Efficient Evaluation of Radial Queries using the Target Tree

Michael D. Morse   Jignesh M. Patel   William I. Grosky

Department of Electrical Engineering and Computer Science
University of Michigan,
{mmorse, jignesh}@eecs.umich.edu

Dept. of Computer Science
University of Michigan-Dearborn
wgrosky@umich.edu

Abstract

In this paper, we propose a novel indexing structure, called the target tree*, which is designed to efficiently answer a new type of spatial query, called a* radial query*. A radial query seeks to find all objects in the spatial data set that intersect with line segments emanating from a single, designated target point. Many existing and emerging biomedical applications use radial queries, including surgical planning in neurosurgery. Traditional spatial indexing structures such as the R*-tree and quadtree perform poorly on such radial queries. A target tree uses a regular hierarchical decomposition of space using wedge shapes that emanate from the target point, resulting in an index structure that is very efficient for evaluating radial queries. We present a detailed performance evaluation of the target tree, comparing with the R*-tree and quadtree indexing methods, and show that the target tree method outperforms these existing methods by at least a factor of 2-10.*

1. Introduction

Computer-assisted surgery (CAS) uses registered three-dimensional images to guide a surgical procedure [8, 9]. High-resolution images of the brain can be obtained using Magnetic Resonance Imaging (MRI) both before and during the surgery. MRI and other medical images undergo a process known as *segmentation*, in which different anatomical features are identified and broken down into polygonal regions suitable for storage [11]. (The segmentation methods themselves are a focus of intense research in CAS, as no single method works well for all types of structures and images.) A key component of computer-assisted neurosurgery is a surgical preplanning step, in which the surgeon needs to plan a path of entry to a tumor that needs to be removed. The goal of this preplanning step is to find a path that causes the least damage. In brain surgeries, this path is invariably a straight line, due to the design of the current generation of surgical instruments. A key component of finding a surgical path is to find all brain structures that intersect with potential surgical trajectories that originate from a number of chosen points on the surface of the brain and end in the tumor.

We use this problem to motivate the efficient evaluation of a new type of spatial query, which we call a *radial query*, which will allow for the efficient evaluation of surgical trajectories in brain surgeries. The radial query is represented by a line in multi-dimensional space. The results of the query are all objects in the data set that intersect the line. An example of such a query is shown in Figure 1. The data set contains four spatial objects, A, B, C, and D. The radial query is the line emanating from the origin (the target point), and the result of this query is the set that contains the objects A and B.

The radial query is specified by two endpoints. One is called the *target point* and the other is called the *query point*. In applications of radial queries, usually many such queries are posed with the same target point, but with varying query points. This is the general context of the radial query which this paper will focus on hereafter.

Radial queries naturally present themselves in a number of applications in addition to the motivating CAS application. Ray tracing in computer graphics is one such example. In ray tracing, rays are shot from the camera position to find objects that intersect with them. For the closest intersection points, additional rays may then be shot. (To limit the computation, these additional rays are often only shot towards the light sources, which can also be modeled as rays being shot from the light source to the intersection point.) Octrees (3-D quadtrees) are often used to evaluate these queries [27, 28]. The target tree is also applicable in these cases. The camera position corresponds directly to the target point, since many rays are projected from the camera position into the scene. Each such ray has a different query point, since each ray is being used to determine the color and intensity of a different pixel in the scene.

Methods for storing and querying spatial data have long been of interest to the database community. Research in methods for managing spatial databases has produced a number of spatial indexing methods that speed up the evaluation of spatial queries. In fact, some of these spatial indexing methods, such as the R*-tree and the quadtree are now also part of commercial database products [1, 2, 25].

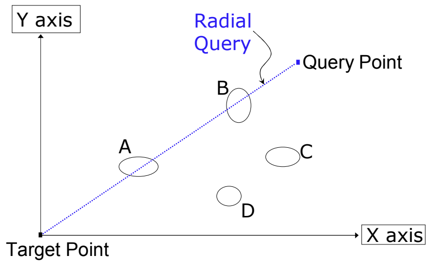


Figure 1: An example of a radial query.

Existing spatial indexing techniques largely focus on speeding up the evaluation of spatial range queries, which requires retrieving all spatial objects that overlap with some (hyper) rectangular region. If the query shape is not a rectangular region, then most spatial indexing techniques convert the query shape into a rectangle, which is then used as the search key. Such a search may result in *false hits,* which are index hits that don't contribute to the final result. If the rectangle is a poor approximation of the query shape, then the number of false hits may be high and query performance will rapidly degrade. For example, if the query shape is a line, then the rectangular approximation has a large amount of *dead-space*, which is the space in the approximation that is not part of the query object. Such dead space is problematic as it requires accessing nodes in the index tree that don't actually contribute to the final result set.

