Dipartimento di Informatica e
Scienze dell'Informazione

An Extensible Framework for Huge Geometric Models

David Canino
DISI, University of Genova, Italy
canino@disi.unige.it

Technical Reports Series **DISI-TR-2009-08**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy

December 2009
http://www.disi.unige.it

Abstract

Nowadays, some gigantic models can be easily produced in many applications and their storage cost often exceeds the RAM size in a common workstation. The model simplification and the multiresolution techniques are a rather mature technologies that in many cases can efficiently manage complex data: however, the RAM size is often a severe bottleneck, even for high–performance graphics workstations. Thus, using an external memory technique is mandatory in this case: it is important to encode huge models in the most efficient way as possible, maintaining the opportunity to analyze, to manage and to display them according to the user requests and choices.

In this paper we define an extensible framework, called *OMSM* (short for *Objects Management in Secondary Memory*), for managing huge models: it supports external memory management of complex models, loading dynamically in RAM only the selected sections. All the functionalities implemented on this framework can be applied to a generic geometric model, independently from its dimension, on low–cost PC platforms.

1 Introduction

Nowadays, the performance of graphics subsystems has enourmously improved, but unfortunately the complexity of graphics applications has also increased. Some huge models can easily produced in many applications, for example by the 3–dimensional scanning of real objects or by using unstructured tetrahedral meshes to represent scalar fields in scientific computing. In order to improve the models accuracy we need an accurate object sampling: the dimension of our models increases and it often exceeds the RAM size in a common workstation. Moreover, the distributed networks provide an extremely widespread channel of transmission, useful for sharing the models: unfortunately, the limited amount of available bandwith could involve very high waiting times. An important example is given by the models used in the *ABA Project* (short for *Allen Brain Atlas*, refer to [ABA06] for more details): this project deals with the influence regions of genes in the mouse brain, by making them available to the researchers on the Web. There are more than 20,000 expressed genes so the whole map exceeds the available RAM in a common workstation. An other important example is the *Saint Matthew* 3D model, recently produced by the *Digital Michelangelo Project*, at the Stanford University (refer to [LPC+00] for more details): this model is composed by about 370M triangles. Hence, these meshes introduce severe overheads and their management has often prohibitive costs, even for the high–performance graphics workstations. Moreover, it is important to encode huge models in the most efficient way as possible, maintaining the opportunity to analyze, to manage and to display them according to the user requests and choices. We can execute some operations on a huge model, for example we can remember:

- the efficient visualization of the geometric models, useful for selective inspection;
- the editing operations, useful to improve the data quality as described in [BEG94], where the authors define the requirements that should be satisfied by a *good* model;
- the retrieval of some topological properties in the model;
- the execution of specific operations on the input model, useful for many aspects: for example we can compute metric measures on the input model, like the computation of curvature, normals and so on.

Increasing the RAM size could be a good and trivial solution, since its cost is going sharply down, but this approach raises some questions:

- this solution is not easily *scalable* because the geometric models could require an arbitrary amount of space and hence increasing RAM size does not really solve the problem;
- this technique is not *feasible* because establishing the correct amount of memory is difficult: you should remember that operating system also requires some RAM to work.

Thus, applying a *simplification* algorithm would seem to be a reasonable approach since we can reduce a model to a manageable size by applying a set of local updates: unfortunately, obtaining the optimal simplification of a model is known to be a *NP–hard* problem, as demonstrated in [AS94]. Hence, many heuristic methods to provide an approximated version of the model have been developed: such techniques simplify the geometry and preserve some attributes of the input mesh. You can refer to [PS97] and [LRC+02] for more details about this topic. The simplification techniques are also suitable to construct a *multiresolution* model. A critical aspect in the interactive graphics is the *resolution* (or *complexity*) of the models to be managed: this aspect can be expressed in terms of density of elementary components (i.e. triangles or tetrahedra). We can define

a *multiresolution model* as a model with the capability of providing some representations of a spatial object at different levels of accuracy and complexity, depending by specific application needs. According to [DdFM+06], a multiresolution model encodes all the updates performed by the simplification process as a partial order, from which a continuous set of meshes at different LODs can be efficiently extracted. These techniques have become relevant in several applications and many multiresolution models have been developed: in [Gar99] and [DdFM+06] the authors describe an historical overview of this type of research, briefly discussing its new perspectives. In literature, we can find many examples of multiresolution models for triangular meshes, as those described in [PH97], [RL00], [LRC+02], and [dFMPS03] and [dFKP04]: we can also remember the multiresolution models based on the tetrahedral meshes, like those described in [DdFM+05] and [DdF06]. However, each multiresolution model is based on a particular type of local operator and it can be considered as a specific instance of the *MT* model (short for *Multi–Tessellation*), that has been introduced in [Pup98]. In fact, the *MT* formalizes a multiresolution model as a coarse base mesh plus a partially ordered set of updates that can be used to obtain refined meshes at variable resolution, independently by the construction strategy. A dimension–independent representation of the *Multi–Tessellation* model has been proposed in [dFPM97] and implemented in the *MT Package*, distributed under the *GPL* license (see [dFMP00] for more details).

Unfortunately, these techniques have memory and time requirements that are far too high to handle very large data sets: moreover, many of these algorithms require a random access to the model parts, thus they cannot be easily adapted to efficiently work with a model exceeding the RAM size. Hence, using an out–of–core technique is mandatory when we need to process an huge mesh: usually, we maintain the entire model in a storage support (namely the *external memory*, for example a local hard–disk or a remote storage area network) and then we dinamically load in RAM only the selected sections, that are small enough to be processed in–core. In such way, we can directly operate on each portion, by using a limited memory footprint and by removing the limit on the input size: however, the I/O among the RAM and the external memory (usually a local disk) is often a bottleneck because an disk access is slower than an access in RAM. Thus, a naive management of the external memory may highly degrade the performance: hence, some data structures and algorithms capable of efficiently working in the external memory are needed. These techniques are also called *EM* (short for *External Memory*) techniques and designing them is an active research area that has produced interesting results in many applicative domains. For example, we can remember some techniques used for:

- the EM models visualization, like those described in [FS01], [CKS02], [SCESL02], [Lin03], [CGG+04], [CGG+05], [RTBS05] and [CBPS06];
- the EM isosurfaces extraction, like those described in [CSS98], [CS99], [CFSW01] and [Chi03];
- the EM surface reconstruction, like that described in [BMR+99].

Furthermore, many *EM* simplification algorithms have been developed, like those described in [Hop98], [Lin00], [Pri00], [LS01], [CMRS03], [ILGS03], [DdFPS06] and [VCL+07]. Also the multiresolution models can exceed the available memory, even if we are using a compact representation: in literature, many EM multiresolution models have been developed, but most of them, as those described in [Lin03] and [CGG+04], do not support some topological operators. In this paper, we concentrate our attention on a particular class of the EM techniques, based on the *space partitioning*: in such techniques we subdivide the input model into hierarchical patches by using the *spatial indexing* structures, useful to efficiently access to the model portions. We assume that each portion can fit to RAM and that it is small enough to be managed in–core.

The remainder of this document is organized as follows: in the Section 2 we give some background notions to be used through this document, while in the Section 3 we give an overview about the research fields interested by our research. In the Section 4 we analyze the properties required by a storing architecture for spatial data and then we propose an extension of a flexible framework for managing huge geometric models, called *OMSM* (short for *Objects Management in Secondary Memory*), that we have introduced in [Can07]. Finally, in the Section 5 we discuss the possible extensions of this prototype.

2 Background Notions

In this Section we give some background notions to be used through this document: in the Section 2.1 we review the *cell* and *simplicial complexes*, the basic combinatorial structures that formalize the mesh definition, while in the Section 2.2 we also review the basic notions about the *topological relations*, that provide the connectivity information among the different parts

of a simplicial complex. We limit our analysis to the Euclidean space $\mathbb{E}^3$: you should refer to [dF04] and [Ago05] for more details about these topics.

2.1 Cell and Simplicial Complexes

In this Section we review the *cell* and the *simplicial complexes*, the basic combinatorial structures that formalize the mesh definition, thus the notion of *geometric model*.

Let x be a vector in $\mathbb{E}^d$ (with $d \geq 1$) and $\|x\|$ the length of x, then we define the *unit d–sphere* as $S^d = \{x \in \mathbb{E}^{d+1}.\|x\| = 1\}$, the *unit d–disk* as $D^d = \{x \in \mathbb{E}^d.\|x\| \leq 1\}$ and the *unit d–ball* as $B^d = \{x \in \mathbb{E}^d.\|x\| < 1\}$.

A *k–cell* in the Euclidean space $\mathbb{E}^n$, with $1 \leq k \leq n$, is a subset of $\mathbb{E}^n$ homeomorphic to B^k: usually k is called *dimension* of the Euclidean cell, for example a point is a 0–cell. An *Euclidean cell complex* in $\mathbb{E}^n$ is a finite set Γ of disjoint cells of dimension at most d (with $0 \leq d \leq n$) such that the boundary of each k-cell γ consists of the union of other cells of Γ with dimension less than k: the set of such cells is denoted with $B(\gamma)$. The maximum d of the cells dimension is called *dimension* of the cell complex Γ, while the subset of $\mathbb{E}^n$ spanned by the cells of Γ is called *domain* of the Euclidean cell complex. If a h–cell $\gamma^{'}$ (with $1 \leq h \leq k$) belongs to $B(\gamma)$ then $\gamma^{'}$ is a *h–face* of γ: if $\gamma^{'} \neq \gamma$ then $\gamma^{'}$ is a *proper face* of γ. A cell that does not belong to the boundary of any other cell is called *top cell*: if all the top cells are d–cells, then Γ is a *regular* cell complex. In an Euclidean cell complex Γ the *star* of a cell γ is defined as $St(\gamma) = \{\gamma\} \cup \{\gamma^{'} \in \Gamma.\gamma \in B(\gamma^{'})\}$, i.e. the set of all the cells that have γ as face, while the *link* of an Euclidean cell γ is defined as the set of cells in Γ forming the combinatorial boundary of the cells in $St(\gamma) \setminus \{\gamma\}$, i.e. the set of all the faces of the cells in $St(\gamma)$ that are not incident in γ.

