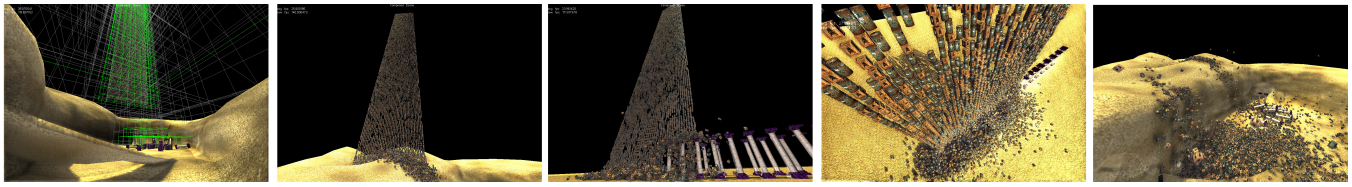
Appeared in the *Proceedings of the ACM SIGMOD Conference*, pp. 169–180, New York, June 2013

Indexing Methods for Moving Object Databases: Games and Other Applications*

Hanan Samet[§] Jagan Sankaranarayanan[‡†] Michael Auerbach[§]
[§]University of Maryland, College Park, MD [‡]NEC Labs America, Cupertino, CA
hjs@cs.umd.edu, jagan@nec-labs.com, mikea@umd.edu



ABSTRACT

Moving object databases arise in numerous applications such as traffic monitoring, crowd tracking, and games. They all require keeping track of objects that move and thus the database of objects must be constantly updated. The cover fieldtree (more commonly known as the loose quadtree and the loose octree, depending on the dimension of the underlying space) is designed to overcome the drawback of spatial data structures that associate objects with their minimum enclosing quadtree (octree) cells which is that the size of these cells depends more on the position of the objects and less on their size. In fact, the size of these cells may be as large as the entire space from which the objects are drawn. The loose quadtree (octree) overcomes this drawback by expanding the size of the space that is spanned by each quadtree (octree) cell c of width w by a cell expansion factor p ($p > 0$) so that the expanded cell is of width $(1+p) \cdot w$ and an object is associated with its minimum enclosing expanded quadtree (octree) cell. It is shown that for an object o with minimum bounding hypercube box b of radius r (i.e., half the length of a side of the hypercube), the maximum possible width w of the minimum enclosing expanded quadtree cell c is just a function of r and p, and is independent of the position of o. Normalizing w via division by $2r$ enables calculating the range of possible expanded quadtree cell sizes as a function of p. For $p \geq 0.5$ the range consists of just two values and usually just one value for $p \geq 1$. This makes updating very simple and fast as for $p \geq 0.5$, there are at most two possible new cells associated with the moved object and thus the update can be done in $O(1)$ time. Experiments with random data showed that the update time to support motion in such an environment is minimized when p is infinitesimally less than 1, with as much as a one order of magnitude increase in the number of updates that can be handled vis-a-vis the $p = 0$ case in a given unit of time. Similar results for updates were obtained for an N-body simulation where improved query performance and scalability were also observed. Finally, in order amplify the paper, a video titled "Crates and Barrels" was produced which is an N-body simulation of 14,000 objects. The video as well as a JAVA applet that illustrates the behavior of the loose quadtree are both available from `http://www.cs.umd.edu/~hjs/loosequad/`.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures

General Terms

Algorithm, Performance

Keywords

game databases, moving objects, spatial data structures, cover fieldtree, loose quadtree, loose octree, spatial indexing, spatial databases, game programming

1. INTRODUCTION

One of the motivations for the development of geographic information systems (GIS) is to keep track of objects (e.g., QUILT [38, 41], and the SAND Browser [11, 36]) for both location-based and feature-based queries [4]. Similar needs arise in game applications (e.g., [9,16]), where the difference is that the objects are not usually static. Instead, they are constantly moving and thus the database of objects must be constantly updated. An attractive method of representing spatial objects to support the tracking process uses an object hierarchy where minimum bounding hypercube boxes (e.g., an R-tree [15,37]) are used to speed up the process of detecting if objects are present or overlap other objects. One of the drawbacks of such a representation is that the hierarchies of different sets of objects

