In Proceedings 14th International Conference on Data Engineering, ICDE '98, pages 550–557. IEEE, 1998.

The Alps at your Fingertips: Virtual Reality and Geoinformation Systems

Renato Pajarola[†]

Thomas Ohler[*] Peter Stucki[‡] Kornel Szabo[‡] Peter Widmayer[†]

[†] Computer Science Department
Institute of Theoretical Computer Science
ETH Zurich, Switzerland

[‡] MultiMedia Laboratory
Computer Science Department
University of Zurich, Switzerland

[*] isys software gmbh
Engelbergerstr. 21
79106 Freiburg i.Br., Germany

Abstract

We advocate a desktop virtual reality (VR) interface to a geographic information system (GIS). The navigational capability to explore large topographic scenes is a powerful metaphor and a natural way of interacting with a GIS. VR systems succeed in providing visual realism and real-time navigation and interaction, but fail to cope with very large amounts of data and to provide the general functionality of information systems. We suggest a way to overcome these problems. We describe a prototype system that integrates two system platforms: A client that runs the VR component interacts via a (local or wide area) network with a server that runs an object oriented database containing geographic data. For the purpose of accessing data efficiently, we describe how to integrate a geometric index into the database, and how to perform the operations that are requested in a real-time trip through the virtual world.

1 Introduction

More than a decade ago, geographic information systems (GIS) have started to play a significant role in a large number of application domains. These applications become more and more complex and require the maintenance of an ever increasing variety and amount of data. To be useful even for a non-expert user, a GIS should allow queries to be posed in a natural way. For the geometric part of a query, it appears natural to let the user explore a virtual world of her choice — indeed quite a powerful metaphor of user interaction [3]. Proper visualization of the scene and all relevant data at the same time is then crucial for the ease of use of the system as a whole [9]. Such a GIS interface could in fact be one of the prime examples of a post-WIMP[1] user interface [17].

A trip through a virtual world is useful only if it provides latency-free interaction. In particular, it must be possible to move through a terrain and see the changes in the scenery in real-time. This requirement for fast data delivery to the simulating VR application calls for an extraordinarily efficient GIS. In particular, the geometric terrain data must be accessed efficiently, while at the same time they should be maintained in the GIS in order to support typical database functionalities, such as transactions and recovery. We propose a way to achieve this efficiency.

The system we propose, ViRGIS (for *Virtual Reality GIS*), maintains three-dimensional terrain data in vector form (such as surface triangulations), raster data (such as those from satellite images and topographic maps), and non-geometric data (such as population counts of cities). It allows a user to move through the scene in real-time by means of a standard input device such as a mouse, and to interact with the data in the GIS (through a point-and-click interface with pop-up windows for non-geometric data). Thus far, we have not implemented virtual reality in- and outputs in the strong sense, such as interaction via 3D input devices or head-mounted displays; our interface is a *desktop VR* or video user interface in the classification of [1]. The basic architecture and concept of ViRGIS also served as a framework for other VR information systems, e.g. in tourism [16].

Real-time walk-through in three-dimensional scenes is nothing new: a number of software tools are available that supportit. Usually, they limit their walk-through capabilities to a scene that is loaded once and stored in main memory. The scene as a whole (but not parts of it) can

[1] WIMP stands for windows, icons, menus, and a pointing device

be replaced by another one. While this is good enough for several applications, it is far from satisfactory for geo-exploration, where gigabytes (or even terabytes) of data need to be explored. Our virtual reality component allows to dynamically maintain a part of a very large scene in main memory, to be visualized using the *IRIS Performer Toolkit*[2]. To get the data from the database into the visualization component, we propose a storage and retrieval scheme for location-oriented dynamic loading that supports geometric proximity queries. Our scheme maintains any geometric index, for instance an *R-tree* [8], directly within an object oriented database in such a way that geometric objects are clustered physically according to their location in space.

Since the GIS data might reside on a workstation to which many users' workstations are connected, we run the graphics software (such as the IRIS Performer Toolkit) on the user side, while the database runs on the database server.

