Eurographics Italian Chapter Conference (2010)
E. Puppo, A. Brogni, and L. De Floriani (Editors)

A Dimension-Independent and Extensible Framework for Huge Geometric Models

David Canino [1]

[1] canino@disi.unige.it, Department of Computer Science, University of Genova, Genova, Italy

Abstract

Nowadays, gigantic models can be easily produced in many applications and their dimension often exceeds the RAM size in a common workstation. Thus, using an external memory technique is mandatory in this case. In this paper, we define a dimension-independent and extensible framework, called Objects Management in Secondary Memory *(OMSM), for managing huge models. The OMSM framework can be easily adapted to the users needs through dynamic plugins, providing many techniques to be integrated in a storing architecture.*

Categories and Subject Descriptors (according to ACM CCS): H.3.1 [Information Systems]: Information Storage and Retrieval—Content Analysis and Indexing

1. Introduction

Nowadays, huge models can be easily produced in many applications, for example by the 3D analysis of brain [ABA06]. The dimension of such models can often exceed the RAM size and, thus, their management has prohibitive costs, even for high-performance graphics workstations. Moreover, it is important to encode geometric models in the most efficient way as possible, maintaining the opportunity to apply editing operators in order to improve their quality [BEG94] or to visualize them. Increasing the RAM size could be a trivial solution, since its cost is going sharply down. But establishing the correct amount of RAM is difficult, since models dimension is variable.

Simplification techniques could be a reasonable solution for this problem, since they reduce models size through local updates. Unfortunately, obtaining the optimal simplification of a model is known to be a *NP-hard* problem [AS94]. Many heuristic methods to simplify a geometric model have been developed [LRC*02]. Such techniques are also suitable to construct a *multiresolution* model [DDFM*06], i.e. a model with the capability of providing representations of a object at different levels of accuracy.

Unfortunately, also these techniques have high RAM requirements and they cannot be efficiently used with a huge model. Thus, using an out-of-core technique is mandatory: we maintain the entire model in *external memory* (*EM*) and we dinamically load in RAM only portions small enough to be processed in-core. Hence, we remove limits over the model size. However, we remark an EM access is slower than an access in RAM: if an efficient control of accesses is not performed, then I/O performance can be degraded. In literature, many EM techniques have been designed, e.g. the EM visualization [SCESL02, CBPS06] and the EM simplification [Hop98, Pri00, CMRS03, VCL*07].

In this paper, we introduce the *Objects Management in Secondary Memory* (*OMSM*) framework in order to manage huge geometric models. The OMSM framework can be easily adapted to the users needs through dynamic plugins, integrating many techniques.

The remainder of this document is organized as follows. In Sect. 2, we review basic notions about the spatial indexing techniques, while in Sect. 3, we propose the requirements to be satisfied by an extensible storing architecture. In Sect. 4, we propose a complete description of the OMSM framework. Finally, we discuss its possible extensions in Sect. 5.

2. Spatial Indexing Techniques

Emerging database applications may need a large variety of data being supported, in general *multidimensional data*, i.e. points representing locations in a high-dimensional space. In this paper, we consider as input unstructured collections of geometric objects embedded in the same Euclidean space, i.e. the *spatial objects*. In order to efficiently access and update spatial objects, we need new access structures beyond the B-tree [BMC72] and thus we must introduce *spatial indexing indices*. Such structures provide an efficient data organization, related to sorting. In literature, many spatial indexing structures have been developed [Sam06].

In this paper, we focus our attention on a particular class of indices, called *space partitioning trees*. Such structures recursively decompose the input model domain into disjoint and not overlapped regions, called *buckets*. We call *leaf* a bucket that is not subdivided, while we call *internal bucket* a bucket that is subdivided. The most suitable representations for such structures is the *multi-way tree of order m*, i.e. trees where each node has at most m children. In such trees, we can recognize *leaves*, i.e. nodes without children, and *internal* nodes, i.e. nodes with m children (at most). In our context, each node recursively describes a bucket of the input domain decomposition. In particular, the root node describes the input model domain, while an internal node n describes an internal bucket b_n. As consequence, the children of n recursively describe all the buckets in the decomposition of b_n and m must be the maximum number of subdivisions for an internal bucket. Moreover, a leaf node of a multiway tree describes a leaf bucket, since it has not any children. Usually, data are contained only in leaves.