The naïve R*-tree technique of approximating an arbitrary query shape with a rectangular query key can be improved using a method proposed by Goldstein et al. [18]**.** This method views a query as a set of linear constraints, and uses this representation of the query to determine overlap between the query and the index keys.

While the optimization technique of Goldstein et al. [18] results in performance benefits, a fundamental problem with line queries is that the partitioning that is used in the index is still a rectangular structure, which results in dead space in the keys of the index. Such dead space combined with key overlap in the R*-tree can lead to a rapid degradation in search performance [6].

A large number of false hits for radial queries is a problem inherent with any indexing method that uses hyper-rectangular structures for either grouping the data objects (such as the R*-tree) or for partitioning the underlying space (such as the quadtree-based methods). An indexing structure that moves away from the rectangular decompositions of data and space can avoid some of these false hits by more closely approximating the region which it circumscribes. Towards this end, the *target tree* is introduced.

In this paper, we take a novel approach to efficiently evaluate radial queries, and introduce the target tree. We recognize that regular hyper-rectangular decompositions are problematic for radial queries, and adopt a decomposition strategy in which the underlying space is divided into *wedges* that originate from the target point. We develop techniques for recursively splitting these wedges into ever finer grained regions of space. The regions that are produced by this recursive decomposition can be encoded using a simple numbering scheme (similar in flavor to the Hilbert or Z-ordering of the quadtree). Using this encoding scheme, regular B+tree indices can be used to implement the target tree indexing method.

In this paper, we also compare the performance of the target tree with the R*-tree [6] and the octree [28] using an actual implementation in the SHORE storage manager [11]. Our experimental results show that the target tree outperforms these other indexing methods by a factor of 2X or better. In addition, we also show that construction costs for the target tree are low, allowing us to amortize the construction cost even for a modest number of radial queries.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the details of the target tree index, and Section 4 presents an evaluation of the target tree. Finally, Section 5 presents our conclusions.

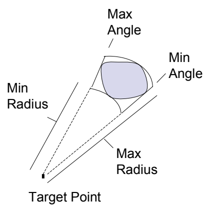
2. Related Work

Radial queries are related to the ray tracing problem in 3-D scene rendering [4]. In this problem, the illumination within a 3-D environment is simulated by backward tracing an individual light-ray from a virtual viewpoint (the initial target point) through the image plane and into the 3-D scene, determining what object it hits first. If an object is hit, more rays are traced from the object intersection point (a new target point) to all the light sources. This 3-D graphics problem has been abstracted into various sorts of visibility problems [8]: point-to-point, point-to-region, and region-to-region.

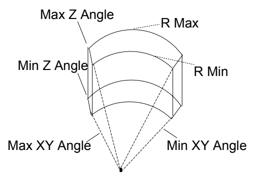
Hierarchical decomposition techniques have been extensively used to solve these problems [17]. These techniques partition the space into bins of various sizes and shapes, and seek to minimize the number of bins that must be tested for intersecting objects.

Many of these hierarchical decomposition techniques utilize a version of the *binary space partitioning tree (BSP),* which is a binary tree whose root corresponds to the convex hull or bounding box of the entire scene, and where a child of each interior node corresponds to one of the two partitions formed by intersecting its parent's spatial region with a given plane. A *kD-tree* is a variant of the BSP tree that has been very popular for solving visibility problems [16]. Octrees [27] and box-trees [5] are other hierarchical techniques that have been used over the years. Grid-based techniques encompass those whose bins are all of the same shape and size [15], as well as irregular grids [29].

While most of these techniques utilize axis-parallel hyper-rectangles for the regions at each level of the hierarchy, [5, 24] utilize other sorts of regions. While



a) MBW in 2-D



(b) MBW in 3-D

Figure 2: Minimum Bounding Wedges (MBW)

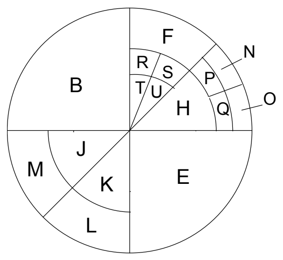


Figure 3: Target Tree Partitioning.

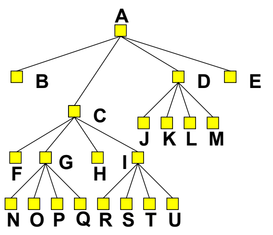


Figure 4: Sample Target Tree.

inherently different from our approach, [22] uses spherical regions in its decomposition.