The performance bottleneck of ViRGIS turns out not to be the database (due to our integration of a geometric index) and not only the network, but also the graphics engine. To reduce the graphics load, ViRGIS displays exact data only where they are useful. In particular, a user should see all the available details of a scene when she can see them in reality, that is, in the close neighborhood of the viewpoint. The farther away we look, the fewer details we can see in reality and need to see in a virtual world. We therefore maintain data in various *levels of detail (LOD)* and access only those levels that are needed. Lower resolution and fewer details are satisfactory not only far from the viewpoint, but also when a flight over a scene is so quick that the eye cannot capture details anyway; this physiological feature corresponds nicely to a reduced graphics load when the image needs to change rapidly. With the LOD concept, we arrive at a sufficiently low amount of graphics data per time unit to be displayed, for typical graphics engine capacities[3].

As to the database side, it turns out to be appropriate to incorporate the geometric index into an object oriented database system, thus combining efficient access with all the advantages of object orientation. Because in a trip through a scene, the LOD changes as we get closer to some location, access to data should not be on the basis of location alone, but also according to the LOD desired. Note in passing that in principle, a geometric index can also be built into a relational database system [11], and a GIS should of course come with such an index in any case (like e.g. the *Smallworld GIS*[4] that comes with an overlapping quadtree, maintained as just one table in the database). For ViRGIS, we chose *ObjectStore*[5] as the underlying object oriented database system. In this paper, we show how an R-tree can be built into an ObjectStore database so as to support all necessary operations sufficiently fast for a real-time trip through a virtual world.

The remainder of this paper is organized as follows. In the next section, we describe the overall system architecture. Section 3 describes how to choose an appropriate geometric index, and Section 4 shows how to integrate it into an object oriented database system. Section 5 discusses the use of varying levels of detail, and Section 6 supports our design decisions with a few experiments. Section 7 concludes the paper.

2 Overall architecture of ViRGIS

The efficiency problem that current VR systems have whenever the memory requirements exceed available main memory, is one of the reasons for combining in ViRGIS two loosely coupled components, one for visualization and one for data handling. Figure 1 shows the overall system architecture of ViRGIS in which these two components communicate over a (local area or wide area) network. Multiple clients are allowed to connect to the database, and the clients can independently request data. In the visualization component, most of the work is performed by the *scene-manager* which takes care of updating the visible scene dynamically according to the user's movements. The visible scene is divided into rectangular patches, and new sets of patches are loaded from the database on demand; outdated or invisible ones are discarded[6]. The scene-manager also maintains the LODs for every patch, and keeps track of what is already loaded and what needs to be requested from the database.

The database component maintains the terrain, triangles, and other geometric objects in a spatial access structure within a database. Raster and image data is stored in a hashing structure which also handles overlapping textures at different resolutions. Each image (or raster data-set) is partitioned into a set of rectangular patches standing for the smallest access units to this image. A hash-directory – a matrix with pointers – allows for fast look-up of every patch. Both data structures, for geometries and raster data, support arbitrary rectangular range queries. A server application provides network access to the databases. It opens the required databases for the different data types and LODs, and then listens for incoming requests. Each request is trans-

[2] The *IRIS Performer Toolkit* is a product of *SiliconGraphics, Inc.*

[3] SGI Indigo2 IMPACT 10000 renders 652k textured and Gouraud shaded triangles per second

[4] *Smallworld GIS* is a product of *Smallworld Systems Ltd.*

[5] *ObjectStore* is a registered trademark of *Object Design, Inc.*

[6] A cache buffer for patches speeds up the update process for recently visited regions.

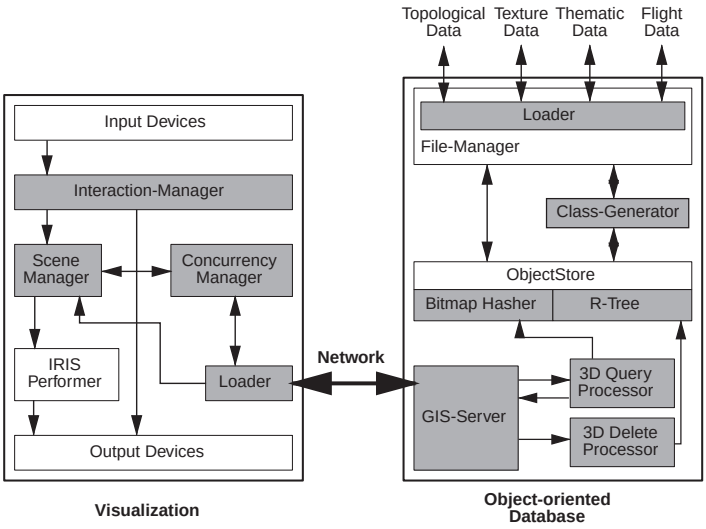

Figure 1. System Architecture

formed into a query on the data structures, and the result is written back on the network in a predefined format.

