THE CELL TREE: AN INDEX FOR GEOMETRIC DATA

by

Oliver Gunther

Memorandum No. UCB/ERL M86/89

9 December 1986

THE CELL TREE: AN INDEX FOR GEOMETRIC DATA

by

Oliver Gunther

Memorandum No. UCB/ERL M86/89

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE CELL TREE: AN INDEX FOR GEOMETRIC DATA

by

Oliver Gunther

Memorandum No. UCB/ERL M86/89

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Cell Tree: An Index for Geometric Data

Oliver Gunther
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

This paper presents the preliminary design of a database index for geometric data, termed cell tree. All data objects in the database are represented as algebraic sums of convex point sets (*cells*). The cell tree indexes the set of cells by means of a binary space partitioning. It is a fully dynamic data structure, i.e. insertions and deletions may be interleaved with searches and no periodic reorganization is required.

1. Introduction

It is well known that current database management systems are not performing very well in domains that involve the processing of complex, multi-dimensional data. Examples of such domains are computer-aided design, computer graphics, image processing, computer vision, robotics, computational geometry, or geographic data processing. Hierarchical data structures provide a convenient representation scheme for this kind of data, based on the *divide and conquer* paradigm. They facilitate the solution of many typical queries such as the following. Range searches ask for all objects that intersect a given search area. In the point location problem, which can be viewed as a degenerate range search, one searches for all objects that contain a given point. Finally, there are set queries such as for the intersection of all objects that meet a given qualification. Most hierarchical data structures are dynamic, i.e. insertions and deletions can be interleaved with queries and no periodic reorganization is required.

In a database environment, hierarchical data structures are frequently used as indices. The canonical example for such an index is the B-tree [Baye72] , a structure that is based on the ordering of one-dimensional key values. More recently, several proposals for multi-dimensional database indices were made, some

This research was sponsored under research contract DAAG29-85-0223 and a scholarship from the German National Scholarship Foundation

of which will be discussed in the sequel.

Section 2 gives a brief survey of the most well-known hierarchical data structures, emphasizing the structures that are most suitable to serve as a database index. Section 3 describes a scheme for a geometric database where all data objects are represented as algebraic sums of convex point sets (*cells*). Section 4 introduces an index for this database, viz., a new hierarchical data structure termed *cell tree*, and describes how to perform search operations. Section 5 gives algorithms to perform insertions and deletions, and section 6 contains our conclusions and plans for further research.

2. Related Data Structures

Hierarchical data structures are based on the principle of recursive decomposition. They can be classified on the basis of the principle guiding the decomposition process on each recursion level. In *tree* structures, the decomposition is guided by the input data. In *trie* structures, the decomposition is independent of the input data. For example, the decomposition may be into subspaces of the same shape as the original space (termed a *regular decomposition*). Both for tree and trie structures, however, the input data determines the recursion depth, i.e. at what point the decomposition is to terminate.

Tries have the general disadvantage that they are not able to represent arbitrary objects precisely without loss of information. Region quadtrees [Same84] are actually tries that organize two-dimensional data. The decomposition process starts from a square that contains all objects to be represented, and proceeds with a recursive subdivision into four equal-sized quadrants. Objects whose boundaries do not fit into the rectilinear partition of the quadtree can only be represented approximately, or in form of an object description attached to the leafs of the quadtree.

A further disadvantage of the region quadtree is that it does not take paging of secondary memory into account. In particular, this becomes problematic for the generalization of region quadtrees to multiple dimensions. The branching factor of the tree is 2^d for d dimensions. At some point nodes will stretch over several pages which may decrease the tree performance significantly.

Finally, the region quadtree representation is very sensitive to the positioning of the objects within the grid. A slight translation or rotation of an object might change its representation in a major way.

In order to overcome some of those difficulties for the case of polygonal data, Samet and Webber proposed the PM quadtree [Same85]. PM quadtrees store polygonal maps (i.e. collections of polygons, possibly containing holes) without any loss of information. They are not overly sensitive to the positioning of the map. However, they are not generalizable to more than two dimensions. Also, they are not very useful for range searches and for set operations on the polygons.

Binary space partitioning (BSP) trees [Fuch80,Fuch83] are binary trees that represent a recursive subdivision of a given space into subspaces by hyperplanes. Each subspace is subdivided independent of its history and of the other subspaces. Each hyperplane corresponds to an interior node of the tree, and each partition corresponds to a leaf. Figure 2.1 gives an example of a BSP and the corresponding BSP tree.