BSP-trees, kD-trees, and octrees are examples of spatial partitioning techniques, while box-trees and the various variants of R*-trees [6, 19] are examples of data partitioning techniques. Each of these approaches has also been used for collision detection algorithms in interactive graphics applications. For these applications, variants of R*-trees have shown much better query times than any of the spatial partitioning schemes [21].

The Pyramid Tree [7] also uses a radially-based spatial coding scheme. The pyramid tree, which is described as deconstructing space like an onion, is an attempt to linearize a multi-dimensional space for query processing. The main goal is dimensionality reduction, so that high dimensional spaces can be indexed using a B+ tree.

The SDSS SkyServer project [30] uses a hierarchical triangular mesh scheme to index large volumes of astronomic data. The triangular mesh is a spherical decomposition of space, but it lacks a radial component. The SkyServer approach also only deals with points. In contrast, the target tree uses differing radial lengths to help partition the entire volume of the indexed region, and is primarily concerned with the management of objects.

Conical queries were studied in [14], which bear some resemblance to radial queries. The paper emphasised disk allocation methods and divided space into hyper-pyramids to achieve a partitioning. The data space partitioning and region bounding techniques we propose to solve radial queries differ significantly from this approach. The space partitioning of [14] essentially partitions the surface of a structure, where as in contrast the target tree partitions the underlying *volume*. Furthermore, the query point for the radial queries can be anywhere in the 3-D space, whereas one can view the conical queries model as representing 3-D objects as a projection on the surface. Finally, the only comparison in [14] is with sequential scan, and not with traditional indexing methods such as quadtrees or R-trees.

Our technique is also related to the *stabbing number* for a query, which is the maximum number of nodes visited in answering a query. Approaches to axis-parallel hierarchical decompositions with low stabbing numbers can be found in [3].

3. Bounding Regions in the Target Tree

In the R*-tree and region quadtrees and octrees, objects in the search space are circumscribed in a *bounding box*. A bounding box is simply a set of two points marking the lower left and upper right corners of the encompassing box. In the context of a target tree, we will construct a *Minimum Bounding Wedge* (MBW) around an object. The MBW of an object consists of the following pieces of information:

In the **2D** case the MBW consists of:

- Two angle values, the minimum and the maximum angles that any point on the surface of the object makes with the positive *x* axis in the *xy* plane.
- Two radius values, corresponding to the minimum and maximum distance of any surface point to the center.

In the **3D** case, in addition to the information stored in the **2D** case, we also store two angle values, corresponding to the minimum and the maximum angle that any point on the surface of the object makes in the *z* dimension with the *xy* plane.

An example of an MBW in **2D** is shown in Figure 2a and example of an MBW in **3D** is shown in Figure 2b.

4. Target Tree

In this section, we describe the target tree. In the interest of space, this paper does not include detailed pseudo code; the interested reader can find these details in [26].

4.1 Description

A target tree is a variable height tree that recursively decomposes the search space around a single *target point*. The index allows for insertion and deletion operations to be intermixed with searches. The tree itself may be stored on disk in a fashion similar to that of a quadtree or an octree.

Figure 3 shows an example of the target tree in the two dimensional case. The topmost node in the tree is a circle whose radius is large enough to cover all the objects that are in the data set. The nodes that compose the tree are called *wedges*. Each wedge is defined by a minimal radius, a maximal radius, a minimal angle, and a maximal

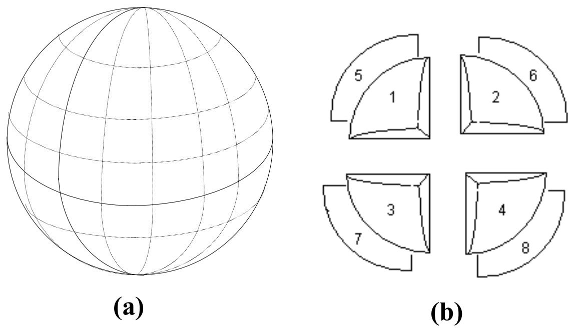


Figure 5: Two depictions of the 3D target trees. Figure **(a)** shows the angular divisions as seen from the surface of the sphere. Figure **(b)** shows the topmost split into the eight initial wedge-shaped pieces.

angle. By convention, each angle is measured from the positive x axis, similar to the polar coordinate system. In Figure 4, we see an example of a two dimensional target tree with a height of 4. The sphere as a whole is represented as node *A* in the tree. Nodes *B, C, D, and E* are at level two. Nodes *F, G, H,* and *I* are the children of node *C* and are at level three. Node *I* is further broken down into nodes *R, S, T,* and *U* at level four. Note that each of these wedges has an arc length of exactly one half that of their parent, except at the root, which splits into four parts, giving each child an arc length of π/2. The lines that separate one wedge from another always travel radially outward (and through) the center point.