3 Spatial access structures

We focus on spatial access structures for maintaining a set of multidimensional interval or point data. The objects to be stored consist of a multidimensional *geometric* key attribute and possibly further attributes. We assume that complex geometric keys like polygons or polylines are approximated by a minimal enclosing axis-parallel rectangle, a *bounding box*. Spatial access structures must support the dictionary operations (insertion, deletion, exact match query) as well as *proximity queries* based on the neighborhood of objects in the Euclidean data space in which the geometric key is defined. An important type of proximity queries are *range queries*: They ask for all objects whose geometric key intersects a given query range. In our application, range queries are used by the scene manager to read new parts of the world scene into main memory.

To support proximity queries efficiently, a spatial access structure aims at grouping together objects that are close in space – such that each group can be stored in a block on external storage. This *clustering* of objects is done to map the neighborhood relation in space to external storage. To guide a search, access structures maintain a set of – usually rectangular – regions. With each region, a *data block* is associated which stores those objects contained in (for some structures: intersecting) this region. The mapping from regions to blocks can either be stored in an appropriate *directory* data structure (as given e.g. for the R-tree [8]) or can be computed for regular partitions of the data space based on a hash function (as used e.g. for Z-hashing [12]). A thorough treatment of spatial access structures and their applications can be found in [14, 15].

One of the most important spatial access structures designed for handling multidimensional non-point data is the R-tree [8]. We have chosen it for our application because of its good range query performance, but also its simplicity and conceptual clarity. Nevertheless, let us emphasize that our approach can easily be adapted not only to other spatial access structures but also to many other external storage data structures.

The R-tree can be seen as a B^+-tree variant with *rectangular regions* as keys. The inner, *directory* nodes maintain set of pairs (*rectangle, pointer*) referring to another node in the tree, and the rectangle component describes the bounding box of the geometric keys of all objects in the subtree of the pointer. A leaf node stores a block of data.

Figure 2 shows on the left side an example R-tree with two directory levels. The root directory node represents three subregions which themselves are covering several regions on the level below. The corresponding tree structure is shown on the right side.

A range query is performed on the R-tree almost the same way as in a B^+-tree. Beginning at the root, the search follows those pointers whose associated regions have a non-empty intersection with the query region. All the objects found in the data blocks, whose geometric key intersects the query region, are processed further for the final result after this rough selection.

Insertion is done by traversing a path from the root to a leaf, in each node following the pointer corresponding to a region which encloses the geometric key of the object to be inserted. If no such region exists, a region which needs minimal enlargement to cover the key is chosen. The ob-