The *simplicial complexes* are similar to cell complexes but their cells, called *simplexes*, are closed and defined by the convex combination of points in the Euclidean space.

Let k be a non–negative integer then an Euclidean *k–simplex* σ is the convex hull of $k+1$ independent points in $\mathbb{E}^n$ (with $k \leq n$), called *vertices* of σ: k is called the *dimension* of simplex σ. A *face* σ of a k–simplex γ is an h–simplex (with $0 \leq h \leq k$) generated by $h+1$ vertices of γ. Two simplexes are called *k–adjacent* if they share a k–face: in particular, two vertices (0–simplexes) are called *adjacent* if they are both incident at a common edge (1–simplex). A *simplicial complex* Σ is a set of simplexes in $\mathbb{E}^n$ of dimension at most d, with $0 \leq d \leq n$ such that:

- all the simplexes spanned by the vertices of a simplex in Σ are also in Σ;
- the intersection of two simplexes in Σ could be either empty or a shared face.

If all the top simplexes in Σ are d–simplexes then Σ is called *uniformly d–dimensional* or *regular*. Moreover, we say that a *h–path* in Σ (with $0 \leq h \leq d-1$) is a sequence of simplexes σ_i in Σ such that all the couples of consecutive simplexes are h–adjacent, while a subset $\Sigma^{'}$ of Σ is a *subcomplex* of Σ if it is a simplicial complex. Thus, Σ is called *h–connected* if and only for each couple of simplexes σ and σ^* in Σ exists an h–path that joins σ and σ^*.

A subset $\mathcal{M}$ of the Euclidean space $\mathbb{E}^n$ is called a *d–manifold* (with $d \leq n$) if and only if every point p of $\mathcal{M}$ has a neighborhood homeomorphic to B^d, while $\mathcal{M}$ is called a *d–manifold with boundary* (with $d \leq n$) if and only if every point p of $\mathcal{M}$ has a neighborhood homeomorphic either to B^d or to B^d intersected with a hyperplane in $\mathbb{E}^n$. If $\mathcal{M}$ does not fulfill these conditions at one or more points, it is called *non–manifold*.

2.2 Topological Relations

The connectivity information among the entities in a simplicial complex is expressed through the *topological relations*, providing an effective framework for a wide spectrum of existing data structures, that can be formally described in terms of the topological entities and relations that they encode. In this Section we define the topological relations for a cell complex, since the simplicial complexes can be considered as special instances of cell complexes.

Let Γ a d–complex and Γ^j the collection of all the j–cells in Γ with $j = 0, 1, \ldots, d$, then we can define $(d+1)^2$ ordered topological relations by considering all the possible ordered pairs (Γ^i, Γ^j) with $i, j = 0, \ldots, d$. If we denote by $\sim_{km}$ the relation for a pair $(\gamma, \gamma^{'})$ in $\Gamma^k \times \Gamma^m$, then:

- $\gamma \sim_{kk} \gamma^{'}$ (with $k \neq 0$) if and only if γ and $\gamma^{'}$ are $(k-1)$–adjacent in Γ;
- $\gamma \sim_{00} \gamma^{'}$ if and only if exists an 1–cell $\gamma^{''}$ in Γ such that $\gamma^{''} \in St(\gamma)$ and $\gamma^{''} \in St(\gamma^{'})$;
- $\gamma \sim_{km} \gamma^{'}$ (with $0 \leq m < k$) if and only if $\gamma^{'} \in B(\gamma)$, i.e. $\gamma^{'}$ belongs to the boundary of γ: this relation is called *boundary* relation;
- $\gamma \sim_{km} \gamma^{'}$ (with $0 \leq k < m$) if and only if $\gamma^{'} \in St(\gamma)$, i.e. $\gamma^{'}$ is bounded by γ: this relation is called *coboundary* relation.

The relations $\sim_{kk}$ and $\sim_{00}$ are also called *adjacency* relations, while the boundary and the coboundary relations are often called *incidence* relations. For each relation $\sim_{km}$, we define a relational operator $\mathcal{R}_{k,m} : \Gamma^k \rightarrow \mathcal{P}(\Gamma^m)$ such that $\mathcal{R}_{k,m}(\gamma) = \{\gamma^{'} \in \Gamma^m . \gamma \sim_{km} \gamma^{'}\}$: this operator provides, for each k–cell γ, the m–cells that are in relation $\sim_{km}$ with γ.

We call *constant* a relation that involves a constant number of entities, while relations that involve a variable number of entities are called *variable*: in a simplicial complex the coboundary and the adjacency relations are variable, while the boundary relations are constant. Thus, we say that an algorithm for retrieving a topological relation $\mathcal{R}$ is *optimal* if and only if it retrieves $\mathcal{R}$ in time linear in the number of entities involved in $\mathcal{R}$.

3 State of Art

In this Section we give an overview about the research fields interested by our research: since many data sets could not fit in RAM, we should apply some out–of–core techniques in order to manage huge meshes. In the Section 3.1 we focus our attention on the *spatial indexing techniques*, the most suitable for our objectives. In the Section 3.2 we review some EM simplification techniques based on the *mesh decomposition*: the analysis of these methods is useful for defining the requirements of a common framework that supports a wide range of spatial indexing structures.

3.1 Spatial Indexing Techniques

In this Section we review the most important properties of the spatial indexing data structures, useful to perform some operations on the spatial data. In literature there has been a tremendous amount of work about these structures and it is not possible to cover it completely in this document: moreover, in our research we are interested only in a particular class of indices, called *space partitioning trees*, since they subdivide the space in disjoint partitions and this property is suitable for the design of an EM simplification technique.

Emerging database applications may need a large variety of data being supported, in general *multidimensional data*: usually, they are a collection of points in high dimensional space and they can represent locations in a real space as well as more general records. In this research, we consider an unstructured collection of geometric objects, embedded in the Euclidean space: these data are usually called *spatial data* and a database system that manage them is called *spatial database*. Hence, we must introduce some *access structures* (or *indices*) in order to efficiently access the spatial data: these techniques provide an efficient organization of the input data, in connection with the sorting. Since the nature of spatial data, a spatial indexing structure is needed to achieve a high degree of success in each of the following aspects:

- efficient query, called also *objects retrieval*;
- efficient storage – the index overhead should not be greater than the data itself;
- efficient update – the index has to allow objects changes/updates, i.e. removals and insertions, although these operations could be very expensive.

The relative frequencies of the operations are application–dependent: thus, we must study separately the performance of each operation in order to estimate the effect on overall performance of a particular structure. A spatial indexing structure that can achieve optimal success for all the metrics is difficult to find (if not impossible): for example, the *BSP–tree*, introduced in [FKN80], is very efficient in the objects storage and retrieval, but it does not efficiently support the objects updates. In literature, there has been a large amount of work about the spatial indexing data structures and it is not possible to cover it in this document: in [Sam06] the author proposes an excellent survey about these structures, by reviewing the foundations

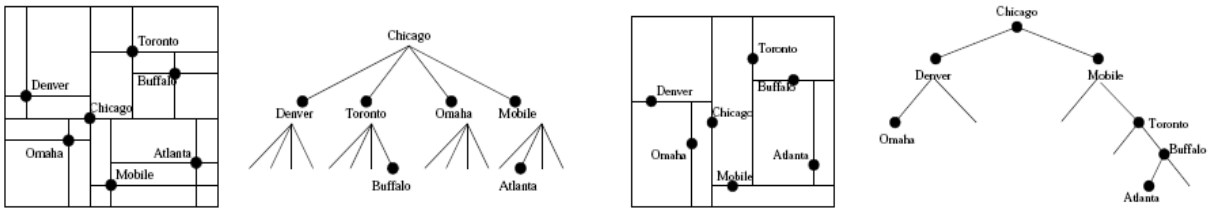
of multidimensional and metric access data structures. In this paper, we focus our attention on a particular class of indices, called *space partitioning trees*: they are hierarchical data structures that decompose the space into disjoint regions, called *buckets*, in a similar way than the divide–and–conquer methods. In these data structures each bucket stores all the objects contained in its corresponding region, thus the most suitable representation is the *multi–way tree*, where each node describes a bucket. We call *leaf* an atomic bucket (i.e. a bucket that is not divided), while we call *internal node* an intermediate node (i.e. a bucket that is divided). Usually, the data are contained only in the leaves, while the internal nodes are useful to guide the data access: however, this property is not always true, since it can change from an index to an other. In such structures, the number of partitions and the resolution (i.e. the number of times that the decomposition process is applied) may be fixed beforehand or it may be governed by some properties of the input data, in fact we can remember:

- *space–driven* partitioning trees, where the space is decomposed regardless of the data distribution: famous examples of such indexes are the *region quadtree*, the *PM quadtree* and the *PR quadtree* (see respectively [FB74], [SW85] and [Ore82] for more details).
- *data–driven* partitioning trees, where we split the dataset using some criteria, dependent from the data distribution: famous examples of such indexes are the *point quadtree* and the *point k–d tree* (see respectively [FB74] and [Ben75] for more details).

The *quadtree* and the *k–d tree* are perhaps the most common spatial indexes in the literature, since they have been described using both the space–driven and both the data–driven approach.

Quadtree – The quadtree is a multdimensional search tree, that can be considered as the generalization of a binary search tree. It is a multiway tree where each node is divided in 2^k hyper–rectangles, where k is the dimension of the space where we consider the input data: for example, in $\mathbb{E}^2$ each node has four children, as depicted in the Figure 1(a), that shows an example of a point quadtree. You should refer to [FB74] and [Sam06] for more details about this index.

