Tetrahedral Trees: A Family of Hierarchical Spatial Indexes for Tetrahedral Meshes

RICCARDO FELLEGARA, German Aerospace Center (DLR)
LEILA DE FLORIANI, University of Maryland at College Park
PAOLA MAGILLO, University of Genova
KENNETH WEISS, Lawrence Livermore National Laboratory

We address the problem of performing efficient spatial and topological queries on large tetrahedral meshes with arbitrary topology and complex boundaries. Such meshes arise in several application domains, such as 3D Geographic Information Systems (GISs), scientific visualization and finite element analysis. To this aim, we propose *Tetrahedral trees*, a family of spatial indexes based on a nested space subdivision (an octree or a kD-tree) and defined by several different subdivision criteria. We provide efficient algorithms for spatial and topological queries on Tetrahedral trees and compare to state-of-the-art approaches. Our results indicate that Tetrahedral trees are an improvement over R*-trees for querying tetrahedral meshes; they are more compact, faster in many queries and stable at variations of construction thresholds. They also support spatial queries on more general domains than topological data structures, which explicitly encode adjacency information for efficient navigation, but have difficulties with domains with a non-trivial geometric or topological shape.

Additional Key Words and Phrases: tetrahedral meshes, spatial indexes, octrees, kD-trees, spatial queries, topological queries

ACM Reference Format:
Riccardo Fellegara, Leila De Floriani, Paola Magillo, and Kenneth Weiss. 2020. Tetrahedral Trees: A Family of Hierarchical Spatial Indexes for Tetrahedral Meshes. *ACM Trans. Spatial Algorithms Syst.* 1, 1, Article 1 (January 2020), 34 pages. https://doi.org/10.1145/3385851

1 INTRODUCTION

Tetrahedral meshes are used to discretize space in a broad range of applications across numerous scientific disciplines. They are used, for example, to model scalar and vector fields sampled at irregularly distributed points in space [21, 78], or to model three-dimensional features in Geographic Information Systems (3D GISs) [68, 69, 97]. They also underpin finite element and structural analysis over domains with complex topology and arbitrary shapes [13, 40, 50, 57]. Even when fields are sampled over regular grids, dataset simplification produces unstructured datasets which need to be triangulated through tetrahedral meshes [1, 19, 41, 67]. The popularity of tetrahedral meshes in these applications stems, in part, from their simple representation, their ability to adapt to features at varying spatial and temporal scales, and the availability of efficient and robust mesh generation algorithms even in the presence of complex geometric boundaries [18, 86, 87].

In simulation, visualization and analysis applications, we often require local information about features in the problem domain. For example, to probe a field at an arbitrary location [3, 39, 53], we

Authors' addresses: Riccardo Fellegara, German Aerospace Center (DLR), Braunschweig, Germany, riccardo.fellegara@dlr.de; Leila De Floriani, University of Maryland at College Park, College Park, MD, USA, deflo@umiacs.umd.edu; Paola Magillo, University of Genova, Genova, Italy, paola.magillo@unige.it; Kenneth Weiss, Lawrence Livermore National Laboratory, Livermore, CA, USA, kweiss@llnl.gov.

2374-0353/2020/1-ART1 $15.00
https://doi.org/10.1145/3385851

need to interpolate field values from nearby samples. Similar needs arise when integrating field quantities over spatial regions of interest, and when computing volumetric overlap between the cells of two meshes, for example, to map field data between different spatial discretizations [42].

Such *spatial queries* are typically composed of a few primitive building blocks that can efficiently locate the mesh cells covering the query region. In particular, *point location* queries find the mesh cell containing a given point, while *range* queries, also referred to as *box* or *window* queries, find all mesh cells that overlap an axis-aligned region of space. *Spatial joins* can be used to combine spatial features from multiple meshes. These are also the main query types in spatial databases [63, 72, 85, 96], and can be used to define proximity and containment queries.

After performing a spatial query on a mesh, we are often interested in traversing the local mesh in the vicinity of the query region. For example, in fluid simulations, *tracer particles* are seeded near features of interest and flow along integral paths of vector fields or scalar field gradients [55, 82]. Similarly, visibility [27, 90] and watershed [56, 75] queries traverse the mesh towards a local horizon or critical points. Other applications require local modifications to the mesh connectivity. For example, when inserting new points into a Delaunay triangulation, one needs to locate and repair the invalidated tetrahedra (i.e., those whose circumcircles contain the inserted point) [38, 73].

Despite the vast literature on spatial indexes and queries, few works exploit the rich connectivity information available in tetrahedral meshes while also supporting spatial queries over arbitrary spatial domains. While spatial indexes based on R-trees [45] and Bounding Volume Hierarchies (BVHs) are perhaps the most widely used, e.g. in spatial databases [63] and collision detection [32], they are optimized for collections of discrete objects rather than meshes. As such, mesh elements (tetrahedra, vertices, etc.), which are mutually adjacent or incident, could be stored in distant tree branches, making it difficult to use the results to traverse through the mesh. Alternatively, spatial queries based on *stochastic walks* require only the mesh connectivity but are restricted to convex domains [28, 59]. In this approach, which is popular in the computational geometry literature, queries are initialized at a random mesh cell and traverse through the mesh towards the query point. This approach is very easy to implement, since it does not require a spatial index, but can fail when the domain is not simply-connected.

Here, we propose *Tetrahedral trees*, a family of spatial indexes over tetrahedral meshes which support efficient spatial queries and reconstruction of the local mesh connectivity on the query results. Tetrahedral trees utilize a nested spatial index (an octree, or a kD-tree) to recursively decompose the embedding space of the mesh, and index the cells of the mesh in all leaf blocks of the tree that they intersect. The decomposition can be easily tuned to limit the number of vertices and/or tetrahedra encoded in a leaf block. Although each tetrahedron can be indexed in multiple leaf blocks, our efficient leaf block encoding (adapted from the representation in [35]), which encodes contiguous ranges of cell indices, yields a very compact representation for Tetrahedral trees. The storage overhead for the spatial index is typically only 5-10% higher than the one for an *indexed mesh* representation that only encodes the boundary vertices for each tetrahedron. We note that the indexed representation does not support efficient spatial or topological queries, while the data structure in [35] does not support efficient spatial queries. Our work builds on a preliminary short paper [25] in which we introduced some spatial indexes for tetrahedral meshes and spatial (but not topological) queries on such meshes. As we demonstrate in Sections 6 and 7, our compacted leaf block representation significantly reduces the spatial index overhead while also improving query response times since it acts as a clustering mechanism for the locally indexed submeshes within a leaf block.

The major contributions of this paper are:

- Scalable algorithms for executing spatial queries on tetrahedral meshes with additional support for navigating through the mesh connectivity on the query results.
- Improved encoding and compression of the information indexed within each leaf block and effective tuning of bucketing thresholds that enable the indexing of larger tetrahedral meshes on commodity hardware. We demonstrate the effectiveness of the Tetrahedral trees representation on structured and unstructured tetrahedral meshes with up to 30 million tetrahedra.
- Extension of subdivision rules, originally defined for 2D uncorrelated data, to decompose and organize structured and unstructured tetrahedral meshes. We demonstrate the importance of considering the cardinality of the star of a vertex in a tetrahedral mesh for setting bucketing thresholds and evaluate the effectiveness of these rules on the resulting spatial decomposition and index compression.
- An extensive evaluation of Tetrahedral trees through comparisons, in terms of memory and query times, with representatives of the most commonly used data structures for spatially querying tetrahedral meshes: an adjacency-based topological data structure (the *IA data structure* [65]) and an R^**-tree* [7]. We demonstrate that, across all datasets, Tetrahedral trees are significantly smaller (typically by a factor of 2-5), can be easily tuned and consistently respond to queries faster than the state-of-the-art data structures.

The remainder of this paper is organized as follows. In Sections 2 and 3, we provide background notions on tetrahedral meshes and review related work on spatial indexes as well as topological data structures. In Section 4, we introduce Tetrahedral trees and describe the criteria used to drive their spatial decomposition and the data structures for representing them. We discuss the spatial and topological queries supported by Tetrahedral trees in Section 5. In Section 6, we evaluate the storage requirements and generation times of Tetrahedral trees, comparing with state-of-the-art data structures. We then empirically evaluate the performance of Tetrahedral trees against the state-of-the-art data structures in Section 7. We conclude in Section 8 by outlining directions for future work.

2 BACKGROUND

In this section, we review some background notions about tetrahedral meshes. Let k be a non-negative integer. A *k-simplex* σ is the convex hull of $k+1$ independent points in Euclidean space. These points are called the *vertices* of simplex σ and k is its *dimension*. A 0-simplex is a vertex, a 1-simplex is an edge, a 2-simplex is a triangle and a 3-simplex is a tetrahedron. An *h-face* σ' of a k-simplex σ is an h-simplex ($0 \leq h \leq k$) generated by $h+1$ vertices of σ. Dually, σ is said to be a *coface* of σ'. For instance, the triangles generated by three vertices of a tetrahedron σ are its 2-faces and σ is a coface of each of these triangles.

A *tetrahedral mesh* Σ is a collection of vertices, edges, triangles and tetrahedra. In this paper, we are concerned with tetrahedral meshes that are *conforming*, *pure* and have a *manifold* domain. In a conforming mesh Σ, each face of a simplex in Σ also belongs to Σ and for each pair of simplices σ and τ in Σ, σ and τ are either disjoint, or they intersect along a common face. In a pure tetrahedral mesh, all vertices, edges and triangles in Σ are faces of a tetrahedron in Σ. Finally, a *manifold* object is a subset of the Euclidean space for which the neighborhood of each internal point is homeomorphic to an open ball, and the neighborhood of each boundary point to an open half ball.

The *boundary* of a simplex σ is the set of all faces of σ. Dually, the *star* of a simplex σ in a tetrahedral mesh Σ is the set of its cofaces in Σ. For instance, the star of a vertex v is the set of edges, triangles and tetrahedra incident in v. Two simplices are said to be mutually *incident* if one

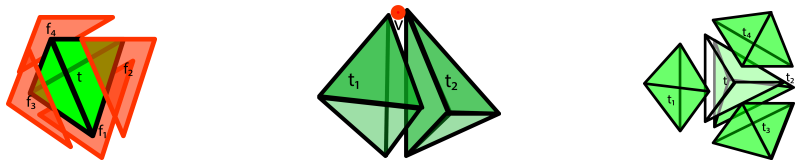


Fig. 1. Examples of topological relations. (a) Tetrahedron-Face boundary relation for tetrahedron t consists of the four triangular faces $\{f_1, f_2, f_3, f_4\}$. (b) Vertex-Tetrahedron co-boundary relation for vertex v consists (in this example) of its two incident tetrahedra $\{t_1, t_2\}$. (c) Tetrahedron-Tetrahedron adjacency relation for tetrahedron t consists of its four adjacent tetrahedra $\{t_1, t_2, t_3, t_4\}$.

of them is a face of the other, while two h-simplices are *j-adjacent*, with $0 \leq j < h$, if they share a j-face. Two vertices are called *adjacent* if they are the boundary vertices of an edge.

The connectivity among the simplices of a tetrahedral mesh is identified through so-called *topological (or connectivity) relations*, which are based on the notions of incidence and adjacency. Let us consider a tetrahedral mesh Σ and a p-simplex σ in Σ, with $0 \leq p \leq 3$ (i.e., a vertex, edge, triangle or tetrahedron). We have the following *topological relations* for σ [26]:

- *boundary relation* $R_{p,q}$, with $0 \leq q < p$, relates σ to its q-simplex faces within Σ.
- *co-boundary relation* $R_{p,q}$, with $p < q \leq 3$, relates σ to its q-simplex cofaces within Σ.
- *adjacency relation* $R_{p,p}$ relates σ to the p-simplices in Σ that are p-adjacent to σ.

An alternative notation for topological relations is to use the name of the involved p- (and q-) dimensional simplices. Using this notation, the boundary relations for a tetrahedron σ are: *Tetrahedron-Face* ($TF(\sigma)$), i.e., the set of triangles bounding σ (see Figure 1(a)), *Tetrahedron-Edge* ($TE(\sigma)$), i.e., the set of edges on the boundary of σ, and *Tetrahedron-Vertex* ($TV(\sigma)$), i.e., the set of vertices on the boundary of σ.

The co-boundary relations of a tetrahedron σ are empty since tetrahedra are the simplices of highest dimension in a tetrahedral mesh, and, thus, are not on the boundary of any other simplex. For a vertex v in a tetrahedral mesh, we have the following co-boundary relations: *Vertex-Tetrahedron* ($VT(v)$) (see Figure 1(b)), *Vertex-Face* ($VF(v)$), and *Vertex-Edge* ($VE(v)$) which consist of the tetrahedra, triangles and edges incident in v, respectively.

Adjacency relation *Tetrahedron-Tetrahedron* ($TT(\sigma)$) for a tetrahedron σ is the set of tetrahedra sharing a triangle face with σ (see Figure 1(c)). The *Vertex-Vertex* ($VV(v)$) adjacency relation consists of the set of vertices having an edge in common with a given vertex v. Topological relations for faces (triangles) and edges are defined in a similar fashion.

Topological data structures explicitly encode a subset of all possible topological relations in a tetrahedral mesh and are, thus, efficient in supporting navigation, but they are not well suited for spatial queries, like point location or range queries. Conversely, spatial indexes are specifically designed to efficiently support spatial queries but are not typically aware of mesh connectivity.

3 RELATED WORK

In this section, we review data structures related to the new spatial indexes presented in this work.

3.1 Spatial indexes

Spatial indexes differ in their decomposition of the embedding space of the data, often into a set of parallelepipeds, which we refer to as *blocks*. In an *object-based* decomposition, blocks are

constructed by considering bounding regions of the entities they index, while in a *space-based* decomposition, blocks are obtained by partitioning the embedding space. *Space-based hierarchical* spatial indexes recursively subdivide the embedding space into *non-overlapping* blocks according to a fixed pattern, where the *children* of a block partition the space associated with their *parent* block. For this reason, hierarchical non-overlapping space-based decompositions are also known as *nested* decompositions. The original data are generally associated with *leaf blocks*, i.e., those without children, while the *internal blocks* are used to drive spatial queries.

The main classification of hierarchical spatial indexes is into those using *regular* refinement and those using *bisection* refinement [11]. Regular refinement of hyper-rectangular blocks generates *quadtrees* [5], in 2D, and *octrees*, in 3D, while the bisection refinement of axis-aligned hyper-rectangles bisected by axis-aligned hyperplanes generates *kD-trees*. These decompositions have been originally defined for indexing point data and subdivide the space into blocks of equal size (generating *PR quadtrees* and *PR kD-trees*, where PR stands for Point Region [64]), or using the positions of the points to define subdivision planes (generating Point quadtrees and kD-trees [36]). Octree-based decompositions for points are widely used for indexing and analyzing Light Detection and Ranging (*LiDAR*) acquisitions [31, 58, 74, 79]. While there is a vast literature on spatial indexes, we review only those relevant to the work presented in this paper. Please see [80] for an extensive treatment of the subject.

The class of *Polygonal Map* (PM)-quadtrees [81] extends PR-quadtrees to represent polygonal maps, viewed as collections of edges in 2D space. There are four variants, namely, the *PM_1-quadtree*, the *PM_2-quadtree*, the *PM_3-quadtree* and the randomized *PMR-quadtree*. They all maintain a list of edges in the leaf blocks but differ in their subdivision rules. A PM_3-quadtree has the same structure as the PR-quadtree built on the vertices of the polygonal map, but, in addition, it stores all the edges that intersect each leaf block. The other PM-quadtrees apply subdivision based on the number and configuration of edges in a block. The subdivision rule for the PM_1 and PM_2-quadtrees is applied recursively during the insertion of an edge while that of the PMR-quadtree [61] is only applied once per insertion. This gives rise to a probabilistic behavior where the order in which the segments are inserted affects the shape of the resulting tree. As proven in [54], the number of blocks in a PMR-quadtree is proportional to the number of line segments and is independent of the maximum depth of the tree.