3. Requirements for a Storing Architecture

In this section, we propose requirements to be satisfied by a generic storing architecture for geometric models. Our target is to design a generic and extensible framework for describing storing architectures. Unfortunately, few generic frameworks for spatial indices have been proposed.

For example, we can mention the *eXtensible and fleXible Library* (*XXL Library*) [vdBBD*01]. It supports 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, then we must modify all the affected indexing structures. The XXL Library does not decouple indexing structures from storage systems and thus it is not easily extensible and customizable.

The *Generalized Search Tree* (*GiST*) [HNP95] is an other framework relevant for our research. It generalizes a height-balanced and single-rooted search tree with variable fanout. 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 must implement some methods, used internally to control the tree behaviour and to simulate the required index. The GiST does not support the space partitioning trees. This project is not supported anymore and the currently available implementation does not work on recent platforms.

The *Space-Partitioning GiST* (*SP-GiST*) [AI01] is a GiST extension. It describes a generalized indexing structure with different behavioral properties, including space partitioning indices. The currently available implementation works only with the *PostgreSQL* database server [Psq96, EEA06] and it supports only bidimensional data.

Thus, the design of storing architectures for spatial objects should consider these three aspects:

- a *spatial index* for improving the operations performance: in our case, we must apply a space partitioning index.
- the *subdivision* of spatial indexing nodes into clusters. A cluster is a set of nodes, considered as an atomic unit according to a *nodes clustering policy*: in this context, an I/O operation is performed on a cluster. Thus, we can group nodes in order to minimize the number of EM accesses: for example, a cluster can contain a single node, but this clustering policy is not optimal [DSS96].
- the *dynamic management* of clusters among a storage support and RAM. Clusters must be efficiently written and read from a storage support, e.g. on a XML file or inside an embedded database [Bdb06].

Moreover, a storing architecture should be able to to manage a large variety of spatial objects. Many techniques have been developed in order to resolve the above problems and, thus, we should choose the most suitable ones for our application. Unfortunately, storing architectures, currently available in literature, provide a-priori choices for each of the above aspects, e.g. the *GiST* framework allows to customize indexing structures and it manages only a local database. Thus, such frameworks are not extensible and they cannot be easily adapted to the users needs: in this case, users must adapt themselves to what the storing architecture offers. However, the three above requirements are *independent* from each other, e.g. we can design a clustering policy discarding the type of spatial index we are using or we can transfer a cluster on a storage support discarding its internal structure.

These issues drive our desire for a common, unifying and extensible framework to describe a storing architecture. This framework should provide an effortless way to customize each capability of a storing architecture (i.e. the space partitioning tree, the clustering policy and the storage support to be used). Each technique has specific advantages and disadvantages, making it suitable for different applications. Thus, selecting a technique for each capability allows to adapt the storing architecture to any users needs. We can reach this goal by adopting a common programming interface in order to efficiently integrate a wide range of techniques. This design allows the abstraction of dynamic software components (one for each of the above aspects) in accordance with the *Interface* design pattern [GHJV95]. This solution should be also applied for spatial objects. Interfaces control not only what functionality an implementation will have, but also dictate how interaction will occur via the offered services. This allows software components to communicate in precisely the same fashion regardless of what implementation a software component is using. Moreover, this design allows to remove an instance of a component and to replace it with another one implementing the same interface, even at run-time. In other words, we need a dynamic plugins system, in accordance with the *Dynamic Linkage* design pattern [GHJV95, SB98]. Each plugin can be shared with other researchers and tried out with its own implementations in novel ways not necessarily thought by the original author. Moreover, this design choice should help developers to promote reusability, easier maintenance and code familiarity.

4. An Extensible Storing Architecture

In this section, we introduce the *Objects Management in Secondary Memory* (*OMSM*) framework, satisfying all the requirements described in Sect. 3.