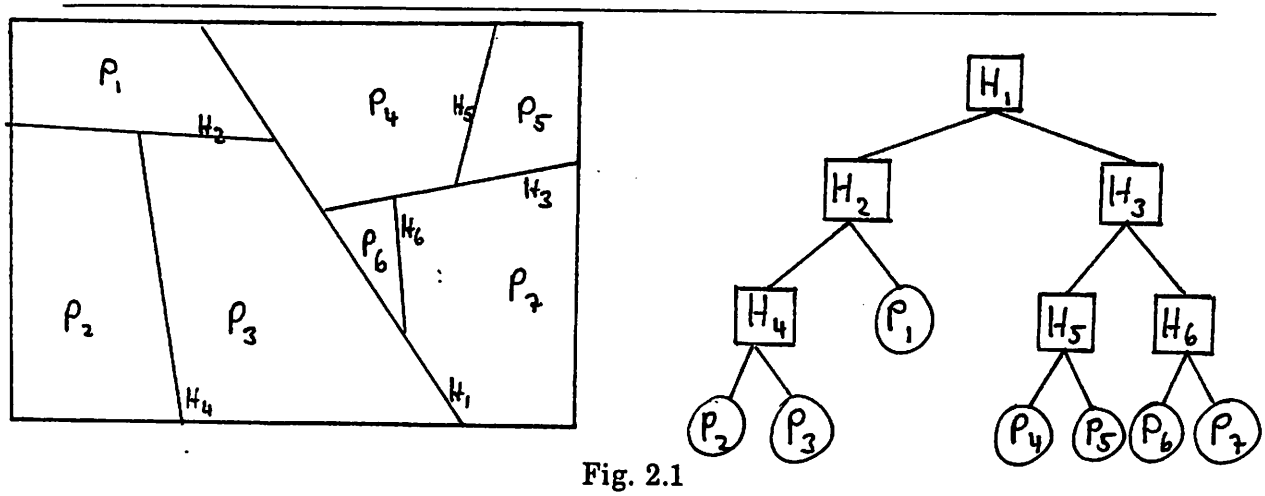

Fig. 2.1

BSP trees provide another way to represent polygonal data, but they are typically very deep which has a negative impact on tree performance. Also, insertion and deletion of objects is very hard, i.e. they are not very dynamic. Finally, they do not account for paging of secondary memory.

The point quadtree [Fink74] is a multidimensional generalization of a binary search tree. As the region quadtree, the point quadtree is sensitive to the positioning of the objects and has a branching factor of 2^d for d dimensions. In order to avoid the large branching factor, k-d trees [Bent75] and k-d-b trees [Robi81] have been developed. Both structures are binary trees. The k-d-b trees are designed for paged memory, whereas k-d trees do not take paging of secondary memory into account. All these data structures are useful only for point data. Although point deletion is fairly complicated, the data structures are fully dynamic.

R-trees, proposed by Guttman [Gutt84] , are a generalization of B-trees [Baye72,Come79] to higher dimensions. They are used to retrieve data for non-point geometric objects according to their locations in a multi-dimensional space. R-trees are designed for data residing on paged secondary memory, and for use as a database index. They are based on the nesting of multi-dimensional rectilinear boxes that, at the lowest level, are wrapped around the actual data objects. Therefore, R-trees do not provide an exact representation of non-rectilinear data objects and, consequently, do not give exact answers for this case. For example, a range search on an R-tree only yields a set of boxes whose enclosed objects *may* intersect the search space. One is left with the problem of testing the *object* for intersection with the search space and, optionally, computing the intersection. The search efficiency of R-trees is limited, because the rectilinear boxes may be too rough an estimate for the data objects enclosed. Especially for point location problems, R-trees are inappropriate because the boxes on one level may be overlapping. This means that one may have to follow several search paths for the same search point. The latter problem led to the development of optimization techniques to minimize the overlap [Rous85] and of the R^+-tree [Ston86] where the boxes on the same tree level are non-overlapping.

For a more detailed survey of hierarchical data structures see, for example, [Bent79] or [Same84].

3. A Geometric Database Scheme Based on Convex Chains

Consider a database consisting of a collection of (possibly self-intersecting) regular* d-dimensional point sets in Euclidean space $\mathbf{E}^d$. In order to support search and set operations efficiently, we represent the data objects as *convex chains*, i.e. as sums of convex point sets [Whit57,Gunt86]. Formally, each data object P is represented as a convex chain in $\mathbf{E}^d$,

$$x_P = \sum_{k=1}^{m} p_i$$

The p_i are d-dimensional convex regular point sets that are not necessarily bounded (*cells*). Note that we do not require the cells to be mutually disjoint. Disjointness is hard to maintain and provides no particular advantages for the operators we intend to support.

