Introduction to bounding volume hierarchies
(draft)

Herman J. Haverkort

18 May 2004

Abstract

This paper is an excerpt from Chapter 1 of my PhD thesis [Hav04], written to introduce my publications about bounding volume hierachies [Aga02, Arg04, Hav04BG].

1 Computational geometry and geometric data structures

Computational geometry is the area of algorithms research that deals with computations on geometric objects. Examples of such objects are points, lines and polygons in the plane—which may represent a city plan—or balls, blocks and more complex shapes in three dimensions—which may represent the interior of a power plant. In these cases, the geometric objects represent physical objects in the real world. But this is not always the case. For example, a database storing the age and salary of a company's employees can also be thought of as a database that stores points in a two-dimensional space: each point represents an employee, with one coordinate indicating the age and the other coordinate indicating the salary of the employee. Therefore, geometric computations are found in many applications of computers: databases, computer-aided design, geographical information systems, flight simulators, other virtual reality applications, robotics, computer vision and route planning are just a few examples.

To do efficient computations on geometric objects, it is crucial that we can store, search, and sometimes update, sets of geometric objects efficiently. When the objects are one-dimensional points, this is relatively easy: we can sort them by their coordinate in that dimension, and put them in memory in that order. This makes it possible to find points fast. It is like looking up words in a dictionary: thanks to the ordering, we can find a word without turning all pages one by one.

When an object cannot be described by a single point in a one-dimensional space, it is less clear how to store a set of objects effectively. For example, I have a collection of music CD's. I would like to order them by the year in which the music was written, so that I can find all music from a particular era fast. My CD's usually contain several works of music written in a range of years. This makes it impossible to characterise a CD by a single point on a time line: a CD is rather like a group of points, or like a segment of the time line. How should I order my CD's? By the oldest work on the CD, maybe? But then even the most recent work might be put in the very first place on the shelf, if it just happens to be on the same CD as the oldest work. If I sort like this, how can I be sure of finding a work from a certain era without checking all earlier CD's too?

When geometric objects have more than one dimension, the problem becomes even more difficult. But in many applications, a lot of questions about a set of geometric objects need to be answered fast. For example, a flight simulator should not need to scan the complete hard disk to determine if the plane is going to hit a mountain in the next second. In such applications, it is essential that geometric objects are stored in such a way that relevant objects can be identified quickly, while irrelevant objects are ignored without checking them one by one. Often we can do this by sorting the objects into groups. If we do this in a clever way, we can, hopefully, discriminate quickly between groups with potentially relevant objects and groups without such objects. Therefore, finding useful groupings of objects is a key issue in many problems in computational geometry.

A set of geometric objects that is sorted, partitioned into groups and/or otherwise preprocessed, so that certain queries about the set can be answered efficiently, is called a *geometric data structure*. The goal of research into such data structures is to make them as efficient as possible with respect to storage space, the time needed to build the data structure, the time needed to insert or delete objects, and the time needed to answer queries. Examples of such queries are: which objects lie (partly) inside a given viewing window? Or: which object is closest to a given query point? Of course, we would like our data structure to facilitate fast answers, not for just one particular query point or window, but for *any* query that might be asked. One cannot usually expect to optimize for all of the objectives mentioned at the same time. In general, the faster the queries, the larger the demands on storage, preprocessing and update time.

We do not usually measure the running times of data structure algorithms by counting milliseconds. We could, of course, but with computers getting faster all the time, this would make our results outdated even before they are published. Rather we ask the question: how well will a data structure be able to take advantage of bigger and faster computers? To answer that question, we analyse in what way the number of basic operations

performed by the central processor depends on the input size (the number of objects stored) and the output size (the number of objects retrieved). The first is usually denoted by n, the second by k. We will write that algorithms have a running time of, for example, $O(n)$ or $O(n^3)$. In the first case, the running time is a linear function of n. This means that if we can double the speed of our hardware, this algorithm can process twice as much data in the same time. In the second case, the running time is a cubic function of n, which means that our double-speed computer will enable us to handle only 26% more data with this algorithm. This means that even if the second algorithm would be a little faster in practice on the current hardware, the first algorithm is probably more promising in the future.

If the amount of data is so big that we cannot keep all of it in main memory while working on it, we count the number of disk accesses rather than the number of operations. In that case we analyse how the running times depend on three parameters: the input size n, the output size k, and the amount of data transferred in one disk access.

In the most basic form, geometric data structures store points and we want to be able to retrieve, for any query range, the points that are inside, or sometimes the points that are closest to that query range. This type of data structures has been well studied and structures have been developed for simplex range queries, axis-parallel (hyper-)rectangular range queries, circular or (hyper-)spherical range queries, and point queries. With $O(n)$ space, one can build a data structure for n points in d dimensions that reports all points inside a simplex in time $O(n^{1-1/d} + k)$ [Mat93], where k is the number of points reported. For queries with axis-parallel rectangles, one can use the same data structure, or the much simpler *kd-tree* with the same query time (see e.g. [Brg97KOS] for a description). With more space, one can often get faster queries. For example, a *layered range tree* answers axis-parallel rectangle queries in time $O(\log^{d-1} n+k)$, using $O(n \log^{d-1} n)$ space [Brg97KOS]. There are also other data structures whose query times depend more heavily on the output size and less on the input size. For more about data structures for points, see, for example, the survey by Agarwal and Erickson [Aga98E].

