Simplification of Geometric Models in Secondary Memory

David Canino

Abstract

This document is an extented abstract of our Master's Thesis defended at the D.I.S.I., Universitá degli Studi di Genova: you can refer to [Can07] for the original version of our Master's Thesis, written in the Italian language. In our Master's Thesis we have analyzed the state of art about the out–of–core simplification techniques and we have designed a prototype of an extensible framework for managing huge geometric models, called *OMSM* (short for *Objects Management in Secondary Memory*).

1 Introduction

In the recent computers the performance of graphics subsystems has been enormously improved, but unfortunately the complexity of graphics applications has also increased. Some gigantic meshes can be easily produced in many applications, for example by three–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 model increases and it often exceeds the RAM size in a common workstation. Moreover, distributed networks provide an extremely widespread channel of transmission, useful for sharing the models: their size and the limited amount of available bandwidth 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, making them available to the researchers on the Web. There are more than 20,000 area of 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: it is composed by about 370M triangles (see [LPC+00] for more details). Hence, these meshes introduce severe overheads and their management has often prohibitive costs, even for the high–performance graphics workstations. Increasing the RAM size could be a good 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 also the operating system requires some RAM.

Hence, it is important to encode huge models in the most efficient way as possible, allowing the user to analyze and to display them. The management of a huge mesh should allow some operations on it, for example we can remember:

- the efficient visualization of the models, useful for selective inspection;
- the mesh editing operations, useful to improve the data quality as described in [BEG94], where the author defines the requirements that shoud be satisfied by a *good* mesh;
- the retrieval of some topological properties in the model;
- the execution of specific operations, useful for many aspects: for example we can compute metric measures on the input model.

This problem has led to many results on mesh *simplification* and on *multiresolution* (also called *LOD*, short for *Level–of–Detail*) mesh–based models. We can reduce the model size by applying a set of local simplification

updates: unfortunately, obtaining the optimal simplification of a mesh is known to be a *NP–hard* problem, as described in [AS94]. Hence, many heuristic methods to provide an approximated version of the simplicial complex have been developed: such techniques simplify the geometry and preserve some attributes of the mesh. You can refer to [PS97] and [LRC+02] for more details about this topic. We can also define a *multiresolution model* as a model with the capability of provide some representations of a spatial object at different levels of accuracy and complexity, depending on specific application needs. According to [PS97] and [DdFM+06], a multiresolution model encodes 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. Such 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 and briefly discuss its new perspectives. In literature we can find many examples of LOD representations for triangular meshes, as those described in [LRC+02], [dFMPS03] and [dFKP04] and : we can also remember LOD models based on tetrahedral meshes, as those described in [DdFM+05] and [dFD06]. However, each multiresolution model is based on a specific type of local operator and it can be seen as a specific instance of the *MT* model (short for *Multi–Tessellation*), 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 of construction strategy. A dimension–independent representation of the *MT* has been proposed in [dFPM97] and implemented in the *MT Package* (see [dFMP00] for more details).

Unfortunately, we could encounter some problems when we apply these techniques to very large data sets, since the input mesh itself can exceed the available RAM (even we are using a compact data structure) and the *simplification* algorithms usually require all the data in–core. 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 (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, by using a limited memory footprint and by removing the limit on the input size: in this case, the I/O among the memory levels is often the bottleneck, since a disk access is slower than an access in RAM. Furthermore, a naive management of the external memory may highly degrade the performance: hence, some data structures and algorithms capable of efficiently working in 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 applications fields. For example, we can remember some techniques used for:

- the 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].

Moreover, many *EM* simplification algorithms have been developed, like those described in [Hop98], [Lin00], [Pri00], [LS01], [CMRS03], [ILGS03] and [DdFPS06]. Also the LOD representations of a huge geometric model can exceed the available memory, even if we are using a compact representation: in literature, many out–of–core multiresolution models have been developed, but most of them, like those described in [Lin03] and [CGG+04], do not support some topological operators.

The remainder of this document is organized as follows: in the Section 2 we review some background notions about the objects modeling, while in the Section 3 we propose an overview about the research fields interested by our thesis. Finally, in the Section 4 we analyze the properties required by a storing architecture for spatial data and then we design a first prototype for an extensible framework for managing huge geometric models, called *OMSM* (short for *Objects Management in Secondary Memory*).

