Proc. 11th International Conference on Multi Media Modeling (MMM 2005), Melbourne, Australia, pp. 352-357

Database Support for Haptic Exploration in Very Large Virtual Environments

Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, Matthias Renz
University of Munich, Germany
{kriegel, kunath, pfeifle, renz}@dbs.informatik.uni-muenchen.de

Abstract

The efficient management of complex objects has become an enabling technology for modern multimedia information systems as well as for many novel database applications. Unfortunately, the integration of modern database systems into human centered virtual reality applications including multimodal simulations fails to achieve the indispensably required interactive response times. In this paper, we present an approach which achieves efficient query processing along with industrial-strength database support for real time haptic rendering systems which compute force feedback (haptic display). Our approach externalizes and accelerates the approved main-memory Voxmap-PointShell™ (VPS) approach. We group numerous independent database queries together according to a cost model which takes statistical information reflecting the actual data distribution into account. The performance of our approach is experimentally evaluated using a realistic data-set CAR, provided by our industrial partner, a German car manufacturer. Our results show that we can achieve satisfying rendering frame rates using the presented access techniques.

1. Introduction

In the past decades the interest in the science of haptics has increased enormously. In combination with Virtual Reality (VR) applications, haptic rendering enables to simulate a physical environment in such a way that humans can readily visualize, feel, explore and interact with the objects in the environment.

A realistic virtual environment typically consists of thousands of objects which may occupy gigabytes of storage space, especially for high resolution objects. In order to allow multiple users concurrently exploring the same data space we have to provide shared access to the data. In order to fulfill these requirements we integrate an off-the-shelf database system into a haptic rendering system for the management of the complete environment data. Thereby, we use the substantial advantages of modern database systems, such as logical and physical data independence, concurrency control, recovery and security [1]. In our approach, we employ the object-relational data model, as it is widely accepted and implemented by many database systems. Its extensibility is a necessary precondition for the seamless embedding of spatial data types and operations.

2. Related Work

To the best of our knowledge, there does not exist any published work that addresses the issue how to combine haptic rendering methods with novel database technologies, so that the force computation can be externalized. For that reason, we separately point out the related work on haptic rendering and on spatial query processing.

Haptic Rendering. There are numerous haptic rendering algorithms for virtual simulations, differing in object and surface modeling, a survey is given in [5]. The *Voxmap-PointShell™* (VPS) approach of McNeely, Puterbaugh and Troy [6] is very promising due to constant sample rates, independent of the static environment. In this paper, we present an externalization of the VPS approach which still provides high haptic rendering frame rates.

Relational Spatial Query Processing. The database integration into the haptic rendering system requires that collision queries are forwarded to the database query engine. In this paper, we focus on the acceleration of these database queries, in order to achieve the performance requirements of the haptic rendering systems.

Numerous spatial query algorithms have been proposed over the last decades. For a general overview on spatial index structures, we refer the reader to the surveys of Manolopoulos, Theodoridis and Tsotras [7] or Gaede and Günther [2]. Most of them rely on the paradigm of multi-step query processing. A fast *filter step* excludes all objects that cannot satisfy the join predicate. The subsequent *refinement step* is applied to the result set which are returned from the *filter step*.

Recently, a general survey on the paradigm of relational index structures has been published [4]. The basic idea of relational access methods relies on the exploitation of

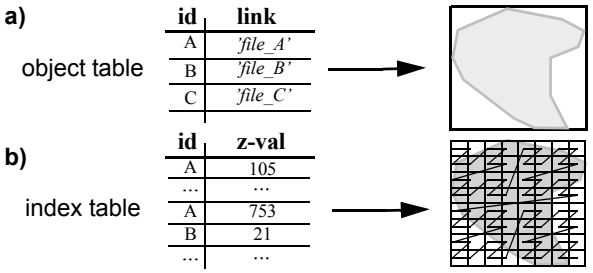

Figure 1: Relational data model.

the built-in functionality of existing database systems. A relational access method delegates the management of persistent data to an underlying relational database system by strictly implementing the index definition and manipulation on top of an SQL interface. Thereby, the SQL layer of the ORDBMS is employed as a virtual machine managing persistent data.