2 Data structures for object data: bounding-volume hierarchies

Designing efficient data structures becomes significantly more difficult if the objects stored are not points, but objects that have some shape and size, such as line segments, balls or polyhedra. Theoretically efficient solutions for such problems are often too complicated and bear too much overhead to be useful in practice. It becomes even more difficult if we want a data structure that supports multiple types of queries at the same time. One can cheat, of course, by just taking a few data structures together and store each object multiple times: once in each structure. But this increases the storage requirements and also puts on us the burden of maintaining several structures.

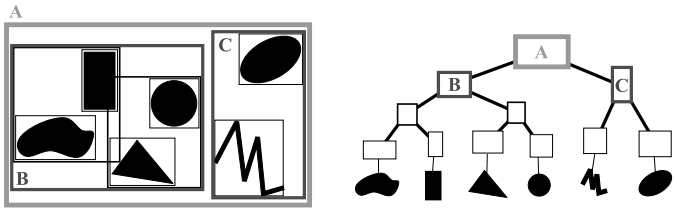

Figure 1: Example of a bounding-volume hierarchy, using rectangles as bounding volumes.

In practice, so-called *bounding-volume hierarchies* often provide a good solution. They are easy to implement, and although a bounding-volume hierarchy for n objects does not store more than $2n$ pointers and geometric objects, it can be used for different types of queries. A query in a bounding-volume hierarchy does not go directly for the answer to the query; rather it generates a set of candidate answers, which then need to be checked one by one. In practice, the set of candidate answers is usually small enough to make this approach efficient. For a bounding-volume hierarchy to be useful, it should allow fast generation of candidate answers, and it should select the candidates such that they are likely to be true answers.

Below, I will first explain what a bounding-volume hierarchy is and how it is used. After that, I will explain what issues have to be addressed when designing a bounding-volume hierarchy. I will then focus on a particular class of bounding-volume hierarchies, namely R-trees, and give an overview of our results on R-trees. To conclude, I will suggest a few subjects for further research in this area.

2.1 Definition and usage

A bounding-volume hierarchy is a tree structure on a set of geometric objects (the data objects). Each object is stored in a leaf of the tree. Each internal node stores for each of its children ν an additional geometric object $V(\nu)$, that encloses all data objects that are stored in descendants of ν. In other words, $V(\nu)$ is a bounding volume for the descendants of ν. For an example, see Figure 1.

Bounding-volume hierarchies can be used to do many types of queries on the set of data objects. For example, the algorithm in Figure 2 finds all objects that intersect a query range Q and are stored in descendants of node ν. To find all data input objects that intersect Q, start the algorithm with the root of the hierarchy as ν. The query will then descend into the tree, visiting exactly those nodes whose bounding volumes intersect Q. The bounding-volume hierarchy can also be used for other types of queries, such as nearest-neighbour queries (see Figure 3).

The algorithms can easily be adapted to hierarchies with leaves that store multiple data objects.

Algorithm *Intersected* (Q,ν)
1. for every child μ of ν
2. **if** $V(\mu)$ intersects Q **then**
3. **if** μ is a leaf **then** { *object* M *stored in* μ *is a candidate answer*}
4. **if** M intersects Q **then**
5. report M
6. **else**
7. *Intersected* (Q,μ).

Figure 2: Finding all objects that intersect Q. To find all objects that lie *completely* inside Q, replace the intersection test in line 4 by a test if M lies inside Q. To find all objects that completely *contain* Q, replace the tests in line 2 and 4 by a test if Q is completely contained in $V(\mu)$, or in M, respectively.

Algorithm *Closest* (Q,ν)
1. *smallestDistanceFoundSoFar* $\leftarrow \infty$
2. $P \leftarrow$ an empty priority queue
3. repeat
4. **if** ν is a leaf **then** { *the object* N *stored in* ν *is a candidate answer* }
5. **if** distance between N and $Q <$ *smallestDistanceFoundSoFar* **then**
6 . *smallestDistanceFoundSoFar* $\leftarrow$ distance between N and Q
7 . *closestObject* $\leftarrow N$
8 . **else**
9 . **for** every child μ of ν
11. insert μ in P with priority (distance between $V(\mu)$ and Q)
12. $\nu \leftarrow$ the node with lowest priority in P; let p be its priority
13. remove ν from P
14. until $p >$ *smallestDistanceFoundSoFar* or P is empty
15. return *closestObject*.

Figure 3: Finding the object closest to Q.