K–d tree – The k–d tree is a multidimensional search tree, useful for answering queries about a set of points in the k–dimensional space: it is a binary search tree with the distinction that at each level a different dimension (called *discriminator key*) is tested to determine the dimension in which a branch is made. For the k–dimensional space at level l, the dimension number $l \bmod k + 1$ is used, where the root is at level 0: thus, the first dimension is used at the root, the second dimension at level 1 and so on until all the dimensions have been used and the dimensions are used again beginning at level k. The Figure 1(b) shows an example of a point k–d tree, while you should refer to [Ben75] and [Sam06] for more details about this index.



(a) (b)

Figure 1: the spatial decompositions and the tree representations for two spatial indexes: (a) a point–quadtree and (b) a point k–d tree. Figures courtesy of [Sam06].

Many indexing techniques, aiming at solving disparate problems and optimized for different types of spatial queries, have appeared lately in the literature and each technique has specific advantages and disadvantages, that make it suitable for different application domains and datasets. Therefore, the task of selecting an appropriate access method, depending on particular application needs, is a rather challenging problem: a spatial index library that can combine a wide range of indexing techniques under a common application programming interface can thus be a valuable tool, since it will enable efficient integration of a variety of structures in a consistent and straightforward way. Unfortunately, a similar framework is not available, according to our experience: towards the definition of a framework with such properties, in the Section 4 we propose an extension of the *OMSM* framework, that we have introduced in [Can07].

3.2 *EM* Simplification Techniques based on the Mesh Decomposition

In this Section we review some EM simplification techniques based on the *mesh decomposition*: these methods split the input mesh into separate patches that are small enough to be simplified individually in–core by using a conventional simplification algorithm, as those described in [SZL92], [Hop96], [GH97], [PS97] and [LRC+02]. One of the most common simplification operators is the *edge–collapse*: given an edge $e = (v_u, v_t)$, this coarsening operation (usually indicate with *ecol*) contracts v_u and v_t into a single vertex v_s, the two faces incident at edge e vanish and the other triangles incident in v_u and v_t become incident in the new vertex v_s. This update is invertible and the inverse operator is called *vertex–split*, usually indicated with *vsplit*: the Figure 2 shows the updates introduced by the edge–collapse and the vertex–split trasformations on the input mesh.

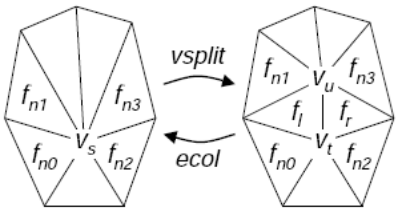


Figure 2: the vertex–split refinement operation (*vsplit*) and its inverse, the edge–collapse coarsening operation (*ecol*). Figure courtesy of [Hop98].

One of the first algorithms has been described in [Hop98] and it is designed for the simplification of terrain models: in this method, the input mesh is hierarchically divided in blocks and then each block is simplified by collapsing edges that are not incident on the boundary of a block. Once that each block has been simplified, the algorithm traverses bottom–up the hierarchical structure by merging sibling cells and again simplifying. The Figure 3 shows a rough scheme of this process.

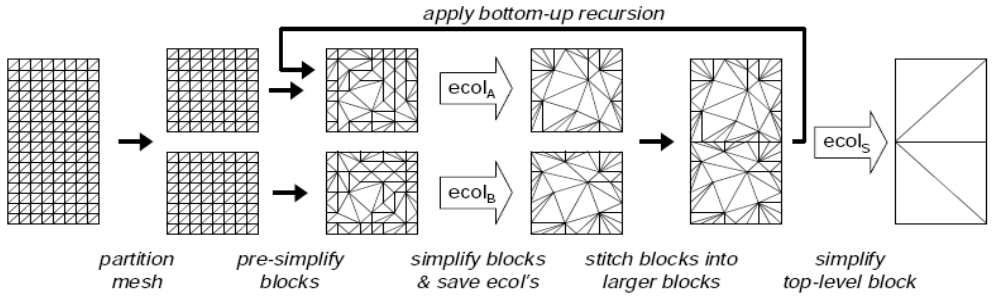


Figure 3: a rough scheme of the simplification technique described in [Hop98]. Figure courtesy of [Hop98].

This approach has either a bottleneck on the output size (because a complete bottom–up traversal of the structure is required to remove elements incident on the inter–cell boundaries) or on the simplification accuracy (intermediate results present unpleasant runs of original high resolution elements located on cells boundaries). Moreover, this approach cannot be extended easily to support other geometric processing tasks, because the elements shared by adjacent blocks cannot be modified unless the blocks are merged. In [Pri00] the author has generalized this method to arbitrary triangular meshes, mantaining a similar organization: unfortunately, the new method suffers the same disadvantages over the inter–blocks boundaries.

In [ESC00] the authors propose an algorithm that works on irregularly distributed data and subdivides the mesh at full resolution into patches. Each path is described by an indexed mesh with explicit topology and all the edges (with the adjacent faces) are kept into an external memory heap, ordered according an error criterion based on the edge length, although the edge length does not ensure high accuracy in simplification. However, implementing an ordering criteria based on the QEM (described in [GH97]) is not easy, due to the complex evaluation of the QEM for each edge. Given k edges on top of the

heap (k is a value depending on the core memory size), this method loads in RAM the associated adjacent faces pairs, reconstructs the mesh portions spanned by these edges and collapses all the edges that have their incidents faces in RAM. This approach reaches a good computational efficiency if we are able to load in RAM large contiguous regions: unfortunately, the shortest edges could be uniformly scattered and it could happen that many spanned sub–meshes loaded in RAM consist of few triangles, therefore requiring a very frequent loading/unloading of very small regions. A positive advantage of this method is that the simplification order performed by the external memory implementation is exactly identical to the one used by an analogous in–core solution, thus the mesh produced by the external memory implementation is identical to the one of the in–core solution.

Another important technique based on the space partitioning is described in [CMRS03], where the authors propose a simplification method for triangle meshes embedded in $\mathbb{E}^3$ by using the *OEMM* data structure (short for *Octree–based External Memory Mesh*): the octree subdivision stops when the set of triangles associated to a leaf fits in a disk page. Each mesh portion, contained in a leaf, is independently simplified through iterative edge collapses: once simplified, the leaves can be merged and simplified further, like in [Hop98]. The problem on the boundaries among the portions has been resolved: the subdivision phase is performed using octree–based regular splits and the elements spanning adjacent cells are identified in the construction phase. Hence, the boundary elements contained in the interior of the loaded region can be treated as any other element thanks to a *tagging strategy*, that allows an easy detection and management of the elements located on the boundary of the current region.

Beside the specific limitations of each one of the EM techniques, most of them have been designed to support just simplification, and extending these approaches to support also other geometric processing algorithms can be not straightforward. However, it is clear that great attention must be paid to the subdivion step and in particular on the patches boundaries in order to maintain the topological consistency of the simplified mesh. Thus, it is important to choose a subdivision scheme that produces only disjoint partitions (i.e. partitions that overlap only over their boundary) in order to simplify the operations over the input mesh: as consequence, the space partitioning trees are a good choice for these methods.

4 An Extensible Storage Architecture

In this Section we propose an extension of a flexible framework for managing huge geometric models, called *OMSM* (short for *Objects Management in Secondary Memory*), that we have introduced in [Can07]: in the Section 4.1 we analyze the requirements of a storing architecture for spatial data, focusing our attention on some aspects that are important for our research. In the Section 4.2 we give a detailed description of the *OMSM* framework, explaining how it satisfies the above requirements.

4.1 The Requirements of a Storing Architecture for Spatial Data

In this Section we analyze the requirements of a storing architecture for spatial data, focusing our attention on some important aspects in our research.

As we have already observed in the Section 3.1, the emerging database applications require new indexing structures beyond the B–tree, a famous database index introduced in [BM72]: in fact, the new applications may need different index structures to suit the big variety of data being supported, i.e. video, images and multidimensional data. In this paper we limit our analysis to spatial data, i.e. data with a location component: thus, efficient data structures are highly needed to facilitate access and to support the different spatial queries against these data. Many indexing techniques, aiming at solving disparate problems and optimized for different types of spatial queries, have appeared lately in the literature and each technique has specific advantages and disadvantages, that make it suitable for different application domains and datasets. Hence, the task of selecting an appropriate access method, depending on particular application needs, is a rather challenging problem: a spatial index library that can combine a wide range of indexing techniques under a common application programming interface can thus be a valuable tool, since it will enable efficient integration of a variety of structures in a consistent and straightforward way. The major issue of such an undertaking is that most part of the indexing structures have a wide range of distinctive properties, difficult to compromise under a common framework. Another important issue is that the indexing structures should provide the functionalities for exploiting the semantics of application–specific data types through easy customization, while making sure that the meaningful queries can still be formulated for the specific data types. Moreover, it is crucial to adopt a common programming interface in order to promote reusability, easier maintenance and code familiarity, especially for large projects

where many developers are involved. A such framework should capture the most important design characteristics into a concise set of interfaces: this design will help developers to concentrate on other aspects of the client applications, promoting faster and easier implementation. The interfaces should be easily extensible in order to address future needs without necessitating revisions to client code: a similar framework should provide a dynamically extensible plugins system that can support not only the existing access techniques, but also the future ones.

In literature many examples of generic frameworks for spatial indexes have been proposed: for example, we can rembember the *XXL* (short for the *eXtensible and fleXible Library*, see [vdBBD$^+$01] for more details). This framework offers some components for the development and for the integration of spatial index structures, for the access to raw disks and for the optimization of a query. Moreover, the *XXL* can support a large variety of advanced spatial queries by generalizing an incremental best–first search query strategy: unfortunately, its querying interfaces are index–specific thus if we define custom queries, we must implement them by hand, modifying all the affected index structures. Moreover, the *XXL* library does not support different storage requirements and does not decouples the index structure implementation from the storage system: in other words, it is not easily extensible and customizable.