3. Haptic Rendering

In the VPS approach [6], the virtual environment consists of objects which can be divided into dynamic and static objects.

Static Object Model. The static environment is collectively represented by a single spatial occupancy map called a voxmap (volume map). It is created by rasterizing the static environment space. The collection of the discrete volume elements (voxels) of the object spaces builds the voxmap.

Dynamic Object Model. The dynamic object is described by a collection of points (*PointShell*), which models its surface. Each point is assigned a surface normal vector, pointing inside the object.

Force Computation. The haptic rendering algorithm includes a fast collision detection technique based on probing the voxmap with the surface point samples of the *PointShell*. By using the normal vectors of the *PointShell*, an approximate collision force can be computed in constant time for each point-voxel interpenetration. The time consumed to render a single frame depends only on the number of *PointShell* points.

4. Relational Embedding

In this section, we show how to embed the haptic rendering method into a relational schema in order to enable the integration into a commercial database management system. Let us first mention, that the *Point-Shell* representing the dynamic object is small enough, so that it easily fits into main memory. The static environment is stored in a relational table which we call *object-table*. The *object-table* contains a set of tuples (id, link) where *id* denotes a unique object identifier

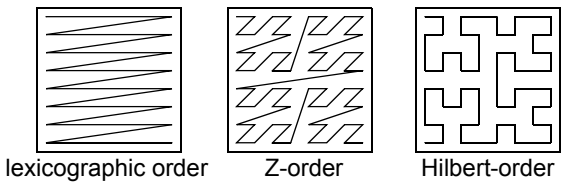

Figure 2: Examples of space-filling curves in the two-dimensional case.

and *link* refers to an external file containing the vectorized object as high order surface representation (cf. Figure 1a)

4.1 Spatial Access Method

In order to carry out point-voxel queries efficiently, we propose a simple relational access method, which consists of the *object table* and an additional *index table* storing all index data exclusively derived from the *object table* (cf. Figure 1b). Each object from the user table which is used in the haptic simulation is converted into a voxelized form and stored in the index table.

Definition 1 (*voxelized object*)
Let O be the domain of all object identifiers and let $id \in O$ be an object identifier. Furthermore, let IN^d be the domain of d-dimensional points. Then we call a pair $O_{voxel} = (id,\ \{v_1,\ ...,\ v_n\}) \in O \times 2^{IN^d}$ a d-dimensional voxelized object. We call each of the v_i an object voxel, where $i \in \{1, .., n\}$.

We employ a space-filling curve to transform the 3-dimensional object voxels into a set of linear ordered values, each represented by one integer value *z-val*. Examples for space filling curves include the lexicographic-, Z-, or Hilbert-order (cf. Figure 2). Due to a good trade-off between spatial clustering properties and computational complexity, we employ the Z-order. The index table consists of tuples (*id*, *z_val*), where the foreign key *id* denotes the identifier of the associated object and *z-val* denotes the encoded position of the voxel using the Z-order. Since the *z-val* values are 1-dimensional, the objects can be dynamically indexed by the *z-val* attribute using built-in index structures, e.g. a B^+-tree.

4.2 Simple Query Processing.

We express the haptic query (contact-force computation) on top of the SQL engine as shown in Figure 3.

```
SELECT    force(PS.point,PS.z_val)
FROM      PointShell :PS
WHERE EXISTS (
            SELECT 1
            FROM   INDEX_TABLE I
            WHERE  I.z_val = PS.z_val )
```

Figure 3: SQL statement for haptic query.

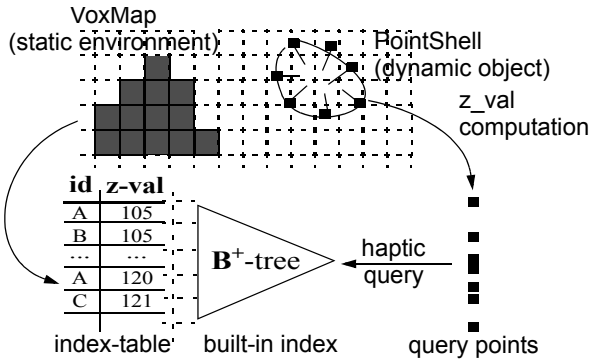

Figure 4: Relational embedding of the haptic query loop.