2.2 Designing bounding-volume hierarchies

When designing a bounding-volume hierarchy, we have to decide what kind of bounding volumes to use, what the structure of the hierarchy should look like, and how to order the data objects in the tree.

The shape of the bounding volumes. The choice of bounding volume is determined by a trade-off between two objectives. On one hand, we would like to use bounding volumes that have a very simple shape. Thus, we need only few bytes to store them and intersection tests and distance computations are simple and fast. On the other hand, we would like to have bounding volumes that fit the corresponding data objects very tightly. Thus, we try to avoid going into subtrees that will not lead to any object that satisfies our query. On one extreme, we could use the full space as the bounding volume for everything. On the other extreme, we would use the union of the data objects as their bounding volume. Both extremes are pointless. In the first case we would traverse the complete tree for every query; in the second case, intersection tests would be just as complex as doing a complete query.

In practice, the most commonly used bounding volume is an axis-parallel (hyper-)rectangle—we will just call them boxes. The minimum (best-fit) bounding box for a given set of data objects is easy to compute, needs only few bytes of storage, and robust intersection tests are easy to implement and extremely fast. Experiments have been done with a number of other shapes though. Among them are the set-theoretic difference of two boxes [Ary00], oriented—that is: non-axis-aligned—bounding boxes [Bar96, Got96], spheres [Oos90] (with little success), the intersection of a box and a sphere [Kat97], the Minkowski sum of a box and a sphere [Lar00], a circular section of a spherical shell [Krs98], pie slices [Bar96], and discretely oriented polytopes (k-DOP's) [Jag90, Klo98], for example octagons [Sit99] or bounded aspect ratio k-DOP's [Dun99].

Circles and spheres seem to leave too little freedom to adjust the shape to fit the objects inside. But some of the more complex shapes might actually work well. It is difficult to get a clear picture from comparitive studies on this issue. Some authors who compared axis-aligned bounding boxes with discretely-oriented octagons (in two dimensions) or oriented bounding boxes (in three dimensions) reported that in the end, axis-aligned bounding boxes often seem to work better, despite the bad fit to the data; see Van den Bergen [Bgn99] and Sitzmann and Stuckey [Sit99]. Sitzmann, however, also reported positive results for octagon hierarchies on data consisting of randomly oriented line segments. The right type of bounding volume

might, in fact, depend on the input: some of the non-standard bounding volumes are specifically aimed at fitting the triangles used to approximate smooth surfaces in virtual reality applications. Finding the right type of bounding volume definitively remains as a subject for futher study.

In our research, we decided to try to establish the best performance that can be achieved with axis-aligned bounding-box hierarchies, both from a theoretical and from a practical point of view.

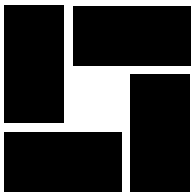

Figure 4: A set of rectangles for which an overlap-free hierarchy of degree two is impossible.

The structure of the hierarchy. Since a bounding-volume hierarchy is a tree structure by definition, the main choice left is to decide on the degree of the nodes, that is: the number of children and/or input objects stored in a node. The optimal degree depends on the way in which the bounding-volume hierarchy is used. The cost of processing a node in the hierarchy is composed of the costs of accessing the location of the node in memory, the cost of reading the node's children pointers and their bounding volumes, and the cost of the intersection or distance computations on those bounding volumes. If the hierarchy is stored on disk, the access cost tends to be high: the disk's head must be moved to the correct physical location. Once the disk head is at the correct position, a complete block of data is read into main memory at once. Computations on those data are relatively cheap, since these are done in main memory. Therefore, high-degree nodes that fill a full block of data are preferred. On the other hand, if the hierarchy is stored in main memory, our main concern is to keep the number of intersection or distance computations down. For queries that do not yield too many answers, this is best achieved by making many low-degree nodes. For example, two nodes that are irrelevant to our query can often be skipped faster if we construct a parent node that gets these two nodes as its children. A single distance computation on the parent's bounding volume may then reveal that we can skip its two children without doing distance computations on them. Of course, this potential advantage of having many low-degree nodes only materializes if usually, the parent node will indeed be skipped if both of its children are, and usually, we do not need to go into both children after all. Whether or not this is actually the case depends on the data, the queries, and the way in which the data objects are distributed in the tree.

Another issue with regard to the structure of the hierarchy is its height. If we want to be able to go from the root to any data object fast, small height is a necessary condition, but not a sufficient one. The main problem is that the bounding volumes of a node's children may intersect. If the object lies inside their intersection, there is no way to tell which child has the object as a descendant. However, small height may still be useful to guarantee that update algorithms can run fast. Most algorithms to insert or delete an object run in time $O(h)$, where h is the height of the tree. Small height is most easily guaranteed by requiring that all leaves are at the same depth. This a sufficient, but not a necessary condition, to guarantee that the tree has height $O(\log_t n)$, where n is the number of objects stored, and t is the minimum degree of the nodes.