2 Background Notions

In this Section we review some background notions about the objects modeling: in the Section 2.1 we introduce the *cell* and the *simplicial complexes*, the basic combinatorial structures that formalize the mesh definition, while in the Section 2.2 we review some notions about the *topological relations*, that provide the connectivity information among the different parts of a simplicial complex.

2.1 Cell and Simplicial Complexes

The first problem that we must solve is the efficient management of the surfaces and the scalar fields in a PC: in literature, there are two most common approaches for representing a geometric entity $\mathcal{O}$. In the first case, we give an analytic description of $\mathcal{O}$, while in the second one we decompose $\mathcal{O}$ in a set of independent portions. The analytical representations require many parameters so their use is not simple, especially for huge objects. However, we can use these techniques only in particular cases, like the *fractals* modeling: you should refer to [Jul22], [Man77], [Vos88], [MM91] and [Mus93] for more details.

Thus, we should follow the second approach: in literature, the *cell* and the *simplicial complexes* are the basic combinatorial structures for representing real objects and scalar fields. We limit our analysis to the Euclidean space $\mathbb{E}^3$: you should refer to [dF04] and [Ago05] for more details.

Let x be a vector in $\mathbb{E}^d$ (with $d \geq 1$) and $\|x\|$ the length of x, then we can 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 which 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: usually k is called the *dimension* of simplex σ. A *face* σ of a k–simplex γ is an h–simplex (with $0 \leq h < k$) generated by $h+1$ vertices of γ: usually we use the notation $\sigma \subseteq \gamma$. Two simplexes are called *k–adjacent* if they share a k–face: in particular, two vertices (0–simplexes) are called *adjacent* if they belong to 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

In a simplicial complex, the connectivity information can be expressed through *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 can 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 adjacency relations are variable, while 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 the Art

In this Section we propose an overview about the research fields interested by our thesis. In the Section 3.1 we introduce some in–core *simplification* algorithms, while in the Section 3.2 we describe the foundations of the spatial indexing techniques, suitable to handle huge geometric models: finally, in the Section 3.3 we review some methods that have been proposed in the literature for the simplification of geometric models that cannot be handled in RAM, i.e. the *EM simplification* techniques.

3.1 In–core Simplification Algorithms

In this Section we introduce some in–core *simplification* algorithms: these techniques have an important role, in fact they reduce the input model to a manageable size. Unfortunately, obtaining the optimal simplification is known to be a *NP–hard* problem, as described in [AS94] and a large amount of heuristic methods have been developed. In the Section 3.1.1 we describe some techniques used for simplifying the triangular meshes, while in the Section 3.1.2 those for simplifying the tetrahedral meshes.

3.1.1 Simplification of Triangulated Surfaces

In this Section, we review some *simplification* methods for triangulated surfaces, discussing some different approaches that have been developed in literature.

One of the most common methods is based on the *vertex clustering*: this operator spatially subdivides the mesh vertices into clusters and then substitutes all the vertices within the same cluster with a representative point. An important example of these methods is the *uniform vertex clustering*, proposed in [RB93]: this method uses a regular grid for subdividing the mesh. Such algorithms are fast, but they produce poor quality approximations, introducing warping effects in the resulting mesh.

An other important simplification technique is based on the *vertex decimation*, introduced in [SZL92]: in this method a vertex $\bar{v}$ is selected for removal, all the faces incident in $\bar{v}$ must be removed and the resulting hole is retriangulated. Since retriangulation requires a projection of the local surface onto a plane, this algorithm is limited to the manifold surfaces: some variants (as that described in [CCMS97]) exist and they are based on more accurate error metrics, like the localized Hausdorff distance. These techniques are quite efficient in time and space, but they have some difficulties in preserving smooth surfaces.

An important class of simplification algorithms is based on the *edge collapse* modification. Given an edge $e = (v_s, v_t)$, the *edge collapse* contracts e into a single vertex v_s: if the vertex v_s is one of the edge e endpoints, then this modification is called *half edge–collapse*, otherwise it is called *full edge collapse*. In both cases, the

two faces incident at edge e vanish and the other triangles incident in v_s and v_t become incident in the new vertex v_s: usually, this operation is indicated with $ecol(v_s, v_t)$. This update is invertible and the inverse operator is called *vertex split*: usually, it is indicated with $vsplit(v_s, v_s, v_t)$. The Figure 1 shows the updates introduced by the half edge–collapse and the vertex–split trasformations on a triangular mesh.

