Spatial Indexing on Tetrahedral Meshes

Leila De Floriani
Department of Computer Science
University of Genova
Genova, Italy
deflo@disi.unige.it

Riccardo Fellegara
Department of Computer Science
University of Genova
Genova, Italy
riccardo.fellegara@gmail.com

Paola Magillo
Department of Computer Science
University of Genova
Genova, Italy
magillo@disi.unige.it

ABSTRACT

We address the problem of performing spatial queries on tetrahedral meshes. These latter arise in several application domains including 3D GIS, scientific visualization, finite element analysis. We have defined and implemented a family of spatial indexes, that we call tetrahedral trees. Tetrahedral trees subdivide a cubic domain containing the mesh in an octree or 3D kd-tree fashion, with three different subdivision criteria. Here, we present and compare such indexes, their memory usage, and spatial queries on them.

Categories and Subject Descriptors
I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object Representations

Keywords
Tetrahedral meshes, Spatial Indexes, Octrees, kd-trees

1. INTRODUCTION

Tetrahedral meshes are used in several applications, such as physics, chemistry, medicine, etc., to model three-dimensional scalar fields, defined at regularly or irregularly distributed data points, and also three-dimensional features in 3D GISs. The meshes arising from finite element analysis do not have a regular structure and the domain has usually an arbitrary shape. In 3D GIS, the meshes arise from the discretization of irregular geometries and are still irregular. Even when scalar fields are sampled over regular grids, the simplification of the data set, and the need to include critical points in the model, lead to reduced data sets that have no longer a regular structure. In all such cases, meshes are typically large, and spatial indexes are needed to access them and perform spatial queries efficiently.

There is very little work in the literature on techniques for performing spatial queries on tetrahedral meshes, and thus our attempt here is to fill such gap.

Hierarchical spatial indexes for points in the 3D Euclidean space are provided by PR-octrees and PR-kd-trees [9]. In such indexes, the shape of the tree is independent of the order in which the points are inserted, and the points are only in the leaf blocks.

The PMR quadtree [7] is a spatial index for a collection of edges in the plane, not necessarily forming a polygonal map, and can be used for spatial objects in the plane [4]. If the insertion of an edge causes the number of edges in a leaf block to exceed a predefined threshold, the block is split only once at this time. This choice avoids excessive splitting, but makes the shape of the resulting tree dependent on the order in which the segments are inserted. The number of nodes in a PMR-quadtree is proportional to the number of line segments and is independent of the maximum depth of the tree [5]. This provides a theoretical justification for the good behavior and use of the PMR quadtree for line segments.

The class of PM-quadtrees [10] extends the PR-quadtree (which is for points in the plane) to represent polygonal maps considered as a structured collection of edges. There are three variations, namely the PM1-quadtree, the PM2-quadtree and the PM3-quadtree, which differ in the criterion to define the content of a leaf block. They all maintain a list of edges in the leaf blocks. In [2] we have extended the PM2-quadtree to produce a hierarchical spatial index for triangle meshes. In such index, a block is a leaf if and only if either it contains one vertex, and all intersecting triangles are incident in that vertex, or it contains no vertex, and all intersecting triangles have a common vertex lying outside the leaf block.

PM-octrees are used to index the boundary of a polyhedral object in space [1, 6, 9]. In particular, PM1, PM2, and PM3-octrees have been proposed, where the subdivision rule is similar to the corresponding quadtrees but considers faces instead of edges.

Pointer-less representations (i.e., which do not encode the tree structures with parent-child links) have been developed in the literature mainly for quadtrees, and octrees, see for instance the linear quadtrees [3]. The idea is to represent only the leaf blocks in the tree. The location and size of each leaf block are encoded in a so-called location code, namely a sequence of bits denoting the corresponding root-to-leaf path. The result is used as a key into an auxiliary data structure, such as a hash-table or a B-tree. A PMR-quadtree implementation using location codes is presented in [4].