The first attempt to extend a PM-quadtree to index triangle meshes is the *PM_2-Triangle quadtree* [24], which consists of a spatial index superimposed on a topological data structure (in this case the IA data structure [65]). The former is used to execute spatial queries, while topological queries are answered by working directly on the IA data structure. In a PM_2-Triangle quadtree, each leaf block contains at most one vertex, which dramatically limits the scalability of this representation. To overcome this issue, an extension of a bucketed PM_3-quadtree to triangle meshes has been recently proposed in [34]. This representation encodes minimal connectivity for a triangle mesh combined with a spatial index. The representation is minimal in the sense that each triangle encodes only the vertices in its boundary, thus enabling the efficient extraction of the full connectivity locally, while the spatial index enables the navigation of the mesh at a global scale and allows for an efficient execution of spatial queries.

The PM-quadtree family has also been extended to encode polyhedral surfaces [15, 60, 80], using octrees to index the boundary cells of these objects. The leaf blocks of these *PM-octrees* either explicitly store the boundary elements present in the block [15], or the plane equations of the indexed elements [60]. *Space Partition (SP)-octrees* [14] maintain information within their internal blocks in addition to their leaf blocks. Thus, internal blocks provide an approximate object description. The subdivision of the space stops when the object portion lying in the block can be defined as the intersection of planes (i.e., it is locally convex, or locally concave).

Object-based hierarchical spatial indexes are widely used in database management systems (*DBMS*) such as Oracle Spatial [63] and PostGIS, a spatial extension for PostgreSQL [72], to encode collections of disconnected objects in space. The most representative of such spatial indexes are *R-trees* [45] and their numerous variants which can be defined in terms of the DBMS's underlying *B-tree* representation [22]. R-trees hierarchically group nearby objects by their *minimum bounding rectangles*, which can overlap. The *R-tree* [45] is a *dynamic* data structure, i.e., the indexed objects are inserted one-by-one, but numerous variants have been defined with specific packing techniques, i.e., inserting an a-priori known *static* set of objects into the structure to optimize the storage overhead and the retrieval performance. The *dynamic* family includes the *Packed R-tree* [77] and the R^+*-tree* [84], while the *static* family includes the *P-tree* [52], the R^**-tree* [7] and the *Priority R-tree* [4].

Spatial indexes have also been used in the scientific visualization literature to probe field data [39, 53]. Langbein et al. [53] combine a kD-tree to index the vertices of a polyhedral mesh with a topological data structure encoding the *Cell-Vertex* and *Vertex-Cell* relations of the mesh. During a point location query, the kD-tree is used to locate a vertex near the query point. A polyhedron incident to this vertex is then used to navigate through the mesh towards the query point. The spatial index in the *celltree* data structure [39] is based on the *bounding interval hierarchy* scheme. A celltree is a binary tree of axis-aligned boxes generated by bisection refinement, similar to a kD-tree. However, unlike a kD-tree, the child blocks do not form an exact partition of their parent. Rather, during refinement, the cells in a block are disjointly distributed into its two children and each block maintains the bounding box of its indexed cells. Each cell is indexed in a single leaf block, but answering a point location query requires visiting multiple branches of the celltree.

3.2 Topological data structures for tetrahedral meshes

There has been much research in designing efficient topological representations for triangle meshes [26] and significantly less for tetrahedral meshes. The *Indexed mesh data structure* is the most compact existing data structure for a tetrahedral mesh. It uses an array to store the coordinates of the vertices, and a separate array to encode the *Tetrahedron-Vertex* (TV) relation, i.e., the indices of the four vertices of each tetrahedron. This data structure is sufficient for rendering and interpolation within each tetrahedron, but does not support an efficient navigation of the mesh connectivity. The Indexed data structure with Adjacencies (*IA data structure*) [62, 65] extends the indexed data structure by also encoding the *Tetrahedron-Tetrahedron* (TT) relation and a partial *Vertex-Tetrahedron* (VT^*) relation, i.e., one tetrahedron in the star of each vertex instead of the entire set of incident tetrahedra. This enables the efficient extraction of all topological relations.

The *Sorted Opposite Table (SOT) data structure* [44] extends the *Corner Table* (*CoT*), defined in [76] for triangle meshes, to tetrahedral meshes, and introduces a compact version of the IA data structure. The SOT data structure explicitly encodes only the TT relation in an array, and implicitly encodes the VT relation by rearranging the order of the tetrahedra within the tetrahedra array. The TV relation is implicitly represented as well, and it can be reconstructed at run-time through a traversal of TT relation. Since modifications to the mesh require non-local reconstructions of the associated data structures, the SOT data structure is most suitable for static meshes.

The *PR-star octree* [93] is a hierarchical topological data structure for tetrahedral meshes which has recently been extended to encode simplicial complexes in arbitrary dimensions [35]. Although it uses a spatial index to encode the mesh, it does not support spatial queries. Specifically, the leaf blocks of the tree encode only the tetrahedra incident in their indexed vertices and are used to reconstruct the local mesh connectivity. It has been proven to be effective within applications, such as local curvature estimations, mesh validation and simplification [93], and morphological feature extractions [94].

Out-of-core approaches based on topological data structures have been proposed for the efficient execution of spatial queries [47, 48, 66] on unstructured tetrahedral meshes. Papadomanolakis et al. [66] present a representation optimized for point and range queries on tetrahedral meshes in a database. The query processing technique defined for spatial queries, called *Directed Local Search* (*DLS*), takes advantage of mesh connectivity, and uses a B-tree (provided by the underlying DBMS) for indexing the tables encoding *Tetrahedron-Vertex (TV)* and *Tetrahedron-Tetrahedron (TT)* relations. This data structure is equivalent to the IA data structure and suffers from the same limitations at answering spatial queries, i.e., it is guaranteed to succeed only when the domain of the tetrahedral mesh is convex.

In [47, 48] a *relational DBMS* supporting finite element analysis operations is described, and a data structure similar to the indexed data structure is proposed, encoding the vertices, the edges and tetrahedra of a tetrahedral mesh, plus the *Tetrahedron-Vertex* (*TV*), the *Tetrahedron-Edge* (*TE*) and *Edge-Vertex* (*EV*) relations (i.e., the vertices and edges in the boundary of each tetrahedron, and the vertices in the boundary of each edge).

4 TETRAHEDRAL TREES

Tetrahedral trees are a family of spatial indexes for tetrahedral meshes based on the recursive subdivision of an initial cubic domain containing the mesh through median planes into eight or two blocks for octrees and kD-trees, respectively. The specific indexes differ in the criterion guiding the subdivision, and in the information stored in the leaf blocks. We use the generic name *tree* to indicate both octrees and kD-trees.

We consider a block to be *closed* at the three square faces incident in its lower-left corner, and *open* at the remaining faces. More precisely, a block consists of all points (x, y, z) such that $x_1 \leq x < x_2$, $y_1 \leq y < y_2$, and $z_1 \leq z < z_2$, where (x_1, y_1, z_1) is the lower-left corner and (x_2, y_2, z_2) is the upper-right corner of the block. Exceptionally, for blocks whose x_2, y_2 or z_2 lies on the domain boundary, the corresponding faces are considered closed. In all cases, an *empty* block is a leaf block which does not intersect any tetrahedra from the mesh.

In Section 4.1, we define subdivision rules driving our spatial subdivisions, and in Section 4.2, we describe the Tetrahedral trees resulting from these subdivisions and provide algorithms for generating them. In Section 4.3, we present the data structures we developed for encoding Tetrahedral trees.

4.1 Subdivision rules

We define three subdivision rules for generating Tetrahedral trees over tetrahedral meshes: one based on the vertices and two on the tetrahedra of tetrahedral mesh Σ. These rules are *bucketed* extensions of rules in the PR and PM families that have been shown to be more effective on points, and line segments compared with the original PR- and PM-quadtrees. Specifically, our vertex rule is a bucketed extension of the Point-Region (PR) rule on 2D and 3D points in space, while the two rules on tetrahedra are bucketed extensions of those underlying PM_2-quadtrees and octrees and PMR-quadtrees, respectively. As we demonstrate in Section 6, bucketing is an important aspect of our rules, as it enables the indexing of much larger datasets, a clear limitation for all the rules available in the literature. In our design phase we also considered other options, such as a subdivision rule based on the centroid of each tetrahedron where each tetrahedron is indexed by a single leaf block. After an initial evaluation, we discarded these since queries could no longer be satisfied entirely by the geometry within a single leaf block.

Vertex-based subdivision rule. This subdivision rule considers a threshold, k_V, on the number of vertices indexed in a block b:

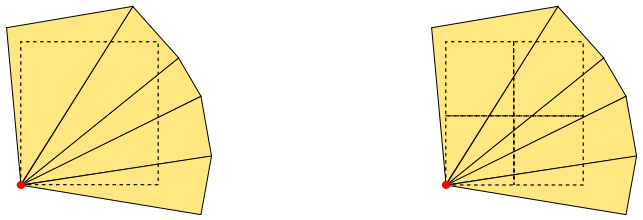


Fig. 2. (a) A situation in which a leaf contains more than $k_T = 4$ tetrahedra (shown in 2D). All tetrahedra are incident in the red vertex on the block boundary. (b) If we divide the block, we create another block in the same situation, leading to an infinite subdivision.

(i) a leaf block b may index up to k_V vertices;
(ii) if a leaf block b indexes more than k_V vertices it is recursively split until condition (i) is met.

Tetrahedron-based subdivision rule. This subdivision rule considers a threshold value, k_T, on the number of tetrahedra which can intersect a leaf block b:

(i) a leaf block b may index up to k_T tetrahedra, unless the next condition is verified;
(ii) a leaf block b may index more than k_T tetrahedra if and only if all tetrahedra intersecting b are incident in a common vertex v, which can be either inside or outside b;
(iii) otherwise, the block is recursively split until either condition (i) or (ii) is met.

Rule (ii) avoids splitting a block b when the same configuration would be repeated in one of the children. Figure 2 illustrates this problem in 2D. If we split the leaf block in Figure 2(a), we get the four blocks in Figure 2(b), where the bottom-left leaf block has the same configuration that started the subdivision process.

Randomized tetrahedron-based subdivision rule. This subdivision rule is still based on a *splitting* threshold value, k_T, on the number of tetrahedra. A block b is subdivided if it intersects more than k_T tetrahedra, but, unlike in the other two subdivision rules, this is done only once per insertion. It extends the principle underlying PMR-quadtrees [61], and it is guaranteed to result in a finite number of subdivision steps.

4.2 The Tetrahedral trees family

Each member of the Tetrahedral Tree family pairs an underlying tree topology (octree or kD-tree) with one or more subdivision rules defined in the previous section, which also determines what is stored in its leaf blocks. We consider eight members of the Tetrahedral tree family defined by two tree topologies and four rule sets, as summarized in Table 1.

P-Tree. A *Point-based Tetrahedral tree* uses the *vertex-based* subdivision rule. It is a direct extension of a bucketed PR-quadtree/octree or kD-tree [64], since the subdivision is based only on the vertices of the tetrahedral mesh. Each leaf block b stores a reference to the vertices inside it and to the tetrahedra intersecting it. Each vertex always belongs to a single leaf, while a tetrahedron σ belongs to all leaf blocks that are intersected by σ. A leaf block in a P-Tree may contain an arbitrary number of tetrahedra and there may exist leaf blocks which do not contain vertices, but only tetrahedra intersecting them.

Given a user-defined threshold k_V, we generate a P-Tree by inserting the vertices one at a time, only visiting the internal blocks that *geometrically* contain each vertex v. When a leaf block b is

Table 1. Leaf block thresholds used to build each type of Tetrahedral tree and spatial information encoded within each leaf block.

index	threshold applies to		encodes list of	
	vertices (k_V)	tetrahedra (k_T)	vertices (b_V)	tetrahedra (b_T)
P-Ttree	✓		✓	✓
PT-Ttree	✓	✓	✓	✓
T-Ttree		✓		✓
RT-Ttree		✓		✓

reached, v is added to b, and the overflow condition for b is checked. If b contains more than k_V vertices, it is recursively split until an overflow occurs in the resulting leaf blocks. After inserting all vertices into the P-Ttree, the refinement is fixed. The tetrahedra of T are then inserted into the leaf blocks that intersect them without triggering further refinements.

T-Ttree. A *Tetrahedron-based Tetrahedral tree* uses only the *tetrahedron-based* subdivision rule. As such, it is a direct generalization of the PM_2-quadtree/octree to tetrahedral meshes, adding also a bucketing threshold on the number of tetrahedra per block. The subdivision of a block b is driven by the tetrahedra intersecting b. Each leaf block b stores only the set of tetrahedra intersecting it.

The generation strategy for T-Ttrees is different from the one for P-Ttrees, as here we add tetrahedra directly, without previously inserting their vertices. Then, given a tetrahedron σ, the leaf blocks that intersect σ are identified, and σ is added to these blocks. For each of such block b, if b contains a number of tetrahedra which is less or equal to the threshold value k_T, σ is added to b. Otherwise, it is checked if all tetrahedra intersecting b plus tetrahedron σ are incident in a common vertex. If the condition above is satisfied σ is inserted in b, otherwise b is split and all tetrahedra of b plus σ are recursively re-inserted in all child blocks intersecting them.

PT-Ttree. A *Point-Tetrahedron-based Tetrahedral tree* uses both the *vertex-based* and the *tetrahedron-based* subdivision rules. It generalizes PM_2-quadtree/octree to tetrahedral meshes adding bucketing thresholds on the number of vertices and of tetrahedra. The subdivision of a block b is determined by the vertices inside b and by the tetrahedra intersecting b. Similarly to a P-Ttree, each leaf block stores references to the vertices and tetrahedra intersecting it.

In the generation of a PT-Ttree, the vertices are first inserted as in the P-Ttree with vertex threshold k_V. Then, the tetrahedra are inserted as in T-Ttree with a tetrahedra threshold k_T.

RT-Ttree. A *Randomized-Tetrahedron Tetrahedral tree* applies the *randomized tetrahedron-based* subdivision rule. As such, it extends the PMR-quadtree to tetrahedra introducing also a splitting threshold guiding block subdivision. Unlike the previous three indexes, the final shape of the tree depends on the insertion order of the tetrahedra. Similarly to a T-Ttree, each leaf block b stores the set of tetrahedra intersecting b. Note that in a RT-Ttree the number of tetrahedra associated with a leaf block can be greater than the splitting threshold k_T. It has been proven in [61] for a PMR quadtree built on the edges of a map that the number of edges intersected by a leaf block cannot exceed the sum of the splitting threshold and of the depth of the leaf block. The same result holds for an RT-Ttree, i.e., the number of tetrahedra in a leaf block of a RT-Ttree can be at most equal to $d + k_T$, where d is the depth of the leaf and k_T is the splitting threshold. For example, if we build a RT-Ttree in the 2D example of Figure 2, the block in Figure 2(a) is split after inserting the

fifth triangle, since $k_T = 4$, resulting in the subdivision shown in Figure 2(b). Note that the bottom left leaf block intersects $k_T + d = 4 + 1$ triangles.

In the generation of an RT-Ttree, for each tetrahedron σ, the leaf blocks intersecting σ are located. For each such leaf block b, if b contains a number of tetrahedra which is less than or equal to threshold k_T, then σ is added to b. Otherwise, b is split, and all tetrahedra, plus σ, are added to every child block intersecting them.

4.3 Encoding Tetrahedral trees

A Tetrahedral tree consists of three components:

- a tetrahedral mesh Σ,
- the tree structure of the containment hierarchy, and
- the information associated with the leaf blocks.