* A point set is regular if it is the closure of its interior [Tilo80].

We consider a point $t \in E^d$ inside P if and only if it is inside any of the cells, i.e.

$$t \in P \iff t \in p_i \text{ for some } i = 1 \ldots m$$

Two chains are *equivalent* if they represent the same point set, i.e. if

$$t \in P \iff t \in Q$$

Convex chains are a simple and powerful tool to describe various kinds of geometric objects. They may be used to describe any simple (i.e. non self-intersecting) point set in E^d (fig. 3.1), or self-intersecting polyhedra of any shape (fig. 3.2, 3.3). Unlike simple point sets, convex chains are closed under all regularized* set operators (fig. 3.1).

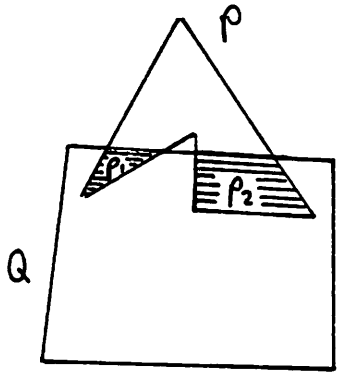

Fig. 3.1: $x_{P \cap Q} = p_1 + p_2$

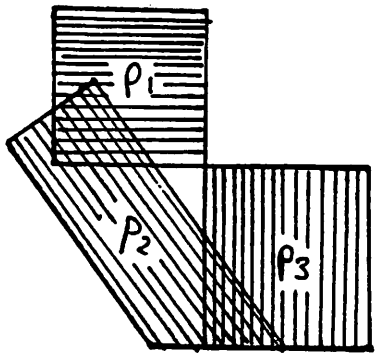

Fig. 3.2: $p_1 + p_2 + p_3$

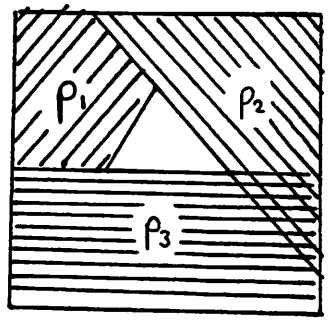

Fig. 3.3: $p_1 + p_2 + p_3$

Cells are stored in the form

$$(cid, S, D)$$

Here, *cid* is a unique identifier which can be used to retrieve the cell. S is a description of the cell's shape, and D is the set of data objects P whose convex chains x_P contain the cell.

Data objects are stored in the form

$$(did, C, A)$$

* The regularized set operators, as defined by Tilove [Tilo80], include regularized intersection, union, and difference. They differ from the corresponding simple set operators by an additional step making the result regular. This way, the dimension of the result is equal to the lowest dimension of any of the operands. In this paper all set operators that are defined on point sets are assumed to be regularized.

Here, *did* is a unique identifier, and A denotes further attributes of the data object which are of no importance in this context. C is the set of cells in the corresponding convex chain. Any cell p and any data object P are to meet the integrity constraint $p \in P.C \iff P \in p.D$ for pointer consistency.

Although the decomposition of the original data objects into cells will take some preprocessing time, we believe that it will eventually pay off by making searches and updates simpler and faster. Note that this decomposition is completely transparent to the user. Cells and the C-part of the data object representations cannot be seen or manipulated by the user. The cost of maintaining the above integrity constraint should therefore be negligible.

4. The Cell Tree

4.1. Description

A cell tree is an index structure for the set of cells in a database. As the R-tree, to which it is related, a cell tree is a height-balanced tree. A search or, in particular, a point location should therefore require visiting only a small number of nodes. Tree nodes correspond to disk pages if the index is disk-resident. The index if fully dynamic; insertions and deletions can be interleaved with searches and no periodic reorganization is required.

Each leaf node entry is a pointer to the representation of a cell. In the following, $E.C$ denotes the cell associated with a leaf node entry E. $E.D$ denotes the set of data objects whose corresponding convex chains contain the cell $E.C$. M_l denotes the maximum number of entries that fit in one leaf node, and $m_l \leq M_l/2$ is a parameter specifying the minimum number of entries in a leaf node.