*This work was supported in part by the National Science Foundation under grants CCF-05-15241, IIS-0713501, IIS-10-18475, IIS-12-19023, Microsoft Research, Google, NVIDIA, the E.T.S. Walton Visitor Award of the Science Foundation of Ireland, and the National Center for Geocomputation at the National University of Ireland at Maynooth.

†Work done while the author was at the University of Maryland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

are not in registration thereby making set operations between the two sets such as unions and intersections more complex.

A solution is to use a hierarchy of congruent cells while still not decomposing the objects. In this case, the hierarchy is based on a regular decomposition of the underlying space such as a region quadtree (e.g., [37]) and then associates each object with its minimum enclosing quadtree cell. Methods that employ this technique include the MX-CIF quadtree [1, 24, 45], multilayer grid file [44], R-file [19], filter tree [40] (used for spatial join algorithms [18,20,21]), and SQ-histogram [3] (used for selectivity estimation in processing spatial queries) where the primary difference lies in the nature of the access structure that is used. For example, Figure 1a is the cell decomposition induced by the MX-CIF quadtree for a collection of rectangle objects, while Figure 1b is its tree representation. Notice that more than one object is associated with some of the nodes in the tree which means that the objects have the same minimum enclosing quadtree cell (e.g., the root and its NE child, where the children are referred to as NW, NE, SW, and SE denoting the Northwest, Northeast, Southwest, and Southeast quadrants, respectively, of corresponding cells).

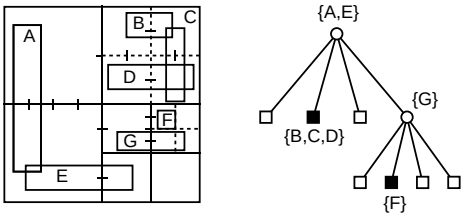


Figure 1: (a) Cell decomposition induced by the MX-CIF quadtree for a collection of rectangle objects and (b) its tree representation (from [37]).

The drawback of these methods is that the size of these minimum enclosing quadtree cells depends on the position of the centroids of the objects and is independent of the size of the objects, subject to a minimum which is the size of the object. In fact, it may be as large as the entire space from which the objects are drawn. This has bad ramifications for applications where the objects move including games, traffic monitoring, and streaming. In particular, if the objects are moved even slightly, then they usually need to be reinserted in the structure.

The cover fieldtree [12,13] and the more commonly known loose quadtree (octree) [47] are designed to overcome this independence of the size of the minimum enclosing quadtree cell and the size of the object (see also the expanded MX-CIF quadtree [2], multiple shifted quadtree methods [7, 26, 27], and the partition fieldtree [12, 13]). This is done by expanding the size of the space that is spanned by each quadtree cell c of width w by a cell expansion factor p ($p > 0$) so that the expanded cell is of width $(1 + p) \cdot w$ and an object is associated with its minimum enclosing expanded quadtree (octree) cell. The notion of an expanded quadtree cell can also be seen in the quadtree medial axis transform [34, 35]. For example, letting $p = 1$, Figure 2 is the loose quadtree corresponding to the collection of objects in Figure 1(a) and its MX-CIF quadtree in Figure 1(b). In this example, there are only two differences between the loose and MX-CIF quadtrees:

1. Rectangle object E is associated with the SW child of the root of the loose quadtree instead of with the root of the MX-CIF quadtree.

2. Rectangle object B is associated with the NW child of the NE child of the root of the loose quadtree instead of with the NE child of the root of the MX-CIF quadtree.