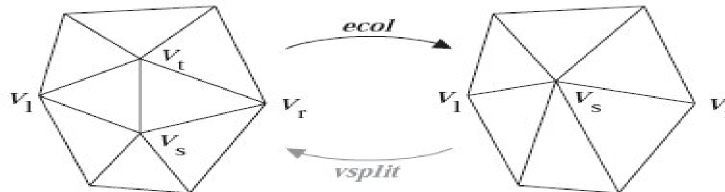


Figure 1: the result of a half edge–collapse and the vertex–split operators, respectively indicated with $ecol$ and $vsplit$. Figure courtesy of [Hop96].

Many algorithms based on the edge–collapse operation follow a greedy procedure to select a sequence of edges to be collapsed: usually they assign a cost and the edge of lowest cost is contracted at each step. In literature, many techniques have been developed: here, we review two important examples, namely the *Progressive Meshes* and the *Quadric Error Metric*.

The *Progressive Meshes*, introduced in [Hop96], are used for the model trasmission on a network: the edge–collapse cost for each edge is modeled by an energy functional, based on the geometric error. This technique produces the highest quality results, but it requires huge running times.

The *Quadric Error Metric*, introduced in [GH97], defines the approximation error in terms of distances from a set of planes: a symmetric 4×4 matrix is assigned to each vertex v and it measures the sum of squared distances of v from all the set planes. The geometric interpretation of this functional is a *quadric*: this metric sacrifices some precision in assessing the approximation error, but it requires short running times.

3.1.2 Simplification of Tetrahedral Meshes

Some of the decimation techniques introduced in Section 3.1.1 can be extended in order to simplify a *volumetric dataset*, i.e. a set of points V in the Euclidean space $\mathbb{R}^3$ (usually extracted from a volume and connected by a tetrahedral mesh) and a collection of field values associated to the points in V. In this Section, we review some simplification algorithms for tetrahedral meshes, but many other techniques have been proposed: for more details you should refer to [CdFM$^+$04], [DdFM$^+$05] and [dFD06].

The first simplification technique for a tetrahedral mesh has been proposed in [Wil93]: a random subset of vertices (and their incident tetrahedra) are removed and the resulting hole is filled with the Delaunay tetrahedralization, as described in [dBSvKO97]. This technique is based on uncontrolled subsampling and there is no control over the accuracy of the simplified mesh.

In [GS98] the authors define the *Progressive Tetrahedralization*, similar to the *Progressive Meshes*, that we have reviewed in Section 3.1.1: this model is based on the edge–collapse operator and proposes many cost functions to drive the decimation proces. Besides preserving the topological and geometric features as boundary, this technique elegantly handles previously undiscussed cases of *negative tetrahedra* (caused by *flipping*) and tetrahedron–boundary intersections at concave interiors. However, since the expensive dynamic mesh–consistency tests for these special cases, its time complexity is discouraging for rapid simplification of very large datasets.

In [THJ98] the authors propose a tetrahedral collapse as a sequence of three edge–collapses, while keeping the overall error (based on a unique spline–segment representation of each tetrahedron) below a tolerance range. They also discuss problems and difficulties specific to the tetrahedral mesh simplification, presenting a framework to employ edge–collapse to decimate tetrahedra: however, because the decimation strategy is based upon successive edge–collapses, the authors mention the overwhelming overheads for maintaining an edge data structure for massive volumetric datasets. Furthermore, the time complexity of the algorithm presented was not evident in the results or discussion.

In [CCM$^+$00] the authors present a framework for simplifying a tetrahedral mesh, based on the edge collapse: the local accumulation, the gradient difference and some brute force strategies are explored to predict

and evaluate errors. Thus, this paper is a good survey for accurate error constraints implementations and offers various options that need to be tested with rigorous decimation strategies. Moreover, they describe two kinds of errors that can be introduced during the decimation process like the *field error* and the *domain error*: the field error is introduced by approximating the input dataset with a coarser representation, while the domain error is caused by the simplification of the complex boundary.