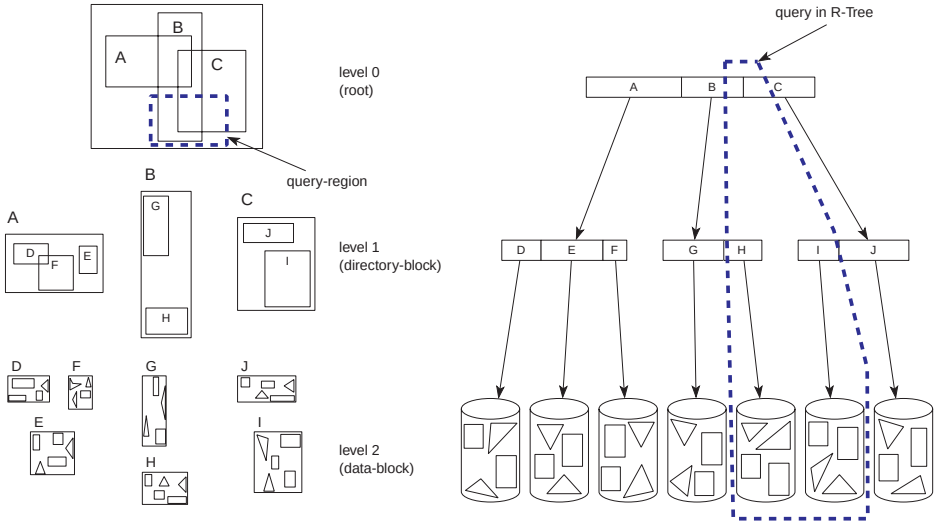

Figure 2. R-tree regions and structure

ject is inserted into the data block where the search ends. On overflow, *splitting* occurs similar to the the B^+-tree and may propagate up to the root. A detailed description of the algorithms can be found in [8]; a variation of the R-tree that outperforms the original, the R^*-tree, is described in [2].

Usually, the R-tree directory nodes as well as the data blocks are maintained as external storage blocks in a file. Then the contents of a node or a data block can be read or written by one block access. Now, when embedding an access structure like the R-tree into a database system, to preserve its efficiency as good as possible in comparison to a file-based implementation, the following requirements must be fulfilled:

1. The database system must preserve the clustering of objects or entries in a data structure block, i.e. objects or entries grouped in a node of the data structure must also be grouped in a block in the database system.

2. The database system must provide access to a single block avoiding unnecessary overhead in time, e.g. as caused by reading several blocks which possibly are not needed by the algorithm of the structure.

As simple as these requirements may appear, as difficult it is to achieve these goals when implementing an access structure on top of a commercially available database system. In [11], it is shown that in principle a hierarchical access structure (the LSD-tree [10]) can be implemented on top of the relational DBMS *Oracle*. However, to fulfill the requirements above, the approach must exploit some *Oracle*-specific features, which leads to a solution of restricted generality. Moreover, the implementation had to be done on top of the SQL-interface, leading to a considerable overhead in time. For a highly efficient integration of an access structure with a database system, the underlying database system must allow the direct control of clustering and access pattern used for the physical storage of blocks. While relational DBMSs do not give this freedom to the user, many object-oriented database management systems (OODBMSs) do. In the next section, we show how this feature can be exploited to build a highly performant implementation of a hierarchical access structure in an OODBMS.

4 Implementation of an R-tree in ObjectStore

4.1 ObjectStore

ObjectStore is a commercially available object-oriented database system which provides persistence of C++ objects overriding the standard allocation methods. *ObjectStore* also features the usual DBMS transaction mechanisms like recovery and concurrency control. In addition to persistent objects, references to other objects are kept persistent too, which allows to navigate through permanently stored object systems. The persistence of references, which means correct retrieval of dereferenced objects from the database, is achieved by a *Virtual Memory Mapping Architecture (VMMA)* which handles dereferencing of persistent pointers.

Each *ObjectStore* database is divided into *segments* and these again may have *object-clusters* as subdivisions. Segments and object-clusters both consist of fixed sized memory *pages* which are managed by the built-in VMMA. In contrast to object-clusters, whose size is a fixed number of pages, segments are flexible in their size, and grow or

shrink according to the data stored in them. The organization of segments and object-clusters, and the reload strategy of pages from the server can be set by the programmer to optimize the application's performance.

4.2 Implementation

Usually, data blocks and directory nodes of spatial access structures correspond directly to blocks on external storage. In our approach we use *ObjectStore* to achieve the persistence of the data, thus the access structure is modeled as an abstract *ObjectStore* data type, and the algorithms use the built-in memory allocation facilities to manage the required clustering.

4.2.1 Classes

First, we will describe the classes which make up the R-tree in *ObjectStore*. Basically, we created a small framework with one interface class which manages access to the R-tree. Figure 3 shows the classes and the membership relations of the R-tree and the geometric objects which can currently be stored in the access structure. The R-tree can handle any objects which are instances of a subclass of Geometry.