Non-leaf nodes contain entries of the form

$$(cp, P, C)$$

Here, *cp* is a child pointer, i.e. the address of a lower level node in the cell tree. P is a convex, not necessarily bounded d-dimensional polyhedron. All cells in the database that overlap P are in the subtree that is rooted at this lower level node. C is a convex subset of P, such that for each cell p in the subtree, C contains $(p \bigcap P)$. C provides a more accurate localization of these cells, which may speed up search queries. In the following, $E.cp$, $E.P$, and $E.C$ denote the corresponding attributes of a non-leaf node entry E. M_{nl} denotes the maximum number of entries fitting in one non-leaf node, and $m_{nl} \leq M_{nl}/2$ is a parameter specifying the minimum number of entries in a non-leaf node. Finally, given a node N, its entry

in its parent node is denoted by E_N, and the entries in N are denoted by $E_i(N)$.

A cell tree satisfies the following properties.

(1) Every leaf node contains between m_l and M_l entries, and every non-leaf node contains between m_{nl} and M_{nl} entries unless it is the root.

(2) For each entry (cp, P, C) in a non-leaf node, the subtree that cp points to contains a cell p if and only if p overlaps the convex polyhedron P.

(3) For each entry (cp, P, C) in a non-leaf node, $C \subseteq P$ is a convex polyhedron that can be specified as the intersection of P with at most k halfspaces in $\mathbf{E}^d$. For each cell p in the subtree pointed to by cp, it is $(p \cap P) \subseteq C$.

(4) For each non-leaf node N, the polyhedra $E_i(N).P$ form a binary space partitioning (BSP) of $E_N.P$.

(5) The root node has at least two children unless it is a leaf.

(6) All leaves are on the same level.

Figures 4.1 and 4.2 show the structure of a cell tree and a corresponding arrangement of data objects, decomposed into cells.

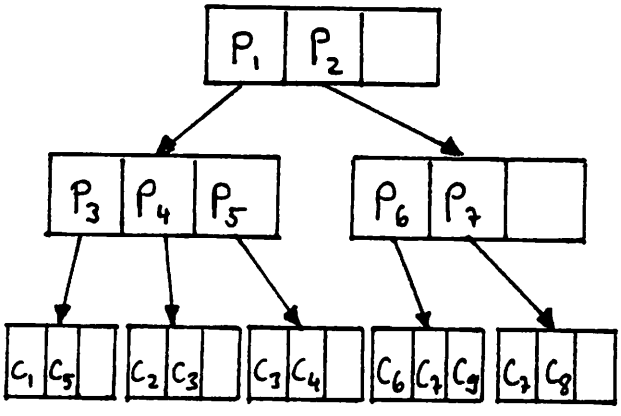

Figure 4.1

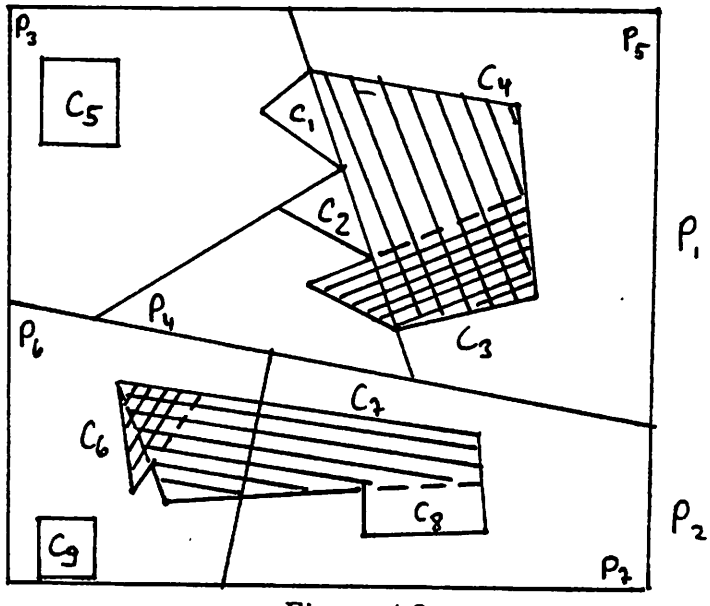

Figure 4.2

In order to analyze the space requirements of a cell tree, we denote the page size by ps, and the number of bytes required to store a number or a pointer by q. Each leaf node entry requires exactly q bytes, hence it is $M_l = \lfloor ps/q \rfloor$. Each non-leaf node entry $E_i(N)$ requires q bytes for the pointer $E_i(N).cp$, and $k \cdot d \cdot q$ bytes for the k $(d-1)$-dimensional hyperplanes that specify $E_i(N).C$ if $E_i(N).P$ is known.