The target tree in three dimensions partitions space radially outward from the center point in the same way as is done in the two-dimensional case. In addition to the data stored in two dimensions, each tree node contains two z dimension angles. An example of this partitioning is shown in Figure 5a. The radial cuts that also define the nodes in the tree are not shown.

Each node is split into eight child regions. A node is split along the bisectors of both its minimum and maximum angles in the xy plane and its minimum and maximum angles extending in the z dimension. Figure 6b shows a parent node with these bisectors indicated with dashed lines. A node is further split using a *spherical cut*. A spherical cut, shown in Figure 6c, is a division of a node about a radius of a particular length. For example, in Figure 6d, the child wedges produced from the parent (Figure 6a) are further split using a spherical cut. The minimum and maximum radii for each wedge are set by this procedure. When performing this operation, we pick a radius such that all resulting child wedges enclose equal volumes. Our current implementation of the target tree uses this subdivision, though other types of cuts could easily be used; for example, one could simply choose the middle distance between the two spherical radii.

The wedges in a target tree are assigned a unique *key* consisting of the pair *(level, wedge-code)*. These *wedge codes* may be assigned radially going outward to facilitate radial queries. Like the Hilbert or Z-ordering in the quadtree [28], the radial ordering of the target tree allows nodes in the target tree to be indexed in a B+-tree index.

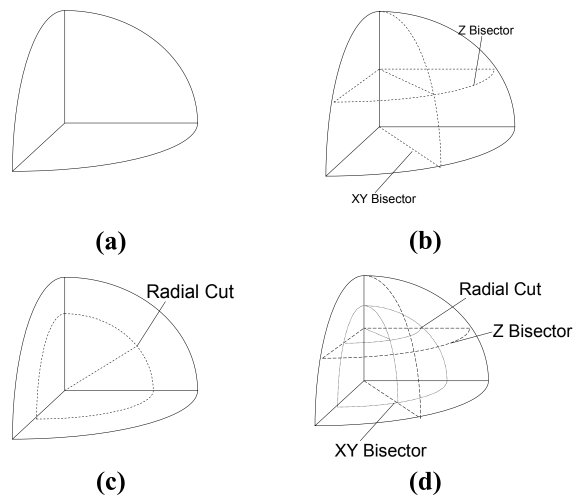


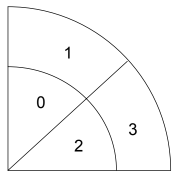
Figure 6. Figure **(a)** depicts one of the eight top level wedges in the target tree (for 3 dimensions). Figure **(b)** contains the angular bisectors of the wedge, and **(c)** shows the radial cut for the wedge. This is combined in **(d)** to produce the eight children of the original node.

However, similarities with Hilbert-style orderings end here. The Hilbert ordering is an attempt to place nodes in the tree that are close in space geographically with codes that are very similar to each other. The radial codes of a target tree do not have this property. Instead, the codes in the target tree for nodes at a particular level increase numerically along a sequence of wedges that share the same arc as you move away from the center. Consequently, on a radial query, all wedges that might be searched at a level will have codes increasing in sequence. The radial codes for a wedge at levels three and four are presented in Figure 7a and 7b, respectively. The codes need not be dense; since the target tree is a variable depth tree, some codes at various levels may go unassigned. In Figure 7c, we see such a variable depth tree. Even though the wedge with key (3, 0) has not split to form wedges at level 4, the children of wedge (3, 1) have not been assigned code 0 or 1. Instead, they are assigned codes assuming that the other wedges around them are either there already, or might be there at some point in the future. Each key is thus unique for a particular node, whether it is in the tree or not.

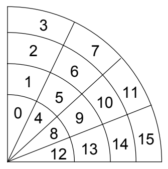
The target tree can be stored and searched using a B+-tree by simply using the key for a target tree node, which is a (level, wedge-code) pair, as the B+-tree index key. Our implementation uses this method.

4.2 Insert

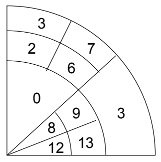
A spatial object in a three-dimensional space consists of a set of surface points which, when connected, form lines, or polygonal surfaces that make up the boundaries of the



(a) Level 3 codes

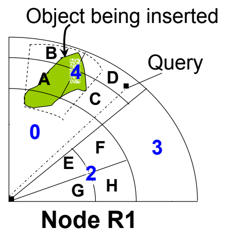


(b) Level 4 codes

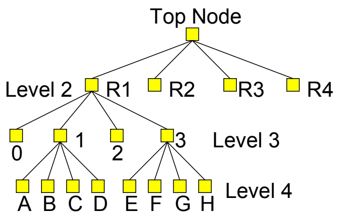


(c) Intermixed level 3 and 4 codes

Figure 7: Radial Codes



(a)



(b)

Figure 8. **(a)** is an example of a MBW being inserted. **(b)** is an example of a search query.

region. We will consider inserting regions defined by some arbitrary number of such surface points in the search space into the target tree.