The distribution of the objects in the hierarchy. Finally, the way in which the objects are distributed in the hierarchy may have a huge impact on its performance. One of the major issues is that overlap between bounding volumes in the same node can make search paths branch and spread out into large parts of the hierarchy. Therefore, it is important to keep the amount of overlap small. Unfortunately, overlap cannot be avoided completely. Points can always be distributed among the different parts of the hierarchy in such a way that the bounding boxes in a node do not overlap, but with other objects this is not always possible. Figure 4 shows a set of rectangles that does not admit of an overlap-free hierarchy of bounding rectangles (or other convex bounding volumes) with nodes of degree two. The only way to avoid overlap is to cut data objects into smaller parts (clipping), but this comes at a cost: it would take more storage space, and while collecting the answers to a query, time may be wasted retrieving pointers to objects which we had found already through another part.

Moreover, minimizing the amount of overlap does not necessarily lead to optimal query efficiency, as is illustrated by the following example. In Figure 5, we divided the line segments into groups of four: each group corresponds to a node just above leaf level in a hierarchy with nodes of degree four. In the top figure, we did the grouping so that we minimize the overlap between the bounding boxes of the nodes. A query with the grey square will visit 8 nodes on this level. In the lower figure, the line segments are grouped in another way. A query with the grey square will now visit only 4 nodes on this level.

If minimizing overlap is not enough to guarantee optimal queries, then how should we distribute or group the objects in the hierarchy? It is this issue that was the main subject of my PhD research.

2.3 R-Trees

We restricted our study to hierarchies that use axis-parallel boxes as bounding volumes. Extending the study to other types of bounding volumes is an obvious subject for further research but it lies beyond the scope of our work. Bounding-volume hierarchies that use axis-parallel boxes as bounding volumes and have nodes of high degree are also known as R-trees. The R-tree was originally introduced by Guttman [Gut84]. His study

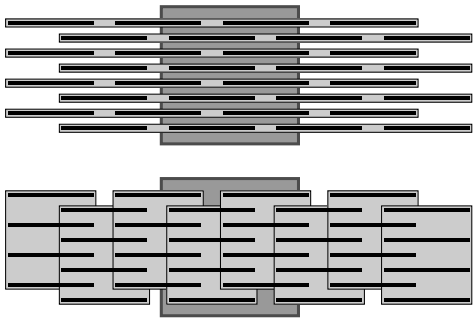

Figure 5: Minimizing overlap does not always lead to optimal query efficiency.

has inspired two decades of research about how to distribute the data objects in an R-tree, some authors designing new distribution algorithms from scratch, others suggesting optimization heuristics to be used in conjunction with known methods.

R-trees are parametrized by the maximum degree of the nodes, denoted by t in this paper. This parameter is set to match the characteristics of the hardware used: usually the tree is stored on disk, and t is chosen such that a node fills a full block on the disk. For in-memory applications, smaller values of t would be used. The minimum degree of the nodes is set to a fixed fraction of t; in the R-tree variants studied it ranged from 10% to 50% of t. R-trees usually store data objects in the leaves only and have all leaves on the same level in the tree, although some authors have designed variants where this is not the case (e.g. [Agg97, Kan97, Ros01]).

Essentially three types of algorithms have been designed to distribute the objects in an R-tree:

by repeated insertion: One defines an insertion algorithm that strives to optimize the tree locally; a complete tree is built by inserting the data objects one by one, e.g. [Ang97, Bkr92, Bmn90, Grc98b, Ros01]. Usually, deletion algorithms are provided as well.

by recursive partitioning: One defines an algorithm to distribute any number of data objects among up to t subtrees; the tree is built by applying the partitioning algorithm recursively top-down, e.g. [Agg97, Grc98a, Whi96]. The resulting data structure can be maintained either by using insertion and deletion heuristics as above (and, for example, rebalancing the complete tree during quiet hours), or by using the logarithmic method [Aga01APV].

by linear ordering: One defines a function that maps each data object to a one-dimensional value; the tree can then be built and maintained as a standard B-tree that uses the function values as keys [Brg00, Bhm99, Kam93, Kam94].

For an extensive survey on R-trees, see Manolopoulos et al. [Man03].

When comparing the query efficiency of R-trees built by such algorithms, one should distinguish between a static environment (the tree is built once and not changed afterwards), and a dynamic environment (the tree is continuously updated). In a dynamic environment it may be very difficult to maintain an "ideal" distribution of objects over the tree. The insertion of an object can, in principle, change the ideal distribution a lot. To allow for reasonably efficient update operations, one has to relax the ideal a bit. As a result, static trees, built with a partitioning or a linear-ordering algorithm, usually allow for more efficient queries than their dynamic counterparts or insertion-based algorithms.