Another approach, called *TetFusion*, has been presented in [CM02], where the authors introduce a reversible atomic operation for tetrahedral meshes, the *tetrahedron collapse*: they iteratively collapse a tetrahedron onto its barycenter. This technique steers a rapid and controlled progressive simplification of a tetrahedral mesh and also takes care of mesh–inconsistency problems. This algorithm features a high decimation ratio and inherently discourages some cases of self–intersection of boundary, element boundary intersection at concave boundary–regions and negative volume tetrahedra.

3.2 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 thesis, 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:

- the *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 2(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 mod $k+1$ 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 2(b) shows an example of a point k–d tree, while you should refer to [Ben75] and [Sam06] for more details about this index.

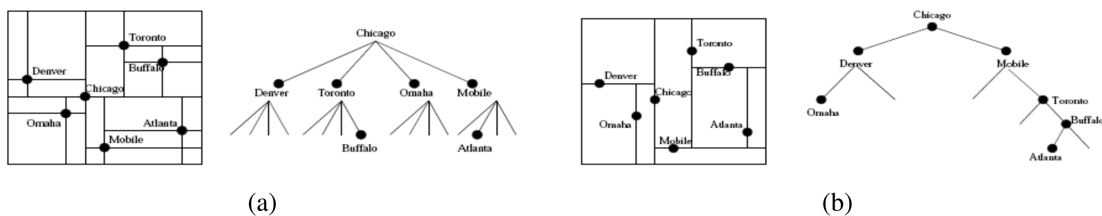


Figure 2: the spatial decompositions and the tree representations for two spatial indexes: (a) a point–quadtree and (b) a point k–d tree. Figures courtesy of [Sam06].

3.3 *EM* Simplification Algorithms

Many applications can require very large meshes, consisting of millions of simplexes: despite the rapid improvement in the hardware performances, these meshes overload the RAM size in a common workstation, thus we cannot apply the simplification techniques, discussed in the Section 3.1. Hence, using an out–of–core technique is mandatory in this case: we can load in RAM only a small portion of the mesh and the expensive operations, as the simplification, must be performed only on data held in–core. In [DdFPS06] the authors review some methods that have been proposed in the literature for the simplification of terrains models that cannot be managed in RAM: however, many of these techniques can be extended in order to support also more general kinds of data (e.g., triangular meshes describing the boundary of 3D objects). The EM simplification algorithms can be roughly subdivided in three major classes: in the Section 3.3.1 we review some algorithms based on the *vertex clustering*, in the Section 3.3.2 we describe some techniques based on the *mesh decomposition*, while in the Section 3.3.3 we propose some results based on the *mesh streaming* technique.

3.3.1 *EM* Simplification Techniques based on the Vertex Clustering

In this Section we review some EM simplification methods based on the *vertex clustering* technique, introduced in [RB93]: such methods are very fast and require a small amount of memory, but they are not progressive. The resolution of the simplified mesh is a priori determined by the resolution of the buckets grid and no intermediate representations are produced during the simplification process.

One of the first methods is described in [Lin00], where the author adopts a clustering approach based on a uniform grid over a triangular mesh and the quadric metric error for choosing the representative point for each cluster, improving the mesh quality. The current mesh is stored as a redundant list of three vertices positions per triangle and the simplified mesh is incrementally constructed and kept in–core.

In [LS01] the authors propose further improvements over the method described in [Lin00], by increasing the quality of approximation and by reducing memory requirements. This technique stores the intermediate results on disk instead of in RAM and in this way it removes the bottleneck of having enough memory for the intermediate steps: the whole process is made essentially independent of the available memory on the host

computer. This approach also improves the quality of the mesh, preserving surface boundaries and optimizing the position of the representative vertex of a grid cell.

The method presented in [Lin00] has also been extended in [SG01], where the authors design a simplification algorithm, that executes two steps over the input mesh. During the first step the mesh is analyzed and an adaptive space partitioning, based on a *BSP–tree*, is performed: this structure is an important spatial index introduced in [FKN80]. Using this approach, we can adapt the distribution of grid cells to the more detailed portions of the surface. In the following step, the algorithm applies the *vertex clustering* to the cells produced at the first step: the accuracy of the output mesh is better than the other methods, but this algorithm requires more RAM than the technique described in [Lin00] since it maintains a *BSP–tree* and additional quadrics information.