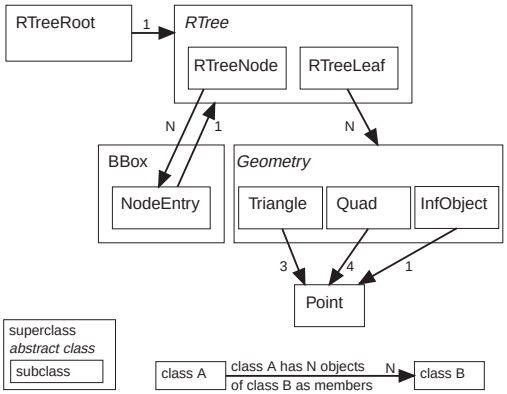

Figure 3. Class Framework

RTreeRoot is the interface class to the R-tree access structure and provides the necessary protocol to the user to build up an R-tree and to retrieve elements from it. The abstract class RTree implements the common interface protocol of the directory- and leaf-nodes. This protocol includes the dictionary operations *insert*, *delete* and *range query*. The directory nodes of type RTreeNode handle the recusive subdivisions; the leaf nodes or data blocks, where the geometric objects are actually stored in, are of type RTreeLeaf. RTreeNode and RTreeLeaf both implement the same split strategy to distribute the elements over two nodes or two leaves if an overflow occurs. The elements of a directory node are objects of type NodeEntry, which is a subclass of BBox, the actual key of such an entry, and contains a pointer to an RTree object (RTreeLeaf or RTreeNode).

As mentioned above, data objects must inherit the protocol from Geometry, and provide the intersection tests to guide the search through the tree. Currently two geometric objects, Triangle and Quad, are implemented to model a digital elevation model (DEM), and a geometric point object exists to maintain non geometric attributes associated to a location in space.

The interface provided by RTreeRoot includes methods which *create* a new R-tree or *open* an existing one, and setup the necessary working environment for the efficient use of the access structure. On overflow during an insert, the *split* method is called and is propagated upwards as necessary. An underflow is resolved by collecting all entries of the node and consecutive re-insertion of them. RTreeNode and RTreeLeaf implement Guttmans split algorithms with quadratic, $O(m^2)$, run-time complexity [8] and with $O(m \log m)$ complexity [2] for m elements to split.

4.2.2 Clustering

The previous section explained how the abstract data type R-tree is modeled using inheritance and aggregation. Nothing, however, was said about the *physical memory layout* which is the essential idea and the critical point of performance of an R-tree implementation.

ObjectStore organizes its memory space in *pages* which are the smallest network transfer and disk storage units, and thus are very similar to disk blocks. Furthermore, the fetch policy of *ObjectStore* and the size of object-clusters may uniformly be set to one page (or any other fixed number of pages), which permits control over the physical memory layout as desired. Additionally, *ObjectStore's* caching strategy supports an effective use of a hierarchical, tree-like data structure because the client's page replacing algorithm is based on a *least recently used* principle, and this helps to keep the frequently used top nodes in the client's cache.

Figure 4 shows the resulting physical memory layout in terms of the usage of segments and object-clusters. Each complete R-tree structure is stored in one segment and each node covers exactly one object-cluster within the same segment. The R-tree interface object RTreeRoot is not tied to any object-cluster like the other nodes because it doesn't hold any data to be clustered, and is used whenever accessing the R-tree. All objects inheriting from RTree cover one object-cluster, and follow the clustering principle by holding the node's elements in that same object-cluster. Especially the leaf nodes must keep the geometric objects they reference in the same object-cluster. The constructors of RTreeNode and RTreeLeaf take full advantage of any

sized object-cluster, holding as many NodeEntry or Geometry as possible.

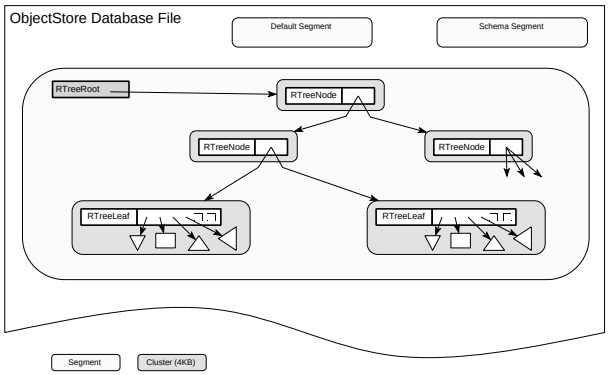

Figure 4. Clustering

5 Taking advantage of levels of detail

The appropriate use of different levels of detail (LOD) for different parts of the visible scene can significantly reduce the number of geometric primitives which have to be rendered. Hence, a larger scene can be displayed at the same frame-rate. Furthermore, the use of different LODs also leads to an enhanced realistic impression of the scene, since the farther the objects are, the lower we choose their LOD. The LOD strategy may differ between applications: A *view-point centered* strategy gives lower LODs to objects farther away from the view-point itself, a *view-direction oriented* strategy assigns lower LODs to objects farther away from the view-direction line. Other strategies might be considered as well.