Despite the huge body of research on R-trees, until recently, very little was known about the query times that can be guaranteed for worst-case data and queries. From Kanth and Singh [Kan98] and De Berg et al. [Brg00] some lower bounds for intersection queries with axis-parallel rectangles were known: query times better than $\Omega((n/t)^{1-1/d} + k/t)$ can, in general, not be guaranteed. Here n is the number of data objects, t is the degree of the nodes, k is the number of object bounding boxes intersected, and d is the number of dimensions. There were no algorithms to construct R-trees that can guarantee to do any query faster than a full traversal of the complete hierarchy, even if there are no answers to be reported. The only results in that direction were by De Berg et al. [Brg00], but they could guarantee fast queries only for relatively small query ranges. Other research on R-trees was mainly experimental, or of a statistical nature, making statements about expected query times under certain assumptions on the distribution of the data and/or the queries. To our knowledge, our algorithms [Aga02, Arg04] are the first algorithms that construct R-trees that guarantee worst-case query times better than $\Omega(n)$ for all axis-parallel rectangle range queries.

Note that in the bound mentioned above, as well as in all results mentioned below, k is not the number of objects intersected, but the number of data object *bounding boxes* intersected. If k would be the number of data objects intersected, it would be very difficult to prove anything about the efficiency of R-trees. Even if the objects are disjoint, their bounding boxes may in the worst case intersect in a single point, leading to a query time of $\Omega(n/t)$. For an example, see Figure 6. In three dimensions, there are sets of line segments such that *any* hierarchy of *convex* bounding volumes on such a set, needs $\Omega(n/t)$ time to answer a query with an axis-parallel line in the worst case [Bar96]. However, even if the objects intersected cannot be identified efficiently in the worst case, this is no reason to give up on at least identifying the object *bounding boxes* intersected efficiently. From now on, we will assume that the data objects stored in our hierarchies are in fact bounding boxes, and k will be the number of such bounding boxes intersected by the query range.

2.4 Our results

Given the fact that we use axis-parallel boxes as bounding volumes and given the maximum degree of the nodes, we set out to optimize the structure of the tree for fast intersection queries.

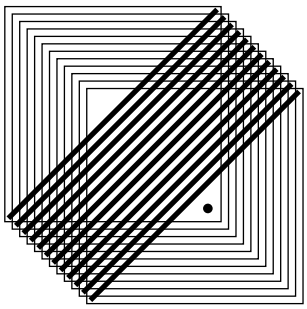

Figure 6: All bounding boxes of these line segments overlap in a single point. A query with that point needs to examine the complete hierarchy.

We chose to optimize for intersection queries since such queries are an important application to start with, and they are indicative of the efficiency of some other types of queries as well. For example, queries for objects intersecting a rectangle and queries for objects contained in a rectangle visit exactly the same nodes, and nearest-neighbour queries with a point visit exactly those nodes which would be visited by a intersection query with a circle centered on that point and just touching the nearest neighbour. Therefore, a good performance on intersection queries is crucial and can be expected to be a good indication of the performance of several other types of queries. To avoid redundancy in the data structure, we excluded clipping variants from our studies.

Our research has led to three articles: *"Box-trees and R-trees with near-optimal query time"* [Aga02], *"The Priority R-Tree: a practically efficient and worst-case-optimal R-tree"* [Arg04], and: *"Box-trees for collision checking in industrial installations"* [Hav04BG]. All of them are, in the first place, about static R-trees, that is: R-trees that are not updated anymore, once built (although in [Arg04] we also discuss updates with the logarithmic method).

In the first paper [Aga02] we prove that there are sets of rectangles, such that in *any* R-tree on such a set, there are queries that yield no answers but nevertheless visit $\Omega((n/t)^{1-1/d})$ nodes. It is not so much this bound itself which is interesting: it was already known from Kanth and Singh [Kan98] and De Berg et al. [Brg00]. What is interesting is the type of data that can bring out this worst-case behaviour. We show that such worst-case sets of rectangles and queries exist even if any one or two of the following restrictions apply:

- no point is contained in the bounding boxes of more than a constant number of data rectangles (in other words: they don't overlap much);
- the aspect ratio of the query rectangles is bounded by a constant (in other words: the query rectangles are not extremely long and thin);
- we have only two dimensions.

Only if all three of these restrictions apply, we cannot do our lower bound construction. In fact, for that case we show how to construct R-trees that can answer any axis-parallel rectangle query by visiting $O(\log^2 n + k)$ nodes.

Note that all our lower bounds, like the previous bounds by Kanth and Singh [Kan98] and De Berg *et al.*[Brg00], do not hold for replicating data structures, that is, data structures that may store each object (or a pointer to it) more than once.