The *GiST* (short for *Generalized Search Tree*, refer [HNP95] for more details) library contains an other framework relevant for our research: it generalizes a height–balanced and single–rooted search tree with variable fanout. In essence, the *GiST* is a parameterized tree that can be customized with user–defined data types and functions defined on such types, that guide the structural and searching behavior in the tree. Each node consists of a set of predicate/pointer pairs: the *pointers* are used to link a node with its children, while the *predicates* are the user–defined data types stored in the tree. The user, apart from choosing a predicate domain (e.g., the set of natural numbers, rectangles or a unit square universe and so on), must also implement some methods (i.e., *consistent*, *union*, *penalty* and *pickSplit*), that are used internally by *GiST* to control the behaviour of the tree. By using a simple interface, *GiST* can support a wide variety of search trees and their corresponding querying capabilities, including *B–trees* (see [BM72] for more details) and the *R–trees* (see [Gut84] for more details). Unfortunately, the class of space partitioning trees, rewieved in the Section 3.1, is not supported by the *GiST*: moreover, this project is not supported anymore and the current implementation does not work over the recent platforms.

The *SP–GiST* (short for *Space–Partitioning GiST*, see [AI01] for more details) is an important extension of the *GiST*: it provides a novel set of external interfaces in order to furnish a generalized index structure that can be customized to support a large class of spatial indexes with different structural and behavioral properties. This framework allows the creation of the space–driven and the data–driven partitioning structures and also of balanced and unbalanced structures: the *SP–GiST* also supports the *k–D trees*, the *tries*, the *quadtrees* and their variants (see respectively [Ben75], [Fre60] and [FB74] for more details). Unfortunately, the *SP–GiST* is designed to work only inside the *PostgreSQL* database server and it supports only bidimensional data.

Thus, according to our experience and by analyzing the current solutions, the design of a storage architecture for spatial objects should be based on these three aspects:

- a *spatial index* for improving the operations performance, in particular a space partitioning tree;
- the *subdivision* of the index nodes into clusters according to a certain policy: for us, a cluster is a set of nodes that we can consider as an atomic unit (according with a *nodes clustering policy*). In this context, an I/O operation must be performed on a single cluster.
- the *dynamic management* of clusters among a storage support and the RAM.

These three aspects can be considered independent and the techniques used for their management may be combined, obtaining different storage architectures. Some storage architectures, currently available in literature, provide a range of a–priori choices, especially as regards the last two aspects, while it is possible to customize only the indexing structures: for example, the *GiST* framework allows only to customize the indexing structures and can manage only a local database, i.e. resident on the local machine. Unfortunately, a similar framework is not available, according to our experience: it would be very attractive from the point of view of database system since the implementation of a full fledged indexing structure with the appropriate concurrency and recovery mechanism is not a trivial process.

4.2 The OMSM Framework Description

Towards the definition of a completed framework with the properties that we have discussed in the Section 4.1, in this Section we describe an extended version of the *OMSM* (short for *Objects Management in Secondary Memory*) framework, that we have introduced in [Can07].

The *OMSM* framework can manage a huge set of spatial objects with different topological properties, but embedded in the same Euclidean space: in the Section 4.2.1 we describe how the spatial objects are managed and stored. The *OMSM* framework has a layered organization, depicted in the Figure 4.

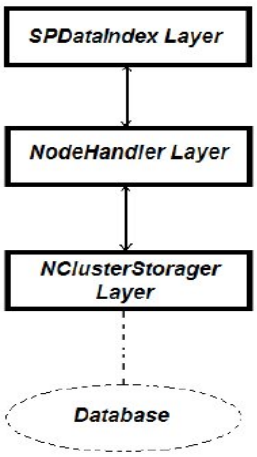


Figure 4: the layers of the *OMSM* framework: the dotted lines with the database clusters remark that we do not know its services set because we do not know how it works, until we have chosen it.

Each layer, called *OMSM layer*, is targeted to the management of one of the three aspects described in the Section 4.1 and the behavior of the *OMSM* framework can be described as interaction among these three layers: moreover, we assume that each layer is independent from the other ones. The layers that belong to this framework are:

- the *SPDataIndex* layer, where the user can choose the *space partitioning tree* to be used and can execute queries and updates: we give a detailed description of this layer in the Section 4.2.2.
- the *NodeHandler* layer, where the user can modify the *nodes clustering policy* to be used, useful for minimizing the number of I/O operations: we remember that a cluster is a set of nodes that can be considered as an atomic unit. In this context, an I/O operation must be performed on a single cluster: we give a detailed description of this layer in the Section 4.2.3.
- the *NClusterStorager* layer, where the user can choose techniques used to dynamically manage the clusters among storage supports and RAM (especially at low–level): we give a detailed description of this layer in the Section 4.2.4.

The organization of this framework could seem quite trivial, but the novelty of our approach lies in its design: a large part of the design effort is devoted to the definition of a deplayment system capable to deal with a wide array of different techniques. This goal has been reached by providing a plugin–based multichannel engine: different plugins are available for the different tasks and the third–party applications (i.e. the plugins) are able to connect and take advantage of the functions they require in a fully decoupled and well–documented fashion. Each layer offers a set of services, described by a well–established interface: the basic *OMSM* work–flow is to instantiate a plugin and then establish the *plugin connection* through the required interface by registering it in the framework. In this way, we can have a lightweight implementation of the framework, focusing on the communication interface between the plugins (i.e. dynamic libraries), while we can specialize the behavior of each layer. This generalization process is also valid for the spatial objects thus we can say that an *OMSM plugin* can be either a specialization of one the three layers in the Figure 4 or a specialization of a spatial object: moreover, a *OMSM plugin* is coupled with a XML description in order to simplify the registration process. As conseguence, the *OMSM* framework is modular and easily

extensible: in fact, it provides a dynamically extensible plugins system that can support not only existing access techniques, but also the future ones. Thus, by writing the appropriate extension to this framework, a new technique can be made available without messing with all the structure and the user can choose how the available plugins must be combined, by adapting the *OMSM* framework to his needs.

However, this approach is orthogonal to the generic frameworks that have discussed in the Section 4.1: these frameworks address the implementation issues behind new access methods by removing the burden of writing structural maintenance code from the developer. The *OMSM* framework does not aim to simplify the development process of the index structures per se, but more importantly, the development of the applications that use them: in that respect, it can be used in combination with all other index structure developer frameworks. Employing *OMSM* is an easy process that requires the implementation of simple adapter classes (plugins) that will make index structures (and structures for nodes clustering and clusters storagers) compliant to its interfaces, conflating these two conflicting design viewpoints. Ostensibly, existing libraries can be used for simplifying client code development as well – and share, indeed, some similarities with *OMSM* (especially in the interfaces for inserting and deleting elements from the indices) – but, given that they are not targeted primarily for that purpose, they are not easily extensible (without the subsequent need for client code revisions), they do not promote transparent usage of diverse indices (since they use index specific interfaces) and they cannot be easily used in tandem with any other index library (existing or not).

The implementation of the *OMSM* framework has required a considerable amount of work and it is contained in the *OMSM Library*, written in C++: this library satisfies the POSIX standards and it is supported by the most common platforms like *GN/Linux*, *Apple MacosX* and *Microsoft Windows*.[1] Moreover, this library also contains a set of built–in *OMSM plugins*:

- some plugins for the point–quadtree, the point k–d tree, the hybrid PR k-d trie and the hybrid PR quadtrie: you can refer [Sam06] for more details about these indexing structures;
- a plugin for the single–node policy clustering policy (i.e. a cluster contain only one node);
- a plugin for storing clusters in an embedded database: an embedded database system is a *DBMS* (short for *DataBase Management System*), that is tightly integrated with an application software that requires access to stored data in such a way that the database system is *"hidden"* from the application end–user, requiring little or no ongoing maintenance. In this plugin we use the famous *Oracle Berkeley DB* (see [Bdb06] for more details): it is suitable for the data model of this layer (see the Section 4.2.4), since it stores arbitrary key/data pairs as byte arrays.
- some plugins for storing points, segments, triangles and polygons (refer to the Figure 6 for more details): in these plugins, we consider a geometric model as an unstructured set of independent geometric entities. However, this approach is limited and not suitable for our objectives since a geometric model is usually described by a simplicial complex and by its topological relations, as described in the Section 2.2.

Each plugin can be dynamically loaded in the *OMSM* framework, by using an XML description: thus, the rough prototype that we have developed can be expanded in many directions. This library will be released as soon as possible with an Open Source License.

4.2.1 The *OMSM* spatial objects

In this Section we give a detailed description of the spatial objects management in the *OMSM* framework: we focus our attention on the low–level representation and on the services offered by a single object. In order to describe some technical aspects, we use a object–oriented pseudocode.

One of the most important problems in a storing architecture (like the *OMSM* framework) is the *persistence* of the data to be managed. Usually the data to be stored are described by objects, according to the object–oriented paradigm: for example, they are C++ or Java objects. In this context, we plan to make persistent all the modifications applied to the database: moreover, we also require to fetch objects from the storage support and to load them in RAM. In our case, the objects to be managed represent spatial data, hence we are also interested in their indexing in a space partitioning tree and in extracting geometric properties from them: as consequence, we must decouple the persistence properties from the spatial ones in order to obtain

[1] All the rights and the registered trademarks belong to their owners.

a flexible and extensible system in the same spirit of the *OMSM* framework. A general solution is based on the definition of a hierarchy of interfaces: each interface offers a set of services and the user can easily write an extension, that satisfy these interfaces. In other words, we apply the same technique used for the layers implementations.