In this paper, we define a family of spatial indexes for tetrahedral meshes, that we call tetrahedral trees. Such indexes use three different criteria to guide the space subdivi-
sion, and can be either octrees or 3D kd-trees. We provide an extended comparison of all six possible instances of tetrahedral trees, based on their space overhead and efficiency in spatial queries.

The remainder of this paper is organized as follows. In Section 2, we present our new spatial indexes for tetrahedral meshes, and their memory requirements. In Section 3, we present experiments to evaluate and compare the various spatial indexes. Finally, in Sections 4, we draw some concluding remarks and directions for future work.

2. THE TETRAHEDRAL TREES

Tetrahedral trees are based on a recursive subdivision of an initial cubic domain by tracing median planes, into eight (all three median planes) or two (just one plane, cycling over subdivision levels) blocks for octrees and kd-trees, respectively. The specific indexes differ in the criterion guiding the subdivision. We use the generic name tree indicating both an octree and a kd-tree. The names of our indexes are inspired by the names of the variants of quadtrees and 2D mesh. We call

A PR-Tetrahedral Tree (PR-T Tree) is a direct extension of the PR (Point-Region) octree, or kd-tree, since the subdivision is based on the vertices of the tetrahedral mesh. We call \( k_t \) the capacity of a block, i.e., the maximum allowed number of vertices per block. The subdivision criteria for the current block \( b \) is thus as follows. If \( b \) contains a number of vertices less or equal to \( k_t \), then \( b \) is a leaf; otherwise, \( b \) is recursively subdivided into eight or two, depending on whether we are building an octree or a kd-tree. Each leaf block \( b \) stores the set of vertices contained in and the set of tetrahedra intersecting it. Note that a vertex always belongs to a single leaf, while a tetrahedron may belong to an arbitrary number of leaves. Moreover, there are leaf blocks which do not contain vertices, but are intersected by tetrahedra.

A PMR-Tetrahedral Tree (PMR-T Tree) extends the principle underlying PMR-quadtrees. The subdivision is based on the tetrahedra and not on the vertices. A threshold value \( k_t \) defines the maximum allowed number of tetrahedra intersecting a given block \( b \). The subdivision rule is the following one: a block \( b \) is subdivided, if it intersects more than \( k_t \) tetrahedra, but this is done only once. This is different from all other methods, where the subdivision condition is always checked recursively on the children of a given block. Unlike other indexes considered here, the final shape of the tree depends on the insertion order of the tetrahedra. Each leaf node in a PMR-T tree stores the set of tetrahedra intersecting the corresponding block. Such tetrahedra may be more than \( k_t \), but they cannot be more than \( d + k_t \), where \( d \) is the depth of the leaf (\( d = 0 \) at the root).

A PM-Tetrahedral Tree (PM-T Tree) is an octree or kd-tree in which the subdivision of a block \( b \) is determined by the vertices inside \( b \) and by the tetrahedra intersecting \( b \). The subdivision rules are as follows. A block \( b \) which contains less than or exactly \( k_t \) vertices and intersects less than or exactly \( k_t \) tetrahedra is a leaf block. A block \( b \) which contains less than or exactly \( k_t \) vertices but more than \( k_t \) tetrahedra is a leaf block if and only if (i) all tetrahedra intersecting \( b \) are incident in the same vertex which must be either contained in the block, or located on some of the faces of the block (see Figure 1), or (ii) all tetrahedra intersecting \( b \) are incident in the same edge either intersecting the block, or lying on some of the faces of the block. Each leaf block \( b \) stores the set of vertices contained in it, and the set of tetrahedra intersecting it.

We assume that the tetrahedral mesh is encoded in an indexed data structure, as detailed below. Both vertices and tetrahedra of the mesh are stored in arrays. Each entry in the vertex array stores the three coordinates of a vertex \( v \). Each entry in the tetrahedron array stores the indexes of the four vertices of a tetrahedron \( t \) within the vertex array.