A hybrid technique is described in [GS02], by combining the *vertex clustering* and the *iterative edge–collapse* in two steps: in the first step, this technique performs an uniform vertex clustering as in the methods described above, while in the second step some edge–collapses are iteratively performed in order to simplify the mesh, guided by the quadric metric error. This method assumes that the mesh obtained after the first step is small enough to perform the second step in main memory.

3.3.2 *EM* Simplification Techniques based on the Mesh Decomposition

In this Section we review some EM simplification technques 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 the Section 3.1. Great attention must be paid to the patches boundaries in order to maintain the topological consistency of the simplified mesh: these techniques are progressive and allow to create a dependency relation among the updates.

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 simplification process.

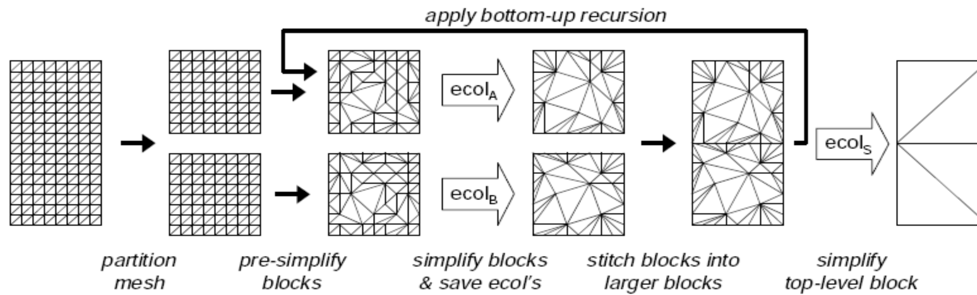


Figure 3: a rough scheme of the simplification technique described in [Hop98]. Figure courtesy of [Hop98].

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.

3.3.3 *EM* Simplification Techniques based on the Streaming Meshes

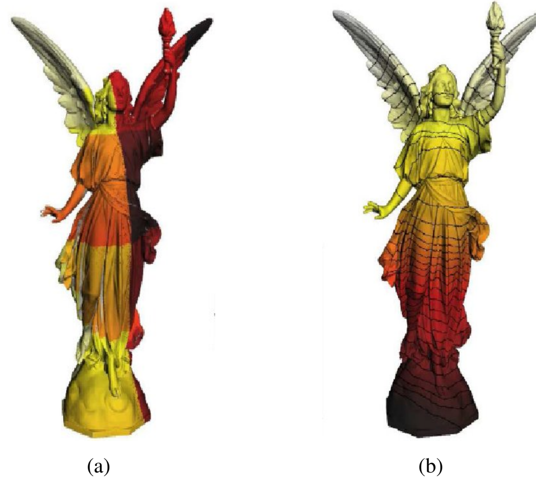
In this Section we review some EM simplification methods based on the *sequential analysis* of a mesh during the scanning of the input file that contains the mesh itself, providing a compact representation of the input data. The philosophy underlying the streaming techniques is that a strictly sequential processing order is followed, by mantaining the input data locality: each element is loaded only a time in RAM and the result is written to secondary memory as soon as possible. The major problem is dealing with the initial format of the input, that cannot guarantee the *coherence* of the data: the current mesh formats are usually *indexed* and do not impose any constraints on the order of vertices and simplexes: for example, the three vertices of a triangle can be located anywhere in the vertices array and need not be close to each other, while adjacent triangles may reference vertices at opposite ends of the array. This flexibility was convenient for in–core meshes, but has become an important bottleneck with huge datasets: one approach to overcome the problems associated with indexed data is working on *dereferenced simplexes soup*, which streams from disk to memory in increments of single simplexes and either attempts to handle their connectivity. The main idea is decomposing the input model into portions, called *Streaming Meshes*, mantaining the data coherence: this objective could be reached by interleaving indexed vertices and triangles and providing information when vertices are last referenced.

In [IL05] the authors define the underlying theory for working with this new representations: in particular, they describe desirable qualities for streaming meshes, some methods for converting from a traditional format to a streaming one and several EM algorithms for reordering meshes with poor coherence. Furthermore, they provide metrics that characterize the coherence of a mesh layout and suggest appropriate strategies for improving the *streamability* i.e. an intuitive measure of the longest duration of a mesh portion in RAM. The Figure 4 shows two versions of *Lucy* model, obtained by *The Stanford 3D Scanning Repository* (see [Sta94]): the colors scale indicates the streamability of each triangle. The Figure 4(a) shows the original layout, while the Figure 4(b) shows the layout after reordering the vertices and triangles arrays: as we can easily understand, the coherent regions are very fragmented in the original model and many I/O operations are needed to analyze it.