The input of this nested SQL query is the index-table and a collection of points derived from the PointShell (c.f. Figure 4). We assume that the 3-dimensional PointShell points are mapped to the corresponding z-val values associated with the voxel grid of our static environment. In the projection part of the SQL statement, we use the function *force*() which is a user-defined aggregate function as provided in the SQL:1999 standard. This function collects all points of the PointShell which intersect an object voxel of the index-table and returns the aggregated contact force vector. The exist-quantifier is necessary to filter out redundant results.

5. Cost Based Query Acceleration

Our approach aims at reducing the total query cost associated with the built-in B^+-tree. If we assume that our PointShell consists of n points, the simple query process as proposed in Section 4.2 leads to n point queries on the index table (cf. Figure 5a). Thus, we have to navigate n times through the built-in index (B^+-tree) directory. The general idea of our approach is to minimize the overall navigational cost of the built-in index. This is achieved by a cost-based grouping of n query points into m query ranges, where $m << n$ holds. For each query range we have to traverse the built-in index directory only once, and read the range query result by a single

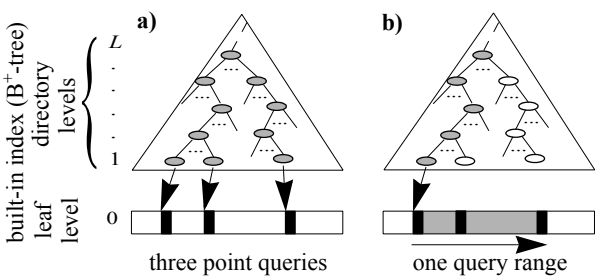

Figure 5: General Idea.
a) Simple query process b) Range query approach

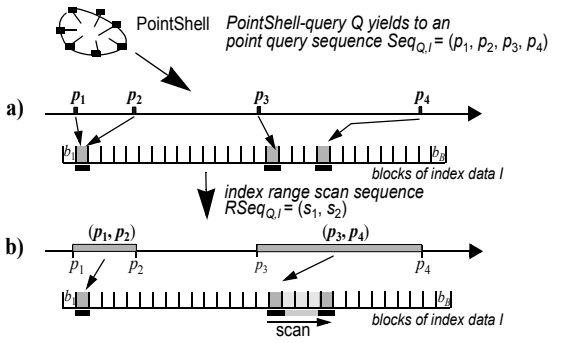

Figure 6: Accelerated query processing
a) Point query sequence **b)** Index range scan sequence

scan on the leaf level of the built-in index as shown in Figure 5b.

In the following, we assume that all points of the PointShell are transformed to the corresponding z-val values. We start with a straightforward approach of a PointShell query, the *Point Query Sequence*:

Point Query Sequence. The *PointShell* query Q leads to an ordered sequence $Seq_{Q,I} = (\langle p_1,..., p_n\rangle)$ of query points on the index table I, where $p_i < p_{i+1}$ for all $i \in \{1, .., n\text{-}1\}$ (cf. Figure 6a). For $Seq_{Q,I}$ the following assumptions hold:

- The elements r_i stored in the index are of the same type as p_i. Furthermore, we assume that the elements r_i can be regarded as a linear ordered list $L(I) = \langle r_1,...,r_N\rangle$ for which $r_1 \le ... \le r_N$ holds.
- We assume that the disk blocks b_i of the index obey a linear ordering $\le$ and fulfill the following property: $r' \le r'' \Leftrightarrow b(r') \le b(r'')$, where $b(r)$ denotes the disk block of the index I, which contains the entry r.