The cost for encoding the tetrahedral mesh is the same as for all Tetrahedral trees, while the cost of the other two components is specific for each of them. The *connectivity* of the tetrahedral mesh Σ is encoded in an indexed mesh data structure, which uses two arrays V and T, for the *vertices* and *tetrahedra*, respectively. The vertex array encodes the spatial embedding of mesh Σ by storing the three vertex coordinates for each vertex of Σ. The tetrahedron array encodes, for each tetrahedron σ in Σ, the indexes of its four vertices in the vertex array.

The tree describing the nesting structure of the index is encoded using an explicit pointer-based tree data structure, in which each block b of the tree is represented with a single data type. Our encoding of the tree hierarchy is independent of the tree topology, i.e., it can be used on octree and kD-tree decompositions without any modification. Each block b of a Tetrahedral tree contains a pointer to its parent block and to an array of children. This latter is empty if b is a leaf block. For all Tetrahedral trees, each leaf block b also contains an array b_T of references to the tetrahedra intersecting b. The leaf blocks in a P-Ttree and a PT-Ttree also contains an array b_V of references to the vertices indexed by b.

We encode these local leaf block vertex and tetrahedra arrays using the approach proposed in [35] for a topological data structure for simplicial complexes in arbitrary dimensions based on vertex clustering. This approach compresses the index arrays using *sequential range encoding (SRE)*, a variant of *run-length encoding* [49] that represents each *run* of consecutive indexes with a pair of integers encoding the starting index of the run and a count of the number of elements in the run. With this encoding, we can easily intersperse individual indices (encoded using a positive integer) and runs of indices (which begin with a negative integer) within the same array (see Figure 3(a)). The effectiveness of this compression increases when paired with a procedure that reorders the mesh vertices and elements to better exploit their locality (see [35] for additional details). In our case, we use the spatial locality induced by the tree decomposition to reorder the mesh and leaf block arrays after we construct the index. That is, a run represents a set of vertices or tetrahedra indexed by the same leaf block or within neighboring leaf blocks (i.e., those indexed by the same set of leaf blocks). Since each vertex is indexed in a single leaf block, we can represent each local vertex array with a single run (i.e. using just two integers).

The novelty of our approach lies in applying the SRE compression method, originally defined for a topological data structure to a spatial index and in exploiting properties of SRE to accelerate query response times (see Section 5). As we demonstrate empirically in Section 6, SRE compression retains its effectiveness for spatial indexes underlying Tetrahedral Trees, where each tetrahedron can be indexed by an arbitrary number of leaf blocks (rather than at most four as in [35]).

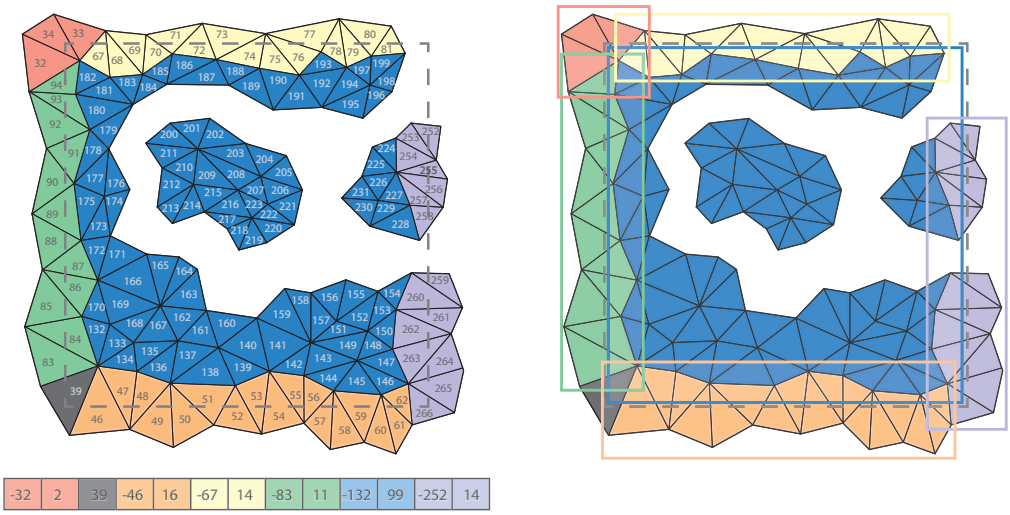


Fig. 3. In a *Sequential-Range Encoded (SRE)* block, *runs* of contiguous tetrahedron indices are compressed into a pair of integers encoding the starting index and length of the run. This block (shown in 2D) contains 163 tetrahedra with indices {*32-34, 39, 46-62, 67-81, 83-94, 132-231, 252-266*} using an array of length 13.
(a) Indices of tetrahedra within a block (dotted gray square) and corresponding compressed SRE representation of the runs. (b) Bounding boxes of the runs can be used to accelerate spatial queries. Note that entries within a run are not necessarily in the same connected component of the mesh.

5 QUERYING A TETRAHEDRAL TREE

In this section, we discuss how both spatial and topological queries are implemented within Tetrahedral trees. These are the building blocks for many higher level algorithms on tetrahedral meshes.

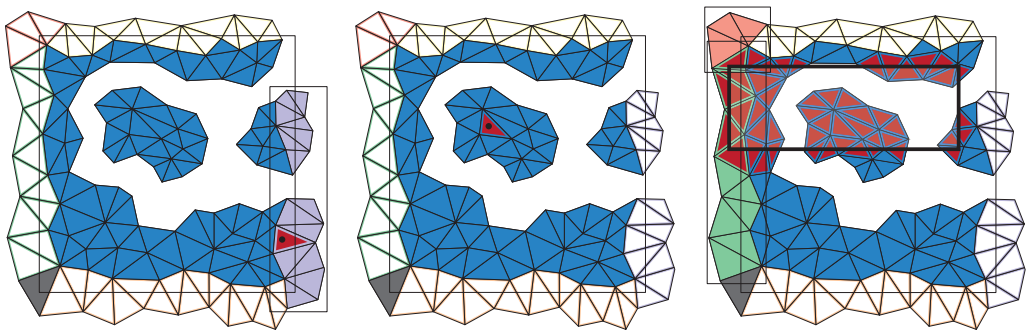
5.1 Spatial queries

The two most common spatial queries on a tetrahedral mesh Σ are:

Point location query. Given a query point p, find the tetrahedron σ in Σ containing p. In case several tetrahedra contain p (i.e., p lies on a shared vertex, edge, or triangular face), just one such tetrahedra is reported.

Range query. Given an axis aligned query box ρ, find the (possibly empty) set of all tetrahedra in Σ that have a non-empty intersection with ρ.

In all variants of the Tetrahedral tree, our querying algorithms perform a top-down tree traversal to locate the leaf block containing query point p, or the leaf blocks intersecting query box ρ. Each target leaf block indexes a set of tetrahedra which we check against point p, or box ρ, to decide whether it must be reported in the answer. Since these fine-grained geometric tests consume the bulk of our query response time, we attempt to reduce the number of these tests by exploiting the spatial locality of the index runs within our SRE index lists. Specifically, for each run of consecutive tetrahedra in the index list, we first compute the bounding box of its vertices and test this against the query geometry before applying the more expensive geometric tests on the tetrahedra (see



(a) Point query 1 (b) Point query 2 (c) Range query

Fig. 4. *Run-aware* point location and range queries on an SRE compressed block (shown in 2D with black circular query points in (a) and (b) and rectangular query range in (c)). Geometry in runs whose bounding boxes do not intersect the query point or box (hollow triangles) can be immediately discarded. The remaining candidates (filled triangles) require further bounding box and query-dependent geometric tests to find the desired triangles (dark red filled triangles).

Algorithm 1 POINT_LOCATION(b,p)

Input: b is a leaf block of the tree, and b_T is the tetrahedra array indexed by b
Input: p is a query point

```
1: for each entry r in b_T do
2:   if r is a run then
3:     extract the bounding box b_box of r
4:     if b_box contains p then
5:       for each tetrahedron σ in r do
6:         if σ contains p then
7:           return σ
8:   else if r contains p then // r is a tetrahedron reference
9:     return r
```

Figure 3(b)). As we demonstrate in Section 7, this optimization yields a significant (query-dependent) speedup in execution times.

Algorithm 1 provides a pseudo-code description of our *run-aware* point location algorithm within a leaf block b containing the query point. If the current entry is a run r of tetrahedra, we compute its bounding box b_{box} (row 3) and test it against the query point p (row 4). If p intersects b_{box}, then we test the individual tetrahedra belonging to the run (rows 5–7), otherwise we proceed with the next entry of the array. Figure 4(a) and (b) present examples of our run-aware *point location* query. For simplicity, we use a 2D example, in which we consider a triangle instead of a tetrahedral mesh. As we iterate through the index array, we compute the bounding box of each run (see Figure 3(b)) and discard triangles in runs whose bounding box does not intersect the query point (hollow triangles in Figure 4(a) and (b)). We then execute point-in-triangle tests only on those triangles belonging to the runs whose bounding box contains p and on those which do not belong to any run (filled triangles in Figure 4(a) and (b)).

Algorithm 2 RANGE_QUERY(b,ρ,result)

Input: b is a leaf block in the hierarchy, with b_T the tetrahedra array indexed by b
Input: ρ is a query box
Input: *result* is the result list containing the tetrahedra intersecting ρ

```
1:  if b is completely contained in ρ then
    // simply add the tetrahedra to the result
2:    for each tetrahedron σ in b_T do
3:      add σ to result
4:  else
5:    for each entry r in b_T do
6:      if r is a run then
7:        extract the bounding box b_box of r
8:        if b_box is completely contained in ρ then
          // expand the run by adding the tetrahedra to the result
9:          for each tetrahedron σ in r do
10:           add σ to result
11:       else if b_box intersects ρ then   // expand the run
12:         for each tetrahedron σ in r do
13:           if σ intersects ρ then
14:             add σ to result
15:     else if r intersects ρ then   // r is a tetrahedron reference
16:       add r to result
```

Algorithm 2 provides pseudo-code description of our *run-aware* range query within a leaf block b. Before testing any of the tetrahedra in b against the query box ρ, we check if b is completely contained in ρ, thus executing just one *box-in-box* test (row 1). If so, we directly add all of its tetrahedra to the output, without executing any *tetrahedron-in-box* tests. Otherwise, for each run r, we test if r is completely contained in ρ (row 8), or if ρ and r's bounding box intersect (row 11). In the first case, we directly add all the tetrahedra in the run to the output list without any additional tests (rows 9-10). In the second case, we test each tetrahedron in the run (rows 12–14). If the current element is not a run, we execute a *tetrahedron-in-box* test on it (rows 15–16). Figure 4(c) shows an example range query on the block from Figure 3.

5.2 Topological queries

In Tetrahedral trees, boundary relation *Tetrahedron-Vertex (TV)* is stored globally for each tetrahedron in the mesh as part of the underlying indexed mesh representation. Coboundary and adjacency relations can be generated on the fly either for the whole domain, or inside a region of interest. We describe here, for brevity, only the algorithms for answering topological queries in a region of interest, which are formulated as follows:

Range topological query. Given a query box ρ, compute the desired topological relations for the (possibly empty) set of simplices from the mesh that intersect ρ.

The algorithm for identifying the portion of the index and thus the tetrahedra involved in the query is very similar to the one for the range query (as described in Section 5.1). As examples, we will focus on the *range Vertex-Tetrahedron (VT)* query and the *range Tetrahedron-Tetrahedron (TT)* query. For efficiency, in both queries we use the run-aware optimizations described in Section 5.1.

Algorithm 3 EXTRACT_VT(b,ρ,$VTlist$)

Input: b is a leaf block in the hierarchy intersecting ρ, with b_T the tetrahedra array of b
Input: ρ is a query box
Input: $VTlist$ is the list of all VT(v), for v in ρ
1: **for each** tetrahedron σ in b_T **do**
2: **for each** vertex v in TV(σ) **do**
3: **if** v in b **and** ρ contains v **then**
4: add σ to VT(v) in $VTlist$

A *range Vertex-Tetrahedron (VT)* query returns, for each vertex in the specified range, the tetrahedra incident in it. Algorithm 3 provides a pseudo-code description of the algorithm for performing the query. The algorithm considers a leaf block b and extracts the vertices of the boundary of each tetrahedron σ in b (using *Tetrahedron-Vertex (TV)* relation). If a vertex v is indexed by b and is contained in box ρ, the index of σ in T is added to the list of tetrahedra in the VT list of v.

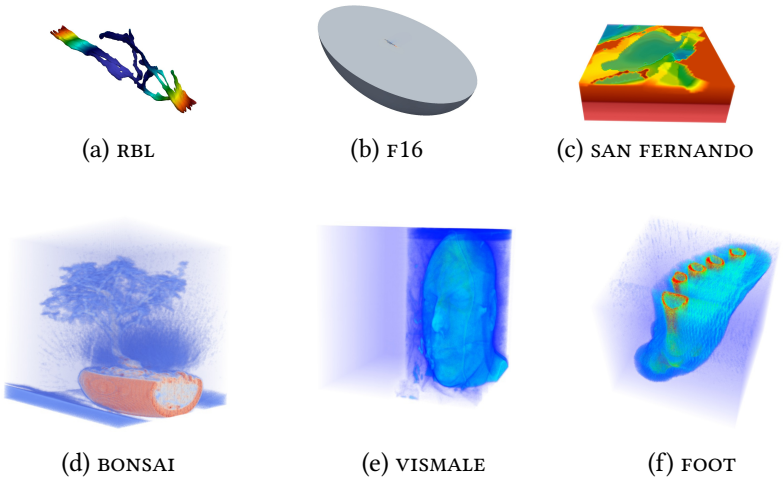
A *range Tetrahedron-Tetrahedron (TT)* query returns, for each tetrahedron σ in the given range, the tetrahedra sharing a face with σ. The algorithm iterates over the leaf blocks intersecting query box ρ. For each such leaf block b, and for each tetrahedron σ in b that intersects ρ, it first extracts the faces of σ from TV(σ) as triples of vertex indices. For each face f of σ, an entry is inserted in a list L consisting of the three vertices of f plus the index of σ in the global tetrahedron array. Lexicographically sorting list L pairs the tetrahedra that have a face in common. The TT relation is then built by iterating over L: for every pair of consecutive entries in L having the same triple of vertices, the two tetrahedra σ_1 and σ_2 in the two entries are marked as face-adjacent.

6 EVALUATION OF STORAGE COSTS AND GENERATION TIMES

In this section, we present an experimental evaluation of the storage cost and of the generation time of the different Tetrahedral trees over a testbed of datasets constructed using different threshold parameters k_V and k_T for the number of vertices and tetrahedra allowed in a leaf block of the tree. Our evaluation considers limit cases for the thresholds (i.e., 1 or ∞) as well as a statistic t_{avg} representing the average number of leaf blocks in which a tetrahedron is indexed. This is an important indicator of the quality of the spatial decomposition, as it highlights if the decomposition is either too coarse (i.e., each tetrahedron appears in very few leaf blocks) or too fine (i.e., each tetrahedron intersects a large number of leaf blocks). We also compare Tetrahedral trees against representatives of the two most commonly used data structures for spatial queries: the IA topological data structure [65] and the R*-tree [7]. To evaluate the storage and computational benefits of the run-based leaf block encoding, we compare Tetrahedral trees with this encoding against an uncompressed variant that does not compress the vertex and tetrahedron arrays in the leaf blocks. This is equivalent to the data structure in [25].

We have performed our experiments on six unstructured tetrahedral meshes ranging in size from 4 to 30 million tetrahedra (see Figure 5), originating from biomedical and engineering applications. The BONSAI, VISMALE and FOOT datasets were derived from a regular grid using *regular simplex bisection* [92], leading to adaptive semi-regular tetrahedral meshes. They were then made irregular through a simplification process, based on half-edge collapse, that removed approximately 15% of the vertices. The remaining meshes (RBL, F16, SAN FERNANDO) are originally irregular meshes.