The first step is the definition of an interface common to all the objects, that must be stored in the *OMSM* framework. In literature there is not a common solution for ensuring the objects persistence or it is often language or platform dependent: for example, the C++ language does not support the objects persistence and furthermore we must refer to third–party solutions like those described in [Cor91], [Net02] and in [Bdb06]. To reach our goal, we have designed a persistence system based on the representation of an object $\mathcal{O}$ through a bytes sequence, mantained in little–endian order. Each object $\mathcal{O}$ has an internal state, composed by some internal fields: the sequence of bytes that describes $\mathcal{O}$ can be obtained by concatenating the descriptions of the internal fields in a certain order (the order is not important). This persistence system is described by the *RawData* interface, that offers the following services:

```
bytes_sequence getBytes()

int getBytesNumber()
```

The *getBytes* service (a method in the object–oriented paradigm) extracts the bytes sequence that describe the object, while the *getBytesNumber* method returns its length, expressed in bytes. The use of these services is straightforward when we want to store an object, while the construction of a new object (from a bytes sequence that could describe it) is delegated to the plugin that satisfy this interface, since the adopted representation details are well–defined only inside the plugin: in other words, the internal state and the order of each field in the bytes sequence will be well–defined only inside a plugin, provided by the user. However, the *RawData* is not directly exported for plugins: it is more convenient to export only the interfaces specialized for particular targets, for example for spatial objects. In other words, the *RawData* interface is useful for defining a common set of services that must be satisfied by a generic object in the *OMSM* framework: the Figure 5 shows the complete hierarchy of interfaces, that describe all the objects to be stored in the *OMSM* framework.

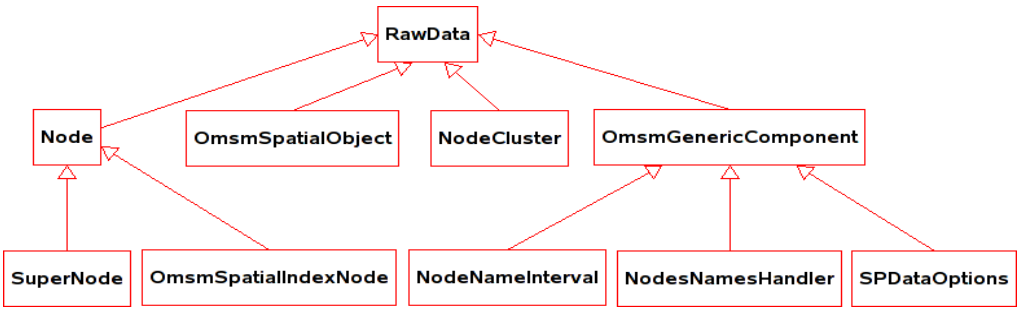


Figure 5: the complete hierarchy of all the interfaces that extend the *RawData* one and that describe all the objects that we can store in the *OMSM* framework.

It is clear that we must store not only spatial data, but also other types of data, like indexes nodes, clusters and so on: in order to catch this objective, we propose many extensions of the *RawData* interface: the subtree with *Node* as root contains interfaces specialized to the nodes management (we give a complete definition of these interfaces in the Section 4.2.2, the subtree with *OmsmSpatialObject* as root contains interfaces specialized to the spatial objects, while the *NodeCluster* interface describes a cluster of nodes (we give a complete definition of this interface in the Section 4.2.3). Finally, the subtree with *OmsmGenericComponent* as root contains interfaces that describe a unstructured collection of generic objects that can be stored in the *OMSM* framework: the most important interface is the *SPDataOptions* one, that is useful for customizing the framework. These interfaces do not offer relevant services: you should refer to [Can07] for more details.

In the *OMSM* framework it is possible to store geometric and spatial objects, thus we must also guarantee the possibility to query and to extract the properties of a spatial object, not only to make it persistent. Given a spatial object $\mathcal{S}$, it must be possible (at least):

- to understand the dimension d of the Euclidean space $\mathbb{E}^d$ where $\mathcal{S}$ is embedded;

- to extract the geometric aspects of $\mathcal{S}$ (for example the vertices of $\mathcal{S}$);
- to compare $\mathcal{S}$ with an other spatial object;
- to extract a d–dimensional point, called *representative point*: this point identifies $\mathcal{S}$ and it will be used inside a space partitioning tree (refer to the Section 4.2.2 for more details).

Hence, we extend the *RawData* interface, introducing the *OmsmSpatialObject* one: it is a more specific interface, targeted for the spatial objects management. In addition to the services offered by the *RawData* interface, it provides (at least) the following methods:

```
int getTopologicalDimension()

int getSpaceDimension()

Point getRepresentativePoint()

list<Point> getEuclideanVertices()

boolean sameObject(OmsmSpatialObject o)
```

Let $\mathcal{S}$ a spatial object, then:

- the *getTopologicalDimension* method returns the topological dimension of $\mathcal{S}$ (refer to the Section 2 for more details);
- the *SpaceDimension* method returns the dimension d of the Euclidean space $\mathbb{E}^d$ where $\mathcal{S}$ is embedded: usually, it is 3;
- the *getRepresentativePoint* returns the d–dimensional representative point of $\mathcal{S}$;
- the *getEuclideanVertices* method returns all the vertices of $\mathcal{S}$;
- the *sameObject* method compares $\mathcal{S}$ with an other spatial object o, checking if they are equal: the output value, returned by this service, communicates the outcome of this operation in a trivial way.

Since this interface is specialized for spatial and geometric objects, the *OmsmSpatialObject* interface is exported as plugin: each plugin that describes a specific spatial object must implement this common interface and must have a XML description in order to be registered in the *OMSM* framework. The Figure 6 shows a possible hierarchy of interfaces that extend the *OmsmSpatialObject* one in order to describe specific spatial objects (i.e. points, lines, triangles,...).

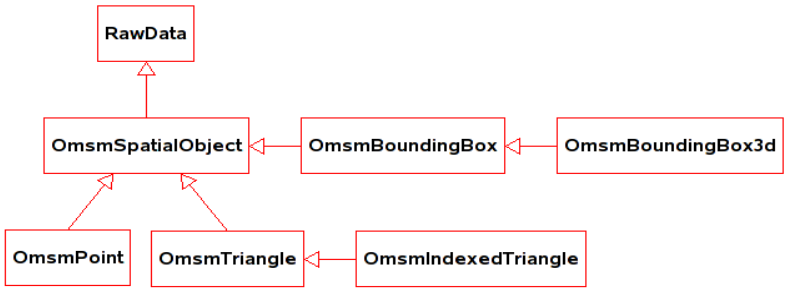


Figure 6: a possible hierarchy of the interfaces that extend the *OmsmSpatialObject* one in order to describe specific spatial objects in the *OMSM* framework.

In particular, all the built–in plugins provided by the *OMSM Library* are shown in the Figure 6:

- the *OmsmPoint* interface describes a multi–dimensional point;

- the *OmsmTriangle* and *OmsmIndexedTriangle* respectively describe a plain triangle and an indexed triangle (i.e. a triangle composed by three references for its vertices);
- the *OmsmBoundingBox* and *OmsmBoundingBox3d* respectively describe a dimension–independent hyper–rectangle and a specialized box in the Euclidean space $\mathbb{E}^3$.

In these plugins, we consider a geometric model as an unstructured set of independent geometric entities: however, this approach is limited and not suitable for our objectives since a geometric model is usually described by a simplicial complex and by its topological relations, as described in the Section 2.2.

These representations could seem very "poor", but they are useful for reducing the input object complexity and for providing a generic data model, that can hide implementative details: for example, in the Section 4.2.4 we reduce a complex object like a cluster to a plain sequence of bytes.

4.2.2 The *SPDataIndex* layer

In this Section we give a detailed description of the services offered by the *SPDataIndex* layer, the most external one in the *OMSM* framework, as depicted in the Figure 4. In order to describe some technical aspects, we use a object–oriented pseudocode.

The *SPDataIndex* layer allows the user to interact with the data stored in the *OMSM* framework by offering a set of services. Moreover, it hides the implementative details of the spatial data management: in fact, an user can create an instance of this layer and then it is possible to invoke its services without knowing anything about its implementation. The main objective of this layer is to efficiently manage a huge set of spatial objects, described by the *OmsmSpatialObject* interface (see the Section 4.2.1 for more details) and to provide a custom implementation of a space partitioning tree. In other words, this layer provides an abstract description of a component that receives some instances of the *OmsmSpatialObject* interface (i.e. the spatial objects to be managed) as input and produces the nodes of the corresponding space partitioning tree, that must be stored in the *NodeHandler* layer (see the Section 4.2.3 for more details). These goals have been obtained by providing a suitable data model that allows to discard all the implementative details about the specific index and the spatial object that we are managing.