The streaming meshes generalize the *Processing Sequences*, introduced in [ILGS03]: a processing sequence represents a mesh as an interleaved ordering of indexed triangles and vertices, by providing a mechanism for storing user data on the stream boundaries (e.g. for mapping between external and in–core vertex indices). The authors also design a simplification technique for triangular meshes, based on the processing sequences and similar to that described in [Lin00]. Mesh access is restricted to a fixed traversal order, but the full connectivity and the geometry information is available for the active elements, thus the simplification step is performed only on the in–core portions.

In [VCL$^+$07] the authors extend these streaming algorithms to the tetrahedral meshes, making some improvements: first, the algorithm converts standard indexed meshes and optionally reorders them in order to improve their streamability and then some portions of the streaming mesh are incrementally loaded into a fixed–size main memory buffer and then simplified using the quadric–based method. When the in–core portions of the mesh reach the user–prescribed resolution, the simplified elements are outputted to disk or to a downstream processing module. This algorithm keeps just a subset of the tetrahedra in–core and thus we do not know the entire mesh connectivity: to ensure that the simplified mesh is crack–free, we also store the boundary (called *stream boundary*) between the set of tetrahedra currently in RAM and all the remaining elements on the

storage support. Furthermore, this algorithm does not allow to collapse any edge that has one or both vertices on the stream boundary, hence the output mesh will have unsimplified areas along the stream boundaries: to avoid this problem the authors follow the algorithm proposed in [WK03]. According to our experience, this is the first algorithm that provides an EM simplification method for the tetrahedral meshes.



(a) (b)

Figure 4: examples of coherence in the mesh layout and streamability distribution of triangles. The original layout (a) is incoherent hence the colors distribution is very fragmented, while the layout after reordering of simplexes (b) is coherent, in fact the colors fragmentation is limited. Figures courtesy of [IL05].

4 An Extensible Storage Architecture

In this Section we analyze the properties required by a storing architecture for spatial data and then we design a first prototype of an extensible framework for managing huge geometric models, that we call *OMSM* (short for *Objects Management in Secondary Memory*) framework.

As we have already observed in the Section 3.2, 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 3.2, 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 attracti-
ve 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.

Towards the definition of a completed framework with the properties that we have discussed in this Section, we introduce a rough prototype of the *OMSM* (short for *Objects Management in Secondary Memory*) framework: it can manage a huge set of spatial objects in a storage support, embedded in the same Euclidean space. In this structure, we consider each of the previous aspects as independent from the other and we manage it into a specific component, called *OMSM layer*: as consequence, the *OMSM* structure is modular and easily extensible, adapting itself to many storage needs. The Figure 5 shows the complete organization of the *OMSM* layers.

The layers that belong to the *OMSM* framework are:

- the *SPDataIndex* component, where the user can choose the *space partitioning tree* to be used and can execute queries and updates.
- the *NodeHandler* component, where the user can modify the *nodes clustering policy* to be used, useful for minimizing the number of I/O operations.
- the *NClusterStorager* component, where the can choose techniques used to manage the clusters, directly on the storage support.

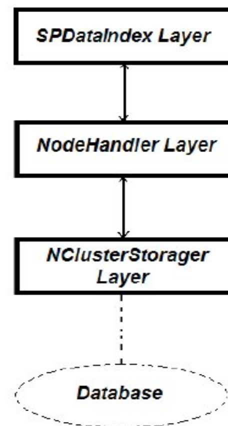


Figure 5: the layers of the *OMSM* framework: the dotted lines with the database clusters remark that we do not know its services set because we do not know how it works, until we have chosen it.

The *SPDataIndex* layer – this layer is the most external one in the *OMSM* architecture and it allows the user to interact with the data stored in the architecture, by offering a set of services and by hiding the implementation details and specific techniques used for the spatial objects management. It is simple to understand from the Figure 5 that once created a particular instance of this layer, it is no longer interesting to recall the offered features and an user can use this layer without knowing anything about its structure: this design choice makes easier to build applications that interact with the stored data. In this layer, the user can specify which spatial index must be used and this choice is totally transparent for the other layers of the *OMSM* framework. The underlying idea is to keep in RAM only the tree root, while the nodes will be dynamically loaded from the inferior layers, if it is needed. In this layer, the data model is described by a generic spatial object that could be indexed by using its *representative point* as key: this point describes the object properties and identify it, for example it could be its center of gravity.