This framework manages a huge set of spatial objects with different topological properties, but embedded in the same Euclidean space. In Sect. 4.1, we describe the spatial objects management in the OMSM framework. Our framework has a layered structure, as depicted in Fig. 1. Each layer is called *OMSM layer* and it is targeted to manage one of the three aspects introduced in Sect. 3:

- the *SPDataIndex* layer, where users can choose the space partitioning tree to be used: in Sect. 4.2 we give a precise description of this layer.
- the *NodeHandler* layer, where it is possible to choose how nodes of the spatial indexing tree must be grouped according to a clustering policy. In Sect. 4.3, we give a precise description of this layer.
- the *NClusterStorager* layer, where it is possible to choose how clusters must be transferred between RAM and a particular storage support (i.e. a hard-disk). In Sect. 4.4. we give a precise description of this layer.

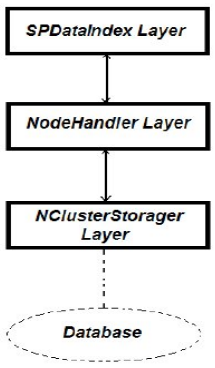

Figure 1: *the layered organization of the OMSM framework.*

Within each layer, an entity implements its functionalities, described by an interface. Each entity interacts only with the layer immediately beneath and provides facilities to the layer above it through services offered by its interfaces. Moreover, data encapsulation is used in each layer in order to abstract its data model. Thus, each layer works on a view of the data to be managed, by discarding not useful content. In this way, each layer is independent from the other one, satisfying the *Interface* design pattern. This communication schema allows also to replace an entity with an other one implementing the same interface, without messing the entire architecture and satisfying the *Dynamic Linkage* design pattern.

4.1. The OMSM spatial objects

One of the most important problems in a storing architecture (as in the OMSM framework) is the data *persistence*. In our case, we are also interested in indexing and in extracting geometric properties from spatial objects. Thus, we decouple persistence properties from the spatial ones in order to obtain an extensible system in the same spirit of the OMSM framework. We apply a solution based on interfaces hierarchy: each interface offers a set of services and the user can write a custom extension, satisfying such interfaces.

The first step is the definition of an interface common to all the objects to be stored in the OMSM framework. In order to reach our goal, we have designed a persistence system based on the representation of an object through a bytes sequence, maintained in little-endian order. Each object $\mathcal{O}$ has an internal state, composed by internal fields. The bytes sequence describing $\mathcal{O}$ can be obtained by recursively concatenating descriptions of the internal fields. Such bytes sequence must be provided (in content and in format) by the user inside a plugin. This persistence system is described by the *RawData* interface, offering the services:

- *bytes_sequence getBytes()*;
- *int getBytesNumber()*.

The *getBytes()* service returns the bytes sequence describing an object $\mathcal{O}$, while the *getBytesNumber()* service returns its length, expressed in bytes.

In the OMSM framework it is also possible to store and to query spatial objects. Given a spatial object $\mathcal{S}$, then it must be possible:

- to extract geometric properties of $\mathcal{S}$ (e.g. its vertices);
- to compare $\mathcal{S}$ with an other spatial object;
- to extract the dimension d of the Euclidean space where $\mathcal{S}$ is embedded;
- to extract the d-dimensional *representative point* of $\mathcal{S}$, i.e. a point used as unique identifier of $\mathcal{S}$ inside a space partitioning tree (see Sect. 4.2).

In order to describe a spatial object, we extend the *RawData* interface, introducing the *OmsmSpatialObject* one. Also in this case, we assume to have a custom definition of a spatial object (usually application-dependent): at the mean time, we can have a collection formed by different types of spatial objects. The representative point definition follows the same approach: one of most common candidates is the spatial object baricentre, but it is not the unique choice.

In addition to the services offered by the *RawData* interface, the *OmsmSpatialObject* interface offers the following services:

- *int getSpaceDimension();*
- *Point getRepresentativePoint();*
- *list<Point> getEuclideanVertices();*
- *bool sameObject(OmsmSpatialObject o).*

Let $\mathcal{S}$ be a spatial object, i.e. an instance of the *OmsmSpatialObject* interface. The *getSpaceDimension()* service returns the dimension d (usually $d = 3$) of the Euclidean space where $\mathcal{S}$ is embedded. The *getRepresentativePoint()* service returns the d-dimensional representative point of $\mathcal{S}$, while the *getEuclideanVertices()* returns all the vertices of $\mathcal{S}$. Finally, the *sameObject()* service compares the input object $\mathcal{S}$ with an other spatial object o, checking if they are equal.