A range of problems arise when considering LODs for a digital terrain model. First of all, the different LODs could be represented as uniform height fields, as a discrete number of separate triangulations, or as dynamic point sets or triangulations which are updated according to the LOD parameter. Connected to the representation is the selection of relevant terrain points for a specific LOD, in the first case sub-, super-sampling and interpolation can be used. The second choice allows any type of selection if a suitable triangulation can be computed on it. In the last case the selection is closely related with the triangulation itself. Height fields are easy to maintain, both on the database side and for visualization. However, their adaptivity to the actual terrain – and therefore the visual impression – is quite bad, because the point selection depends on position and not on terrain features, such that smoothing and omission of relevant details are the consequences. Maintaining a fixed set of different triangulations allows fast selection of the triangles for one specific LOD out of a discrete number of predefined ones. An overview on how to calculate a surface triangulation or select a point-set for a specific LOD can be found in [13]. However, no continuous or error-driven LOD-threshold can be supported, and the data storage costs are very high. Providing a terrain representation which supports continuous LOD selection and produces a terrain-adaptive triangulation is quite hard, because to do so, the triangulation has to be constructed on demand for every selection (this can be very time-consuming). This is also true for hierarchical triangulation models [7] which also have problems answering rectangular range queries as needed in a dynamic environment.

Because the LOD changes as we get closer to some location, it should be efficient to dynamically load additional data from disk in order to refine a region that is already in core at a low resolution. This goal is difficult to achieve for surface triangulations [6, 4, 5], because different LODs in adjacent triangles may lead to cracks in the terrain, the well-studied *sliver* polygons, unless special care is taken to neatly stitch adjacent triangles together – not an easy task.

In ViRGIS, the visualization uses a viewpoint centered LOD strategy as shown in Figure 5, which also incorporates a speed threshold to select a lower LOD. We maintain a fixed number of different LODs to speed up the interaction between the visualization and the database component. The database handles a triangulation for every LOD; the triangulation may be generated with any of the methods mentioned in [13].

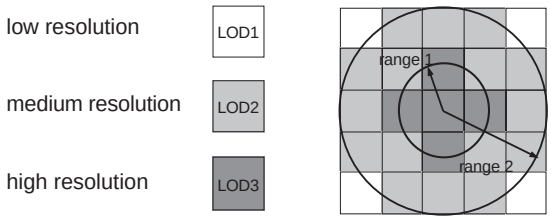

Figure 5. LOD Strategy

6 Experiments

For our specific computing environment, experiments have shown some of the advantages and disadvantages of using different LODs. First, let us look at the static setting where no update of the visible scene is necessary. The visualization component holds 5×5 patches, each of size 5000×5000 meters, where the numbers of triangles in the three LODs for one patch are 72, 286 and 965. We compared two different settings: (I) All patches are of medium LOD, and (II) different LODs are used, as shown in Figure 6. This results in 7150 triangles to be displayed in setting (I), and $1 \times 965 + 8 \times 286 + 16 \times 72 = 4405$ triangles in setting (II). Thus, with the use of different LODs, the rendering complexity can be reduced significantly without

significant loss in image quality, and therefore a larger visible scene or a higher frame-rate can be achieved.