The insertion algorithm progresses as follows: Starting at the root of the tree, the MBW of the object to be inserted is checked to see which, if any, of the children of the root the object may lie within. The object will be inserted into any and all children whose space partition contains the MBW, either in whole or in part. This process will recurse through the nodes in the tree until the object is inserted into a leaf or leaves that have space for it. If an object must be inserted into a leaf, and the leaf has reached a full capacity, the leaf is split, producing a new internal node in the tree and some number of new children, depending on the dimensionality. The new MBW, along with all the old MBWs that were contained in the old leaf, are inserted into any of the new children whose space partition intersects that of their respective MBWs. The algorithm for splitting a node, which was described in section 4.1 is called during the insert operation whenever a leaf page overflows.

To illustrate the insertion of a minimum bounding wedge into a target tree, consider inserting the object and MBW shown in Figure 8a into the tree represented in Figure 8b. The insertion algorithm will begin by testing the four largest pie shaped wedges at level 2. Let **R1**, **R2**, **R3**, and **R4** denote these wedges. Each of these wedges is checked to see if it overlaps the MBW of the object being inserted. Only **R1** (shown in 8a) overlaps with the MBW of the object, since the object lies entirely in the first quadrant. Next, the four nodes at level 3, namely nodes labeled **0**, **1**, **2**, and **3** in Figure 8a, will be tested to see if they overlap the MBW. Nodes **2** (which contains children **E**, **F**, **G**, and **H**) and **3** will be eliminated. Node **0** is a leaf, so it will include a reference to the object and its MBW. Node **2** is not a leaf, so the insertion will continue with its four children, **A**, **B**, **C**, and **D**. Each overlaps the object's MBW and will have a reference placed in them. At the leaf level inserts, the insertion could have caused a further node split, if the node fill threshold was exceeded.

4.3 Search

A search consists of a query point in the three-dimensional space. For the search, assume that the target tree has been loaded into a B+ tree, where the key of each node in the target tree also serves as the key of the B+ index. The search seeks to determine which, if any, of the objects in the tree intersect a line drawn between the query point and the center point. The complexity of the search is bounded by at most *O(MaxDepth)* scans of the B+ tree, essentially one scan for each key level. In the search algorithm, we obtain the code of the target tree node closest to the center point and scan from that (level, code) key pair to the key pair that contains the code of the target tree node that actually contains our query point at that level.

To illustrate how the search progresses, consider as an example a query point that lies in region **D** of Figure 8a. The insertion will first check the parent, spherical node at level **1** by probing the B+ tree. The search progresses top down, first scanning the top level node, then node **R1** at the second level. At the third level, the search touches two nodes when it scans from node **0** to node **1**. Here, two container classes will be retrieved from the B+ tree and checked for intersecting pieces. At the fourth level, the search will scan for potentially four nodes, the two children of node **0** and the two children of node **1** that intersect the radial query. Only two of them, nodes **C** and **D** will be found since node **0** itself is a leaf. Finally, the B+-tree will be probed for entries at the fifth level. None will be found so the search will terminate.

4.4 Deletion and Updates

The deletion algorithm simply deletes all the entries for the object from the B+-tree. Updates to existing entries are simply treated as a deletion followed by an insertion operation.

5. Experimental Evaluation

5.1 Experimental Setup

In this section, we present experimental results comparing the performance of the target tree with the R*-tree [6] and the quadtree/octree [28].

All indexing techniques are implemented in the SHORE storage manager [11]. SHORE currently supports two indexing methods: 2D R*-trees [6] and B+-trees [12]. The R*-tree key definition in SHORE was modified in a fairly straightforward fashion to support 3D R*-trees. In addition, the R*-tree search algorithm was modified to implement the linear constraint method of Goldstein et al. [18]. The quadtree and octree indices are implemented by generating integer codes that are indexed using a B+-tree. (This method is similar to the method described in [25].) The target tree is implemented by indexing the (level, code) pair using a B+-tree. The B+-tree has one entry for each leaf level entry in the quadtree/target tree.

The leaf level of the quadtree and the target tree is implemented by using an *intermediate file* in which there is one page for each entry in the B+-tree index (which also corresponds to a leaf entry in the target tree or the quadtree). The pointers in the B+-tree entries point to the pages in this file.