4.2. The SPDataIndex layer

The *SPDataIndex* layer allows users to interact with the data stored in the OMSM framework, hiding the implementative details. The main objective of this layer is to efficiently manage a huge set of spatial objects, described as instances of the *OmsmSpatialObject* interface (see Sect. 4.1) and to provide a custom implementation of a space partitioning tree. In other words, this layer describes a component that receives some instances of the *OmsmSpatialObject* interface (i.e. the spatial objects to be managed) as input and produces the corresponding space partitioning tree as output. All the output nodes must be sent to the *NodeHandler* layer (see Sect. 4.3) in order to update clusters.

These goals are obtained providing a suitable data model that allows to discard all the implementative details. In this layer, the data model is a generic node of a space partitioning tree, described by the *OmsmSpatialIndexNode* interface. This node contains one or more spatial objects, indexed through a representative point (or a linear combination of each representative point). In other words, a spatial object is stored in the node where its representative point is stored. In this way, all the spatial objects can be represented through their representative points, reducing their complexity. This design choice is transparent for the other OMSM layers. Since instances of the *OmsmSpatialIndexNode* interface must be stored in the OMSM framework, this interface must extend the *RawData* one in order to guarantee persistence. In order to hide the specific spatial index we are currently using, the *OmsmSpatialIndexNode* interface must also extend the *Node* interface (abstracting a generic node in the structure), as depicted in Fig. 2. In Sect. 4.3, we give a detailed motivation of this design choice.

In addition to the services offered by the *RawData* interface, the *OmsmSpatialIndexNode* interface offers the following services:

- *void addSpatialObject(OmsmSpatialObject o, int lev)*;
- *void removeSpatialObject(OmsmSpatialObject o, int lev)*;
- *bool searchSpatialObject(OmsmSpatialObject o, int lev)*;
- *list<OmsmSpatialObject> getObjects()*;
- *bool isLeaf().*

Let $\mathcal{N}$ be a node of a spatial index, described by an instance of the *OmsmSpatialIndexNode* interface. The *addSpatialObject()* service adds a spatial object o in $\mathcal{N}$, located at level *lev*. The *removeSpatialObject()* service removes the spatial object o from $\mathcal{N}$, located at level *lev*, while the *searchSpatialObject()* service looks for the spatial object o in $\mathcal{N}$ (and recursively in its children, if they exist), located at level *lev*. Finally, the *getObjects()* service returns all the objects stored in $\mathcal{N}$, while the *isLeaf()* service checks if $\mathcal{N}$ is a leaf.

In order to minimize memory requirements, we maintain in RAM only the root node of the space partitioning tree, while the other nodes are dynamically loaded from the *NodeHandler* layer (see Sect. 4.3). However, this design choice is a bootleneck and we are investigating about it.

We stated that the *SPDataIndex* layer is the most external layer in the OMSM framework and users interact with our storing architecture through this layer. We intend to provide a general system not optimized and tailored to a specific application. Thus, the *SPDataIndex* layer must offer (at least) this limited set of services:

- *void open(string dbname, bool createOn, bool rdOnly)*;
- *void close();*
- *void addSpatialObject(OmsmSpatialObject o).*

The *open()* service connects the current instance of the OMSM framework with a 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 updates of the required data source. The *close()* service disconnects the current instance of the OMSM framework from its data source. Finally, the *addSpatialObject()* service stores a spatial object o in the current instance of the OMSM framework.

4.3. The NodeHandler layer

The *NodeHandler* layer groups nodes of a spatial index into clusters in order to improve the I/O bandwith. We recall that a cluster is a set of nodes, considered as an atomic unit: an I/O operation, delegated to the *NClusterStorager* layer (see Sect. 4.4), must be performed on a single cluster. The goal of this operation is to provide a compromise between the high latency of EM accesses and the amount of data being transferred on a storage support. Moreover, we should understand what nodes could be useful in the future, avoiding an EM access. A cluster can contain only one node, but this clustering policy is not optimal [DSS96]. In this case, we have a lot of high latency operations and each of them transfers only one node. Thus, each future operation will require a new EM access. This aspect is very critical in a storing architecture and efficient clustering policies must be designed.