In this layer, the data model is a generic node of a space partitioning tree, described by the *OmsmSpatialIndexNode* interface: a similar node can contain a set of a spatial objects, that can be indexed by using their representative point. In this way, all the spatial objects can be represented by their representative point, by reducing their complexity and this design choice is totally transparent for the other layers of the *OMSM* framework. The instances of the *OmsmSpatialIndexNode* interface must be stored in the *OMSM* framework, thus this interface must extend the *RawData* one in order to guarantee their persistence: moreover, it must also extend the *Node* interface, as depicted in the Figure 7: in the Section 4.2.3 we give a detailed motivation of this design choice. In addition to the services offered by the *RawData* interface, the *OmsmSpatialIndexNode* provides the following services:

```
void addSpatialObject(OmsmSpatialObject o, int lev)

void removeSpatialObject(OmsmSpatialObject o, int lev)

boolean searchSpatialObject(OmsmSpatialObject o, int lev)

node_id forward(OmsmSpatialObject o, int lev)

list<OmsmSpatialObject> getObjects()

boolean isLeaf()
```

Let $\mathcal{N}$ a node of a space partitioning tree, then:

- the *addSpatialObject* method adds a spatial object *o* in $\mathcal{N}$, that is currently located at level *lev* (i.e. if $\mathcal{N}$ is the root then *lev* will be 0): this service could require some further splitting operations in the current space partitioning tree;

- the *removeSpatialObject* method removes the spatial object o from $\mathcal{N}$, that is currently located at level *lev*: this service could require some expensive operations for reorganizing the current space partitioning tree;
- the *searchSpatialObject* method looks for the spatial object o in $\mathcal{N}$, that is currently located at level *lev*: this service could require a recursive visit on the subtree rooted at $\mathcal{N}$, if $\mathcal{N}$ is an internal node (i.e. a node that is splitted and does not contain any objects);
- the *forward* method returns the identifier of the node where we can probably found the spatial object o, if a such node exists: this node is a child of $\mathcal{N}$ (currently located at level *lev*). If $\mathcal{N}$ is a leaf (i.e. an atomic node that is not splitted anymore and that usually contains some objects), then this operation is undefined;
- the *getObjects* method returns a list containing all the objects stored in $\mathcal{N}$: if $\mathcal{N}$ is an internal node, then this operation is undefined;
- the *isLeaf* method checks if $\mathcal{N}$ is a leaf: the output value, returned by this service, communicates the outcome of this operation in a trivial way.

Since this interface is specialized for nodes of a generic spatial partitioning tree, the *OmsmSpatialIndexNode* interface is exported as a plugin and each plugin that describes a specific node must implement this common interface and must have a XML description in order to be registered in the *OMSM* framework. The Figure 7 shows a possible hierarchy of interfaces that extend the *OmsmSpatialIndexNode* one in order to describe specific indexes, in particular, all the built–in plugins provided by the *OMSM* library are shown:

- the *OctreeIndexNode* describes a node of a point–quadtree, a data–driven space partitioning tree that we have reviewed in the Section 3.1: you should refer to [FB74] and [Sam06] for more details about this index.
- the *KdtreeIndexNode* describes a node of a point k-d tree, a data–driven space partitioning tree that we have reviewed in the Section 3.1: you should refer to [Ben75] and [Sam06] for more details about this index.
- the *BPRQIndexNode* and the *BPRKIndexNode* interfaces respectively describe a node of the *hybrid PR quadtrie* and of the *hybrid PR k-d trie*. These indexes reflect the structure similarity between the trie and the quadtree (or the k–d tree): each leaf node can contain a certain number c of objects at most and it must be splitted if there are more than c objects. Instead of placing a subdivision point in a fixed position (i.e. the domain midpoint), these indexes use the *sliding midpoint* technique (introduced in [MA97]) in order to minimize the number of empty leaves. You should refer to [MA97], [MM99] and [Sam06] for more details.

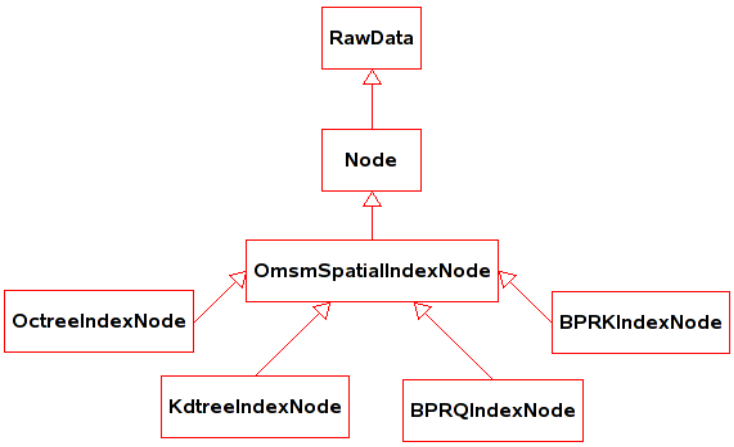


Figure 7: a possible hierarchy of the interfaces that extend the *OmsmSpatialIndexNode* one in order to describe specific indexes in the *OMSM* framework.

In order to minimize the memory requirements, we mantain in RAM only the root of the space partitioning tree, while the other nodes are dynamically loaded from the *NodeHandler* layer, according to the navigation needs: however, this design choice could be a bottleneck and we are investigating about it.

The external interface of the *SPDataIndex* is quite different from the original one, introduced in [Can07] since we consider only the particular node of a space partitioning tree as a plugin and not the whole layer. The new *SPDataIndex* interface offers a very limited set of services:

```
void open(string dbname, boolean createOn, boolean rdOnly)

void close()
```

The *open* method is useful for connecting the current instance of the *OMSM* framework with the data source, whose name is contained in the *dbname* string: in this service, the boolean flag *createOn* is used for enabling the construction of a new data source, while the *rdonly* flag is useful for blocking the modifications on the required data source. The *close* method disconnects the current instance of the *OMSM* framework from its data source: after invoking this method, an user cannot access to a data source anymore.

The *SPDataIndex* interface offers a limited set of services because we intend to provide a general querying system (not optimized for a specific application), by limiting the number of the available queries, that could be required by the applications. The *OMSM* framework could be extended in many directions, for example in order to support spatial queries for the *GIS* (short for *Geographic Information System*) applications, but also for managing a simplicial complex (see the Section 2 for more details): these two applications require different queries, thus the user can extend the *SPDataIndex* interface by adding custom queries and new capabilities. In this way, the *OMSM* framework becomes suitable to support (and to perform) some operations on geometric models no matter what modeling primitives or spatial data structures are used. As consequence, we offer a very concise and straightforward interface in order to query arbitrary index structures in a uniform manner with the use of standardized design patterns and to formulate novel queries without having to revise the library in any way.

4.2.3 The *NodeHandler* layer

In this Section we give a detailed description of the services offered by the *NodeHandler* layer, perhaps the most important one in the *OMSM* framework (refer to the Figure 4). In order to describe some technical aspects, we use a object–oriented pseudocode.

The *NodeHandler* layer has perhaps the major role in the *OMSM* architecture since it manages some aspects about the *nodes clustering*: we can group the spatial indexes nodes into clusters in order to improve the I/O bandwith. For us, a cluster is a set of nodes that could be considered as an atomic unit (according to a nodes clustering policy): each I/O operation, delegated to the *NClusterStorager* layer (see the Section 4.2.4 for more details), must be performed on a single cluster. Considering the physical storage of the tree nodes, a direct and simple implementation of the clustering, is to assign a cluster for each node: unfortunately, this simple assignment will not be efficient for very sparse nodes, as described in [DSS96]. In order to overcome these limitations, in literature there has been a large amoun of work about the nodes clustering: for example, we can remember the techniques described in [DSS96] and in [DdFPS06].

In this layer we allow the user to provide a custom nodes clustering policy, that can be more suitable for the type of operations to be performed on the current index, enhancing the query response time of the *OMSM*. In other words, this layer provides an abstract description of a component that receives some instances of the *OmsmSpatialIndexNode* interface (produced by the *SPDataIndex* layer, see the Section 4.2.2 for more details) as input and produces a set of clusters as output, managed by the *NClusterStorager* layer (see the Section 4.2.4 for more details). The nodes clustering policy, used in this process, is directly implemented in all the plugins specialized for this layer and it is *"hidden"* by the *NodeHandler* interface: this goal has been obtained by providing a suitable data model that allows to discard all the implementative details introduced in the other layers. In this layer, the data model is described by a common entity, called *Node*, that is useful for describing the two types of nodes in the *OMSM* framework:

- the indexing structures nodes, described by the *OmsmSpatialIndexNode* interface;

- a special node, called *SuperNode*: it contains critical informations about the current configuration of the *OMSM* framework and it can be considered as the super–block in a filesystem.

The instances of the *Node* interface, that does not offer relevant services, must be stored in the *OMSM* framework, thus this interface must extend the *RawData* one in order to guarantee the objects persistence: moreover, an instance of the *Node* interface has a generic code that identifies it. The Figure 8 shows all the inheritance relations among these interfaces.

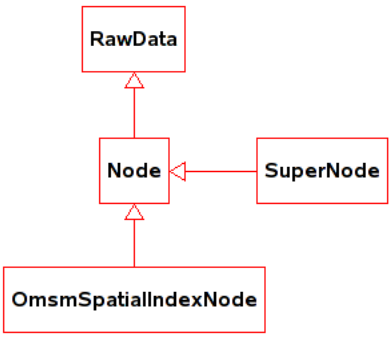


Figure 8: the hierarchy of interfaces that extend the *Node* one in order to describe all the nodes types in the *OMSM* framework.

Thus, the *NodeHandler* layer can manage only instances of the *Node* interface: in such way, we can discard all the implementative details introduced by the other layers, without worrying about the content of the *Node* instances. However, the unique constraint to be satisfied involves the *SuperNode*: the cluster used for its storage, called *Super–Cluster*, must not contain any other node and it will be always mantained in RAM. The clusters containing instances of the *OmsmSpatialIndexNode* interface could be destroyed since a LRU cache is used in order to maintain in RAM only the most recently used clusters.

Thus, in this layer we provide a user–defined clustering of nodes and each instance of the *Node* interface must be inserted in a cluster, described by the *NodeCluster* interface: this interface does not offer relevant services, but it must extend the *RawData* one, as depicted in the Figure 5, in order to guarantee its persistence. An important aspect of this interface is to provide an efficient and extensible representation as a bytes sequence: in this case, we have a good example of the capabilities of our framework. If a cluster $\mathcal{C}$ contains the $n_1, \ldots, n_k$ nodes and if we denote with $\overline{b}$ the extraction of a bytes sequence from an object b then the binary representation of the $\mathcal{C}$ content becomes:

$$\overline{\mathcal{C}} = \overline{k} \cdot \overline{l_1} \cdot \overline{n_1} \cdot \ldots \overline{l_k} \cdot \overline{n_k}$$

where $\| \ldots \|$ returns the length of a bytes sequence (expressed in bytes) and $l_i = \|\overline{n_i}\|$, for $i = 1, \ldots, k$. Since a generic node has been exported as a plugin (or it is the unique instance of the *SuperNode* in the framework), we obtain an user–defined representation of the cluster $\mathcal{C}$, easily extensible: in such way, we reduce this component to a sequence of bytes that can be managed by the *NClusterStorager* layer.