For every mesh, we have built sixteen Tetrahedral trees (see Table 2). For each of the four subdivision rules, we have generated two spatial indexes based on octrees, and two spatial indexes



(a) RBL (b) F16 (c) SAN FERNANDO

(d) BONSAI (e) VISMALE (f) FOOT

Fig. 5. Tetrahedral meshes used in the experiments.

based on kD-trees, by using two different values of the relevant thresholds, k_V and/or k_T depending on the specific subdivision rules. The two thresholds have been selected in order to obtain similar values of t_{avg} (see t_{avg} column in Table 2) for the octree and the kD-tree subdivision rules, thus allowing a direct comparison between them. We have chosen the values in such a way that the tree built with the larger threshold contains roughly half as many blocks of the other tree. With the values we used for k_T, the storage requirements of RT-TTREES and T-TTREES are the same and, thus, Table 2 shows only those of T-TTREES. The hardware configuration used for these experiments is an Intel CPU i7-3930K, at 3.2 Gigahertz, and with 64 gigabytes of RAM.

6.1 Experimental evaluation of Tetrahedral trees

We begin by comparing the storage requirements for the different Tetrahedral tree variants. In Table 3, the columns labeled *tree* show the storage cost of the spatial index, while those labeled *tot* also take into account the storage of the underlying tetrahedral mesh.

We first compare the SRE-compressed Tetrahedral trees against their corresponding uncompressed Tetrahedral trees. Thanks to the spatial reorganization and *SRE* compression, the total size of the tetrahedron arrays in the leaf blocks is an order of magnitude smaller than its uncompressed counterparts. In comparison to uncompressed trees, Tetrahedral trees save, on average, 90% of the storage for encoding the spatial index. This translates to a 30% savings in overall storage requirements. The remainder of this section will focus on the SRE-compressed Tetrahedral trees. We will return to the uncompressed trees in our analysis of query performance in Section 7.

Comparing the storage overhead of the Tetrahedral trees spatial index (in column *tree*) against that of the indexed mesh (in column *Base Mesh*) further highlights the benefits of our run-based SRE encoding. Even though each tetrahedron is indexed by approximately two leaf blocks (see t_{avg} in Table 2), the overhead of the spatial index component of Tetrahedral trees remains quite low. Specifically, Tetrahedral trees built using the smaller threshold values have only about 5-10% overhead while those with built with larger thresholds have about half of that, i.e., 3-5% overhead.

Next, comparing the different tree topologies, we observe that, in general, octrees tend to be more compact for higher thresholds, while kD-trees are more compact for smaller ones, with differences ranging from around 5% to 35%. This trend is not only correlated to the bucketing threshold and to

Table 2. Statistics about the experimental datasets and Tetrahedral trees built on them. $|b|$ is the total number of blocks, while t_{avg} represents the average number of leaf blocks in which a tetrahedron is indexed. For the chosen parameters, RT-TTrees and T-TTrees are equivalent, and thus we report only the statistics of T-TTrees.

Data	Σ			k_V	k_T	indexes					
						P-TTree		PT-TTree		T-TTree	
	$\|V\|$	$\|T\|$				$\|b\|$	t_{avg}	$\|b\|$	t_{avg}	$\|b\|$	t_{avg}
RBL	730K	3.89M	octree	100	1000	32.6K	2.15	37.6K	2.25	34.9K	2.22
				200	1200	16.8K	1.86	22.9K	1.99	22.9K	1.99
			kD-tree	50	500	50.1K	2.15	57.5K	2.22	55.1K	2.20
				100	700	26.2K	1.90	36.9K	2.04	36.7K	2.04
F16	1.12M	6.35M	octree	200	1200	24.0K	2.02	38.6K	2.24	38.5K	2.24
				400	1600	12.8K	1.79	28.3K	2.10	28.3K	2.10
			kD-tree	100	600	35.2K	2.00	68.3K	2.29	68.3K	2.29
				150	800	23.7K	1.86	48.7K	2.14	48.7K	2.14
SAN FERN	2.46M	14.0M	octree	125	1200	78.3K	2.08	87.4K	2.14	63.9K	2.02
				300	1800	34.9K	1.85	35.6K	1.85	35.6K	1.85
			kD-tree	50	800	141K	2.11	142K	2.11	101K	2.00
				125	1200	653K	1.84	68.1K	1.86	58.5K	1.81
BONSAI	4.25M	24.4M	octree	55	550	252K	2.07	368K	2.23	360K	2.22
				150	800	115K	1.86	142K	1.92	142K	1.92
			kD-tree	50	300	263K	1.99	535K	2.29	535K	2.29
				75	600	173K	1.85	211K	1.91	211K	1.91
VISMALE	4.65M	26.5M	octree	50	500	318K	2.14	418K	2.26	416K	2.25
				200	800	124K	1.88	141K	1.92	141K	1.92
			kD-tree	30	300	429K	2.17	553K	2.28	552K	2.28
				75	600	187K	1.86	231K	1.92	231K	1.92
FOOT	5.02M	29.5M	octree	55	600	339K	2.11	426K	2.21	375K	2.15
				115	1000	125K	1.86	128K	1.86	127K	1.86
			kD-tree	35	350	388K	2.11	482K	2.18	480K	2.18
				75	700	192K	1.84	195K	1.85	194K	1.84

the corresponding spatial decomposition, but also, and more interestingly, to the effectiveness of the compression of sequential index runs.

Finally, comparing the different tree variants, P-TTrees, PT-TTrees and T-TTrees, we observe that, for similar values of t_{avg}, the octree-based P-TTree is generally the most compact. For example, P-TTrees require half the storage on the BONSAI and F16 datasets as the others. However, for the SAN FERNANDO dataset, T-TTrees are more compact, requiring about 5% to 50% less storage.

In summary, we observe that the SRE compression allows an order of magnitude reduction compared to uncompressed trees. On SRE-based trees, the use of either an octree or a kD-tree does not influence the storage cost significantly. Even through P-TTrees are often the most compact, the differences in storage cost among the four indexes are quite small.

Considering generation times (see Table 4), we observe that creating the initial spatial decomposition accounts for up to 90% of the overall generation times. There is little difference among the various subdivision strategies (see column *tree*); P-T and RT-TTrees are about 5% faster than T-T and PT-TTrees due to their simpler subdivision rule. Conversely, the compression stage for a

Table 3. Storage costs, expressed in *megabytes* (*MBs*), for Tetrahedral Trees (based on the thresholds shown in Table 2), IA data structure and R*-trees. Column *Base mesh* shows the tetrahedral mesh storage, column *conn.* shows the storage required to encode the connectivity for IA, column *tree* shows the storage to encode the index for Tetrahedral Trees and R*-trees. *uncompr.* and *SRE* columns show the storage for uncompressed and SRE-compressed Tetrahedral trees. For R*-trees, we also show the branching factor (b_F column).

Data	Base mesh	Tetrahedral trees									IA		R*-tree		
		index	octree				kD-tree								
			uncompr.		SRE		uncompr.		SRE						
			tree	tot	tree	tot	tree	tot	tree	tot	conn.	tot	b_F	tree	tot
RBL	76.0	P-Ttree	36.2	112	5.19	81.2	37.0	113	5.97	82.0	62.1	138	4	329	405
			31.1	107	2.46	78.5	32.2	108	3.13	79.2					
		PT-Ttree	37.9	114	6.21	82.2	38.3	114	6.95	83.0			8	91.9	168
			33.3	109	3.58	79.6	34.7	111	4.53	80.5					
		T-Ttree	34.1	110	5.49	81.5	34.6	111	6.13	82.1			16	46.7	123
			30.2	106	3.32	79.3	31.5	107	4.10	80.1					
F16	123	P-Ttree	54.3	177	5.68	128	54.3	177	5.74	128	101	224	8	137	259
			48.3	171	3.31	126	50.3	173	4.05	127					
		PT-Ttree	60.4	183	8.70	131	62.9	185	10.4	133			16	70.7	193
			56.4	179	6.64	129	58.2	181	7.75	130					
		T-Ttree	55.6	178	8.25	131	57.8	180	9.66	132			32	45.5	168
			51.8	174	6.32	129	53.4	176	7.19	130					
SAN FERN	270	P-Ttree	124	394	12.3	282	128	398	16.3	286	223	492	8	318	588
			109	379	5.81	275	111	380	7.68	277					
		PT-Ttree	127	397	13.6	283	129	398	16.4	286			16	186	456
			110	380	5.93	276	112	381	8.04	278					
		T-Ttree	110	380	9.63	279	110	380	10.7	280			32	112	381
			100	370	5.52	275	98	368	6.41	276					
BONSAI	470	P-Ttree	221	691	34.7	505	214	684	27.5	498	389	859	16	288	758
			195	665	16.3	486	197	667	18.2	488					
		PT-Ttree	241	711	50.3	520	254	724	55.0	525			32	183	653
			201	671	19.9	490	204	674	22.2	492					
		T-Ttree	219	689	45.3	515	231	701	48.9	519			64	136	606
			183	654	18.2	488	185	655	19.8	490					
VISMALE	511	P-Ttree	249	760	43.8	555	257	769	44.4	556	423	934	16	312	823
			214	725	17.7	529	215	726	19.9	531					
		PT-Ttree	265	777	57.1	568	274	785	57.1	568			32	198	710
			219	730	19.9	531	223	734	24.3	536					
		T-Ttree	242	754	52.1	563	250	761	50.7	562			64	147	659
			199	711	18.3	530	203	714	21.6	533					
FOOT	565	P-Ttree	272	838	47.3	613	274	839	40.2	605	470	1035	32	221	786
			234	799	18.0	583	235	801	20.3	586					
		PT-Ttree	288	853	59.1	624	287	852	50.4	616			64	164	729
			235	800	18.4	584	236	801	20.6	586					
		T-Ttree	255	821	48.2	613	262	827	44.8	610			128	138	703
			214	779	16.8	582	214	780	18.3	584					

P-T index is the fastest, requiring 80% less time than than T-T and RT-T indexes and 10% less time than PT-T indexes. The compression is slower for T-T and RT-Ttrees since they must reconstruct their vertex indices (see Section 4.3). We next note that kD-trees generate the uncompressed spatial index about twice as fast as their octree counterparts, while compressing the spatial index requires about the same time for the two decompositions.

Analysis of limit cases. We will now shift our focus to the limit cases for our spatial indexes, when the threshold values are set to a minimum value, i.e., $k_V = 1$ and $k_T = 1$. The dual extrema

Table 4. Generation timings, expressed in *seconds*, for Tetrahedral Trees (based on the thresholds shown in Table 2) and for R*-trees. Column *tree* shows the time required for generating the Tetrahedral Tree and R*-tree spatial decompositions, while *compr.* shows the time required for compressing the spatial decomposition for Tetrahedral trees. Column *tot* shows the overall generation timings. For R*-trees, we also show the branching factor (b_F column).

Data	Tetrahedral trees							R*-tree	
	index	octree			kD-tree				
		tree	compr.	tot	tree	compr.	tot	b_F	tree
RBL	P-Tree	58.5	1.48	60.0	24.3	1.51	25.8	4	97.7
		52.9	1.17	54.1	21.9	1.21	23.1		
	PT-Tree	60.5	1.59	62.1	26.9	1.61	28.5	8	51.0
		56.6	1.31	57.9	25.4	1.37	26.8		
	T-Tree	61.1	2.67	63.8	26.8	2.65	29.5		
		55.0	2.33	57.4	24.6	2.37	27.0		
	RT-Tree	55.5	3.22	58.7	24.4	2.66	27.1	16	63.4
		52.0	2.38	54.4	22.5	2.38	24.8		
F16	P-Tree	141	7.42	149	57.2	7.27	64.5	8	94.6
		133	5.81	139	54.6	6.24	60.8		
	PT-Tree	150	8.87	159	69.2	9.32	78.6	16	115
		146	7.95	153	64.8	8.31	73.1		
	T-Tree	148	12.1	160	67.4	12.4	79.7		
		144	11.1	156	64.3	11.3	75.6		
	RT-Tree	143	12.0	155	63.9	12.4	76.3	32	182
		139	11.0	150	60.9	11.3	72.2		
SAN FERN	P-Tree	187	5.00	192	77.2	5.30	82.5	8	177
		173	4.09	177	70.5	4.17	74.7		
	PT-Tree	193	5.17	198	77.7	5.41	83.1	16	206
		178	4.11	182	79.8	4.31	84.1		
	T-Tree	189	8.29	197	78.2	8.24	86.5		
		172	7.46	180	71.4	7.30	78.7		
	RT-Tree	174	8.54	182	71.9	8.24	80.1	32	328
		164	7.64	172	65.8	7.33	73.2		
BONSAI	P-Tree	353	9.78	363	134	8.70	142	16	320
		330	7.74	338	127	7.59	135		
	PT-Tree	375	11.3	387	170	11.3	181	32	509
		340	8.09	348	180	8.22	188		
	T-Tree	370	17.5	387	158	17.6	176		
		335	14.3	349	138	14.1	152		
	RT-Tree	348	18.3	366	145	17.7	163	64	813
		321	14.7	335	129	14.1	143		
VISMALE	P-Tree	396	11.1	407	155	11.1	166	16	351
		365	8.44	373	141	8.45	149		
	PT-Tree	414	12.3	427	166	12.3	178	32	555
		380	8.78	389	190	9.14	200		
	T-Tree	409	19.4	428	172	19.3	192		
		368	15.5	384	153	15.6	168		
	RT-Tree	394	20.4	414	159	19.4	179	64	898
		361	16.0	377	142	15.6	157		
FOOT	P-Tree	440	11.9	452	171	11.2	183	32	624
		403	8.91	412	158	8.86	167		
	PT-Tree	457	13.3	471	189	12.2	201	64	990
		407	8.95	416	158	8.99	167		
	T-Tree	452	20.0	472	185	19.5	205		
		402	16.1	418	165	15.9	181		
	RT-Tree	419	21.0	440	174	19.5	193	128	1664
		384	16.6	401	153	15.9	169		

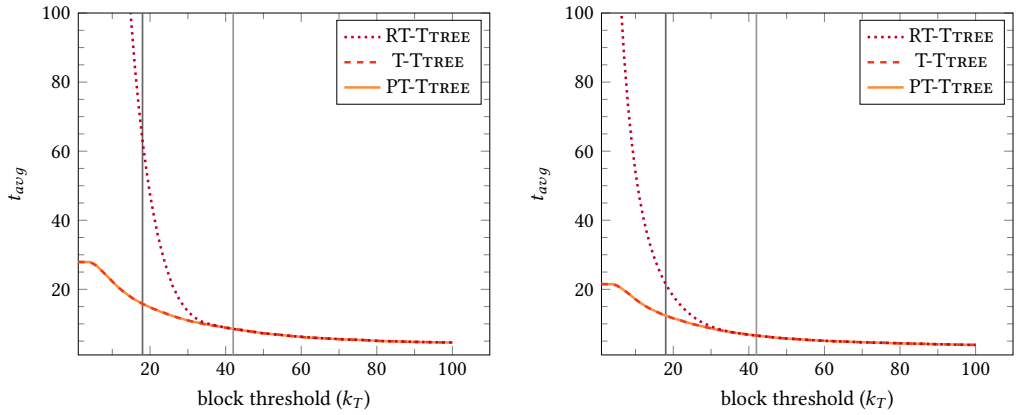


Fig. 6. Average references per tetrahedron (t_{avg}) vs. block threshold for octree-based (a), and kD-tree-based (b) Tetrahedral tree representations of the SPX dataset. The vertical lines highlight the average (dark gray) and maximum (light gray) cardinality of the VT relation. For PT-TTREES, we set $k_V = k_T$.

values, when k_V or k_T are *infinite* (i.e., a value greater than the number of vertices or tetrahedra respectively), are not considered further since this results in just a single block indexing the whole domain. With SRE-encoding, this is essentially equivalent to having just the indexed mesh data structure.