In the same paper [Aga02] we also give an algorithm for the construction of axis-aligned-bounding-box hierarchies with nodes of degree two that achieves optimal query time $\Theta(n^{1-1/d} + k)$ in the general case. In a follow-up paper, *"The Priority R-Tree"* [Arg04], we extend this method to get optimal $\Theta((n/t)^{1-1/d} + k/t)$ query time on nodes of degree t (assuming that I/O-operations dominate). That paper describes the method for nodes of degree t in detail; it is not necessary to read the other paper first to understand it. In the follow-up paper we also present experimental results in two dimensions. The results indicate that our algorithm creates R-trees that are efficient in practice, while being more robust than the heuristic approaches known so far.

One may wonder if it is possible to construct R-trees that combine the good properties of both constructions mentioned above: get $O((n/t)^{1-1/d} + k/t)$ query in the general case, and $O(\log^2 n + k)$ query time if the three restrictions mentioned above apply. In the first paper [Aga02] we describe a construction, called kd-interval tree, that goes a long way towards achieving this goal. A kd-interval tree in two dimensions answers axis-parallel range queries in time $O(\sqrt{\frac{n}{t}} + k)$ and point queries in time $O(\log^2 + k)$, provided that the data rectangles don't overlap much. As overlap among the data rectangles increases, the point query performance degenerates gradually into $O(\sqrt{\frac{n}{t}} + k)$. One could use similar techniques as in our PR-trees [Arg04] to get a better dependency on the degree t in the k-term.

Our lower bound constructions [Aga02] show that it is not possible to achieve something similar in more than two dimensions: there are sets of disjoint data boxes that make any R-tree that guarantees polylogarithmic query times for point queries, spend near-linear time on some (hyper-)cube queries.

In another follow-up paper, *"Box-trees for collision checking in industrial installations"* [Hav04BG], we look into the three-dimensional situation further. The data boxes in the lower-bound construction mentioned above do not look extremely strange: they can be arbitrarily close to a unit cube in shape and size. There is one peculiar thing about them though: they must be arranged in such a way that certain cubic query ranges yield no answers while there are a *lot* of data boxes nearby. It turns out that if we accept that such cases are difficult (but probably rare in practice), and if we accept that certain arrangements of extremely flat data boxes are difficult (but probably rare in practice), we can build a three-dimensional kd-interval-tree with polylogarithmic query time for the remaining cases (the cases we expect to find in practice). We prove that these

query times are achieved not only for queries with boxes but also for queries with other query ranges of constant complexity. In *Box-trees for collision checking in industrial installations"* [Hav04BG] we describe how to build a tree with nodes of low degree; one may use the transformation algorithms described in our first paper [Aga02] to transform the tree into a real R-tree with high-degree nodes.

To distinguish between arrangements of boxes that should be handled efficiently, and arrangements of boxes that may be considered difficult, we define the *slicing number* of a set of data objects as follows: let the slicing number λ_C with respect to a cube C be the maximum number of data object bounding boxes that intersect four parallel edges of C; then the overall slicing number λ is the maximum value of λ_C over all possible cubes C. A low slicing number means that the data boxes do not overlap much and that there are no arrangements of lots of extremely flat data boxes very close to each other.

The main results for point and axis-aligned rectangle queries can be summarized in Figure 7, where we use the following notation:

n: the total number of data objects in the hierarchy;

k: the number of data object bounding boxes that intersect the query range;

k_ε: (with $\varepsilon > 0$) the number of data object bounding boxes that intersect the query range, or lie within a distance of ε times the diameter of the query range;

t: the maximum degree of the nodes in the hierarchy;

α: the maximum aspect ratio (width/height or height/width) of the query range;

2.5 Subjects for further research

The 3-dimensional kd-interval-tree mentioned above has good theoretical bounds for low-degree nodes, but when turned into an R-tree (using the technique explained in [Aga02]), the dependency on the degree of the nodes is not as good as one would wish. We cannot yet say if data sets of realistic size and structure will nevertheless bring out the strength of the kd-interval-tree, and if so, for what types of data and queries this method would indeed be the method of choice.

In [Arg04] we compare our PR-tree to two variants of the Hilbert-R-tree, which is an R-tree based on ordering objects along the Hilbert space filling curve [Kam94]. Although the Hilbert-R-tree cannot guarantee worst-case query times, and does not outperform the PR-tree, it still has advantages: it is built faster and it is much easier to implement and maintain. We tested two variants of the Hilbert-R-tree in two dimensions: one in which each data object is represented by its center point, and one in which each data object is represented by a *four*-dimensional point whose coordinates are those of the object's bounding rectangle. Naturally, the second variant is more robust when the data consists of rectangles. However, the experiments also show that the second variant is *weaker* on some sets of *point* objects. It makes one wonder if this unwanted behaviour cannot be avoided. Can we design a space-filling curve, to be used as the basis for an R-tree, which is good for both point and rectangle data?

The next big question that remains is: what is the best type of bounding volume? It might depend on the type of queries we want to perform. Are axis-aligned bounding boxes the best choice for axis-aligned rectangle queries? What would be the best bet for general range queries in two dimensions? Do the results on octagons by Sitzmann and Stuckey [Sit99] suggest that octagons, sometimes helpful, sometimes harmful, are just a little bit too much? Would the optimum be found at discretely oriented hexagons? And how would that be in three dimensions? Dodecahedra?