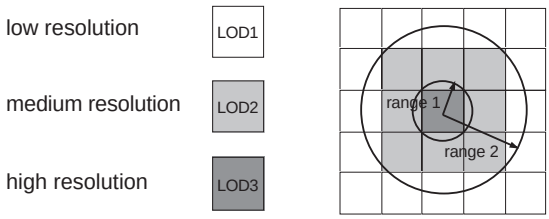

Figure 6. LODs in Experiments

Our network is fast enough that its performance influences the system only if we move through the scenery at very high speed. Then, the update over the network needs more time to transfer the requested data than the viewpoint needs to pass that region. To cope with that problem we can simply lower the LOD accordingly to reduce the amount of data. Therefore, the scene update is only the second bottleneck, clearly less critical than the rendering performance bottleneck. Nevertheless, we compared the network update time – database access and transfer over the network – to the pure database access time, and we also compared single medium LOD update to multiple LOD update. The database maintains about 64 times the area of the visible scene for every LOD. Figure 7 displays the loading times for the initial 25 patches in every LOD, and shows that the network needs roughly 5 to 10 times longer to transfer one patch than the database needs to retrieve it, i.e., for processing the query and reporting the result. Furthermore, we can see that the loading time increases more slowly than the number of triangles for the different LODs, and also that cache misses in the database system might cause the peaks in Figure 7 a).

Figure 8 shows the loading times, again for the database only (a) and together with the network (b), for dynamic updates of the visible scene. The y-axis denotes the cumulated milliseconds used to query (a) and transmit (b) the data for the initial, visible startup-scene and the three updates. Using the same two LOD settings as above – single medium LOD and the multiple LOD from Figure 6 – we observed three updates according to the user's movements of the viewpoint. In the medium LOD setting, each update had to query and transfer $5 \times 286 = 1430$ triangles, with multiple LODs $1 \times 965 + 3 \times 286 + 5 \times 72 = 2183$ triangles had to be transmitted. Again the network turns out to be the bottleneck compared to the database access time, and this outweighs the differences in the LOD strategies.

7 Conclusion

Although a lot of sophisticated spatial access structures have been developed in the past decade, not many such structures have been integrated into commercially available database systems by the software producers itself; the Smallworld GIS with its overlapping quadtree is one of the few exceptions. In this paper we have shown at an example that object-oriented database systems, providing the necessary facilities of controlling clustering and block access, can be used as a good basis for such an integration. Our approach can easily be adapted to other OODBMSs if they provide persistent abstract data types and physical clustering of objects. The ability of creating persistent abstract data types should be a basic ingredient of every OODBMS, and also the physical clustering is important in general for the retrieval performance of complex abstract data types. Therefore, these two requirements don't seem to limit the applicability of our approach to other OODBMSs. Furthermore, also objects with non-geometric attributes that have a spatial relation may be stored in and accessed through our data structure if they provide the correct interface.

We have taken advantage of the geometric index to prototypically implement a virtual reality user interface to a geographic information system. Neither the geometric index nor the OODBM system nor the visualization software are unique in their choice; other geometric data structures and other available software components might replace them. Practical experience with the proposed virtual reality interface is needed in order to assess its ease of use and the way to proceed.

Acknowledgments

We acknowledge the contributions of our diploma students P. Aschwanden, F. Kagi, M. Roth, S. Verscharen and S. Odendahl to ViRGIS.

References

[1] P. W. Agnew and A. S. Kellerman. *Distributed Multimedia – Technologies, Applications, and Opportunities in the Digital Information Industry: A Guide for Users and Providers*. ACM Press and Addison Wesley Publ. Co., Reading, Massachusetts, 1996.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, pages 322–331. ACM, 1990.

[3] R. Coyne. *Designing Information Technology in the Postmodern Age*. The MIT Press, Cambridge, Massachusetts, 1995.

[4] M. de Berg. Visualization of TINs. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of Geographic Information Systems, Summerschool, Udine*, Lecture Notes in Computer Science. Springer Verlag, 1997.