To further understand the loose quadtree and its behavior, see the publicly available web site at `http://www.cs.umd.edu/~hjs/loosequad/`. The web site also contains a video titled "Crates and Barrels" that shows an N-body simulation containing 14,000 objects. The video illustrates the improvement in in performance when using a loose quadtree over an MX-CIF quadtree.

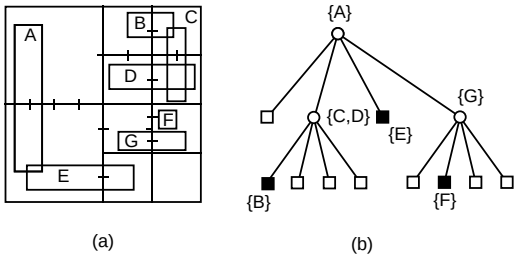


Figure 2: (a) Cell decomposition induced by the loose quadtree for a collection of rectangle objects identical to those in Figure 1, and (b) its tree representation (from [37]).

Ulrich [47] has shown that given a loose quadtree cell c of width w and cell expansion factor p, the radius r of the minimum bounding hypercube box b of the smallest object o that could possibly be associated with c must be greater than $pw/4$.

Our contribution is the realization that the real utility of the loose quadtree is best evaluated in terms of the inverse of the above relation as we are interested in minimizing the maximum possible width w of c given an object o with minimum bounding hypercube box b of radius r (i.e., half the length of a side of the hypercube) denoted by MBHR(o,b,r). This is because reducing w is the real motivation and goal for the development of the loose quadtree as an alternative to the MX-CIF quadtree for which w can be as large as the width of the underlying space. We achieve our goal in Section 3 by examining the range of the relative widths of c and b as this provides a way of taking into account the constraints imposed by the fact that the range of values of w is limited to powers of 2. In particular, the novelty of our work lies in our showing this range to be just a function of p, and hence independent of the position of o. Moreover, we prove that for $p \geq 0.5$, the relative widths of c and b take on at most two values, and usually just one value for $p \geq 1$. This makes updating the index very simple when objects are moving as there are at most two possible new cells associated with a moved object, instead of $\log_2$ of the width of the space in which the objects are embedded (which can be as large as 16 assuming a $2^{16} \times 2^{16}$ embedding space as used by us). In other words, we have shown how to update in $O(1)$ time for $p \geq 0.5$ which is of great importance as there is no longer a need to perform a search for the appropriate quadtree cell.

The rest of this paper is organized as follows. Section 2 discusses related work in the context of the moving object databases literature. Section 3 shows how to achieve position independence for the width of the minimum enclosing quadtree cell c by examining the range of the relative widths of c and the minimum bounding hypercube box b of object o, denoted by MBH(o,b), and also how to take into account the constraints imposed by the fact that the range of values of the width of c is limited to powers of 2. Section 4 presents a cell insertion algorithm for the loose quadtree. Section 5 discusses the ramifications of the results of Section 3. Section 6 contains an experimental evaluation of the loose quadtree with respect to the extent that it needs to be updated on account of object motion for different values of p and object size distribution, Section 7 shows the results of using the Loose Quadtree in an N-body simulation which is typical of the type of functionality needed in modern video games and hence is conducted in a main memory

environment unlike the experiments in Section 6 which used secondary storage. Concluding remarks are drawn in Section 8.

2. RELATED WORK

As pointed out in [43], updates to spatial indices, such as those that occur due to motions of the objects, require a coarse tree-level locking instead of object-level as entire sub-trees may have to locked to facilitate deletion of an object from its prior position and reinsertion into its newer position. Several approaches [22, 29, 33, 46, 49] have been proposed which we broadly classify based on the strategy that they use to minimize the updates to the spatial index. If the movement of the data is *predictive*, or in other words, the future position of the object is known for a short future time period, then the structure can be optimized to answer queries for a short period without having to rebuild the structure. This is the strategy adopted by the TPR-tree [33], its variant the TPR*-tree [46].