For each node of the tree corresponding to a block \( b \), we store the node type (an integer value); for leaf blocks we also store a list of tetrahedra and possibly a list of vertices (depending on the specific index): such lists contain the indexes of vertices and tetrahedra in the corresponding arrays. Since we are just interested in comparing the different indexes, our prototype implementation explicitly represents the tree structure by using pointers.

We denote with \( N_V \) and \( N_T \) the number of vertices and tetrahedra in the mesh, with \( N_N \) the number of nodes in the tree, with \( N_L \) the number of leaf nodes, and with \( N_{TL} \) the total number of tetrahedra stored in the lists at leaves. Thus, \( N_{TL} \) counts each tetrahedron repeated as many times as the number of leaves containing it. Note that the total number of vertices stored in the lists at leaves is the same as \( N_V \) (since each vertex is in exactly one leaf block). We assume that integers and pointers have a unit cost. The cost for a node includes a fixed part for its type; for leaves, we also have the total length of their lists, and, for internal nodes, the total number of parent-child pointers. The space requirements of the three indexes are summarized in Table 1. Table 1 simply reports the overhead in storage space due to the spatial index, disregarding the storage cost for the input mesh. Note that values of \( N_V, N_L, N_{TL} \) are different in the different tetrahedral trees for the same input mesh. An estimate of the maximum depth of a tetrahedral tree depends on parameters related with the input mesh. Let us consider first tetrahedral octrees. For simplicity, we assume that the mesh domain is the unit cube. In an octree, at each level the length of the block edge is halved. Therefore, the height \( h \) and the edge length \( l \) of the deepest leaf are related by \( l = 1/2^h \). In a PR-T octree, it may be necessary

<table>
<thead>
<tr>
<th>Index</th>
<th>node types</th>
<th>lists in leaves</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-T</td>
<td>( N_N )</td>
<td>( N_V + N_{TL} )</td>
<td>((N_N - N_L)N_C)</td>
</tr>
<tr>
<td>PMR-T</td>
<td>( N_N )</td>
<td>( N_{TL} )</td>
<td>((N_N - N_L)N_C)</td>
</tr>
<tr>
<td>PM-T</td>
<td>( N_N )</td>
<td>( N_V + N_{TL} )</td>
<td>((N_N - N_L)N_C)</td>
</tr>
</tbody>
</table>

Table 1: Space requirements for the three spatial indexes: \( N_C = 8 \) for octrees, and \( N_C = 2 \) for kd-trees.
to perform the recursive subdivision until the diagonal of a block is equal to the minimum distance \(d_v\), between two input vertices. We have that \(\sqrt{3} = d_v\), which leads to \(h = \log_2(\sqrt{3}/d_v) = \log_2(\sqrt{3} - \log_2 d_v)\).

For the PM-tetrahedral octree, the same relation with \(k_v\) holds, but, in addition, it may be necessary to subdivide until the diagonal of a block is equal to the minimum tetrahedron height \(d_t\) and the minimum face height \(d_f\). Thus, the maximum height \(h\) of a PM-T octree is given by \(h = \log(\sqrt{3} - \min(\log_2 d_v, \log_2 d_t, \log_2 d_f))\).

For the tetrahedral kd-trees, the same block size is reached in three as many subdivision levels. Therefore, the above bounds must be multiplied by 3.

### 3. EXPERIMENTAL EVALUATION

We have performed experiments on ten tetrahedral meshes raising from 3K to 300K vertices, which include regular meshes (vertices are on a cubic grid, with five or six tetrahedra inside each cube), semi-regular meshes with cuboidal domain (obtained by simplifying a regular mesh by means of long-edge bisection) both with a cuboidal domain and with an irregular domain, and irregular meshes (scattered vertices connected through a Delaunay tetrahedralization). Figure 2 shows the various mesh types, and Table 2 summarizes the meshes used in the experiments. For every mesh, we have built the six tetrahedral trees with different values of the relevant parameter(s) between \(k_v\) and \(k_t\).