Range Query Sequences. Consecutive query points of the point query sequence $Seq_{Q,I} = (\langle p_1,..., p_n\rangle)$ can be grouped into subsequences $(\langle seq_1,..., seq_m\rangle)$ of the form $(\langle\langle p_1,...,p_{l(1)}\rangle, \langle p_{l(1)+1},...,p_{l(2)}\rangle, .., \langle p_{l(m\text{-}1)+1},..., p_n\rangle\rangle)$ where $l(i\text{-}1) < l(i)$ and $1 \le l(i) \le n$ for all $i \in \{1, .., m \le n\}$. Each subsequence seq_i builds the new query range s_i, which is bounded by the first and last point of seq_i. Thus, the large amount of point queries is reduced to a small sequence $RSeq_{Q,I}$ of query ranges associated with the PointShell Q and index table I. When carrying out a range query $s = (p_u, p_v)$ derived from the subsequence $\langle p_u,...,p_v\rangle$, we traverse the index directory only once and perform a range scan $(p_u\, p_v)$ on the leaf-level, as for example (p_3, p_4) in Figure 6b. Thereby we read false hits from the index table I, which have to be filtered out in a subsequent refinement step.

The cost $C(s)$ associated with a *range query* s are composed of the I/O part required to access the query

result and the CPU cost which is required for the refinement step: $C(s) = C^{I/O}(s) + C^{CPU}(s)$.

I/O cost. The I/O cost $C^{I/O}(s)$ associated with one range scan $s = (p_u, p_v)$ are composed from two parts $C^{I/O}(s) = C_n^{I/O}(s) + C_s^{I/O}(s)$, with the following properties:

(i) $C_n^{I/O}(s) = C_n^{I/O}(p_u)$ (*navigational cost*)

(ii) $C_s^{I/O}(s) = C_s^{I/O}((p_u, p_v))$ (*scan cost*)

CPU cost. The CPU cost $C^{CPU}(s)$ associated with one range scan $s = (p_u, p_v)$ denote the cost which are required to perform the refinement operation for all tuples resulting from the range scan: $C^{CPU}(s) = C^{CPU}(\langle r', .., r'' \rangle)$, where $\forall r \in L(I) : (r' \le r \le r'') \Leftrightarrow (p_u \le r \le p_v)$.

The total cost $C(RSeq_{Q,I})$ associated with a *range query sequence* is the sum of the cost of the single range queries.

The navigational cost $C_n^{I/O}(s)$ are independent of the actual query range s and can easily be estimated by C_n^{est}, e.g. by the height of the B^+-directory. In contrast, both cost $C_s^{I/O}(s)$ and $C^{CPU}(s)$ depend on the size of the query range s. In the following we present our cost based grouping which is based on an estimation method for the selectivity of query ranges.

Grouping Rules. The crucial question is how we can achieve the most profitable set of query ranges covering the query points. Consider the two extreme cases: If we group all query points into one single query range we save almost all navigational cost of the B^+-tree (the navigational cost to find the starting point is required only), but we might obtain too many false hits which increase the I/O cost associated with the range scan. On the other hand, if we group only query points with adjacent z-values together into one query range, we probably obtain too many short ranges which lead to a high overhead of the navigational cost. Consequently, a good selection of the query points which should be grouped into one range is required. A good grouping should take the following "*grouping rules*" into consideration:

Rule 1:	The number of query ranges should be small.
Rule 2:	The approximation error of all query ranges should be small.

The first rule guarantees that the navigational cost of the built-in index is kept small, whereas the second rule guarantees that many unnecessary candidate tests of the refinement step can be omitted, as the number of false hits included in the range query, i.e. the approximation error, is small. A good query response behavior results from an optimal trade-off between these two grouping rules.

For the computation of appropriate query ranges we apply the introduced cost model, extended by selectivity estimations which are based on statistical informations.

Selectivity Estimation based on Statistics. In [3], it was shown that using quantiles ('equi-count histograms') is more suitable for estimating the selectivity and the corresponding I/O cost associated with a query range scan than using histograms ('equi-width histograms'). Fortunately, most ORDBMS comprise efficient built-in functions to compute single-column statistics, particularly for cost-based query optimization. The basic idea of our quantile-based selectivity estimation is to exploit these built-in index statistics rather than to add and maintain user-defined histograms. We start with the definition of a quantile vector, the typical statistics type supported by relational database kernels.