When $k_V = 1$ and/or $k_T = 1$, the decomposition is a dense subdivision of the embedding space and the tree storage requirements become very high, as each tetrahedron appears in a large number of leaf blocks. These cases have a similar decomposition strategy as the PM-octrees which were defined for polygonal meshes bounding a surface rather than tetrahedral meshes [15, 60]. For P-TTREES with $k_V = 1$ on the RBL, F16 and SAN FERNANDO datasets, the number of blocks are approximately twice to three times the number of vertices, and t_{avg} grows to approximately 10 for octrees, and 7 for kD-trees. Even using SRE-compression, the storage overhead for our leaf blocks grows to around 50% and 75% of the storage cost of the indexed mesh.

We tried generating indexes with $k_T = 1$ using RT-T, PT-T and T-T subdivisions on our testbed meshes, but the subdivision of embedding space became too fine, and we ran out of memory. Thus, we have generated them on two smaller datasets, SPX (2.9K vertices and 13K tetrahedra) and FIGHTER2 (257K vertices and 1.4M tetrahedra). Even on these two smaller datasets, with $k_T = 1$, an RT-T tree could not be generated, but we were able to generate PT-TTREES and T-TTREES. For SPX, the number of blocks is 5.5 times the number of tetrahedra, while for FIGHTER2, the number of blocks is from 3.8 to 5 times the number of tetrahedra. For SPX, t_{avg} is 28 on the octree and 21.5 on the kD-tree, while for FIGHTER2, t_{avg} is 25 on the octree and 16 on the kD-tree.

This result implies that the approach with $k_T = 1$, proposed for triangle meshes in [24], is not viable in the 3D case. The motivation is that, while the average number of triangles incident in a vertex in a triangle mesh is equal to 6, the average number of tetrahedra incident in a vertex in a tetrahedral mesh is around 23 [6], causing a much finer refinement of the embedding space.

Analyzing variations of t_{avg}. We now consider how t_{avg} varies with the values of k_V and k_T. To this aim, we have generated indexes with k_V and k_T ranging from 2 to 100 using RT-T, PT-T, and

T-T subdivisions, on our two smaller datasets, SPX and FIGHTER2. For a PT-T subdivision, which uses both k_V and k_T thresholds, we set $k_V = k_T$.

Our analysis makes use of the average and maximum number of tetrahedra in the star of a vertex of our mesh Σ. We denote the latter as $|VT| = max\{|VT(v)| \text{ for each vertex } v \text{ in } \Sigma\}$. When k_T is greater than $|VT|$, PT-TTREES and T-TTREES generate the same spatial decomposition and, thus, below this value, only the k_V threshold, used by PT-TTREES, is relevant for determining the final tree shape, thereby generating deeper trees. RT-TTREES present degenerate (unnecessary) decompositions for lower threshold values. As shown in Figure 6, the RT-T subdivision with thresholds from 1 to 20 for SPX dataset exhibits two main behaviors: it either goes out-of-memory (i.e., it leads to a nearly infinite subdivision), or it generates a very deep subdivision (with values of t_{avg} larger than 100).

This result demonstrates that the RT-T subdivision is inefficient for k_T values smaller than $|VT|$. In this case, we subdivide the space without any benefit, as in configurations like the one shown in Figure 2. We also observe from Figure 6 that, when k_T is larger than $|VT|$, RT-TTREES, PT-TTREES and T-TTREES subdivide the embedding space in the same way. Thus, the subdivision seems to depend only on the threshold and not on the subdivision rule. These trends have also been observed on other datasets, such as FIGHTER2.

6.2 Comparison against R*-trees and the IA data structure

We conclude this section with a comparison between Tetrahedral trees and representatives of the two most widely used data structures for spatial queries: the IA data structure [65] and the R*-tree [7].

Recall that the IA data structure encodes the *Tetrahedron-Vertex (TV)* relation, as in the global tetrahedron array of the Tetrahedral trees, as well as the *Tetrahedron-Tetrahedron (TT)* and partial *Vertex-Tetrahedron (VT)* relations. As can be seen in the *tree* and *conn*. columns of Table 3, the spatial index underlying a Tetrahedral tree requires about 95% less storage than the connectivity information stored in the IA.

We have implemented a 3D R*-tree indexing data structure for tetrahedral meshes, starting from an open source 2D implementation [2]. For each dataset, we have built three R*-tree indexes, with different branching factors, starting from the optimal ones defined in [7], and we have calibrated these with respect to our tetrahedral meshes. In Table 3, we show the branching factors (under b_F column). Each internal block of the R*-tree contains: (i) a pointer to its parent block, (ii) an array of children blocks, and (iii) the minimum and maximum corner points of the bounding box. Each leaf block stores just a pointer to an integer array for the indexed tetrahedra. Each tetrahedron is indexed in a single leaf block.

Before comparing Tetrahedral trees and R*-trees, we describe how we have selected the branching factors in R*-trees, as they appear remarkably lower that the leaf blocks thresholds used in Tetrahedral trees. In [7], the authors state that the *optimal* branching factor obtained in their experiments was 8, and thus we have started our calibration from this value, choosing powers of two for the other branchings and balancing storage costs against query performance (which we discuss in Section 7). With smaller branching factors, the R*-tree index requires from 150 to 440 MB, depending on the specific mesh (see column *tree*). For larger branching factors, the storage required by the R*-tree index decreases to between 45 and 150 MB. R*-trees always require more memory than Tetrahedral trees: from 85% more, for *larger* branching factors, to 93% more, for *smaller* ones.

In spite of this trend, R*-trees exhibit significantly higher computational overhead when executing spatial queries (see Section 7). This is due to the larger number of leaf blocks (and their indexed tetrahedra) that need to be checked to satisfy spatial queries, as higher branching factors produce

larger overlaps of the bounding boxes. Thus, for example, while performing point locations, R*-trees have to visit several branches of the tree, while decompositions based on octrees and kD-trees only need to visit a single subtree in each level of the index.

The *total* storage requirements account, in the case of Tetrahedral trees and R*-trees, for the indexed mesh plus the spatial index, while, in case of the IA data structure, for the indexed mesh plus the adjacencies, Tetrahedral trees are always more compact: on average 40% smaller than the IA data structure and from 25% to 50% more compact than R*-trees, depending on the branching factor.

Comparing the generation times of Tetrahedral trees and R*-trees (see Table 4), we note that larger R*-tree branching factors lead to larger differences in generation times. For small branching factors, R*-trees have generation timings similar to those of Tetrahedral octrees, and, thus, twice the generation times for Tetrahedral kD-trees. This gap increases as the the branching factor increases, where for the larger branchings, R*-trees use from 40% to 75% more time than Tetrahedral trees.

7 QUERY EVALUATION

In this section, we analyze our experimental performance results for spatial and topological queries on Tetrahedral trees using the meshes and thresholds described in Section 6 along with the *run-aware* algorithmic optimizations described in Sections 5.1 and 5.2. For our experimental comparison, we have also developed an implementation of these queries on the IA data structure [51], and on the R*-tree [7], and we also compare against uncompressed tetrahedral trees.

Queries on the IA data structure. To execute spatial queries on the indexed mesh data structure, the best we can do is to sequentially tests all the tetrahedra in Σ, leading to a complexity which is linear in the number of tetrahedra in the mesh. Several strategies have been proposed in the literature to optimize spatial queries when we have additional connectivity information [23, 28, 29, 59]. We have implemented the *stochastic walk* approach for point location on the IA data structure, as this strategy has been shown to have the best performances [28]. This approach randomly picks a starting tetrahedron σ, and then walks through adjacencies from σ to a target point. This target point is the input point p, in the case of a point location, or a point on the boundary of the range, in the case of a range query.

For point location queries, we return the tetrahedron σ', if it can be found, or an empty output. In the case of a range query, once we find the first tetrahedron σ' that contains the target point, we start a traversal through mesh adjacencies to get all the intersecting tetrahedra. We have adapted the range query algorithm to execute *range Vertex-Tetrahedron (VT)* queries. Given a vertex v, in order to extract the local VT relation, we start from the tetrahedron σ encoded in the partial VT^* relation of v. We then identify all the tetrahedra incident in v by using the *Tetrahedron-Tetrahedron (TT)* relation.

Note that the stochastic walk assumes a convex domain. If a concave area or a hole is found during the execution of a query, the stochastic walk algorithm may not return a complete answer to the query, i.e., it may return a subset of the tetrahedra satisfying the query. For example, while Tetrahedral trees and R*-trees have no trouble responding to the point query in Figure 4(b), query algorithms on an IA would have to revert to linear search to successfully respond to this query. In order to keep track of this behavior, we compute a statistic, that we call the *hit ratio*, as the percentage of fully answered queries. A hit ratio of 100% means that all queries have been answered correctly.

To improve the efficiency of our IA data structure implementation, we have also added a dynamic bit vector, using the *Boost C++* library [83], which greatly reduces the number of geometric tests executed in a single query. The cost of this speed-up is a run-time storage overhead of one bit for

each tested tetrahedron. As a further optimization, we initialize the stochastic walk by randomly picking 100 tetrahedra, and then starting the walk from the one nearest to the target point or range.

Queries on the R-tree.* Spatial queries for an R*-tree begin with a top-down tree traversal to locate the leaf blocks containing the query point or intersecting the query range. For each such leaf block b, we apply the appropriate geometric tests on all tetrahedra indexed by b. As each tetrahedron is indexed in only one leaf block, R*-trees avoids duplicate geometric tests on the same tetrahedron, but blocks in multiple branches of the tree must be tested, since R*-tree blocks can overlap.

The overlapping blocks within an R*-tree's structure also tend to reduce the efficiency of range topological queries since spatially close tetrahedra can be arbitrarily far apart in the R*-tree's index space. Thus, to respond to range topological queries, we must first execute a range query and then post-process the results to extract the desired topological relations. This can be acceptable for smaller ranges, but for larger ranges, the indexing structure no longer helps to reduce the query times, leading to time and storage requirements entirely similar to a brute-force strategy on the entire mesh. In contrast, the range topological queries for Tetrahedral trees incrementally build up the topological relations as each leaf block is being processed.

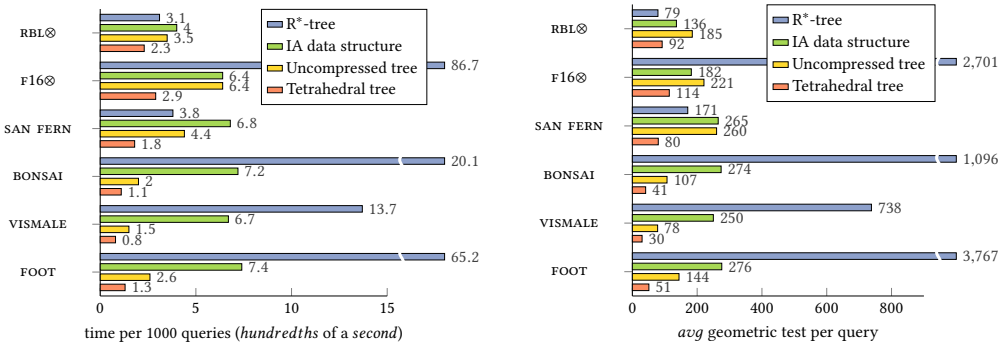
Experimental setup. In our performance analysis of the spatial and topological queries, we first compare the relative efficiency among the various Tetrahedral trees. We then compare the best representative among Tetrahedral trees, the IA data structure, and R*-trees. The charts in Figures 7, 8 and 9 compare the spatial query performance among the IA data structure, the R*-tree, the uncompressed and SRE-encoded Tetrahedral trees. For these comparisons, we plot the R*-trees with the middle branching factor (see Table 2) as these provide the best trade-off between storage and execution times. Similarly, for the Tetrahedral trees, we plot the PT-T Tetrahedral kD-trees constructed using the smaller k_v and k_t thresholds as they have the best overall query performances among the sixteen Tetrahedral trees. Similarly, Figures 10 and 11 compare performances on the topological queries for the IA data structure, and the same R*-trees and Tetrahedral trees. We provide the full query performance results for all Tetrahedral trees and R*-trees in Appendix A and summarize these results in our analysis.

7.1 Spatial queries

For point location queries, we used a set of 1000 randomly generated query points inside the bounding box of the mesh. Similarly, for range queries and range topological queries, we used two sets of 1000 randomly generated query ranges inside the bounding box of the mesh: a set of smaller ranges containing on average from 15 to 45 thousand tetrahedra, and a set of larger ranges containing from 300 to 700 thousand tetrahedra.

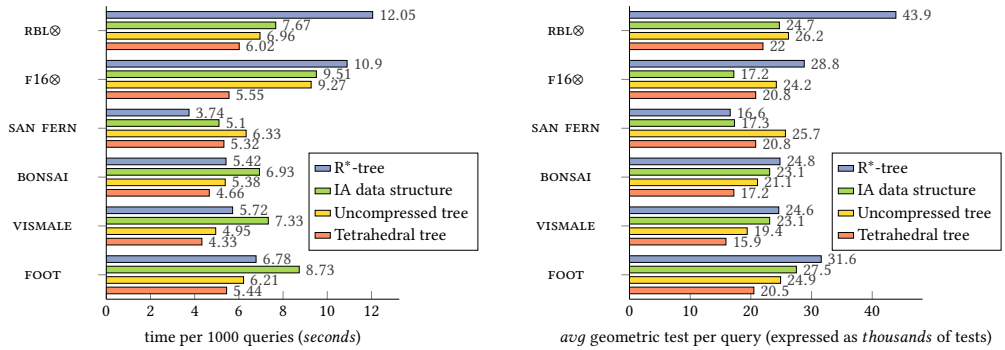
Point location. This query is extremely fast, as can be seen in Figure 7 (and Tables A.I, A.III and A.IV). Thanks to the run-aware optimization (described in Section 5.1), Tetrahedral trees perform 40%–80% fewer geometric tests than their uncompressed counterparts, leading to 30%–70% faster response times.

Compared to the IA data structure, Tetrahedral trees perform about 50%–90% fewer geometric tests on our semi-regular datasets, leading to query times that are 60%–90% faster. For the other datasets, the IA data structure gives only partial results, with hit ratios from 50% to 74%. If we estimate the full ratio, i.e., by normalizing the geometric tests and execution times on the IA data structure by multiplying them by the inverse of the hit ratio, we observe that Tetrahedral trees always perform better than the IA data structure on these irregular datasets with both thresholds.



(a) Point location query times (b) Avg. geometric tests per query

Fig. 7. Point location results for the R*-tree (based on the middle branching factor), the IA data structure and the uncompressed and SRE-compressed PT-T Tetrahedral kD-trees. The IA data structure has hit ratios of 50% and 74% for RBL and F16 datasets (marked with ⊗), respectively.

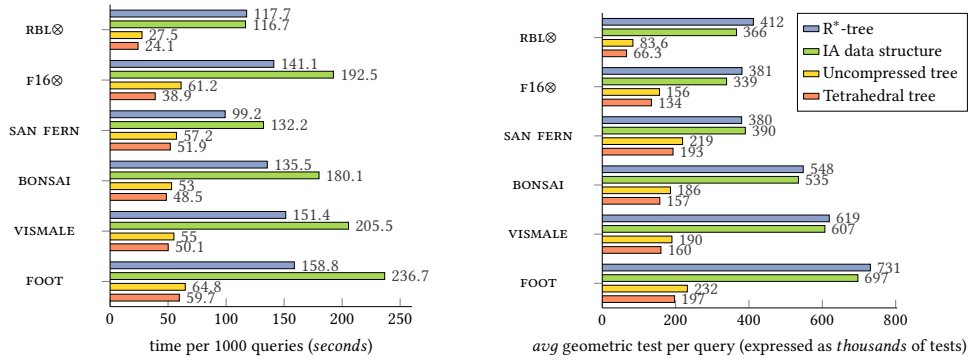


(a) Smaller range query times (b) Avg. geometric tests per query

Fig. 8. Smaller range query results for the R*-tree (based on the middle branching factor), the IA data structure and the uncompressed and SRE-compressed PT-T Tetrahedral kD-trees. The IA data structure has hit ratios of 43% and 72% for RBL and F16 datasets (marked with ⊗), respectively.