We need a special structure, equivalent to a super-block in a filesystem, containing information about the current configuration of the OMSM framework. It is described by the *SuperNode* interface and, for the sake of simplicity, it is considered equivalent to a generic node of a spatial index.Thus, the data model of the *NodeHandler* layer is either an instance of the *OmsmSpatialIndexNode* or the unique instance of the *SuperNode*. Thus, we need a generic interface, called *Node*, in order to hide implementative details between the above interfaces. As consequence, the *NodeHandler* layer manages only instances of the *Node* interface. Instances of such interface must be persistent and, thus, it must extend the *RawData* interface (see Sect. 4.1), as depicted in Fig. 2.

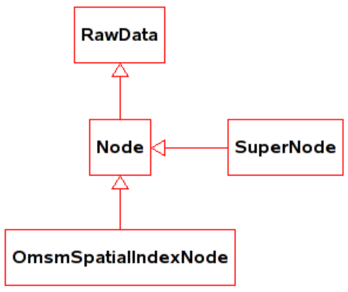

Figure 2: *the inheritance relation between interfaces describing the nodes types in the OMSM framework.*

Now, we can analyze the most important services offered by the *NodeHandler* layer:

- *void addNode(node_id nid, Node n)*;
- *bool removeNode(node_id nid)*;
- *bool searchNode(node_id nid)*;
- *Node getNode(node_id nid)*.

The *addNode()* service adds the node *n* with *nid* as identifier in a cluster of the current subdivision. The *removeNode()* and the *searchNode()* services respectively removes and looks for a node with *nid* as identifier in the current subdivision. Finally, the *getNode()* service loads a node with *nid* as identifier from the current subdivision in clusters.

4.4. The NClusterStorager layer

The *NClusterStorager* layer manages low-level representations of nodes clusters to be written or read from a storage support. The particular type of storage support is *hidden* by the current layer interface: for example, we can access a remote database or write each cluster on an independent file, through its interface. In the *NClusterStorager* layer, the data model is a pair (id, ba), where *id* is the unique identifier for the cluster *c* of interest, while *ba* is the sequence of bytes describing the cluster *c*. We recall that a cluster is an object that can be stored in the OMSM framework, i.e. a subclass of the *RawData* interface. Thus, we can extract its representation in a straightforward way, by invoking the *getBytes()* service (see Sect. 4.1).

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

- *void addCluster(cluster_id cid, byte_sequence ba, int lg)*;
- *bool removeCluster(cluster_id cid)*;
- *bool searchCluster(cluster_id cid)*;
- *void getCluster(cluster_id cid, byte_sequence ba, int lg)*.

The *addCluster()* service adds a cluster in the storage support: the input cluster has *cid* as identifier and it is described by the bytes sequene *ba*, composed by *lg* bytes. The *removeCluster()* and the *getCluster()* services respectively removes and looks for the cluster with *cid* as identifier from the storage support. Finally, the *getCluster()* service loads the cluster with *cid* as identifier from the storage support: if this cluster exists, then its representation (composed by *lg* bytes) will be contained in the bytes sequence *ba*, otherwise the output parameters will be undefined.

5. Concluding remarks and future extensions

In this paper, we have proposed the *Objects Management in Secondary Memory* (*OMSM*) framework for managing huge geometric models. It can be easily adapted to the user needs through dynamic plugins, providing many techniques to be integrated in a storing architecture. Thus, a new technique can be made available without messing with all the structure, by writing an appropriate extension for this framework. Our work is orthogonal to the currently available libraries: such frameworks address the implementation issues behind new methods by removing the burden of writing structural maintenance code from the developer point of view. The OMSM framework aims to simplify applications development: employing the OMSM framework requires to implement plugins compliant to a concise set of interfaces. Existing libraries can be used for simplifying the client code development as well, but they are not targeted primarily to be extensible and they do not promote transparent use of different techniques.