Definition 2 (*quantile vector*).

Let $(M, \le)$ be a totally ordered multi-set. Without loss of generality, let $M = \{m_1, m_2, \ldots, m_N\}$ with $m_j \le m_{j+1}$, $1 \le j < N$. Then, $Q(M, \nu) = (q_0, \ldots, q_\nu) \in M^\nu$ is called a *quantile vector* for M and a *resolution* $\nu \in IN$, iff the following conditions hold:

(i) $q_0 = m_1$

(ii) $\forall i \in 1, \ldots, \nu: \exists j \in 1, \ldots, N: \; q_i = m_j \wedge \frac{j-1}{N} < \frac{i}{\nu} \le \frac{j}{N}$

The multi-set M of our quantile vector $(q_0, \ldots, q_\nu)$ (cf. Definition 2) is formed by the values of the first attribute of the domain values of our index I. By means of these statistics we can estimate the I/O cost $C_s^{I/O}(s)$ associated with one range scan $s = (p_u, p_v)$. In the following formula, B denotes the number of disk blocks at the leaf level of I, ρ denotes the resolution of the quantile vector and *overlap* returns the intersection length of two intersecting ranges.

$$C_s^{I/O}((p_u, p_v)) \approx \frac{\sum_{i=1}^{\rho} \left(\frac{overlap((p_u, p_v), (q_{i-1}, q_i))}{q_i - q_{i-1}} \right)}{(\rho / B)}$$

We can also apply the above formula to estimate the total cost $C_s(s) = C_s^{I/O}(s) + C^{CPU}(s)$ which are required to scan over the gap $g =]\, p_v', p_u''\, [$ between two adjacent point and range queries s' and s'' respectively. The CPU cost can be estimated by $C^{CPU}(g) = k \cdot C_s^{I/O}(g)$, with a parameter $k > 0$, since both the I/O cost and the CPU cost are directly proportional to the size of the result set of the range scan.

We will now discuss how we can use this information for our cost model in order to speed up the PointShell query.

Cost Based Grouping. For each haptic query, there exist a lot of different possibilities to group the PointShell points into a range query sequence.

Lemma 1 (*number of grouping possibilities*).

Let $(\langle p_1, \ldots, p_n \rangle)$ be a point query sequence. Furthermore, let $W = \{(p_u, p_v) \in IN^2, p_u \le p_v\}$ be the domain of ranges

```
CBGroup (PointShell PS, QuantileVector Q) {
  Integer left := compute_zvalue(PS.point[1]);
  Integer right := left;
  Integer next := 0;
  Float nav_cost := estimate_navigational_cost();
  for i := 2 to PS.size() do {
    next := compute_zvalue(PS.point[i]);
    if estimate_gap_cost(right,next,Q) > nav_cost then {
      perform_range_query (left, right);
      left := next;
    }
    right := next;
  }
  perform_range_query (left, right);}
```

Figure 7: Cost based grouping algorithm.

and let $s_1 = (p_{u(1)}, p_{v(1)})$, …, $s_n = (p_{u(n)}, p_{v(n)}) \in W$ be a sequence of range queries where $p_{v(i)} + 1 < p_{u(i+1)}$. Then, there exist $O(2^n)$ different range query sequences.

Proof. The point query sequence consists of n-1 "gaps". For each of these gaps we can decide whether it is included in a range query, or whether it separates two range queries. Thus we have 2^{n-1} different possible range query sequences. ■

Based on the cost formulas of the range query sequence, we can find a cost optimum grouping algorithm. Unfortunately, as shown in the above lemma, there exist exponentially many grouping possibilities, which results in an exponential runtime $O(2^n)$ of an optimum cost-based grouping algorithm, where n denotes the number of PointShell points. In this section, we will present an algorithm with a guaranteed worst-case runtime complexity of $O(n)$ which produces a cost optimal range query sequence helping to accelerate the query process considerably (cf. Section 6).