Point location query times for R*-trees tend to decrease with the branching factor, i.e., fewer tetrahedra in leaf blocks lead to faster executions. However, R*-trees with smaller branching factors require more storage (as discussed in Section 6.2). We observe that Tetrahedral trees perform better than the best R*-tree, requiring, on average, 60% less time and executing 70% fewer geometric tests. Execution times of Tetrahedral trees are also more stable across the input meshes, as they vary from 0.01 to 0.05 seconds, while execution times of the R*-tree with the smallest branching factor (see Table A.III) range from 0.01 to 0.90 seconds, depending on the mesh.

Range queries. For this query, we use a dynamic *bit vector* for Tetrahedral trees to track the tetrahedra that have already been tested. As with queries on the IA data structure, this adds a run-time storage overhead of one bit for each tested tetrahedron. As shown in Figures 8 and 9 (and in Appendix Tables A.I and A.IV), we observe that Tetrahedral trees always execute fewer



(a) Larger range query times (b) Avg. geometric tests per query

Fig. 9. Larger range query results for the R*-tree (based on the middle branching factor), the IA data structure and the uncompressed and SRE-compressed PT-T Tetrahedral kD-trees. The IA data structure has hit ratios of 51% and 92% for RBL and F16 datasets (marked with ⊗).

geometric tests, and are faster than uncompressed trees, with approximately a 10–20% time saving. Therefore, in the following, we will focus just on the SRE-compressed Tetrahedral trees. We did not observe any trends across datasets, subdivision criteria or box sizes to suggest a preference for octrees or kD-trees. However, trees built with smaller thresholds perform fewer tests than those generated with larger ones and, in general, PT-TTREES and T-TTREES perform better than P-TTREES on these queries.

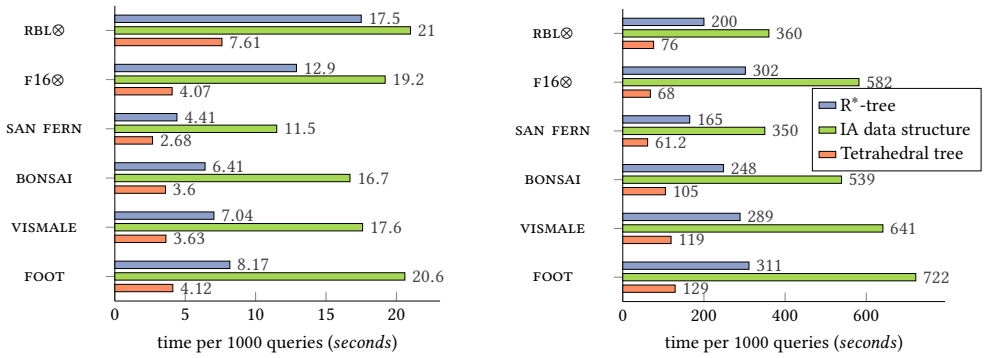
In our larger range queries (see Figure 9 and Table A.I), Tetrahedral trees always perform better than the IA data structure, achieving about a 55% improvement for SAN FERNANDO, and about 70% for the other datasets. Tetrahedral trees generally perform better than the IA data structure on smaller range queries as well, requiring from 30%–50% less time. However, they are a bit slower (about 5%–35%) on the SAN FERNANDO dataset, where the run-aware filtering appears to be less effective in filtering out candidate tetrahedra from more expensive tetrahedra-in-range tests. For the non-convex datasets, the IA data structure had hit ratios from 43% to 92%. By estimating the full ratio, we observe that Tetrahedral trees are always faster, requiring from 10%–50% of the time.

Comparing the performance of Tetrahedral trees against R*-trees (see Figures 8 and 9 and Tables A.I and A.III), we found that Tetrahedral trees are faster and execute fewer *tetrahedron-in-range* tests than R*-trees for range queries, with the exception of the SAN FERNANDO dataset on small ranges, as discussed above. In all other cases, Tetrahedral trees require on average from 15%–80% less time, and execute 20%–80% fewer geometric tests than R*-trees. The performances of R*-trees with other branching factors (shown in Table A.III) indicate a larger performance loss with respect to Tetrahedral trees as the branching factor increases.

7.2 Topological queries

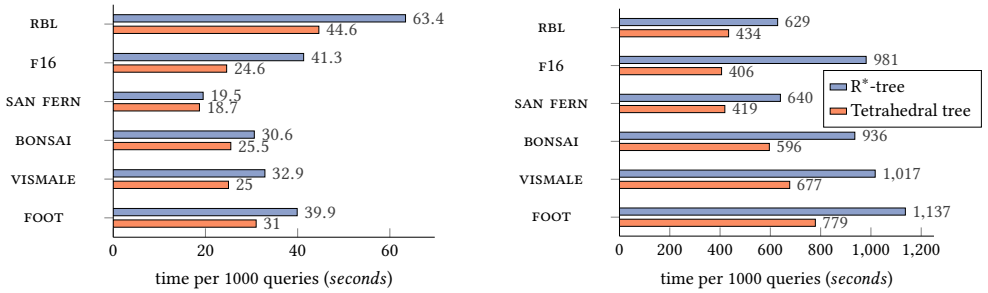
In this subsection, we analyze performance results from executing range-based topological queries over SRE-compressed Tetrahedral trees, the IA data structure and R*-trees.

Range Vertex-Tetrahedron (VT) queries. We use the same ranges as in the range queries and only report timings since the statistics (i.e., the number of *tetrahedron-in-range* executed) are the same as for range queries. P-TTREES and PT-TTREES are more suitable than RT-TTREES and T-TTREES for these queries since they are *vertex-based*, while P-TTREES and PT-TTREES explicitly encode the set



(a) Range VT query (smaller ranges) (b) Range VT query (larger ranges)

Fig. 10. Range Vertex-Tetrahedron (VT) query results for the R*-tree (based on the middle branching factor), the IA data structure and the SRE-compressed PT-T Tetrahedral kD-tree. The IA data structure cannot fully answer the query on some datasets (marked with ⊗).



(a) Range TT query (smaller ranges) (b) Range TT query (larger ranges)

Fig. 11. Range Tetrahedron-Tetrahedron (TT) query results for the R*-tree (based on the middle branching factor), and SRE-compressed PT-T Tetrahedral kD-tree.

of vertices contained in the leaf blocks, RT-Ttrees and T-Ttrees need to reconstruct this during the query, i.e., by executing geometric *point-in-leaf* tests. This can be seen in Table A.II, where Tetrahedral trees based on RT-T and T-T subdivisions are 35%–60% slower compared to their P-T and PT-T counterparts. Thus, we compare only the performances of the Tetrahedral trees based on P-T and PT-T subdivisions against the IA data structure and R*-trees.

As it is evident from Figure 10, Tetrahedral trees always completely answer the query and do so faster than the IA data structure, requiring 70%–80% of the time. They are also faster than R*-trees, with a 35%–70% time savings on smaller ranges, and 60%–80% on larger ones.

Range Tetrahedron-Tetrahedron (TT) queries. For this query, we only compare Tetrahedral trees against R*-trees since the IA data structure explicitly encodes the *Tetrahedron-Tetrahedron* adjacency relation. Recall that range adjacency queries in R*-trees must first execute a range query and then compute the adjacency relations in a separate pass through the data. This requires a higher storage overhead (i.e., an additional list containing the result of the spatial query) compared to querying a Tetrahedral tree, but query performances do not appear to be significantly affected by variations in

the R*-tree branching factor. We can see from Figure 11, and Tables A.II and A.III that answering the range *TT* query on Tetrahedral trees requires up to 60% less time than on R*-trees, with a wider gap for the larger ranges.

8 CONCLUDING REMARKS

We have defined a family of spatial indexes, the *Tetrahedral trees*, that index a tetrahedral mesh using an octree or kD-tree subject to four different refinement strategies. Leveraging ideas from [35], we have applied spatial coherence to reorder and compress the indexed data, thus obtaining a compact encoding for Tetrahedral trees. We have developed efficient algorithms for both topological and spatial queries to take advantage of this compact encoding. The source code for our reference implementation is available at [33].

We have compared the various Tetrahedral trees based on memory usage, generation times and performances in spatial and topological queries. Compared to the three uncompressed spatial indexes defined in [25], the compressed Tetrahedral trees encoding provides an order of magnitude storage saving while also improving query response times. Conversely, the storage differences among the various Tetrahedral trees, using this compact encoding, are relatively small. In general, using an octree or a kD-tree subdivision does not significantly influence the overall query response times. While we have observed that smaller subdivision thresholds lead to slightly faster query execution times. Larger or smaller thresholds do not affect significantly the storage cost of the resulting Tetrahedra trees.

The PT-Ttree exhibits, in general, the best query performances with a moderate memory overhead. We also found that T-T and RT-Ttrees can be slower during the execution of our topological queries since they have to extract the range of the vertices at run time. Even though T-T and RT-Ttrees have similar storage requirements and query performances, the RT-T subdivision is limited by its tetrahedra insertion order which leads to unnecessarily deeper trees, in some cases. Our experiments highlight that this behavior happens when the threshold k_T on the number of tetrahedra per leaf block is below the maximum of the numbers of tetrahedra incident at the vertices. This suggests that the RT-T subdivision can be effective for 2D meshes, as described in [80], but not for tetrahedral meshes in 3D.

We have also compared Tetrahedral trees with representative data structures used for spatial and topological queries in practice: the IA data structure [65] and the R*-tree [7]. Tetrahedral trees outperform the IA data structure and R*-trees on spatial and range topological queries since they utilize the rich connectivity information in the tetrahedral mesh while also supporting domains with complicated geometry and topology. In particular, the IA data structure provides a complete answer to spatial queries only on convex simply-connected meshes. R*-trees require more space and are highly dependent on the branching factor of the internal nodes. Smaller branching factors enable a better discretization of the space and an increased efficiency in query execution but exhibit a larger storage overhead.

In our current implementation of Tetrahedral trees, we require a tetrahedral mesh to be provided as input. While simulation tools produce a 3D mesh directly, in many other applications a 3D point cloud is given, and a 3D triangulation algorithm needs to be applied to this input to produce a tetrahedral mesh. A future development of the work presented here is to design and implement an algorithm for building a Delaunay triangulation from a 3D point cloud which operates directly on Tetrahedral trees. The initial spatial decomposition will be generated based on the point cloud and a Delaunay mesh computed by triangulating the points in each leaf block independently and in parallel, for example using *OpenMP* [16, 17, 73]. The local Delaunay meshes could be combined to obtain a global Delaunay one by extending the *DeWall* algorithm [20] to our hierarchical decomposition.

We also plan to extend Tetrahedral trees to deal with arbitrary 3D simplicial complexes, i.e., with a non-manifold domain and with dangling edges and triangles. One application we are targeting is the identification and reconstruction of individual (physical) trees from huge point clouds originated by airborne or terrestrial Light Detection and Ranging (*LiDAR*) acquisitions. In this context, the structure of the point cloud is inferred by computing an *alpha shape* simplicial complex [30], which is further processed, by using, for instance, topology-based analysis techniques [95]. We plan to devise a distributed strategy for computing an alpha shape on Tetrahedral trees, and a distributed version of the topological analysis algorithm for identifying individual trees.

By using the extension of Tetrahedral trees for 3D simplicial complexes, further applications to geological models [88, 89] could be tackled, which are characterized by non-manifold geometries. This is an active research field [8–10, 12, 37, 43, 46, 70, 71, 91], but the efficient generation of these models is still an open problem, since existing methods do not scale to very large meshes.

ACKNOWLEDGMENTS

This work has been developed while Riccardo Fellegara was with University of Maryland at College Park, USA. This work has been partially supported by the US National Science Foundation under grant number IIS-1910766 and by the University of Maryland under the 2017-2018 BSOS Dean Research Initiative Program. It has also been performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

We wish to thank the reviewers for their valuable feedback. Datasets are courtesy of the *Volvis* repository (BONSAI, F16 and FOOT), the *Volume Library* (VISMALE), the *CMU Unstructured Mesh Suite* (SAN FERNANDO), Jason Sheperd (RBL), Claudio Silva (FIGHTER2) and Peter Williams (SPX).

REFERENCES

[1] P. Alliez and C. Gotsman. 2005. Recent advances in compression of 3D meshes. In *Advances in multiresolution for geometric modelling*. Springer, 3–26.

[2] S. Alsubaiee, A. Behm, and C. Li. 2010. Supporting location-based approximate-keyword queries. In *Proceedings SIGSPATIAL International Conference on Advances in Geographic Information Systems* (San Jose, California) *(GIS '10)*. ACM, New York, NY, USA, 61–70. https://doi.org/10.1145/1869790.1869802

[3] N. Andrysco and X. Tricoche. 2010. Matrix trees. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 963–972.

[4] L. Arge, M. De Berg, H. J. Haverkort, and K. Yi. 2004. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *Proceedings ACM SIGMOD international conference on Management of data*. ACM, 347–358.

[5] R. E. Bank, A. H. Sherman, and A. Weiser. 1983. Refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing, IMACS Transactions on Scientific Computation*, R. Stepleman, M. Carver, R. Peskin, W. F. Ames, and R. Vichnevetsky (Eds.). Vol. 1. North-Holland, Amsterdam, 3–17.

[6] M. W. Beall and M. S. Shephard. 1997. A general topology-based mesh data structure. *Internat. J. Numer. Methods Engrg.* 40, 9 (1997), 1573–1596. https://doi.org/10.1002/(SICI)1097-0207(19970515)40:9<1573::AID-NME128>3.0.CO;2-9

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings ACM SIGMOD Conference*. ACM Press, Atlantic City, NJ, 322–331.

[8] B. Benes and R. Forsbach. 2001. Layered data representation for visual simulation of terrain erosion. In *Spring Conference on Computer Graphics*. IEEE, 80–86.

[9] B. Benes and R. Forsbach. 2002. Visual simulation of hydraulic erosion. *Journal of WSCG* 10 (2002), 79–94.

[10] B. Benes, V. Tesinsky, J. Hornys, and S. K. Bhatia. 2006. Hydraulic erosion. *Computer Animation and Virtual Worlds* 17, 2 (2006), 99–108.

[11] J. L. Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[12] S. Brandel, S. Schneider, M. Perrin, N. Guiard, J.-F. Rainaud, P. Lienhard, and Y. Bertrand. 2005. Automatic building of structured geological models. *Journal of Computing and Information Science in Engineering* 5, 2 (2005), 138–148.

[13] J. C. Caendish, D. A. Field, and W. H. Frey. 1985. An approach to automatic three-dimensional finite element mesh generation. *International journal for numerical methods in engineering* 21, 2 (1985), 329–347.

[14] P. Cano and J.C. Torres. 2002. Representation of polyhedral objects using SP-octrees. *Journal of WSCG* 10, 1 (2002), 95–101.

[15] I. Carlbom, I. Chakravarty, and D. Vanderschel. 1985. A hierarchical data structure for representing the spatial decomposition of 3D objects. *IEEE Computer Graphics and Applications* 5, 4 (1985), 24–31.