A page in the intermediate file contains all the entries that are represented by the leaf node in the quadtree/target tree. These entries consisted of the MBR/MBW and a record id to the actual spatial object in the main data file.

SHORE was compiled with a 32KB page size. This large page size benefited all indexing methods. All experiments were run on a 1.70 GHz Intel Xeon processor machine, running Debian Linux 2.4.13, and configured with a 40GB Fujitsu MAN3367MP SCSI hard drive.

5.2 Data Sets

We used two data sets for our experiments. The first data set is neurological data from the Talairach-Tournoux 1988 Atlas. This atlas is proprietary and was obtained by a special arrangement based on our collaboration with the Neurosurgery Department at Wayne State University. The atlas data was segmented into 111 different distinct brain structures producing a total of 11,369 different three-dimensional objects. For example, one such distinct brain structure is the anterior commissure. The anterior commissure contains sixty different polygons forming a circumscribed convex hull about the region. The total size of this data set is about 800KB. Even though this atlas data is small, we chose to use it in our evaluation since it represents real data for the CAS application (and getting real data in this domain is very hard). We note that as segmentation methods improve, the geometry of the features identified in atlases and by segmentation of high-resolution MRI images is likely to produce data with more detailed geometry. In other words, future data set are likely to be significantly larger than this data set.

The second data set is a 3D animated ray tracing data set [13]. In the computer graphics community, octrees are often used for ray tracing on such data sets [20], which allows us to compare the target tree with this existing method. We also chose to test the R-tree, since it is a popular multi-dimensional indexing technique.

The graphics data set that we use corresponds to a scene which is a complex assortment of machine parts and power plant structures [13]. We picked a complex scene from this data set, which has approximately four million such triangular patches and the data file when stored in SHORE has a disk footprint of 245.1 MB.

For each data set, we vary the buffer pool size from a small value such that only a small amount of the index fits into memory to a large value where the entire index fits in memory.

	Index Size (in MB)	**Index construction time in sec for various buffer pool sizes**		
		4MB	16MB	64MB
R-tree	213.0	317	280	266
Octree	727.0	1482	1076	740
Target tree	1.6GB	1243	811	567

Table 1: Index Sizes and Build Times for the Graphics Data Set

5.3 Index Construction and Sizes

In this section, we first present the time it took to build the three indices for the graphics dataset. The sizes of the indices and the construction costs for the indices for different buffer pool sizes are listed in Table 1.

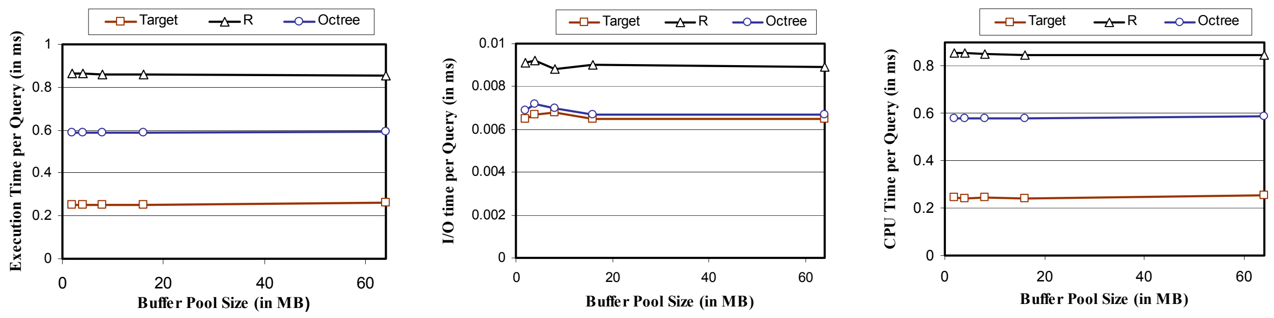
Since the target tree and the octree indices partition the data space and include references to objects that lie either in whole or in part within that space, a fill threshold (or bucket threshold) is needed, to determine at what point to split the node. For both structures, a threshold of 500 was chosen, which is the maximum capacity of a disk page.

The R*-tree is built using a bulk loading mechanism [23]. This makes the build of the R*-tree faster than that of the quadtree or target tree.

As can be seen from Table 1, both the quadtree and the target tree have comparable construction times. Looking at the number of index entries, we see that both the octree and target tree methods incur a modest overhead. The actual sizes of the octree and the target tree include the size of the intermediate file, which has low page occupancy. Consequently the disk footprint for these indices is larger than that of the R-tree. (The intermediate file contributes around 95% of the index footprint for both the octree and the target tree.) In our current implementation each leaf page in the octree and target tree is mapped to an entire page in the intermediate file. More sophisticated methods that allow multiple leaf nodes to share a disk page can improve the disk occupancy.