Our approach closes the gaps between two query points if and only if the cost related to the additional read data is smaller than the navigational cost related to an additional point query. More precisely, for each gap g of the n-1 gaps between the point queries $p_1,..., p_n$ of a point query sequence $Seq_{Q,I} = (\langle p_1,..., p_n\rangle)$ we decide whether to close this gap or to skip it. The cost-based grouping algorithm CBGroup implementing our approach is depicted in Figure 7. With CBGroup, we obtain a range query sequence $RSeq_{Q,I} = (\langle seq_1,..., seq_m\rangle) = (\langle\langle p_1,...,p_{l(1)}\rangle, \langle p_{l(1)+1},...,p_{l(2)}\rangle, .., \langle p_{l(m-1)+1},..., p_n\rangle\rangle)$ which satisfies the following property:

$$\forall j \in 1\ldots(n-1)\colon j \in l(1)\ldots l(m-1) \Leftrightarrow$$

$$C_n^{est}(p_{j+1}) < C_s((p_j + 1, p_{j+1} - 1))$$

In the following, we assume that our selectivity estimation method exactly corresponds to the correct data distribution.

Lemma 2 (*cost-optimal grouping*).
The CBGroup algorithm generates a cost optimal range query sequence.

Proof. We start with a range query sequence *RSeq* which is generated by applying our grouping method to a sequence of point queries *Seq*. Let us assume that there exists another range query sequence *RSeq'* ≠ *RSeq* generated from *Seq* which yields less query cost than *RSeq*. In order to transform the sequence *RSeq* into *RSeq'*, we have to modify the sequence *RSeq* by opening respectively closing at least one gap between adjacent points of *Seq*. For each possible gap, our cost based grouping method has determined the cost optimal solution and each modification of the initial sequence *RSeq* would increase the overall query cost, which contradicts to our assumption. ■

6. Experimental Evaluation

In this section, we evaluate the performance of our approach with a special emphasis on the haptic frame rate. The tests are based on a real-world test data set *CAR* which was provided by our industrial partner, a German car manufacturer, in form of high resolution voxelized 3-dimensional CAD parts. Table 1 shows the properties of this data set.

Table 1: Test Data Set

data set	# voxels	# objects	size of data space
CAR	$14x10^6$	200	2^{33} cells

The query processing functionality of our approach is implemented on top of the Oracle9*i* Server using PL/SQL for the computational main memory based programming. All experiments were performed on a Pentium 4/2600 machine with IDE hard drives. The database block cache was set to 500 disk blocks with a block size of 8 KB and was used exclusively by one active session.

In the following, we examine the benefits of using range query sequences instead of point query sequenc-

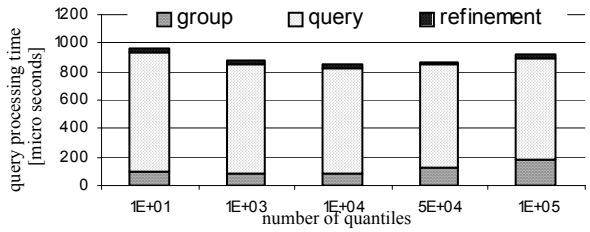

Figure 8: Avg. query processing time for different quantile resolutions

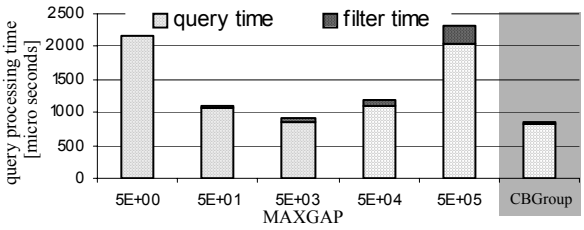

Figure 9: Performance of range query sequences.

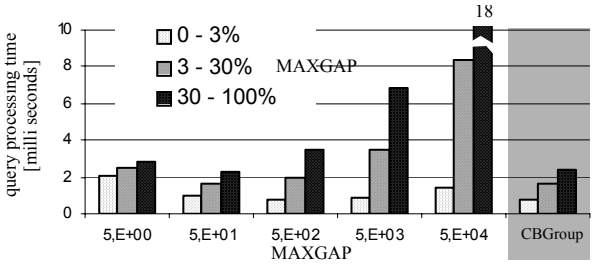

Figure 10: Query processing time for regions having different voxel density

es. We carried out several PointShell queries at different locations and logged the average response times. The query PointShell consists of about 300 points.

