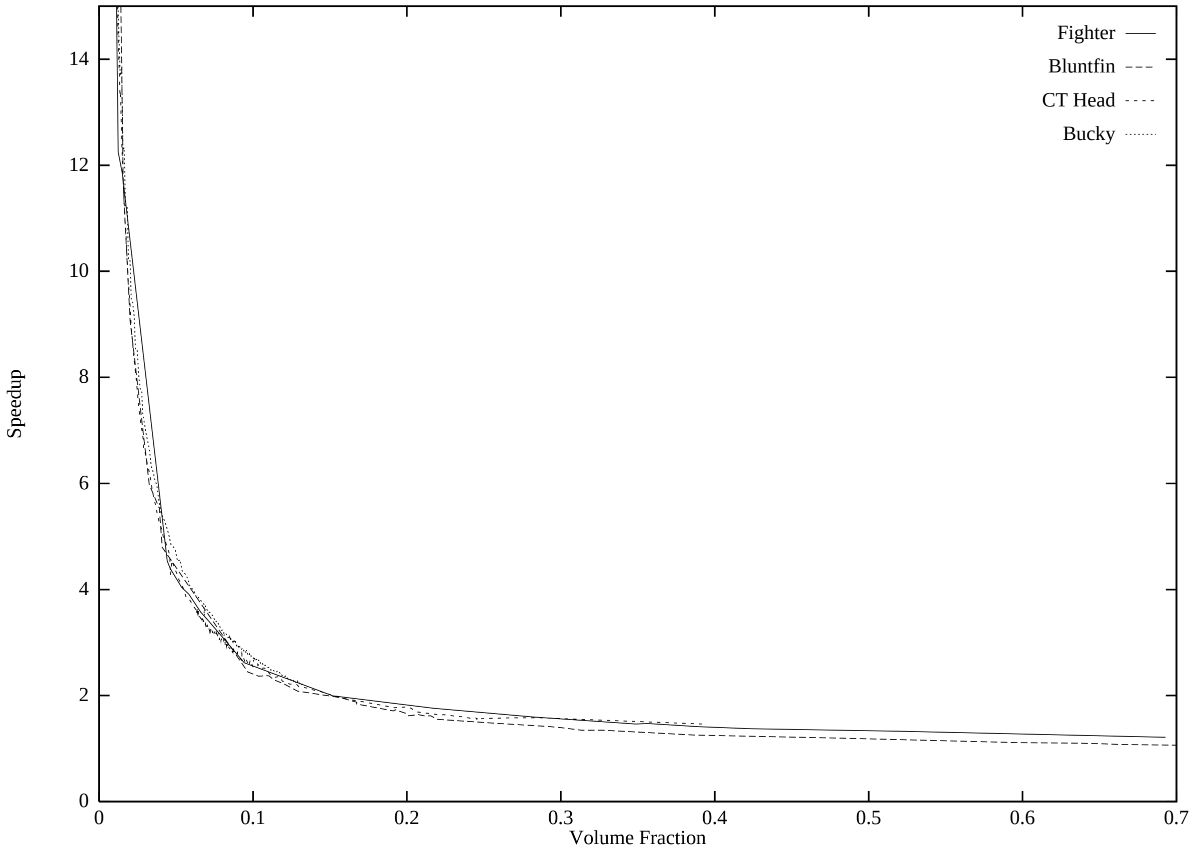

Fig. 14. Speedup vs. Volume Fraction (triangles / cells)

Council (CNR).

References

[1] C. L. Bajaj, V. Pascucci, D. R. Schikore, "Fast Isocontouring for Improved Interactivity" In *1996 Symposium on Volume Visualization Proc.*, San Francisco (CA), Oct. 28-29, pp. 39-46, 1996.

[2] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Multiresolution Modeling and Visualization of Volume Data", *Technical Report 95-22*, Istituto CNUCE - CNR, Pisa, Italy, July 1995.

[3] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Optimal isosurface extraction from irregular volume data", In *1996 Symposium on Volume Visualization Proc.*, San Francisco (CA), Oct. 28-29, pp. 31-38, 1996.

[4] P. Criscione, C. Montani, R. Scateni, and R. Scopigno, "DiscMC: an interactive system for fast fitting isosurfaces on volume data", In *Virtual Environments and Scientific Visualization '96*, M. Goebel, J. David, P. Slavik and J.J. van Wijk eds., Springer Wien, pp. 178-190, 1996.

[5] H. Edelsbrunner, "Dynamic data structures for orthogonal intersection queries", *Technical Report F59*, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.

[6] R. S. Gallagher, "Span filter: an optimization scheme for volume visualization of large finite element models", In *IEEE Visualization '91 Conf. Proc.*, pp. 68-75, 1991.

[7] M. Giles and R. Haimes, "Advanced interactive visualization for CFD", *Computing Systems in Engineering*, 1, pp. 51-62, 1990.

[8] T. Itoh, Y. Yamaguchi, and K. Koyamada, "Volume Thinning for Automatic Isosurface Propagation", In *IEEE Visualization '96 Conf. Proc.*, pp. 303-310, 1991.

[9] T. Itoh and K. Koyamada, "Automatic Isosurface Propagation using an Extrema Graph and Sorted Boundary Cell Lists", *IEEE Transactions on Visualization and Computer Graphics*, 1(4), pp. 319-327, 1995.

[10] M. Laszlo, "Fast generation and display of iso-surfaces wireframe", *CVIGP: Graphical Models and Image Processing*, 54(6), pp. 473-483, 1992.

[11] Y. Livnat, H. Shen, and C. Johnson, C. "A near optimal isosurface extraction algorithm for structured and unstructured grids", *IEEE Transactions on Visualization and Computer Graphics*, 2(1), pp. 73-84, 1996.

[12] W. Lorensen and H. Cline, "Marching cubes: a high resolution 3D surface construction algorithm", *ACM Computer Graphics (SIGGRAPH '87 Conf. Proc.)*, 21(4), pp. 163-170, 1987.

[13] C. Montani, R. Scateni, and R. Scopigno, "Discretized Marching Cubes", In *Visualization '94 Conf. Proc.*, R. Bergeron and A. Kaufman, Eds., IEEE Computer Society Press, pp. 281-287, 1994.

[14] F. Preparata and M. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, 1985.

[15] H. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson, "Isosurfacing in Span Space with Utmost Efficiency (ISSUE)" In *Visualization '96 Conf. Proc.*, San Francisco (CA), Oct. 27-Nov. 1, pp. 287-294, 1996.

[16] H. Shen and C. Johnson, "Sweeping simplices: a fast iso-surface extraction algorithm for unstructured grids", In *IEEE Visualization '95 Conf. Proc.*, pp. 143-150, 1995.

[17] D. Speray and S. Kennon, "Volume probes: interactive data exploration on arbitrary grids", *ACM Computer Graphics (1990 Symposium on Volume Visualization Proc.)*, 24(5), pp. 5-12, November 1990.

[18] M. van Kreveld, "Efficient methods for isoline extraction from a digital elevation model based on Triangulated Irregular Networks", In *Sixth International Symposium on Spatial Data Handling Proc.*, pp. 835-847, 1994.

[19] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation", *ACM Transaction on Graphics*, 11(3), pp. 201-227, July 1992.

[20] G. Wyvill, C. McPheeters, B. Wyvill, "Data structures for soft objects", *The Visual Computer*, 2(4), pp. 227-234.