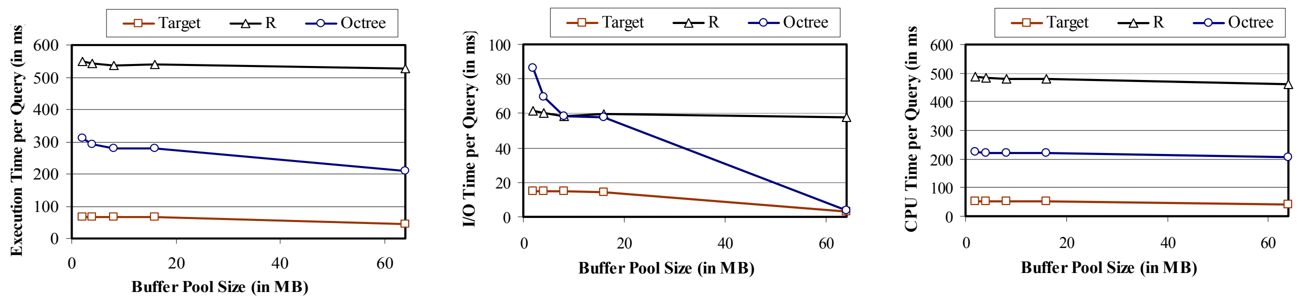
5.4 Performance Evaluation

In this section, we compare the execution of the three indices. For this experiment, we picked a target point at the center of the *universe* of each data set, and randomly picked a thousand query points in the entire universe. Radial queries were constructed from each of these points. We executed this entire batch of queries, and computed per query costs, which we report in this section.



(a) Execution Time **(b) I/O Time** **(c) CPU Time**

Figure 9: Query Performance on the Segmented Brain Data Set (Average costs for a batch of 1,000 queries)



(a) Execution Time **(b) I/O Time** **(c) CPU Time**

Figure 10: Query Performance on the Graphics Data Set (Average per query costs for a batch of 1,000 queries).

First we consider the query performance using the segmented human brain data set. These results are shown in Figure 9 for buffer pool sizes from 2MB to 64MB. We show the average query execution time (Figure 9a), the average disk IO time incurred by each query (Figure 9b), and the CPU costs per query (Figure 9c).

As can be seen from Figure 9a, the target tree outperforms the octree by a factor of 2-3X, and the R*-tree by a factor of 3-4X. These efficiencies come from both better IO characteristics (see Figure 9b), and better CPU characteristics (see Figure 9c).

Next, we consider the query performance using the graphics data set. These results are shown in Figure 10 for 2MB to 64MB buffer pool sizes. We show the average query execution time (Figure 10a), the average disk IO time incurred by each query (Figure 10b), and the CPU costs per query (Figure 10c).

As can be seen from Figure 10a, the target tree outperforms the octree by a factor of 6X, and the R*-tree by a factor of 10X. These efficiencies come from both better IO characteristics (see Figure 10b), and better CPU characteristics (see Figure 10c).

The dramatically higher performance on queries for the target tree comes from the following three factors: First, the R*-tree has internal nodes that have a lot of overlap with each other. This overlap means that the R-tree search has to access many nodes which do not produce any results. The set of nodes considered by the target tree, due to its shape, is far less. Second, it does not suffer from the dead space problem that the R*-tree encounters. If there are no objects in a region, the target tree will not split that area, and query evaluation does not need to explore that portion of space. Third, the R*-tree uses rectangular approximations, which are likely to result in false hits with queries that are not rectangular regions.

The octree also suffers from the use of rectangular decomposition structures. The octree in this case is eight levels deep, and evaluation of the radial queries requires accessing a large number of nodes in this deep tree. The target tree also uses codes, but the codes that match are sequential arranged (see Figure 7), as opposed to the octree in which the Hilbert matching codes are arbitrarily spread across the entire coding space.

6. Conclusions

In this paper we have presented a new type of query, called a radial query, which consists of a query line segment that emanates from a target point. A number of applications, ranging from ray-tracing to computer-assisted surgery, require evaluating a large set of radial queries that emanate from the same target point. Conventional spatial indexing methods perform poorly for this class of queries.

We have proposed a new indexing method, called the *target tree*, which recursively decomposes the space into wedge-like partitions. Like the quadtree, this hierarchical decomposition strategy can be used to produce integer codes that can then be indexed using a regular B+-tree. Consequently the target tree can easily be implemented in existing commercial and free database products. Using a number of real data sets, we show that the target tree outperforms the R*-tree and octree indexing methods by at least a factor of 2X.