Our research has been primarily aimed at two- and three-dimensional settings. Our theoretical results *are* valid for multi-dimensional data as well. Unfortunately, this includes the rather disappointing lower bounds. From this we must conclude that the theoretical approach taken in this thesis, aiming for optimal worst-case query times, may not give us a data structure that is practical for high-dimensional data. In practice, one would like to have a data structure that does not only guarantee optimal query times on the worst possible data, but can also take advantage of easier data to allow for faster queries. Since in many practical situations, we do not have worst-case data, this would lead to a data structure that is much faster in practice. We do not know if our data structures take advantage of easy data or fail to do so. For two-dimensional data, it worked out well—in our experiments, the PR-tree does appear to be efficient—but this success does not necessarily carry over to higher dimensions. Handling high-dimensional data may require more study into questions of the type: what *is* easy data, and how can we design a data structure that simultaneously guarantees worst-case query times and takes advantage of easy data? We have made an attempt is an attempt to deal with the first question in three dimensions [Hav04BG], but it is doubtful if it makes sense to generalize that approach to higher dimensions. The right questions to ask may depend on the number of dimensions. Typical applications for low-dimensional data include motion planning. There we have objects that may have a shape in, for example, four dimensions (three spatial dimensions and one time dimension). But high-dimensional data more often comes from applications where the data objects have no shape and size, but are just points whose coordinates represent the values of non-geometric properties of the objects.

References

[Aga98E] P. K. Agarwal and J. Erickson: Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack (eds.), *Advances in Discrete and Com-*

Results in two dimensions: asymptotic upper bounds $O(...)$			
input rectangles	*tree [paper]*	*point queries*	*rectangle queries*
disjoint	2D kd-interval [Aga02]	$\log^2 n$	$\sqrt{\frac{n}{t}} + k$
disjoint	2D kd-interval+lsf [Aga02]	$\log^2 n$	$\alpha \log^2 n + k$
intersecting	PR [Arg04]	$\sqrt{\frac{n}{t}} + \frac{k}{t}$	$\sqrt{\frac{n}{t}} + \frac{k}{t}$

Results in three dimensions: asymptotic upper bounds $O(...)$			
input boxes	*tree [paper]*	*point queries*	*box queries*
constant slicing nr.	3D kd-int.+lsf [Hav04BG]	$\log^4 n$	$\min_\varepsilon\{(\frac{1}{\epsilon})^2 \log^4 n + k_\epsilon\}$
intersecting	PR [Arg04]	$(\frac{n}{t})^{2/3} + \frac{k}{t}$	$(\frac{n}{t})^{2/3} + \frac{k}{t}$

Figure 7: Summary of our results.

putational Geometry, Contemporary Mathematics 223: 1–56, American Mathematical Society, 1998.

[Aga01APV] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter: A Framework for Index Bulk Loading and Dynamization. In *Proc. 28th Int. Coll. Automata, Languages and Programming (ICALP)*, 2001, pp 115–127.

[Aga02] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort: Box-trees and R-trees with near-optimal query time. In *Discrete Computational Geometry* 28:291–312 (2002), or: H. J. Haverkort, *Results on Geometric Networks and Data Structures*, Ph.D. thesis, Utrecht University, 2004, Chapter 3. In the thesis version, the analysis of an algorithm has been sharpened a little, leading to better bounds on the query times.

[Agg97] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman: The S-Tree: An Efficient Index for Multidimensional Objects. In *Proc. 5th Symp. Spatial Databases (SSD)*, 1997, LNCS 1262:350–373.

[Ang97] C. H. Ang and T. C. Tan: New Linear Node Splitting Algorithm for R-trees. In *Proc. 5th Symp. Spatial Databases (SSD)*, 1997, LNCS 1262: 339–349.

[Arg04] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi: The Priority R-Tree: A Practically Efficient and Worst-Case-Optimal R-Tree. In *Proc. Management of Data (SIGMOD)*, 2004, to appear, or: H. J. Haverkort, *Results on Geometric Networks and Data Structures*, Ph.D. thesis, Utrecht University, 2004, Chapter 4.

[Ary00] A. Arya and D. Mount: Approximate range searching. *Computational Geometry: Theory & Applications* 17(3-4):135–152 (2000).

[Bar96] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal: BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum* 15(3):387–396 (1996).

[Bgn99] G. J. A. van den Bergen: *Collision Detection in Interactive 3D Computer Animation*. Ph.D. thesis, Eindhoven Technical University, 1999.

[Bhm99] C. Böhm, G. Klump, and H.-P. Kriegel: XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In *Proc. 6th Symp. Spatial Databases (SSD)*, 1999, LNCS 1651:75–90.

[Bkr92] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer: Enclosing many boxes by an optimal pair of boxes. In *Proc. 17th Symp. Theoretical Aspects of Computer Science (STACS)*, 1992, LNCS 577:475–486.