The *NodeHandler* layer – this layer has perhaps the major role in the *OMSM* architecture, since it subdivides the spatial index nodes into clusters according to a certain policy: for us, a cluster is a set of nodes that are considered as an atomic unit (according with a *nodes clustering policy*) and each I/O operation, delegated to the *NClusterStorager* layer, must be performed on a single cluster. In this layer, the data model is described by the *OMSM Node*, that includes not only the indexing structures nodes, but also a particular node, called *OMSM Super–Node*: it contains information about the current configuration, like the plugins to be loaded and so on. Consequently, the *NodeHandler* layer can manage *OMSM nodes* without worrying about their content: the only constraint to be satisfied involves the *OMSM Super Node*: the cluster used for its storage cannot handle any other nodes and it has a specific management system. This design choice allows an efficient implementation of the *OMSM* architeture: for example, this cluster will be always mantained in RAM, while the other clusters could be destroyed since a LRU cache is used in order to mantain in RAM only the most recently used clusters.

The *NClusterStorager* layer – this layer manages the low–level representations of the nodes clusters, by operating on a storage support: the type of support is not important, for example it can access to a remote database or write each cluster on an independent file. In this layer, the data model is described by a generic code for the cluster identification and the sequence of bytes that represents each cluster. In such way, we can discard the data structure implementation details and allow additional operations on the bytes sequence, like the encryption or compression.

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 plugins for each layer:

- 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, by using the famous *Oracle Berkeley DB* (see [Bdb06] for more details).

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 Licence.

Our work is orthogonal to the existent libraries in literature 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. The *OMSM* framework does not aim to simplify the development process of the indexing 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 indexing structure developer frameworks. Employing the *OMSM* is an easy process that requires the implementation of simple adapter classes (the *OMSM* plugins), that will make indexing structures compliant to its interfaces, conflating these two conflicting design viewpoints. Ostensibly, existing libraries can be used for simplifying client code development and share, indeed, some similarities with the *OMSM*, (especially in the interfaces for inserting and deleting elements from the indices) but, since they are not targeted primarily for that purposes, they are not easily extensible (without the subsequent need for client code revisions), they do not promote transparent usage of different indices (since they use index specific interfaces) and they cannot be easily used in tandem with any other index library (existing or not).

5 Conclusions

Many applications can require very large meshes, consisting of millions of simplexes: despite the rapid improvement in the hardware performances, these meshes often overload the RAM size of a common workstation. The simplification techniques are a good solution for this problem, since they reduce the size 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 (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, by using a limited memory footprint and by removing the input size limit: in this case, the I/O among the memory levels is often the 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 thesis we have analyzed the state of the art about the out–of–core simplification techniques, by focusing our attention of those techniques based on the spatial decomposition: moreover, we have analyzed the properties required by a storing architecture for spatial data and then we have designed a rough prototype of an extensible framework for managing huge geometric models, called *OMSM* (short for *Objects Management in Secondary Memory*).

References

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

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

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

[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* 5th *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] J. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 1975.

[BM72] R. Bayer and E. McCreight. Organization and Mantenance of Large Ordered Indices. *Acta Informatica*, Volume 1, 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, October–December 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. T. Silva. Progressive Volume Rendering of the Large Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, Volume 12, 2006.

[CCM+00] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of Tetrahedral Volume Data with Accurate Error Evaluation. In *Proceedings of IEEE Visualization*, pages 85–92. IEEE Computer Society, 2000.

[CCMS97] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution Decimation based on Global Error. *The Visual Computer*, Volume 13, 1997.

[CdFM+04] P. Cignoni, L de Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective Refinement Queries for Volume Visualization of the Unstructured Tetrahedral Meshes. *IEEE Transactions on Visualization and Computer Graphics*, Volume 10, 2004.

[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. Adaptive TetraPuzzles: Efficient 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–Triangulations. In *Proceedings of the IEEE Visualization Conference*, 2005.

[Chi03] Y. J. Chiang. Out–of–Core Isosurface Extraction of Time–Varying Fields over Irregular Grids. In *Proceedings of IEEE Visualization '03*, 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.