The polyhedra $E_i(N).P$ form a BSP of $E_N.P$ with no more than M_{nl} partitions. Therefore, the corresponding BSP-tree requires the storage of no more than $M_{nl}-1$ hyperplanes and $2 \cdot M_{nl}-2$ pointers. The total number of bytes to store a full non-leaf node is therefore

$$M_{nl} \cdot (q + k \cdot d \cdot q) + (M_{nl}-1) \cdot d \cdot q + (2 \cdot M_{nl}-2) \cdot q$$
$$= q \cdot (M_{nl} \cdot ((k+1) \cdot d + 3) - d - 2)$$

As one node corresponds to one disk page of ps bytes, we obtain

$$M_{nl} = \left\lfloor \frac{ps/q + d + 2}{(k+1) \cdot d + 3} \right\rfloor$$

Hence, in particular it is $M_{nl} \leq M_l$.

Therefore assuming $m_{nl} \leq m_l$, the height of a cell tree containing N index records is bound by $\lceil \log_{m_{nl}} N \rceil - 1$, because the branching factor of each node is at least m_{nl}. The maximum number of nodes is $\left\lceil \frac{N}{m_l} \right\rceil + \left\lceil \frac{N}{m_l m_{nl}} \right\rceil + \left\lceil \frac{N}{m_l m_{nl}^2} \right\rceil + \ldots + 1$. Except for the root, the worst-case space utilization is m_l/M_l for leaf nodes, and m_{nl}/M_{nl} for non-leaf nodes.

If a new cell is inserted into a cell tree, it may be inserted into no more than $\lceil N/m_l \rceil$ subtrees. Thus, the subsequent insertion of Q cells into a cell tree that is initially empty will yield a cell tree with no more than $1 \cdot m_l + 2 \cdot m_l + \ldots + \lceil Q/m_l \rceil \cdot m_l \approx Q^2/2m_l$ index records. As confirmed by empirical results [Fuch83], the actual number of index records is much smaller. It is usually no more than twice the number of cells, and the largest found by Fuchs *et al.* was 2.33 times.

A new data object is inserted into the tree by inserting each of the cells in the corresponding convex chain separately. The number of cells per object is highly data-dependent. If all data objects are convex (as it is actually the case for layout data, for example), there may be only one cell per data object.

The parameters m_l, m_{nl}, and k are to be varied as part of the performance tuning. Large m_l and m_{nl} (i.e. close to $M_l/2$ or $M_{nl}/2$, respectively) will increase the space efficiency and decrease the height of the tree, which might in turn improve the search performance. On the other hand, large m_l and m_{nl} may cause updates to become very expensive, as tree condensations will occur more frequently and be more complex (see section 5.4). A large value for k allows a more accurate

localization of the cells in a subtree, which might improve the search performance. On the other hand, k and M_{nl} are inversely proportional. A large k will therefore yield a small M_{nl}. This might in turn increase the tree height and decrease the search performance.

4.2. Searching

The cell tree allows efficient searches such as to find all data objects that overlap a search space, where the search space may be of arbitrary shape. We give the algorithm for this search problem; other searches can be implemented by variations of this algorithm.

The search algorithm first decomposes the search space into not necessarily disjoint convex components. For each component the search algorithm descends the tree from the root in a manner similar to a B-tree or an R-tree. At each non-leaf node the search space is decomposed further into several disjoint convex subspaces, and a not necessarily convex remainder space. The remainder space is insignificant to the search and therefore eliminated. The convex subspaces are each passed to one of the subtreees to be decomposed recursively in the same manner. Note that this algorithm differs from the equivalent R-tree algorithm where the subspaces are allowed to overlap, thereby decreasing the search efficiency.

Algorithm **Search**(T,S). Given a cell tree with root node T, find all data objects that overlap a search space S.

S1. [Decompose S.] If S is not convex, find a set of cells S_i such that $\sum_i S_i = S$. For each S_i, **Search**(T,S_i) and stop.

S2. [Search subtree.] If T is not a leaf, check each entry $E_i(T)$ to determine whether $E_i(T).C$ overlaps S. If yes, **Search**(T',$S \bigcap E_i(T).C$) where T' denotes the node $E_i(T).cp$ points to.

S3. [Search leaf node.] If T is a leaf, check all entries $E_i(T)$ to determine whether $E_i(T).C$ overlaps S. If yes, return all data objects in $E_i(T).D$.