[Bmn90] 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. Management of Data (SIGMOD)*, 1990, pp 323–331.

[Brg00] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars: On R-trees with low stabbing number. In *Proc. 8th European Symp. Algorithms (ESA)*, 2000, LNCS 1879:167–178.

[Brg97KOS] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf: *Computational Geometry: Algorithms and Applications*, Springer, 1997.

[Dun99] C. A. Duncan: *Balanced Aspect Ratio Trees*. Ph.D. thesis, John Hopkins University, 1999.

[Got96] S. Gottschalk, M. C. Lin, and D. Manocha: OBB-Tree: a hierarchical structure for rapid interference de-

tection, In *Proc. Computer Graphics (SIGGRAPH)*, 1996, pp 171–180.

[Grc98a] Y. J. García, M. A. López, and S. T. Leutenegger: A Greedy Algorithm for Bulk Loading R-trees, In *Proc. Advances in GIS*, 1998, pp 163-164, and technical report 97-02, University of Denver.

[Grc98b] Y. J. García, M. A. López, and S. T. Leutenegger: On Optimal Node Splitting for R-trees, In *Proc. Very Large Databases (VLDB)*, 1998, pp 334–344.

[Gut84] A. Guttmann: R-trees: a dynamic indexing structure for spatial searching. In *Proc. Management of Data (SIGMOD)*, 1984, pp 47–57.

[Hav04] H. J. Haverkort: *Results on Geometric Networks and Data Structures*. Ph.D. thesis, Utrecht University, 2004.

[Hav04BG] H. J. Haverkort, M. de Berg, and J. Gudmundsson: Box-Trees for Collision Checking in Industrial Installation. In *Computational Geometry: Theory & Applications*, 2004, to appear, or: H. J. Haverkort, *Results on Geometric Networks and Data Structures*, Ph.D. thesis, Utrecht University, 2004, Chapter 5.

[Jag90] H. V. Jagadish: Spatial Search with Polyhedra. In *Proc. Int. Conf. Data Engineering (ICDE)*, 1990, pp 311-319

[Kam93] I. Kamel and C. Faloutsos: On Packing R-trees In *Proc. Conf. Information and Knowledge Management (CIKM)*, 1993, pp 490–499.

[Kam94] I. Kamel and C. Faloutsos: Hilbert R-tree: An improved R-tree using fractals, In *Proc. Very Large Databases (VLDB)*, 1994, pp 500–509.

[Kan97] K. V. Ravi Kanth, D. Agrawal, A. El Abbadi, and A. K. Singh: Indexing Non-uniform Spatial Data. In *Proc. Int. Database Engineering and Applications Symp. (IDEAS)*, 1997, pp 289–298.

[Kan98] K. V. Ravi Kanth, and A. K. Singh: Optimal Dynamic Range Searching in Non-replicating Index Structures. In *Int. Conf. Database Theory (ICDT)*, 1999, LNCS 1540:257-276.

[Kat97] N. Katayama and S. Satoh: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. Management of Data (SIGMOD)*, 1997, pp 369–380.

[Klo98] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan: Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Visualization and Computer Graphics* 4(1):21–36 (1998).

[Krs98] S. Krishnan, A. Pattekar, M. C. Lin and D. Manocha: Spherical shells: a higher order bounding volume for fast proximity queries, *Proc. Workshop Algorithmic Foundations of Robotics (WAFR)*, 1998, pp 177–190.

[Lar00] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha: Fast Distance Queries with Rectangular Swept Sphere Volumes. In *Proc. Int. Conf. Robotics and Automation (ICRA)*, 2000, pp 3719–3726.

[Man03] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis: *R-Trees Have Grown Everywhere*, technical report available at http://www.rtreeportal.org/, 2003.

[Mat93] J. Matoušek: Range Searching with Efficient Hierarchical Cuttings. *Discrete & Computational Geometry* 10:157–182 (1993).

[Oos90] P. van Oosterom: *Reactive Data Structures for Geographic Information Systems*. Ph.D. thesis, Leiden University, 1990.

[Ros01] K. A. Ross, I. Sitzmann, and P. J. Stuckey: Cost-based Unbalanced R-Trees. In *Proc. Statistical and Scientific Database Management (SSDBM)*, 2001, pp 203–212.

[Sit99] I. Sitzmann and P. J. Stuckey: *The O-Tree—A Constraint-Based Index Structure*, technical report, University of Melbourne, 1999.

[Whi96] D. A. White and R. Jain: Similarity Indexing: Algorithms and Performance. In *Storage and Retrieval for Image and Video Databases (SPIE)* 1996, pp 62–73.

[Zho99] Y. Zhou and S. Suri: Analysis of a bounding box heuristic for object intersection, In *Proc. 10th Symp. Discrete Algorithms (SODA)*, 1999, pp 830–839.