In a first experiment, we investigated how different resolutions of the quantile vector influence the performance of our approach (cf. Figure 8). A low quantile resolution leads to a cheap grouping, but badly estimates the query selectivity. Contrary, if we choose a high quantile resolution the query is well adjusted to the respective selectivity, at the expense of the grouping performance. In our experiments we achieved the best results using a quantile resolution of about 10000 which we use for our CBGroup algorithm throughout the next experiments.

In the following experiments, we compare our CBGroup algorithm with another grouping approach which does not use any statistical information at all. The comparison algorithm, called MaxGap approach, groups query points into a range query, in which the gap between two adjacent query points does not exceed a specified MAXGAP parameter. Note that a MAXGAP parameter of 0 corresponds to a point query sequence.

The next experiment shows that the runtime of our cost-optimum CBGroup algorithm corresponds to the best possible runtime achieved by the MaxGap approach. By varying the MAXGAP parameter, we can find the optimum trade-off between the grouping rules of Section 5. Figure 9 shows that using low MAXGAP values results in a low query performance. This is caused by a substantial navigational overhead of the index because we have to carry out many range queries. On the other hand, high MAXGAP values result in a very low filter selectivity which leads to high I/O and refinement cost. We can observe that a good trade-off between the navigational overhead and filter selectivity is achieved using a MAXGAP value of 500. Note, that the results presented in Figure 9 also show that our CBGroup approach outperforms the MaxGap approach for all possible MAXGAP parameters. This is due to the fact that the CBGroup algorithm locally adapts the grouping to the data distribution.

In the last experiment (cf. Figure 10), we performed haptic queries in regions having different voxel density. Thereby, the density denotes the ratio of object voxels to free space. In areas where the voxel density is low, a high MAXGAP value performs best. Although the query ranges are large, which results in low navigational cost, the result set stays small due to low region density. With increasing voxel density, high MAXGAP values result in many false hits, leading to high I/O and refinement cost. For all three experiments shown in Figure 10, our cost-based grouping algorithm adapts automatically to the voxel density.

7. Conclusions

In this paper, we presented an approach which externalizes and accelerates the approved main-memory Voxmap-PointShell approach. We presented a cost-optimum access method which groups different independent point queries together to larger range scans. In a broad experimental evaluation based on a real-world test data set we demonstrated that we achieve an enormous acceleration of the query process. We obtain a frame rate of about 1000 Hz for the haptic rendering loop, which is by far sufficient for many haptic rendering applications.

8. References

[1] Date C. J.: *An Introduction to Database Systems*. Addison Wesley Longman, Boston, MA, 1999.

[2] Gaede V., Günther O.: *Multidimensional Access Methods*. ACM Computing Surveys 30(2): pp. 170-231, 1998.

[3] Kriegel H.-P., Pfeifle M., Pötke M., Seidl T.: *A Cost Model for Interval Intersection Queries on RI-Trees*. SSDBM, pp. 131-141, 2002.

[4] Hans-Peter Kriegel, Martin Pfeifle, Marco Pötke, Thomas Seidl: *The Paradigm of Relational Indexing: a Survey.* BTW, pp. 285-304, 2003.

[5] Lin, Ming C. and Gottschalk, Stefan: *Collision detection between geometric models: a Survey.* Proc. IMA Conference on Mathematics of Surfaces 1998, p. 20.

[6] McNeely W. A., Puterbaugh K. D., Troy J. J.: *Six Degree of Freedom Haptic Rendering Using Voxel Sampling*. ACM SIGGRAPH, pp. 401-408, 1999.

[7] Manolopoulos Y., Theodoridis Y., Tsotras V. J.: *Advanced Database Indexing. Boston.* MA: Kluwer, 2000.