We have tested the spatial indexes for their size requirements, and their efficiency in queries. We have considered two basic queries on tetrahedral meshes common to most applications: point location and window queries. A point location query consists of searching for the tetrahedron \(t\) of the mesh containing a query point \(q\). In case more tetrahedra contain \(q\) (i.e., \(q\) lies on a shared vertex, edge, or face of them), then we report just one of such tetrahedra. A window query returns the (possibly empty) set of all tetrahedra of the mesh that intersect a query box \(b\).

The algorithms for query processing are based on a top-down tree traversal to locate the leaf block containing the query point \(q\), or the leaf blocks intersecting the query box \(b\). Each of such leaf blocks stores a set of tetrahedra. We check each of such tetrahedra against point \(q\), or box \(b\), to decide if it must be reported in the answer.

For point location queries, we have used a set of 100 randomly generated query points within the bounding box of the mesh. For window queries, we have used three sets of 100 randomly generated query windows within the bounding box of the mesh, where the window edge is equal to 5%,

<table>
<thead>
<tr>
<th>Mesh type</th>
<th>vertices</th>
<th>tetrahedra</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>274,625</td>
<td>1,572,864</td>
</tr>
<tr>
<td></td>
<td>32,768</td>
<td>148,955</td>
</tr>
<tr>
<td>semi-regular</td>
<td>147,430</td>
<td>793,760</td>
</tr>
<tr>
<td>(cuboidal domain)</td>
<td>59,410</td>
<td>310,786</td>
</tr>
<tr>
<td></td>
<td>18,938</td>
<td>96,800</td>
</tr>
<tr>
<td>semi-regular</td>
<td>326,683</td>
<td>1,778,510</td>
</tr>
<tr>
<td>(irregular domain)</td>
<td>48,308</td>
<td>163,631</td>
</tr>
<tr>
<td></td>
<td>14,161</td>
<td>59,870</td>
</tr>
<tr>
<td>irregular</td>
<td>48,518</td>
<td>240,122</td>
</tr>
<tr>
<td></td>
<td>2,896</td>
<td>12,936</td>
</tr>
</tbody>
</table>

### Table 2: Meshes used in the experiments.

3.1 OCTREES versus KD-TREES

As expected, the numbers of nodes and leaves are less in the kd-tree than in the octree: the ratio is 0.7 – 0.8 for nodes and 0.4 – 0.5 for leaves, where the PMR-T produces the most relevant node reduction and the PM-T the least one. It is worth to note that regular meshes, where each grid cell is filled with six tetrahedra, tend to have the same number of leaves in the octree and in the kd-tree.

The depth increases about three times (3.2 – 3.3) from octree to kd-tree, matching the theoretical estimate. The number of tetrahedra per leaf in the kd-tree is 1.7 – 1.9 times larger than in the corresponding octree; the largest ratio is achieved in the PR-T tree. Memory consumption decreases when moving from octrees to kd-trees, with a ratio between 0.85 and 0.98.

When considering a kd-tree instead of an octree in the point location query, the number of visited nodes decreases with a ratio about 0.7, and the number of geometric tests (point-in-tetrahedron) increases of about 1.5 – 1.6 times. This is due to the smaller number of nodes and to the larger number of tetrahedra per leaf. The query time gets multiplied by a factor 1.4 – 1.6.

The number of visited nodes in window queries decreases when passing from octree to kd-tree. The ratio is from 0.7 to 0.8, less evident for the PM-T, and more evident for the PR-T and small boxes. The number of geometric tests increases more for small boxes (1.2 – 1.4 times), and less for large boxes (where it remains about equal). The query time increases, especially for small boxes; for large boxes it remains almost the same.

3.2 INFLUENCE OF THE SUBDIVISION CRITERION

We restrict our analysis, without lack of generality, to tetrahedral octrees. For each index, we used four different values for the parameter(s) on each mesh. Given a value of \(k_v\) for the PR-T octree, we choose values of \(k_t\) (used in the PMR-T and PM-T octrees) which are about the average number of tetrahedra per leaf in the PR-T octree.