Acknowledgements

This study is supported in part by a grant received from the Michigan Life Sciences Corridor (Grant No. MEDC-459). We would also like to thank Igor Guskov for valuable discussions on ray-tracing and volume rendering methods that are currently used in computer graphics.

References

1. Informix Spatial DataBlade Module, IBM (http://www-306.ibm.com/software/data/informix/blades/spatial/rtree.html), 2004.
2. Adler, D.W., DB2 Spatial Extender - Spatial data within the RDBMS. In *VLDB*, 2001, 687-690.
3. Agarwal, P.K., Berg, M.d., Gudmundsson, J., Hammar, M. and Haverkort, H.J., Box-trees and R-trees with Near-Optimal Query Time. In *Proceedings of the 17th ACM Symposium on Computational Geometry*, 2001, 124-133.
4. Akenine-Moller, R. and Haines, E. *Real-Time Rendering*. 2nd edition. A.K. Peters Ltd., Natick, Massachusetts, USA, 2002.
5. Barequet, G., Chazelle, B., Guibas, L.J., Mitchell, J.S.B. and Tal, A. BOXTREE: A Hierarchical Representation for Surfaces in 3D. *Computer Graphics Forum*, *15* (3) 1996, 387-396.
6. Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B., The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990, 322-331.
7. Berchtold, S., Bohm, C. and Kriegel, H.-P. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *SIGMOD*, Seattle, WA, USA, 1998, 142-153.
8. Bittner, J. Hierarchical Techniques for Visibility Determination, Postgraduate Study Report DC-PSR-99-05, Department of Computer Science and Engineering, Czech Technical University, Prague, Czech Republic, 1999.
9. Bucholz, R.D. Introduction to Journal of Image Guided Surgery. *J Image Guid Surg (editorial article)*, *1*(1) 1995, 1-3.
10. Bucholz, R.D. and McDurmont, L. From Discovery to Design: Image-guided Surgery. *Clin Neurosurg*, *50* 2003, 13-25.
11. Carey, M.J., DeWitt, D.J., Franklin, M.J., et al., Shoring Up Persistent Applications. In *SIGMOD*, 1994, 383-394.
12. Comer, D. The Ubiquitous B-Tree. *Computing Surveys*, *11* (2) 1979, 121-137.
13. Erikson, C., Manocha, D. and William V. Baxter, I., HLODs for Faster Display of Large Static and Dynamic Environments. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 2001, 111-120.
14. Ferhatosmanoglu, H., Agrawal, D. and Abbadi, A.E. Efficient Processing of Conical Queries. In *CIKM*, 2001, 1-8.
15. Fujimoto, A., Tanaka, T. and Iwata, K. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, *6* (4) 1986, 16-26.
16. Fussell, D. and Subramanian, K.R. Fast Ray Tracing Using K-D Trees, Technical Report TR-88-07, Dept. of Computer Science, University of Texas, Austin, Texas, March 1988.
17. Gaede, V. and Gunther, O. Multidimensional Access Methods. *ACM Computing Survey*, *30* (2) 1998, 170-231.
18. Goldstein, J., Ramakrishnan, R., Shaft, U. and Yu, J.-B., Processing Queries By Linear Constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997, 257-267.
19. Guttman, A., R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984, 47-57.
20. Harvan, V. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News*, *12* (2) December 21, 1999.
21. Held, M., Klosowski, J.T. and Mitchell, J.S.B., Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs. In *Proceedings of the Seventh Canadian Conference on Computational Geometry*, 1995, 205-210.
22. Hubbard, P.M. Collision Detection for Interactive Graphics Applications. *IEEE Transactions on Visualization and Computer Graphics*, *1* (3) 1995, 218-230.
23. Kamel, I. and Faloutsos, C., On Packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management, Washington (CIKM)*, 1993, 490-499.
24. Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H. and Zikan, K. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, *4* (1) 1998, 21-36.
25. Kothuri, R.K.V., Ravada, S. and Abugov, D., Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *SIGMOD*, 2002, 546-557.
26. Morse, M., Patel, J., and Grosky, W., Evaluating Radial Queries with the Target Tree. University of Michigan Technical Report. 2005.
27. Samet, H., Implementing Ray Tracing with Octtrees and Neighbor Finding. In *Computers & Graphics*, 1989, 445-460.
28. Samet, H. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, *16* (2) 1984, 187-260.
29. Silva, C.T. and Mitchell, J.S.B. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics*, *3* (2) 1997, 142-157.
30. Szalay, A.S., Kunszt, P.Z., Thakar, A., Gray, J. and Slutz, D.R. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In *SIGMOD*, 2000, 451-462.