---

5. Updating the Cell Tree

5.1. Insertion

To insert a new data object, one inserts each cell in the corresponding convex chain separately. Inserting index records for new cells is similar to insertion into a B- or R-tree. Index records are added to the leaves. Nodes that overflow are split, and splits propagate up and down the tree. Note, however, that the cell may be inserted into several subtrees. Therefore the insertion of a cell may cause the creation of more than one new index record.

Algorithm **CellInsert**(T,p). Insert a new cell p into a cell tree with root node T.

CI1. [Insert into subtrees.] If T is not a leaf, check each entry $E_i(T)$ to determine whether $E_i(T).P$ overlaps p. If yes, expand $E_i(T).C$ to include $p \bigcap E_i(T).P$, and **CellInsert**(T',p) where T' is the node $E_i(T).cp$ points to.

CI2. [Insert into leaf node.] If T is a leaf node, install a pointer to p as a new entry in T. If T has now more than M_l entries, **SplitNode**(T) to obtain a valid cell tree.

5.2. Deletion

In order to delete a data object J from a cell tree that indexes the object, one processes each cell in the corresponding convex chain separately. For each cell J_i, J is removed from the set $J_i.D$. If $J_i.D$ is empty then, the cell is no more needed. It is removed from storage and from the cell tree.

Algorithm **Delete**(J,T). Delete the data object J from the cell tree with root node T.

D1. [Decompose J.] For each cell $J_i \in J.C$, **CellDelete**(J_i,J,T).

D2. [Condense tree.] For each leaf node N from which cells were deleted, **CondenseTree**(N).

Algorithm **CellDelete**(J_i,J,T). Delete the cell J_i of the data object J from the cell tree with root node T.

CD1.[Search subtree.] If T is not a leaf, check each entry $E_i(T)$ to determine whether $E_i(T).P$ overlaps J_i. If yes, **CellDelete**(J_i,J,T'), where T' denotes the node $E_i(T).cp$ points to.

CD2.[Update leaf node.] If T is a leaf node, for each $E_i(T)$, remove J from $E_i(T).D$. If $E_i(T).D$ is now empty, delete the cell $E_i(T).C$ from storage, delete $E_i(T)$ from T, and contract $E_T.C$, if possible.

5.3. Node Splitting

As mentioned, the polyhedra that correspond to sibling nodes are mutually non-overlapping. This increases the search efficiency, especially for point location problems, but it also makes tree updates more difficult. For that reason, the splitting of a full node is more complicated in a cell tree than in related data structures such as the R-tree.

The splitting is done in two steps. First, we search for a "good" hyperplane along which the split is to be performed, and divide the set of node entries into two subsets. Second, the split is executed and propagated across the tree.

Algorithm **SplitNode**(LN). Given an overloaded leaf node LN in a cell tree, split LN along a hyperplane, and propagate the split upward and downward if necessary.

SN1.[Initialize.] Set $N = LN$.

SN2.[Find hyperplane.] **FindHyperplane**(N). H_1, H_2 denote the two disjoint halfspaces defined by the splitting hyperplane. N_1, N_2 are subnodes of N such that N_k contains all entries $E_i(N)$ where $E_i(N).C$ overlaps H_i. ($k = 1,2$)

SN3.[Grow tree taller.] If N is the root, create a new root whose only entry is (q_N, E^d, CP). Here, CP is a convex polyhedron with at most k faces that encloses all cells in the cell tree, and q_N is a pointer to N.

SN4.[Create new entries.] Let q_1 and q_2 be pointers to the roots of N_1 and N_2, respectively. Create two new entries $E_{N_i} = (q_i, E_N.P \bigcap H_i, E_N.C \bigcap H_i)$ $(i = 1,2)$ and replace E_N by E_{N_1} and E_{N_2}.

SN5.[Propagate split downwards.] Search the subtrees rooted at N_i $(i = 1,2)$ for cells that do not overlap H_i and delete the corresponding leaf node entries.

SN6.[Propagate split upwards.] If N's parent node has now more than M_{nl} entries, set N to N's parent node, and repeat from SN2.

SN7.[Condense tree.] For each leaf node LN from which entries have been deleted, **CondenseTree**(LN).