Now, we can analyze the services offered by the *NodeHandler* interface, that must be exported as a plugin:

```
void addNode(node_id nid, Node n)

boolean removeNode(node_id nid)

boolean searchNode(node_id nid)

Node getNode(node_id nid)
```

The *addNode* method is useful for adding a node *n* in the current nodes clustering by using *nid* as identifier of the new node. The *removeOne* service is useful for removing the node with *nid* as identifier from the current nodes subdivision: the output value, returned by this service, communicates the outcome of this operation in a trivial way. Furthermore, the *searchNode*

method is useful to look for the node with *nid* as identifier in the current nodes subdivision: the output value, returned by this service, communicates the outcome of this operation in a trivial way. Finally, the *getNode* method retrieves the node with *nid* as identifier from the current nodes subdivision, if it exists.

In this context, the most critical operation is the identification of the cluster that contains a node *n*, by using the *n* identifier. In order to catch our objectives, the *linear clustering* techniques are the most suitable, since they are based on the *locational codes*, that identify efficiently a cluster. Examples of such techniques are described in [Mor66], [Gar82], [FR89], [Jag90], [Sag94] and [TOL02].

4.2.4 The *NClusterStorager* layer

In this Section we give a detailed description of the services offered by the *NClusterStorager* layer, the last one in the *OMSM* framework, as depicted in the Figure 4. In order to describe some technical aspects, we use a object–oriented pseudocode.

The *NClusterStorager* layer manages the low–level representations of the nodes cluster, by operating on a storage support in an efficient way. Thus, this layer provides an abstract description of a storage support where we can read or write the clusters, obtained at the *NodeHandler* layer (see the Section 4.2.3 for more details). The type of storage support, directly implemented in the plugins specialized for this layer, is not important and it is *"hidden"* by its interface: for example, we can access to a remote database or write each cluster on an independent file. This goal has been obtained by providing a suitable data model that allows to discard all the implementative details introduced in the other layers. In this layer, the data model is a pair (id, ba) where:

- id a generic code that identifies a cluster: we do not assume a particular structure for this identifier;
- ba is the sequence of bytes that describe a cluster: a cluster is an object that can be stored in the *OMSM* framework, thus we can extract its representation in a straightforward way, by invoking the *getBytes* service (see the Section 4.2.1 for more details).

In such way, we can discard the implementative details and to apply additional transformations on the bytes sequence: for example, we can encrypt or compress it.

Now, we can analyze the services offered by the *NClusterStorager* layer:

```
void addCluster(cluster_id cid, byte_array ba, int lg)

boolean removeCluster(cluster_id cid)

boolean searchCluster(cluster_id cid)

void getCluster(cluster_id cid, out byte_array ba, out int lg)
```

The *addCluster* method is useful to add a cluster in the storage support: the input cluster has *cid* as identifier and it is described by the binary sequence *ba*, composed by *lg* bytes. The *removeCluster* is useful for removing the cluster with *cid* as identifier from the storage support: the output value, returned by this service, communicates the outcome of this operation in a trivial way. Furthermore, the *searchCluster* method is useful to look for the cluster with *cid* as identifier in the storage support: the output value, returned by this service, communicates the outcome of this operation in a trivial way. Finally, we can use the *getCluster* service in order to extract the cluster with *cid* as identifier from the storage support: if this cluster exists, its representation (composed by *lg* bytes) will be contained in *ba*, otherwise the output parameters will be undefined.

This interface could seem very poor, but it is suitable to manage clusters on a generic storage support: a specific plugin for this layer must implement this common interface in order to be registered in the *OMSM* framework.

5 Conclusions

Nowadays, many applications can require very large meshes, consisting of millions of simplexes: despite the rapid improvement in the hardware performances, these meshes often exceed the RAM size of a common workstation. The simplification techniques are a good solution for this problem, since they reduce the dimension of the input models: unfortunately,

we encounter some problems with these techniques, since the simplification algorithms usually require all the data in–core, while the input mesh itself can exceed the available RAM, even we are using a compact data structure. Hence, using an out–of–core technique is mandatory when we want to process an huge mesh: usually, we mantain the entire model in a storage support (i.e. a local hard–disk or a remote storage area network) and then we dynamically load in RAM only the selected sections that are small enough to be processed in–core. With this approach we can directly operate on these portions, using a limited memory footprint and removing the limit on the input size: in this case, the I/O among the memory levels is often a bottleneck because a disk access is slower than an access in RAM. These techniques are also called *EM* (short for *External Memory*) techniques and designing them is an active research area that has produced interesting results in many applications fields. In this paper, we have concentrated our attention on a particular class of the EM techniques, based on the *space partitioning*: in such techniques we subdivide the input model into hierarchical patches by using the *spatial indexing* structures, useful to efficiently access to the model portions. We assume that each portion can fit to RAM and that it is small enough to be managed in–core. Many indexing techniques aiming at solving disparate problems and optimized for different types of spatial queries have appeared lately in the literature and each technique has specific advantages and disadvantages that make it suitable for different application domains and datasets. Therefore, the task of selecting an appropriate access method, depending on particular application needs, is a rather challenging problem: a spatial index library that can combine a wide range of indexing techniques under a common application programming interface can thus prove to be a valuable tool, since it will enable efficient application integration of a variety of structures in a consistent and straightforward way. Unfortunately, a similar framework is not available, according to our experience.

Towards the definition of a framework with such properties, in this paper we have proposed an extension of the *OMSM* framework, that we have introduced in [Can07]: it is a modular and easily extensible framework that can manage an huge set of spatial objects. The *OMSM* provides a dynamically extensible plugins system that can support not only existing access techniques, but also the future ones: thus, by writing the appropriate extension to this framework, a new technique can be made available without messing with all the structure and the user can choose how the available plugins must be combined, adapting the *OMSM* framework to his needs. Our work is orthogonal to the current libraries and their variants: these frameworks address the implementation issues behind new access methods by removing the burden of writing structural maintenance code from the developer point of view. The *OMSM* framework does not aim to simplify the development process of the index structures per se, but more importantly, the development of the applications that use them: in that respect, it can be used in combination with all other index structure developer frameworks. Employing *OMSM* is an easy process that requires the implementation of simple adapter classes (plugins) that will make index structures compliant to its interfaces, conflating these two conflicting design viewpoints. Ostensibly, the existing libraries can be used for simplifying client code development as well – and share, indeed, some similarities with *OMSM* (especially in the interfaces for inserting and deleting elements from the indices) – but, given that they are not targeted primarily for that purpose, they are not easily extensible (without the subsequent need for client code revisions), they do not promote transparent usage of diverse indices (since they use index specific interfaces) and they cannot be easily used in tandem with any other index library (existing or not).

Since its plugins–oriented structure, the *OMSM* framework could be extended in many directions, for example in order to support some spatial queries for *GIS* (short for *Geographic Information System*): in fact, it is suitable to perform operations on geometric models no matter what modeling primitives or spatial data structures are used in the plugins. An important extension of this framework could be the management of a *simplicial complex*: it is not an unstructured set of simplexes (triangles,lines,points and so on), but we must consider its topology, that describes the connectivity information between its components. Usually, a simplicial complex is described by a topological data structure, thus this new improvement could be very important in the *OMSM* framework, because we could offer an unique platform in order to describe an out–of–core version of a topological data structure. As conseguence, we will provide an extensible framework that decouples the spatial indexing structure from the combinatorial description of a mesh, offering a more general solution than the solution described in [dFFMD08]. With this improvement, the *OMSM* framework will become a very general structure, capable of performing most spatial and topological queries on geometric models. We have implemented this framework in the *OMSM Library*: we plan to release this library as soon as possible under a Open Source license.

References

[ABA06] *The Allen Brain Atlas Project*, 2006. Organized by the *The Allen Institute for the Brain Science*, Official website: `http://www.brain-map.org`.

[Ago05] M. Agoston. *Computer Graphics and Geometric Modeling*. Springer–Verlag, 2005.

[AI01] W. Aref and I. Ilyas. SP–GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems (JIIS)*, 2001.

[AS94] P. Agarwal and S. Suri. Surface Approximation and Geometric Partitions. In *Proceedings of the* 5^{th} *ACM–SIAM Symposium about Discrete Algorithms*, 1994.

[Bdb06] *Oracle Berkeley DB*, 2006. Official website: `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[BEG94] W. Bern, D. Eppstein, and J. Gilbert. Provably Good Mesh Generation. *Journal of Computer and Systems Sciences*, Volume 48, June 1994.

[Ben75] L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, Volume 18, 1975.

[BM72] R. Bayer and E. McCreight. Organization and Mantenance of Large Ordered Indices. *Acta Informatica*, Volume 1, pages 173–189, 1972.

[BMR+99] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The Ball–Pivoting Algorithm for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, Volume 5, 1999.

[Can07] D. Canino. Semplificazione di Modelli Geometrici in Memoria Secondaria. Master's thesis, DISI, Universitá degli Studi di Genova, 2007.

[CBPS06] S. Callahan, L. Bavoil, V. Pascucci, and C. Silva. Progressive Volume Rendering of the Large Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, Volume 12, 2006.

[CFSW01] Y. Chiang, R. Farias, C. Silva, and B. Wei. An Unified Infrastructure for Parallel Out–of–Core Extraction and Volume Rendering of Unstructured Grids. In *Proceedings of IEEE Symposium on Parallel and Large–Data Visualization and Graphics*, 2001.