[16] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. 2001. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[17] B. Chapman, G. Jost, and R. V. D. Pas. 2008. *Using OpenMP: Portable Shared Memory Parallel Programming*. Vol. 10. 353 pages. https://doi.org/10.1234/12345678

[18] A. O. Cifuentes and A. Kalbag. 1992. A performance study of tetrahedral and hexahedral elements in 3D finite element structural analysis. *Finite Elements in Analysis and Design* 12, 3-4 (1992), 313–318. https://doi.org/10.1016/0168-874X(92)90040-J

[19] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. 2004. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics* 10, 1 (January-February 2004), 29–45.

[20] P. Cignoni, C. Montani, and R. Scopigno. 1998. DeWall: a fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design* 30, 5 (1998), 333–341.

[21] P. Cignoni, C. Montani, and R. Scopigno. 1998. *Tetrahedra based volume visualization.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–18. https://doi.org/10.1007/978-3-662-03567-2_1

[22] D. Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[23] J.L. De Carufel, C. Dillabaugh, and A. Maheshwari. 2011. Point location in well-shaped meshes using jump-and-walk. In *Canadian Conference on Computational Geometry (CCCG)*. 147–152.

[24] L. De Floriani, M. Facinoli, P. Magillo, and D. Dimitri. 2008. A hierarchical spatial index for triangulated surfaces. In *Proceedings of the Third International Conference on Computer Graphics Theory and Applications (GRAPP)*. 86–91.

[25] L. De Floriani, R. Fellegara, and P. Magillo. 2010. Spatial indexing on tetrahedral meshes. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 506–509.

[26] L. De Floriani and A. Hui. 2005. Data structures for simplicial complexes: an analysis and a comparison. In *Proceedings of the third Eurographics symposium on Geometry processing*. Eurographics Association, 119–es.

[27] L. De Floriani and P. Magillo. 2003. Algorithms for visibility computation on terrains: a survey. *Environment and Planning B* 30, 5 (2003), 709–728.

[28] O. Devillers, S. Pion, and M. Teillaud. 2001. Walking in a triangulation. In *Proceedings of the seventeenth annual symposium on Computational geometry*. ACM, 106–114.

[29] C. Dillabaugh. 2010. I/O efficient path traversal in well-shaped tetrahedral meshes. *Proceedings of the 22nd Annual Canadian Conference on Computational Geometry, CCCG 2010* (10 2010).

[30] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. 1983. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory* 29, 4 (July 1983), 551–559. https://doi.org/10.1109/TIT.1983.1056714

[31] J. Elseberg, D. Borrmann, and A. Nüchter. 2013. One billion points in the cloud–an octree for efficient processing of 3D laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013), 76–88.

[32] C. Ericson. 2004. *Real-Time collision detection.* CRC Press, Inc., Boca Raton, FL, USA.

[33] R. Fellegara. 2019. Tetrahedral trees: a framework for the representation and analysis of tetrahedral meshes. https://github.com/UMDGeoVis/Tetrahedral_trees.

[34] R. Fellegara, F. Iuricich, and De Floriani. 2017. Efficient representation and analysis of triangulated terrains. In *Proceedings ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM.

[35] R Fellegara, K. Weiss, and L. De Floriani. 2017. The Stellar tree: a compact representation for simplicial complexes and beyond. *ArXiv e-prints* (2017). https://doi.org/abs/1707.02211

[36] R.A. Finkel and J.L. Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[37] M. Floater, Y. Halbwachs, O Hjelle, and M. Reimers. 1998. OMEGA: a CAD-based approach to Geological Modelling. In *Modelling 98 Conference*, Vol. 1. 68.

[38] P. J. Frey and P. L. George. 2008. *Mesh generation: Application to Finite Elements* (2nd ed.). Wiley. https://doi.org/10.1002/9780470611166

[39] C. Garth and K. I. Joy. 2010. Fast, memory-efficient cell location in unstructured grids for visualization. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 1541–1550.

[40] C. Geuzaine and J.-F. Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering* 79, 11 (2009), 1309–1331.

[41] C. Gotsman, S. Gumhold, and L. Kobbelt. 2002. Simplification and compression of 3D meshes. In *Tutorials on Multiresolution in Geometric Modelling*. Springer, 319–361.

[42] J. Grandy. 1999. Conservative remapping and region overlays by intersecting arbitrary polyhedra. *J. Comput. Phys.* 148, 2 (1999), 433–466. https://doi.org/10.1006/jcph.1998.6125

[43] R. H. Groshong Jr. 1999. 3D structural geology: a practical guide to surface and subsurface map interpretation. *Berlin: SpringerVerlag* (1999).

[44] T. Gurung and J. Rossignac. 2009. SOT: a compact representation for tetrahedral meshes. In *Proceedings SIAM/ACM Geometric and Physical Modeling (SPM '09)*. San Francisco, USA, 79–88. https://doi.org/10.1145/1629255.1629266

[45] A. Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings ACM SIGMOD*. ACM, 47–57.

[46] Y. Halbwachs and Ø. Hjelle. 2000. Generalized maps in geological modeling: object-oriented design of topological kernels. In *Advances in Software Tools for Scientific Computing*. Springer, 339–356.

[47] G. Heber and J. Gray. 2007. Supporting finite element analysis with a relational database backend, Part I: there is life beyond files. *Arxiv preprint cs/0701159* (2007).

[48] G. Heber and J. Gray. 2007. Supporting Finite Element Analysis with a relational database backend, Part II: database design and access. *Arxiv preprint cs/0701160* (2007).

[49] G. Held and T. Marshall. 1991. *Data compression; techniques and applications: hardware and software considerations*. John Wiley & Sons, Inc.

[50] K. Ho-Le. 1988. Finite element mesh generation methods: a review and classification. *Computer-Aided Design* 20, 1 (1988), 27 – 38. https://doi.org/10.1016/0010-4485(88)90138-8

[51] F. Iuricich, R. Fellegara, and L. De Floriani. 2015. TetraMesh library. https://github.com/UMDGeoVis/TetraMesh.

[52] I. Kamel and C. Faloutsos. 1993. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management*. ACM, 490–499.

[53] M. Langbein, G. Scheuermann, and X. Tricoche. 2003. An efficient point location method for visualization in large unstructured grids. In *VMV*. 27–35.

[54] M. Lindenbaum, H. Samet, and G. R. Hjaltason. 2005. A probabilistic analysis of trie-based sorting of large collections of line segments in spatial databases. *SIAM J. Comput.* 35, 1 (September 2005), 22–58.

[55] A. Melling. 1997. Tracer particles and seeding for particle image velocimetry. *Measurement Science and Technology* 8, 12 (1997), 1406–1416. https://doi.org/10.1088/0957-0233/8/12/005

[56] M. M. Mesmoudi, L. De Floriani, and P. Magillo. 2009. Morphology analysis of 3D scalar fields based on Morse theory and discrete distortion. In *Proceedings ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 187–196. https://doi.org/10.1145/1653771.1653799

[57] MFEM 2010. MFEM: Modular Finite Element Methods library. https://doi.org/10.11578/dc.20171025.1248 mfem.org.

[58] A. S. M. Mosa, B. Schön, M. Bertolotto, and D. F. Laefer. 2012. Evaluating the benefits of octree-based indexing for LiDAR data. *Photogrammetric Engineering & Remote Sensing* 78, 9 (2012), 927–934.

[59] E. P Mucke, I. Saias, and B. Zhu. 1999. Fast randomized point location without preprocessing in two-and three-dimensional Delaunay triangulations. *Computational Geometry* 12, 1 (1999), 63–83.

[60] I. Navazo. 1989. Extended octree representation of general solids with plane faces: model structure and algorithms. *Computer & Graphics* 13, 1 (1989), 5–16.

[61] R.C. Nelson and H. Samet. 1986. A consistent hierarchical representation for vector data. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 197–206.

[62] G. M. Nielson. 1997. Tools for triangulations and tetrahedralizations and constructing functions defined over them. In *Scientific Visualization: overviews, Methodologies and Techniques*, G. M. Nielson, H. Hagen, and H. Müller (Eds.). IEEE Computer Society, Silver Spring, MD, Chapter 20, 429–525.

[63] Oracle Spatial. 2019. Indexing of spatial data. https://docs.oracle.com/en/database/oracle/oracle-database/19/spatl/spatial-concepts.html [Online; accessed 09-March-2019].

[64] J. A. Orenstein. 1982. Multidimensional tries used for associative searching. *Inform. Process. Lett.* 14, 4 (1982), 150–157.

[65] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. 1993. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics (TOG)* 12, 1 (1993), 56–102.

[66] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. 2006. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings ACM SIGMOD*. ACM, 551–562.

[67] J. Peng, C.-S. Kim, and C.-C. J. Kuo. 2005. Technologies for 3D mesh compression: a survey. *Journal of Visual Communication and Image Representation* 16, 6 (2005), 688–733.

[68] F. Penninga and P van Oosterom. 2008. A simplicial complex-based DBMS approach to 3D topographic data modelling. *International Journal of Geographical Information Science* (2008).

[69] F. Penninga, P. van Oosterom, and B.M. Kazar. 2006. A TEN-based DBMS approach for 3D topographic data modelling. In *12th International Symposium on Spatial Data Handling*, A. Riedl, W. Kainz, and G. Elmes (Eds.). Springer, 581–598.

[70] M. Perrin, B. Zhu, J.-F. Rainaud, and S. Schneider. 2005. Knowledge-driven applications for geological modeling. *Journal of Petroleum Science and Engineering* 47, 1-2 (2005), 89–104.

[71] J. Plate, M. Tirtasana, R. Carmona, and B. Fröhlich. 2002. Octreemizer: a hierarchical approach for interactive roaming through very large volumes.. In *VisSym*. 53–60.

[72] PostGIS. 2019. PostGIS 2.5 user manual. https://postgis.net/docs/manual-2.5/using_postgis_dbmanagement.html#idm2246 [Online; accessed 19-March-2019]. Section 4.6: Building indexes.
[73] J. F. Remacle. 2017. A two-level multithreaded Delaunay kernel. *Computer-Aided Design* 85 (2017), 2–9. https://doi.org/10.1016/j.cad.2016.07.018 24th International Meshing Roundtable Special Issue: Advances in Mesh Generation.
[74] P. N. M. Rizki, J. Park, S. Oh, and H. Lee. 2015. STR-octree indexing method for processing LiDAR data. In *2015 IEEE SENSORS*. IEEE, 1–4.
[75] J. Roerdink and A. Meijster. 2000. The watershed transform: definitions, algorithms, and parallelization strategies. *Fundamental Informaticae* 41 (2000), 187–228.
[76] J. Rossignac, A. Safonova, and A. Szymczak. 2001. 3D compression made simple: Edge-Breaker on a Corner Table. In *Proceedings Shape Modeling International*. IEEE Computer Society, Genova, Italy.
[77] N. Roussopoulos and D. Leifker. 1985. Direct spatial search on pictorial databases using packed R-trees. In *ACM SIGMOD Record*, Vol. 14. ACM, 17–31.
[78] T. Roxborough and G. Nielson. 2000. Tetrahedron-based, least-squares, progressive volume models with application to freehand ultrasound data. In *Proceedings IEEE Visualization*. IEEE Computer Society, 93–100. https://doi.org/10.1109/VISUAL.2000.885681
[79] R. B. Rusu and S. Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation*. 1–4. https://doi.org/10.1109/ICRA.2011.5980567
[80] H. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann. 1024 pages.
[81] H. Samet and R.E. Webber. 1985. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics (TOG)* 4, 3 (1985), 182–222.
[82] F. Sauer, J. Xie, and K.-L. Ma. 2017. A combined Eulerian-Lagrangian data representation for large-scale applications. *IEEE Transactions on Visualization and Computer Graphics* 23, 10 (Oct 2017), 2248–2261. https://doi.org/10.1109/TVCG.2016.2620975
[83] B. Schaling. 2014. *The Boost C++ libraries, Second Edition*. B. Schaling.
[84] T. Sellis, N. Roussopoulos, and C. Faloutsos. 1987. The R+-tree: a dynamic index for multi-dimensional objects. *VLDB endowments* (1987).
[85] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C.-T. Lu. 1999. Spatial databases-accomplishments and research needs. *IEEE Transactions on Knowledge and Data Engineering* 11, 1 (Jan. 1999), 45–55. https://doi.org/10.1109/69.755614
[86] H. Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Software* 41, 2 (Feb. 2015), 11:1–11:36. https://doi.org/10.1145/2629697
[87] S.-H. Teng and C. W. Wong. 2000. Unstructured mesh generation: theory, practice, and perspectives. *International Journal of Computational Geometry & Applications* 10, 03 (2000), 227–266. https://doi.org/10.1142/S0218195900000152
[88] A. K. Turner. 2006. Challenges and trends for geological modelling and visualisation. *Bulletin of Engineering Geology and the Environment* 65, 2 (2006), 109–127.
[89] A. K. Turner and C. W. Gable. 2007. A review of geological modeling. *Three-Dimensional Geologic Mapping for Groundwater Applications* (2007), 81.
[90] H.-H. Vu, P. Labatut, J.-P. Pons, and R. Keriven. 2012. High accuracy and visibility-consistent dense multiview stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, 5 (2012), 889–901. https://doi.org/10.1109/TPAMI.2011.172
[91] L. Wang, Y. Yu, K. Zhou, and B. Guo. 2011. Multiscale vector volumes. *ACM Transactions on Graphics (TOG)* 30, 6 (2011), 167.
[92] K. Weiss and L. De Floriani. 2011. Simplex and diamond hierarchies: models and applications. *Computer Graphics Forum* 30, 8 (2011), 2127–2155. https://doi.org/10.1111/j.1467-8659.2011.01853.x
[93] K. Weiss, R. Fellegara, L. De Floriani, and M. Velloso. 2011. The PR-star octree: a spatio-topological data structure for tetrahedral meshes. In *Proceedings ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 92–101.
[94] K. Weiss, F. Iuricich, R. Fellegara, and L. De Floriani. 2013. A primal/dual representation for discrete Morse complexes on tetrahedral meshes. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 361–370.
[95] X. Xu, L. De Floriani, and F. Iuricich. 2018. Individual tree mapping from LiDAR point clouds based on topological tools. *Proceedings 2018 Fall Meeting American Geophysical Union* (Dec 2018), 10–14.
[96] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. 2012. Towards parallel spatial query processing for big spatial data. In *IEEE International Parallel and Distributed Processing Symposium*. 2085–2094. https://doi.org/10.1109/IPDPSW.2012.245
[97] S. Zlatanova, A. Abdul Rahman, and W. Shi. 2004. Topological models and frameworks for 3D spatial objects. *Computers & Geosciences* 30 (2004), 419–428.