Thanks to its extensible structure, the OMSM framework can be extended in many directions and 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 is the management of *simplicial* or *cell complexes* [Ago05]. Such structures are usually described by a topological data structure and thus we can offer an unique platform in order to describe an EM version of a topological data structure. As result, we decouple storage aspects from the combinatorial description of a complex, obtaining a solution more general than the technique 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.

Performance of storing architectures for repeated *inserting* operations could not be satisfactory, when inserting a large amount of data. To overcome this drawback, a technique of *bulk construction* for EM data (and in particular an EM topological data structure) is needed. With such technique, we can obtain a better storage utilization and better query answering performance. Another related problem is *bulk insertion*. In contrast to bulk loading (where an index is built from scratch), such technique aims to update an existing structure with a large set of data. We are investigating about these problems and we are looking for a general solution, in the same spirit of the OMSM framework.

References

[ABA06] The Allen Brain Atlas Project, 2006. The Allen Institute for Brain Science, http://www.brain-map.org.

[Ago05] AGOSTON M.: *Computer graphics and geometric modeling*. Springer, 2005.

[AI01] AREF W., ILYAS I.: SP-GiST: an extensible database index for supporting space partitioning trees. *JIIS Jour.* (2001).

[AS94] AGARWAL P., SURI S.: Surface approximations and geometric partitions. In *Proc. of 5th Symp. about Discr. Algo.* (1994).

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

[BEG94] BERN W., EPPSTEIN D., GILBERT J.: Provably good mesh generation. *Jour. of Comp. and Sys. Sciences* (1994).

[BMC72] BAYER R., MC-CREIGHT E.: Organization and mantenance of large ordered indices. *Acta Inf.* (1972).

[CBPS06] CALLAHAN S., BAVOIL L., PASCUCCI V., SILVA C.: Progressive volume rendering of large unstructured grids. *IEEE Trans. on Vis. and Comp. Graph.* (2006).

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External memory management and simplification of huge meshes. In *IEEE Trans. on Vis. and Comp. Graph.* (2003).

[DDFM*06] DANOVARO E., DE FLORIANI L., MAGILLO P., PUPPO E., SOBRERO D.: Level-of-detail for data analysis and exploration: a historical overview and some new perspectives. *Comp. Graph.* (2006).

[DFFMD08] DE FLORIANI L., FACINOLI M., MAGILLO P., DIMITRI D.: A hierarchical spatial index for triangulated surface. In *Proc. of the GRAPP Conf* (2008).

[DSS96] DIWAN A., SESHADRI S., SUDARSHAN S.: Clustering techniques for minimizing the external path length. In *Proc. of the VLDB Conf.* (1996).

[EEA06] ELTABAKH M., ELTARRAS R., AREF W.: Space-partitioning trees inside PostgreSQL: realizing and performance. In *Proc. of the 22nd ICDE* (2006).

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[HNP95] HELLERSTEIN J., NAUGHTON J., PFEFFER A.: Generalized search trees for the database systems. In *Proc. of the VLDB Conf.* (1995).

[Hop98] HOPPE H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proc. of IEEE Vis. Conf* (1998).

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARNSHEY A., WATSON B., HUEBNER H.: *Level-of-detail for 3d graphics*. Morgan Kaufmann, 2002.

[Pri00] PRINCE C.: *Progressive meshes for large models of arbitrary topology*. Master's thesis, University of Washington, Washington DC, USA, 2000.

[Psq96] PostgreSQL, 1996. http://www.postgresql.org.

[Sam06] SAMET H.: *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[SB98] SMARAGDAKIS Y., BATORY D.: Implementing reusable object-oriented components. In *Proc. of the ICSR Conf.* (1998).

[SCESL02] SILVA C., CHIANG Y., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *Proc. of IEEE Vis. Conf.* (2002).

[VCL*07] VO H., CALLAHAN S., LINDSTROM P., PASCUCCI V., SILVA C.: Streaming simplification of tetrahedral meshes. In *IEEE Trans. on Vis. and Comp. Graph.* (2007).

[vdBBD*01] VAN DEN BERCKER J., BLOHSFELD B., DITTRICH J., KRAMER J., SCHAFER T., SCHNEIDER M., SEEGER B.: XXL - a library approach to supporting efficient implementations of advanced database queries. In *Proc. of the VLDB Conf.* (2001).