FindHyperplane(N) is some heuristic algorithm that finds a hyperplane along which the node N is to be split. Any such hyperplane H has to meet condition (*):

$$m \leq |\{E_i(N):E_i(N).C \text{ overlaps } H_k\}| \leq M \quad (k=1,2)$$

Here, H_k denote the two disjoint halfspaces defined by H, m denotes m_l or m_{nl}, and M denotes M_l or M_{nl}, depending on N being a leaf or a non-leaf node. H should intersect a minimal number of polyhedra $E_i(N).C$, because each such intersection causes the split to propagate down the cell tree. A large number of such intersections may cause the split to become very costly.

Unfortunately, there is not always a hyperplane that fulfills condition (*). In particular, for a leaf node N whose partition $E_N.P$ has a convex subset that is covered by more than M_l cells there is obviously no such hyperplane. In this case it is necessary to subdivide cells in order to find a BSP of $E_N.P$ such that no partition overlaps more than M_l cells. To perform the subdividing may become very costly. In this case, it may well be more efficient to tolerate more than M_l entries and allow overflow pages. A more detailed analysis of this problem is subject to further research and will appear in [Gunt].

In the case of N being a leaf node, **FindHyperplane** can be approached efficiently by l plane sweeps [Prep85] across $\mathbf{E}^d$, along l different directions. The parameter l is to be varied as part of the performance tuning. A large l will cause the splitting operation to be more costly, but it may yield a better hyperplane.

In the case of N being a non-leaf node, the hyperplanes in N's BSP-tree make good candidates for the split. An appropriate heuristic would be to sort the hyperplanes by the number of leafs in the subtree rooted at the corresponding BSP-tree node. These leafs correspond to BSP partitions that will certainly not be intersected by the hyperplane. Then the hyperplanes are inspected in the order of decreasing number of leafs. In particular, the hyperplane H^* that corresponds to the root node of the BSP-tree will be inspected first. This hyperplane does not intersect any polyhedron $E_i(N).C$. It will also fulfill condition (*) with high probability, which can be shown by the following analysis.

The probability that H^* does not fulfill condition (*) is equal to the probability that the number of partitions on any side of H^* is less than m_{nl}. The total number of partitions in an overloaded node is at least $M_{nl}+1$. Hence, after H^* was first established (viz., when $E_N.P$ was split for the first time), at least $M_{nl}-1$ more

partitions were formed by further splittings of $E_N.P$. Assuming that the cells in the subtree rooted at N are distributed equally across the subspace $E_N.P$, the probability that the number of partitions on any side of H^* is less than m_{nl} is

$$0.5^{M_{nl}-1} \cdot [1+(M_{nl}-1)+(M_{nl}-1)(M_{nl}-2)+ \ldots +(M_{nl}-1)(M_{nl}-2) \ldots (M_{nl}-m_{nl}+2)]$$

It is therefore important to keep m_{nl} reasonably low.

Of course, it may be useful to also look at hyperplanes that are not part of the BSP. Also, one may use a plane sweep approach for non-leaf nodes as well.

5.4. Tree Condensation

The tree condensation eliminates underloaded nodes and reinserts their entries on the correct tree level.

Algorithm **CondenseTree**(LN). Given a leaf node LN from which entries have been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate the eliminations across the tree.

CT1.[Initialize.] Set $N=LN$. Set Q, the set of eliminated leaf node entries, to be empty.

CT2.[Shorten tree.] If N is the root and it has only one entry, make the child the new root and let N be the new root.

CT3.[Find parent entry.] If N is the root, go to CT6.

CT4.[Eliminate underloaded node.] If N has less than m_l (m_{nl}) entries, delete E_N from its parent node, add E_N to Q, and extend the polyhedra P of N's siblings to cover $E_N.P$.

CT5.[Move up one level in the tree.] Set N to its parent node and repeat from CT2.

CT6.[Reinsert orphaned leaf node entries.] Reinsert each entry in Q.

The polyhedron extension in step CT4 can be carried out very efficiently as follows. Let N_i denote the siblings of node N. The polyhedra $E_{N_i}.P$ and $E_N.P$ are the partitions of a BSP and stored as a BSP-tree. Let X_N be the BSP-tree leaf corresponding to the partition $E_N.P$. If X_N's parent node is replaced by X_N's sibling, the resulting tree represents a different BSP. This BSP is derived from the original BSP by deleting the partition $E_N.P$ and extending the partitions $E_{N_i}.P$ to cover $E_N.P$. This follows from the following lemma.

Lemma 5.1: Let B denote some BSP, and let X denote some leaf node in the BSP-tree corresponding to B. If X's parent is replaced by X's sibling, the BSP B' that corresponds to the resulting BSP-tree has the following properties:

(i) B' has one partition less than B.