[CM02] P. Chopra and J. Meyer. Tetfusion: an Algorithm for Rapid Tetrahedral Mesh Simplification. In *Proceedings of IEEE Visualization*, pages 133–140. IEEE Computer Society, 2002.

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

[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. T. Silva, and W. Schroeder. Interactive Out–of–Core Isosurface Extraction. In *Proceedings of the IEEE Visualization*. IEEE Press, 1998.

[dBSvKO97] M. de Berg, O. Schwarzkopf, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer–Verlag Publisher, 1997.

[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*, Volume 30, 2006.

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

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

[dFD06] L. de Floriani and E. Danovaro. 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*. Springer–Verlag Publisher, 2006.

[dFKP04] L. de Floriani, L. Kobbelt, and E. Puppo. A Survey on Data Structures for Level–of-Detail Models. In M. Floater N. Dodgson and M. Sabin Editors, editors, *MINGLE: Multi–resolution in Geometric Modeling*. Springer Verlag Publisher, 2004.

[dFMP00] L. de Floriani, P. Magillo, and E. Puppo. *The MT 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 Multresolution Topological Representation for Non–Manifold Meshes. *Computer–Aided Design Journal*, 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*, pages 302–323. Springer–Verlag Publisher, 1997.

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

[FB74] R. Finkel and J. Bentley. Quadtrees: the 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.

[Fre60] E. Fredkin. Trie Memory. *Communications of the ACM (CACM)*, 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.

[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.

[GS98] M. Gross and O. Staadt. Progressive Tetrahedralizations. In *Proceedings of IEEE Visualization*, pages 397–402. IEEE Computer Society, 1998.

[GS02] M. Garland and E. Shaffer. A Multiphase Approach to Efficient Surface Simplification. In *Proceedings of the IEEE Visualization Conference*. IEEE Computer Society, 2002.

[Gut84] A. Guttman. R–Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the International Conference on Management of Data*, 1984.

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the* 21st *International Conference on Very Large Databases*, 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*. IEEE Computer Society, 1998.

[IL05] M. Isenburg and P. Lindstrom. Streaming Meshes. In *Proceedings of the IEEE Visualization Conference*. IEEE Computer Society, 2005.

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

[Jul22] G. Julia. Mémoire sur la Permutabilité des Fractions Rationnelles. *Annales Scientifiques de l'École Normale Supérieure*, 1922.

[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. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of the SIGGRAPH Conference*, 2000.

[LRC$^+$02] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and H. Huebner. *The 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.

[Man77] B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman & Company, 1977.

[MM91] F. Musgrave and B. Mandelbrot. The Art of Fractal Landscapes. *IBM Journal of Research and Development*, 1991.

[Mus93] F. Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University, 1993.

[Ore82] J. Orenstein. Multidimensional Tries used for Associative Searching. *Information Processing Letters*, Volume 14, 1982.

[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 Terrain Surfaces. In *Proceedings of the* 8th *Canadian Conference on Computational Geometry*, 1998.

[RB93] J. Rossignac and P. Borrel. Multi–Resolution 3D Approximations for Rendering Complex Scenes. In T. L. Kunii and B. Falcidieno, editors, *Modeling in Computer Graphics*, pages 455–465. Springer–Verlag Publisher, 1993.

[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.

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

[SCESL02] C. T. 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.

[SG01] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. In *Proceedings of IEEE Visualization*. IEEE Computer Society, 2001.

[Sta94] *The Stanford 3D Scanning Repository*, 1994. Stanford University, Official website: `http://graphics.stanford.edu/data/3Dscanrep`.

[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.

[THJ98] J Trotts, B. Hamann, and K. Joy. Simplification of Tetrahedral Meshes. In *Proceedings of IEEE Visualization*. IEEE Computer Society, 1998.

[VCL+07] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming Simplification of Tetrahedral Meshes. *IEEE Transactions on the 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.

[Vos88] R. Voss. *The Fractals in nature: from characterization to simulation*. In *The Science of Fractal Images*. Springer–Verlag Publisher, 1988.

[Wil93] P. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. PhD thesis, University of Illinois at Urbana–Champaign, 1993.

[WK03] J. Wu and L. Kobbelt. A Stream Algorithm for the Decimation of Massive Meshes. In *Proceedings of Graphics Interface Conference*, 2003.