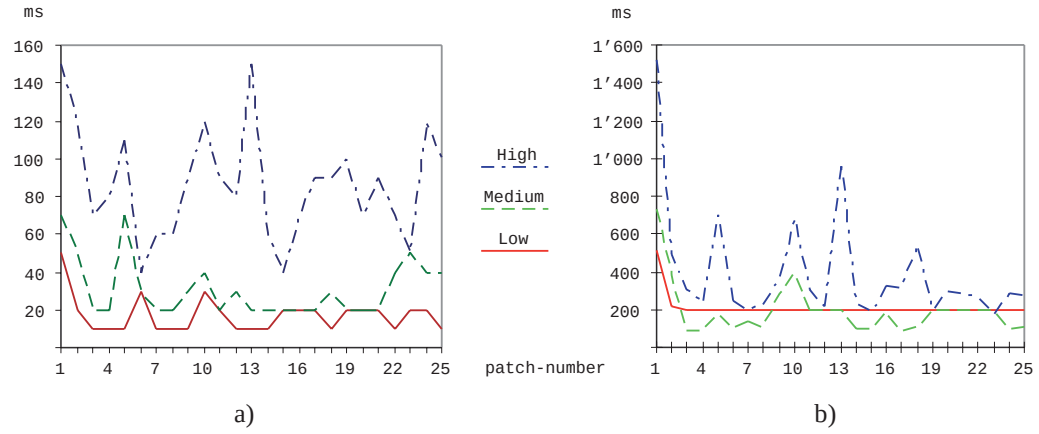

Figure 7. a) Database access, b) plus network transfer.

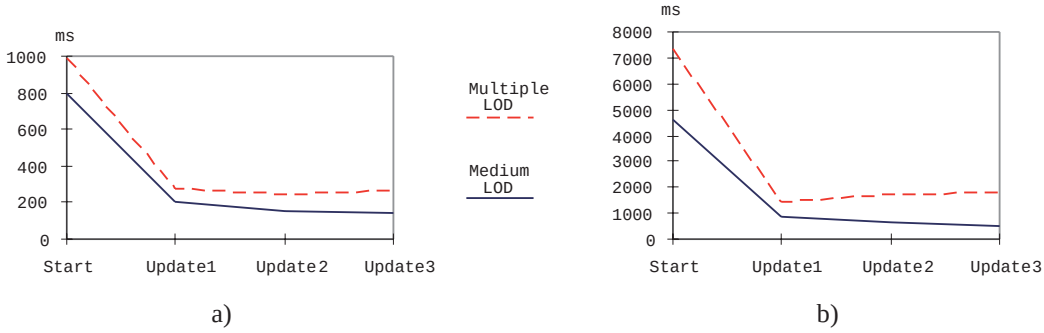

Figure 8. a) Update query in database, b) plus network transfer.

[5] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *11th ACM Symposium on Computational Geometry*, pages C26–C27. ACM, June 5 - 7 1995.

[6] L. De Floriani, P. Marzano, and E. Puppo. Multiresolution models for topographic surface description. *The Visual Computer*, 12(7), August 1996.

[7] L. De Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, 1995.

[8] A. Guttman. A dynamic index structure for spatial searching. In *Proceedings of the* 13^{th} *ACM SIGMOD International Conference on Management of Data*, pages 47–57. ACM, 1984.

[9] H. M. Hearnshaw and D. J. Unwin, editors. *Visualization in Geographical Information Systems*. John Wiley & Sons, Chichester, 1994.

[10] A. Henrich. *Der LSD-Baum: eine mehrdimensionale Zugriffsstruktur und ihre Einsatzmöglichkeiten in Datenbanksystemen*. PhD thesis, FernUniversität Hagen (Germany), 1990.

[11] A. Henrich, A. Hilbert, H.-W. Six, and P. Widmayer. Anbindung einer räumlich clusternden Zugriffsstruktur für geometrische Attribute an ein Standard-Datenbanksystem am Beispiel von Oracle. In *GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft*, volume 270 of *Informatik-Fachberichte*, pages 161–177. Springer-Verlag, 1991.

[12] A. Hutflesz, H.-W. Six, and P. Widmayer. Globally order preserving multidimensional hashing. In *Proceedings of the* 4^{th} *International Conference on Data Engineering*, pages 572–579, 1988.

[13] J. Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *International Journal of Geographic Information Systems*, 5(3):267–285, 1991.

[14] H. Samet. *Applications of Spatial Data Structures: computer graphics, image processing, and GIS*. Addison Wesley Publ. Co., Massachusetts, 1989.

[15] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley Publ. Co., Massachusetts, 1989.

[16] K. Szabo, P. Stucki, P. Aschwanden, T. Ohler, R. Pajarola, and P. Widmayer. A virtual reality based system environment for intuitive walk-throughs and exploration of large-scale tourist information. In *Enter95*, January 1995.

[17] A. van Dam. Post-WIMP user interfaces. *Communications of the ACM*, 40(2):63–67, February 1997.