(ii) Each partition in B' is a superset of some partition in B.

(iii) Each partition in B other than the one corresponding to X is a subset of some partition in B'.

Proof:

(i) The tree transformation decreases the number of leafs by one. Hence, the number of partitions in the corresponding BSP decreases by one as well.

(ii) The tree transformation decreases the number of interior nodes by one. This corresponds to the removal of one of one of the hyperplanes defining the BSP. Hence, the partitions in B' are either identical to some partition in B, or they are derived from some partition in B by removing one of the defining hyperplanes. In any case, they are a superset of some partition in B.

(iii) The tree transformation deletes the leaf corresponding to partition $E_N.P$. This, together with (i) and (ii), implies (iii). □

The reinsertion algorithm attempts to reinsert nodes at the correct tree level without modifying the subtree rooted at that node. This procedure saves existing structures and avoids multiple rebuilding of the same subtree. If the reinsertion on the same level is no more possible, the algorithm attempts to reinsert the descendants of this node on the next lower level.

6. Conclusions

We presented the preliminary design of a database index for multidimensional geometric data, termed cell tree. Compared to related data structures such as the R-tree, we believe that the cell tree is particularly efficient for non-rectilinear data objects and for the point location problem. For the near future, we are planning to work on a theoretical and practical analysis of the cell tree. In addition to a theoretical performance analysis, there is further theoretical work required to obtain better heuristics for node splitting. In order to optimize tree performance, different sets of parameters have to be tested in an experimental implementation.

References

[Baye72] Bayer, R. and E. M. McCreight, Organization and maintenance of large ordered indices, *Acta Informatica* 1, 3 (1972), pages 1-21.

[Bent75] Bentley, J. L., Multidimensional binary search trees used for associative searching, *Comm. of the ACM* 18, 9 (Sept. 1975), pages 509-517.

[Bent79] Bentley, J. L. and J. H. Friedman, Data structures for range searching, *Computing Surveys* 11, 4 (Dec. 1979), pages 397-409.

[Come79] Comer, D., The ubiquitous B-tree, *Computing Surveys* 11, 2 (1979), pages 121-138.

[Fink74] Finkel, R. A. and J. L. Bentley, Quad trees - A data structure for retrieval on composite keys, *Acta Informatica* 4 (1974), pages 1-9.

[Fuch80] Fuchs, H., Z. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, *Computer Graphics* 14, 3 (June 1980).

[Fuch83] Fuchs, H., G. D. Abram, and E. D. Grant, Near real-time shaded display of rigid objects, *Computer Graphics* 17, 3 (Summer 1983), pages 65-72.

[Gunt] Gunther, O., *Data bases and data structures for geometric data,* Ph.D. dissertation, University of California, Berkeley, Ca., in preparation.

[Gunt86] Gunther, O. and E. Wong, *A dual space representation for geometric data,* submitted for publication, 1986.

[Gutt84] Guttman, A., R-trees: A dynamic index structure for spatial searching, in *Proc. of ACM SIGMOD Conference on Management of Data*, Boston, June 1984.

[Prep85] Preparata, F. P. and M. I. Shamos, *Computational geometry,* Springer-Verlag, New York, NY, 1985.

[Robi81] Robinson, J. T., The k-d-b tree: A search structure for large multidimensional dynamic indexes, in *Proc. of ACM SIGMOD Conference on Management of Data*, April 1981.

[Rous85] Roussopoulos, N. and D. Leifker, Direct spatial search on pictorial databases using packed R-trees, in *Proc. of ACM SIGMOD Conference on*

Management of Data, Austin, TX, June 1985.

[Same84] Samet, H., The quadtree and related hierarchical data structures, *Computing Surveys* **16**, 2 (June 1984), pages 187-260.

[Same85] Samet, H. and R. E. Webber, Storing a collection of polygons using quadtrees, *ACM Trans. on Graphics* 4, 3 (July 1985), pages 182-222.

[Ston86] Stonebraker, M., T. Sellis, and E. Hanson, An analysis of rule indexing implementations in data base systems, in *Proc. of the 1st International Conference on Expert Data Base Systems*, April 1986.

[Tilo80] Tilove, R. B., Set membership classification: A unified approach to geometric intersection problems, *IEEE Trans. on Computers* **C-29**, 10 (Oct. 1980), pages 874-883.

[Whit57] Whitney, H., *Geometric integration theory*, Princeton, NJ, 1957.