[CGG+04] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Out–of–Core Construction and Visualization of Gigantic Multiresolution Polygonal Models. In *Proceedings of the ACM SIGGRAPH Conference*, 2004.

[CGG+05] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. GPU–friendly Multi–Triangulation. In *Proceedings of the IEEE Visualization Conference*, 2005.

[Chi03] Y. Chiang. Out–of–Core Isosurface Extraction of Time–Varying Fields over Irregular Grids. In *Proceedings of IEEE Visualization Conference*, pages 217–224, 2003.

[CKS02] W. Correa, J. Klosowski, and C. Silva. *iWALK: Interactive Out–of–Core Rendering of Large Models*. Technical Report TR–653–02, Princeton University, 2002.

[CMRS03] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 9, 2003.

[Cor91] *The Common Object Request Broker Architecture*, 1991. Official website: `http://www.corba.org`.

[CS99] Y. Chiang and C. Silva. External Memory Techniques for the Isosurface Extraction in Scientific Visualization. In J. Abello and J. S. Vitter Editors, editors, *External Memory Algorithms and Visualization*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS Press, 1999.

[CSS98] Y. Chiang, C. Silva, and W. Schroeder. Interactive Out–of–Core Isosurface Extraction. In *Proceedings of the IEEE Visualization*. IEEE Press, 1998.

[DdF06] E. Danovaro and L. de Floriani. Generating, Representing and Querying Level–of–Detail Tetrahedral Meshes. In *Proceedings of Dagsthul Seminar on Scientific Visualization: Extracting Information and Knowledge from Scientific Data Sets*, 2006.

[DdFM+05] E. Danovaro, L. de Floriani, P. Magillo, E. Puppo, D. Sobrero, and N. Sokolovsky. The Half–Edge Tree: a Compact Data Structure for Level–of–Detail Tetrahedral Meshes. In *Proceedings of the International Conference on Shape Modeling*, 2005.

[DdFM+06] E. Danovaro, L. de Floriani, P. Magillo, E. Puppo, and D. Sobrero. Level–of–Detail for Data Analysis and Exploration: a Historical Overview and some new Perspectives. *Computer Graphics*, 30, 2006.

[DdFPS06] E. Danovaro, L. de Floriani, E. Puppo, and H. Samet. Out–of–Core Multiresolution Terrain Modeling. In *Modeling and Management of Geographical Data over Distributed Architectures*. Springer–Verlag, 2006.

[dF04] L. de Floriani. *Geometric Modeling*, 2004. Notes for the *Geometric Modeling* course, Laurea Magistrale in Informatica, Universitá degli Studi di Genova.

[dFFMD08] L. de Floriani, M. Facinoli, P. Magillo, and D. Dimitri. A Hierarchical Spatial Index for Triangulated Surfaces. In *Proceedings of the* 3^{rd} *International Conference on Computer Graphics Theory and Applications*, 2008.

[dFKP04] L. de Floriani, L. Kobbelt, and E. Puppo. A Survey on Data Structures for Level–of–Detail Models. In G. Bertrand, M. Couprie, and L. Perroton, editors, *MINGLE: Multi–resolution in Geometric Modeling*. Springer–Verlag Publisher, 2004.

[dFMP00] L. de Floriani, P. Magillo, and E. Puppo. *The Multi–Tessellation Package*, 2000. Official website: `http://www.disi.unige.it/person/MagilloP/MT_SW/MT/doc/index.html`.

[dFMPS03] L. de Floriani, P. Magillo, E. Puppo, and D. Sobrero. A Multiresolution Topological Representation for Non–manifold Meshes. *Computer–Aided Design Jour.*, Volume 36, number 2, pages 141–159, 2003.

[dFPM97] L. de Floriani, E. Puppo, and P. Magillo. A Formal Approach to the Multiresolution Modeling. In R. Klein, W. Strasser, and R. Rau, editors, *Geometric Modeling: Theory and Practice*. Springer–Verlag, 1997.

[DSS96] A. Diwan, S. Seshadri, and S. Sudarshan. Clustering Techniques for minimizing the External Path Length. In *Proceedings of the VLDB Conference*, pages 342–353, Bombay, India, 1996.

[ESC00] J. El-Sana and Y. Chiang. External Memory View–dependent Simplification. *Computer Graphics Forum*, Volume 19, pages 139–150, 2000.

[FB74] R. Finkel and L. Bentley. Quadtrees: a Data Structure for Retrieval on Composite Keys. *Acta Informatica*, Volume 4, 1974.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On the Visible Surfaces Generation by a priori Tree Structures. *ACM Computer Graphics*, 1980.

[FR89] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *Proceedings of the* 8^{th} *ACM Symposium on Principles of Database Systems*, 1989.

[Fre60] E. Fredkin. Trie Memory. *Communications of the ACM*, Volume 3, 1960.

[FS01] R. Farias and C. Silva. Out–of–Core Rendering of Large Unstructured Grids. *IEEE Computer Graphics and Applications*, Volume 21, 2001.

[Gar82] I. Gargantini. Linear Octree for the Fast Processing of 3–dimensional Objects. *Computer Graphics and Image Processing*, 1982.

[Gar99] M. Garland. Multiresolution Modeling: Survey & Future Opportunities. In *Proceedings of the Eurographics Symposium on Geometry Processing*, 1999.

[GH97] M. Garland and P. Heckbert. Surface Simplification using the Quadric Error Metric. In *Proceedings of the ACM Computer Graphics*, Annual Conference Series, 1997.

[Gut84] A. Guttman. R–trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, 1984.

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for the Database Systems. In *Proceedings of the VLDB Conference*, 1995.

[Hop96] H. Hoppe. The Progressive Meshes. In *Proceedings of the SIGGRAPH Conference*, 1996.

[Hop98] H. Hoppe. Smooth View–Dependent Level–of–Detail Control and its Application to Terrain Rendering. In *Proceedings of the IEEE Visualization Conference*, 1998.

[ILGS03] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large Mesh Simplification using Processing Sequences. In *Proceedings of the IEEE Visualization Conference*, 2003.

[Jag90] H. Jagadish. Linear Clustering of the Objects with Multiple Attributes. In *Proceedings of the ACM–SIGMOD International Conference on the Management Of Data*, 1990.

[Lin00] P. Lindstrom. The Out–of–Core Simplification of Large Polygonal Models. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2000.

[Lin03] P. Lindstrom. Out–of–Core Construction and Visualization of Multiresolution Surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.

[LPC+00] M. Levoy, K. Pulli, B Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Gintzon, S. Anderson, J. Davis, J. Ginsberg, J. Shape, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of SIGGRAPH Conference*, pages 131–144, July 2000.

[LRC+02] D. Luebke, M. Reddy, J. Cohen, A. Varnshey, B. Watson, and H. Huebner. *Level–of–Detail for 3D Graphics*. Morgan Kaufmann Publisher, 2002.

[LS01] P. Lindstrom and C. Silva. A Memory Insensitive Technique for Large Model Simplification. In *Proceedings of the IEEE Visualization Conference*, 2001.

[MA97] D. Mount and S. Arya. ANN: a library for Approximate Nearest Neighbor searching. In *The* 2^{nd} *Annual Fall Workshop on Computational Geometry*, 1997. Official website: `http://www.cs.umd.edu/~mount/ANN`.

[MM99] S. Maneewongvatana and D. Mount. It's okay to be Shinny, if your friends are Fat. In *The* 4^{th} *Annual Fall Workshop on Computational Geometry*, 1999.

[Mor66] G. Morton. A Computer–oriented Geodetic Database and a new Technique in the File Sequencing. Technical report, IBM Ltd., 1966.

[Net02] *The Microsoft .NET Framework*, 2002. Official website: `http://www.microsoft.com/NET`.

[Ore82] J. Orenstein. Multidimensional Tries used for Associative Searching. *Information Processing Letters*, 1982.

[PH97] J. Popovic and H. Hoppe. Progressive Simplicial Complexes. In *Proceedings of the SIGGRAPH Conference*, pages 217–224, 1997.

[Pri00] C. Prince. Progressive Meshes for the Large Models of Arbitrary Topology. Master's thesis, University of Washington, 2000.

[PS97] E. Puppo and R. Scopigno. Simplification, LOD and Multiresolution: Principles and Applications. In *Tutorial Notes of Eurographics Conference*, 1997.

[Pup98] E. Puppo. Variable Resolution Triangulations. *Computational Geometry*, 1998.

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: a Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the SIGGRAPH Conference*, 2000.

[RTBS05] P. Rhodes, X. Tuang, R. Bergeron, and T. Sparr. Out–of-core Visualization using Iterator aware Multidimensional Prefetching. In *Proceedings of the Visualization and Data Analysis*, 2005.

[Sag94] H. Sagan. *Space–filling Curves*. Springer–Verlag, 1994.

[Sam06] H. Samet. *Foundations of the Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[SCESL02] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. The Out–of–Core Algorithms for Scientific Visualization and Computer Graphics. In *Proceedings of the IEEE Visualization Conference*, 2002.

[SW85] H. Samet and E. Webber. Storing a Collection of Polygons using Quadtrees. *ACM Transactions on Graphics*, Volume 4, 1985.

[SZL92] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of Triangles Meshes. In *Proceedings of the SIGGRAPH Conference*, 1992.

[TOL02] T. Tu, D. O'Hallaron, and J. Lopez. ETREE – a Database–oriented Method for generating Large Octree Meshes. In *Proceedings of the* 11^{th} *International Meshing Roundtable*, 2002.

[VCL+07] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming Simplification of Tetrahedral Meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 13, 2007.

[vdBBD+01] J. van den Bercker, B. Blohsfeld, J. Dittrich, J. Kramer, T. Schafer, M. Schneider, and B. Seeger. XXL – a Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of the VLDB Conference*, pages 39–48, 2001.