A APPENDIX

Table A.I. Comparison of total timings (in seconds) and average geometric tests for point locations and range queries on Tetrahedral trees. The *time* columns show the execution timings for 1000 queries, while the *tests* columns show the average geometric tests executed in a single query. The *ok* columns show the results for the Tetrahedral octrees, while the *kD* columns show the results for the Tetrahedral kD-trees. In bold are highlighted the timings and geometric tests shown in the charts.

data		point location				range queries							
						smaller				larger			
		time		tests		time		tests		time		tests	
		ok	kD	ok	kD	ok	kD	ok	kD	ok	kD	ok	kD
RBL	P-T	0.038	0.025	0.17K	0.11K	7.06	6.27	26.6K	22.9K	27.2	24.6	83.5K	70.4K
		0.054	0.043	0.26K	0.20K	7.98	7.53	31.5K	29.4K	29.3	28.1	98.0K	91.6K
	PT-T	0.035	**0.023**	0.16K	**0.09K**	6.86	**6.02**	25.3K	**21.5K**	26.8	**24.1**	79.5K	**66.3K**
		0.046	0.031	0.22K	0.14K	7.53	6.76	29.1K	25.4K	28.3	26.1	91.1K	79.1K
	T-T	0.036	0.023	0.16K	0.09K	6.98	6.14	25.8K	21.7K	27.3	24.5	81.1K	67.2K
		0.046	0.031	0.22K	0.14K	7.57	6.82	29.1K	25.4K	28.5	26.3	91.1K	79.2K
F16	P-T	0.056	0.046	0.25K	0.20K	6.81	6.66	27.0K	26.3K	43.9	44.8	163K	167K
		0.092	0.063	0.44K	0.29K	8.23	7.48	33.7K	30.2K	51.9	49.4	203K	190K
	PT-T	0.039	**0.029**	0.16K	**0.11K**	6.03	**5.55**	23.1K	**20.9K**	39.7	**38.9**	140K	**134K**
		0.050	0.036	0.22K	0.15K	6.47	6.05	25.3K	23.3K	41.9	41.4	153K	149K
	T-T	0.039	0.029	0.17K	0.11K	6.05	5.59	23.1K	20.9K	39.9	39.1	140K	134K
		0.050	0.036	0.22K	0.15K	6.48	6.10	25.3K	23.3K	42.2	41.6	153K	149K
SAN FERN	P-T	0.024	0.018	0.11K	0.08K	6.36	5.33	25.5K	20.9K	59.0	51.9	228K	193K
		0.029	0.036	0.14K	0.18K	6.82	6.90	27.9K	28.4K	63.2	64.4	251K	257K
	PT-T	0.021	**0.018**	0.09K	**0.08K**	6.11	**5.32**	24.2K	**20.8K**	56.9	**51.9**	217K	**193K**
		0.029	0.032	0.13K	0.16K	6.82	6.74	27.9K	27.6K	63.2	63.3	250K	252K
	T-T	0.022	0.023	0.10K	0.11K	6.13	5.82	24.4K	23.0K	58.0	55.9	221K	212K
		0.029	0.034	0.13K	0.17K	6.85	6.81	27.9K	27.8K	63.4	65.2	250K	259K
BONSAI	P-T	0.010	0.018	0.03K	0.07K	5.10	5.46	19.7K	21.3K	50.6	54.4	175K	194K
		0.014	0.024	0.05K	0.11K	6.00	6.02	24.0K	24.1K	56.7	59.1	209K	219K
	PT-T	0.009	**0.011**	0.03K	**0.04K**	4.87	**4.66**	18.4K	**17.2K**	48.7	**48.5**	163K	**157K**
		0.012	0.019	0.04K	0.08K	5.52	5.68	21.8K	22.4K	53.2	56.1	191K	204K
	T-T	0.009	0.011	0.03K	0.04K	4.93	4.71	18.5K	17.2K	49.4	48.8	164K	157K
		0.012	0.020	0.04K	0.08K	5.55	5.72	21.8K	22.4K	53.5	56.6	191K	204K
VISMALE	P-T	0.006	0.009	0.02K	0.03K	4.58	4.57	17.3K	17.2K	51.3	51.8	169K	172K
		0.010	0.016	0.03K	0.06K	5.55	5.56	22.2K	22.2K	58.3	59.3	212K	216K
	PT-T	0.006	**0.008**	0.02K	**0.03K**	4.31	**4.33**	15.9K	**15.9K**	48.9	**50.1**	154K	**160K**
		0.008	0.013	0.02K	0.05K	5.18	5.19	20.5K	20.3K	55.8	56.7	200K	201K
	T-T	0.006	0.008	0.02K	0.03K	4.35	4.36	15.9K	15.9K	49.5	50.6	154K	160K
		0.008	0.013	0.02K	0.05K	5.20	5.21	20.5K	20.3K	56.1	56.8	200K	201K
FOOT	P-T	0.010	0.015	0.04K	0.06K	5.63	5.67	21.5K	21.7K	61.0	61.1	206K	208K
		0.015	0.023	0.05K	0.10K	6.47	6.64	25.9K	26.6K	67.3	68.5	246K	251K
	PT-T	0.010	**0.013**	0.03K	**0.05K**	5.35	**5.44**	20.0K	**20.5K**	58.6	**59.7**	190K	**197K**
		0.014	0.022	0.05K	0.10K	6.43	6.60	25.7K	26.4K	67.0	68.2	244K	249K
	T-T	0.010	0.013	0.04K	0.05K	5.52	5.49	20.7K	20.5K	60.3	60.2	197K	197K
		0.014	0.022	0.05K	0.10K	6.48	6.63	25.7K	26.4K	67.5	68.8	245K	250K

Table A.II. Comparison of timings (in seconds) for executing the range Vertex-Tetrahedron (VT) and range Tetrahedron-Tetrahedron (TT) queries. The columns show the execution timings for 1000 queries. These queries use the same query boxes as for the spatial range queries. The *ok* columns show the results for the Tetrahedral octrees, while the *kD* columns show the results for the Tetrahedral kD-trees. In bold are highlighted the timings and geometric tests shown in the charts.

data		range VT				range TT			
		smaller		larger		smaller		larger	
		ok	kD	ok	kD	ok	kD	ok	kD
RBL	P-T	7.97	7.60	78.0	75.9	46.2	44.6	439	430
		7.58	7.41	73.6	73.4	45.8	44.8	427	421
	PT-T	8.18	**7.61**	79.5	**76.0**	46.6	**44.6**	445	**434**
		7.73	7.55	75.2	75.4	45.6	44.5	430	425
	T-T	12.2	11.9	112	110	47.8	46.2	444	436
		11.1	11.2	102	105	46.4	45.9	431	430
F16	P-T	4.06	4.05	65.8	66.2	25.7	25.3	406	403
		4.12	4.05	63.6	64.4	27.2	26.1	409	405
	PT-T	4.12	**4.07**	68.2	**68.0**	25.3	**24.6**	409	**406**
		4.08	4.04	66.5	67.2	25.6	24.8	405	404
	T-T	6.56	6.48	100	102	26.5	25.9	411	410
		6.34	6.30	95.3	97.0	26.5	25.6	406	406
SAN FERN	P-T	2.94	2.69	62.9	61.2	20.1	18.7	429	420
		3.00	2.70	61.5	58.5	20.5	20.1	426	427
	PT-T	2.94	**2.68**	63.8	**61.2**	20.0	**18.7**	429	**419**
		3.00	2.70	61.1	58.5	20.5	20.0	427	426
	T-T	4.70	4.34	94.7	89.0	21.1	19.9	433	425
		4.48	4.21	87.6	84.1	20.9	20.4	429	433
BONSAI	P-T	4.10	3.87	101	95.7	25.1	24.9	578	581
		4.28	4.05	99.3	95.4	26.4	24.9	577	579
	PT-T	4.16	**3.60**	107	**105**	25.6	**25.5**	594	**596**
		4.22	3.88	98.5	94.8	25.1	24.5	571	578
	T-T	6.10	5.44	156	149	28.0	26.6	618	634
		5.80	5.55	138	135	27.6	26.0	581	600
VISMALE	P-T	4.18	3.58	120	116	25.8	24.2	674	668
		4.44	4.09	116	110	27.3	25.0	659	663
	PT-T	4.27	**3.63**	125	**119**	25.9	**25.0**	688	**677**
		4.45	3.90	115	111	26.1	25.3	655	665
	T-T	6.25	5.42	183	170	28.8	26.7	712	722
		5.81	5.52	158	156	27.2	26.2	667	686
FOOT	P-T	4.94	4.09	136	127	31.3	30.9	780	778
		4.89	4.70	128	122	30.6	30.0	757	757
	PT-T	4.94	**4.12**	140	**129**	31.3	**31.0**	793	**779**
		4.89	4.66	126	123	30.6	30.0	758	757
	T-T	7.23	6.11	198	180	34.4	32.0	813	824
		6.58	6.28	165	158	30.6	29.9	751	754

Table A.III. Comparison of total timings (in seconds) and of average geometric tests (per query) for all the spatial and topological queries executed on our R*-tree implementation using three different branch factor. The *time* columns show the execution timings for 1000 queries, while the *tests* columns show the average geometric tests executed in a single query. In bold are highlighted the timings and geometric tests shown in the charts.

		point		**range query**				**range VT**		**range TT**	
data	b_F	**location**		**smaller**		**larger**		**small**	**large**	**small**	**large**
		time	**tests**	**time**	**tests**	**time**	**tests**	**time**	**time**	**time**	**time**
RBL	4	0.03	0.02K	18.6	42.8K	174	409K	24.2	255	65.9	786
	8	**0.03**	**0.08K**	**12.1**	**43.9K**	**118**	**412K**	**17.5**	**200**	**63.4**	**629**
	16	0.09	0.32K	10.8	45.3K	105	416K	16.3	188	59.7	560
F16	8	0.21	0.49K	11.0	24.8K	153	363K	13.6	308	42.6	1060
	16	**0.87**	**2.70K**	**10.9**	**28.8K**	**141**	**381K**	**12.9**	**302**	**41.3**	**981**
	32	5.20	16.6K	11.1	31.8K	138	395K	12.4	296	42.4	1004
SAN FER	8	0.01	0.05K	3.89	16.0K	110	375K	4.73	176	19.8	687
	16	**0.04**	**0.17K**	**3.74**	**16.6K**	**99**	**380K**	**4.41**	**165**	**19.5**	**640**
	32	0.21	1.12K	4.04	18.6K	102	395K	4.37	161	20.1	627
BONSAI	16	0.04	0.20K	5.22	23.1K	133	534K	6.69	257	30.5	904
	32	**0.20**	**1.10K**	**5.42**	**24.8K**	**136**	**548K**	**6.41**	**248**	**30.6**	**936**
	64	0.67	3.90K	5.71	27.1K	128	566K	6.44	248	30.9	908
VISMA	16	0.03	0.15K	5.56	23.0K	151	606K	7.44	303	32.9	977
	32	**0.14**	**0.74K**	**5.72**	**24.6K**	**151**	**619K**	**7.04**	**289**	**32.9**	**1017**
	64	0.58	3.41K	5.85	26.7K	143	638K	6.93	292	33.2	970
FOOT	32	0.18	0.98K	6.73	29.2K	174	712K	8.33	317	39.9	1186
	64	**0.65**	**3.77K**	**6.78**	**31.6K**	**159**	**731K**	**8.17**	**311**	**39.9**	**1137**
	128	2.73	16.1K	7.50	35.3K	163	759K	8.23	309	40.7	1129

Table A.IV. Comparison of total timings (in seconds) and average geometric tests for point locations and range queries on *Uncompressed Tetrahedral trees* based on the naive uncompressed encoding, described in [25]. The *time* columns show the execution timings for 1000 queries, while the *tests* columns show the average geometric tests executed in a single query. The *ok* columns show the results for *Uncompressed* octrees, while the *kD* columns show the results for *Uncompressed* kD-trees. In bold are highlighted the timings and geometric tests shown in the charts.

data		point location				range queries							
						smaller				larger			
		time		tests		time		tests		time		tests	
		ok	kD	ok	kD	ok	kD	ok	kD	ok	kD	ok	kD
RBL	P-T	0.056	0.040	0.31K	0.21K	7.96	7.23	31.1K	27.7K	30.9	28.2	101K	87.9K
		0.081	0.068	0.46K	0.38K	9.08	8.66	37.1K	35.0K	34.0	32.6	121K	113K
	PT-T	0.051	**0.035**	0.28K	**0.19K**	7.73	**6.96**	29.8K	**26.2K**	30.4	**27.5**	96.4K	**83.6K**
		0.069	0.050	0.39K	0.27K	8.58	7.85	34.2K	30.8K	32.6	30.3	112K	100K
	RT-T	0.053	0.036	0.29K	0.19K	7.93	7.09	30.5K	26.7K	31.2	28.1	100K	85.5K
		0.069	0.050	0.39K	0.27K	8.60	7.92	34.2K	30.8K	32.8	30.6	112K	100K
F16	P-T	0.124	0.110	0.45K	0.40K	11.2	11.2	30.8K	30.4K	68.9	71.3	185K	193K
		0.205	0.150	0.77K	0.55K	13.5	12.5	38.4K	34.7K	81.1	78.2	229K	217K
	PT-T	0.083	**0.064**	0.29K	**0.22K**	9.93	**9.27**	26.4K	**24.2K**	61.9	**61.2**	160K	**156K**
		0.109	0.083	0.39K	0.29K	10.7	10.1	28.9K	27.0K	65.9	65.6	174K	172K
	RT-T	0.083	0.064	0.29K	0.22K	10.0	9.34	26.4K	24.2K	62.5	61.7	160K	156K
		0.109	0.083	0.39K	0.29K	10.8	10.2	28.9K	27.0K	66.4	66.0	174K	172K
SAN FERN	P-T	0.072	0.045	0.44K	0.27K	7.57	6.34	31.5K	25.8K	65.6	57.1	261K	220K
		0.086	0.083	0.53K	0.51K	8.01	7.94	33.8K	33.5K	69.3	69.0	282K	282K
	PT-T	0.060	**0.044**	0.37K	**0.26K**	7.23	**6.33**	29.9K	**25.7K**	63.3	**57.2**	249K	**219K**
		0.085	0.075	0.53K	0.46K	8.01	7.77	33.7K	32.6K	69.4	67.9	281K	276K
	RT-T	0.061	0.054	0.37K	0.32K	7.30	6.79	30.1K	27.8K	64.3	60.7	253K	237K
		0.085	0.076	0.53K	0.47K	8.04	7.82	33.7K	32.8K	69.4	69.7	281K	283K
BONSAI	P-T	0.030	0.035	0.16K	0.20K	5.82	6.35	23.3K	26.0K	55.9	60.1	203K	227K
		0.049	0.050	0.28K	0.29K	6.76	6.99	28.0K	29.1K	62.4	65.0	239K	254K
	PT-T	0.028	**0.020**	0.15K	**0.11K**	5.56	**5.38**	22.0K	**21.1K**	53.9	**53.0**	190K	**186K**
		0.040	0.039	0.22K	0.22K	6.25	6.58	25.5K	27.2K	58.5	62.0	219K	237K
	RT-T	0.028	0.020	0.15K	0.11K	5.64	5.42	22.1K	21.1K	54.6	53.4	191K	186K
		0.040	0.039	0.22K	0.22K	6.28	6.62	25.5K	27.2K	58.7	62.3	219K	237K
VISMALE	P-T	0.019	0.018	0.10K	0.09K	5.30	5.24	20.9K	20.9K	57.0	57.2	198K	204K
		0.049	0.034	0.28K	0.19K	6.34	6.41	26.1K	26.6K	64.3	65.9	242K	252K
	PT-T	0.016	**0.015**	0.08K	**0.08K**	4.99	**4.95**	19.4K	**19.4K**	54.5	**55.0**	182K	**190K**
		0.023	0.026	0.12K	0.14K	5.91	5.96	24.2K	24.5K	61.6	62.4	229K	235K
	RT-T	0.016	0.015	0.08K	0.08K	5.03	4.99	19.4K	19.4K	54.9	55.4	182K	190K
		0.023	0.026	0.12K	0.14K	5.94	6.00	24.2K	24.5K	61.9	62.7	229K	235K
FOOT	P-T	0.028	0.030	0.15K	0.17K	6.55	6.44	26.3K	26.1K	67.9	66.4	243K	244K
		0.051	0.053	0.29K	0.31K	7.42	7.57	30.8K	31.7K	74.0	74.7	282K	289K
	PT-T	0.026	**0.026**	0.14K	**0.14K**	6.24	**6.21**	24.7K	**24.9K**	65.4	**64.8**	227K	**232K**
		0.047	0.051	0.27K	0.29K	7.37	7.51	30.6K	31.4K	73.7	74.3	280K	287K
	RT-T	0.027	0.026	0.15K	0.14K	6.45	6.26	25.5K	24.9K	66.9	65.2	235K	233K
		0.047	0.052	0.27K	0.30K	7.41	7.54	30.6K	31.4K	74.0	74.6	281K	288K