For regular meshes, all indexes tend to produce the same number of nodes, leaves, tetrahedra per leaf, and the same height, for a fixed parameter value. Note that, due to the regularity of the mesh, a certain number of vertices per leaf implies a certain number of tetrahedra, and vice-versa, and, thus, the two parameters \(k_v\) and \(k_t\) of the PM-T octree are not independent. Still, space requirements are different. The PR-T and PM-T trees require the same memory space, which is more than the space required by the PR-T tree, because they maintain a list of vertices in each node.

A similar behavior is observed in semi-regular meshes with a cuboidal domain, which are obtained with moderate simplification of a regular grid. Differences between the three indexes become more evident on such meshes, as simplification increases. In some cases, the PR-T tree is much smaller than the other indexes. This means that several of its leaves contain a number of tetrahedra larger than the average number, and, thus, they are subdivided in the other indexes. Such subdivisions, however, produce leaves with a number of tetrahedra largely below the threshold. For
In one case, the average number of tetrahedra per leaf in the PR-T tree is 56. By imposing \( k_t = 40 \) for other indexes we obtain an average number of tetrahedra per leaf of about 10. In order to obtain 50 – 60 tetrahedra per leaf, we must set \( k_t = 120 \) (and, still, the PM-T tree creates more nodes than the PR-T tree).

For irregular meshes, the PM-T tree is the index with the smallest number of tetrahedra per leaf, but with the largest numbers of nodes and leaves, and with the largest height. On semi-regular meshes with an irregular domain, the behavior is between that on semi-regular and on completely irregular meshes.

The most relevant evaluation of the indexes is on point location. During point location, the average number of visited nodes is in the range 15–40, and the average number of geometric tests is less than 150 (and much less for small mesh). For small boxes, window queries visit on average 20–130 nodes. The average number of geometric tests is for most of the meshes less than 12 times the average output size (number of tetrahedra intersecting the box). Query execution times rarely are above 0.5 milliseconds for point location, and are in the order of milliseconds for box queries.

The point location algorithm visits about the same number of nodes with all spatial indexes. On regular meshes, query times are comparable. On other meshes, the best spatial index depends on the mesh and it is often either the PR-T or the PMR-T tree. However, differences in query times are small.

In window queries, the PMR-T tree performs fewer geometric tests, sometimes half than the other two indexes, especially for large query boxes. The number of tests with the PM-T tree is only little smaller than with the PR-T tree. The PR-T tree visits fewer nodes, but makes more geometric tests, whose cost may become dominant for large query boxes and/or large meshes. Window queries with small boxes exhibit a similar behavior to point location queries, but the PM-T tree is sometimes the best index on meshes with an irregular domain. While enlarging the size of the query box, the number of meshes for which the PMR-T tree is the most efficient index increases.

### 4. CONCLUDING REMARKS

To support spatial queries on tetrahedral meshes, we have proposed a family of spatial indexes, called tetrahedral trees, based on octrees and kd-trees and on three different block subdivision strategies.

Our experiments show that the PMR-T tree has in general a better performance in queries, and it also has a moderate memory overhead. The other two indexes have notably larger size, with no much gain in query time. This result matches the one reported in [8] while comparing PMR-quadtrees and PM-quadtrees used to index the set of segments forming a polygonal map in geographic applications.

Using kd-trees instead of octrees brings slightly larger memory requirements and no gain in query times. Thus, octrees are preferable.

We plan to implement PMR-T octrees by using a pointerless implementation and a standard B-trees. Since the sizes of the tetrahedral meshes are usually huge, we are also considering a GPU implementation.

### 5. ACKNOWLEDGMENTS

This work has been partially supported by the National Science Foundation under grant CCF-0541032.

### 6. REFERENCES