[22, 49] resort to space-filling curves and B-trees in lieu of a more traditional spatial index, such as an R*-tree [6] in order to take advantage of the B-tree's ability to handle high rates of updates. Similarly, [25, 31] transform the position of a moving object using a Hough transformation into a dual space, such that updates are only required when the velocity of the object changes resulting in overall fewer updates. A method for indexing trajectories of moving objects is given in [32], while a method to index objects moving on a road network that takes advantage of the restricted motions of these objects along the road network within prescribed speed limits is given in [29]. To make spatial indices more update efficient, [42, 43] propose a general method to take advantage of the many cores in modern computer architectures even as queries are applied to the spatial data structure.

Our work is unique in the sense that we deal with objects that have extents (i.e., geometries), while most of the work in this area deals with point objects. While the geometry of the moving objects may not be important in vehicular or people tracking applications, it is not so for games and physics-based applications. Note that we cannot assume that the position of the object and its trajectory can be reasonably estimated. A typical game scenario consists of several dynamic objects which move in response to other dynamic or static objects in the scene, making long term prediction of their movement quite difficult. Moreover, rendering the scene requires that the spatial data structures be queried tens of times per second in order to ascertain which objects are visible in the scene, how they are interacting, and even how light interacts with them in order to produce the desired interactivity that users expect from games. A B-tree could be used to speed up updates in lieu of a spatial index such as an R*-tree or a quadtree. In this paper we use two experimental setups: one involving a loose quadtree indexed by a B-tree where the blocks of the quadtree are represented by their location codes, and another using a pointer-based quadtree structure. Finally, in contrast to methods that increase the throughput of a spatial index by harnessing the work of multiple threads updating the spatial index (e.g., [42, 43]), our method avoids as many updates to the spatial index as possible. Our method and multi-threaded update intensive methods are not mutually exclusive in the sense that one does not preclude the application of the other.

3. CALCULATION OF THE MAXIMUM LOOSE QUADTREE CELL WIDTH

A key principle to observe is that in the loose quadtree, the smallest expanded quadtree cell c of width w that contains the object o has the property that the centroid of o (actually of MBHR(o,b,r)) is contained in the non-expanded portion of c. Thus insertion proceeds by finding the smallest quadtree cell c that contains the centroid of b, and whose expanded cell also contains o. The traditional way of finding c is to recursively search the quadtree starting at the root and descend to the appropriate child based on the value of the centroid. In fact, it turns out that there is even an easier way of determining c, which involves little search (i.e., few descents in the quadtree). In particular, we show below that the width w of c must lie within a relatively small range of values, thereby greatly restricting the number of possible cells that must be tested for the inclusion of o.

Recall that one of the key drawbacks of data structures such as the MX-CIF quadtree that associate an object o with the minimum sized quadtree cell c of width w that encloses object o's MBHR(o,b,r) is that w is a function of the position of o, and to a lesser extent, a function of r in the sense that only its minimum is a function of r. In contrast, in the loose quadtree, as we show in the rest of this section, the dependence of w on the position of o is reduced significantly. In particular, we demonstrate that w lies within a range of values that only depend on the radius r of o's MBH(o,b) and the value of the cell expansion factor p. In fact, normalizing w via division by $2r$ leads to Theorem 3.1, given below, which enables the calculation of a range of expanded quadtree cell sizes whose lower and upper bounds only depend on p. As we will see, restricting of the cell sizes to be powers of 2 (i.e., $1, 2, 4, \ldots 2^n$) makes these bounds quite tight, with the range taking on at most two values for $p \approx 1$, which turns out to be the primary p value of interest. Thus the size of the containing quadtree cell is almost the size of the object or the size of the next larger quadtree cell.

THEOREM 3.1. *The ratio $w/2r$ of the widths of the expanded minimum enclosing quadtree cell c and MBH(o,b) obeys*

$$\frac{1}{1+p} \leq \frac{w}{2r} < \frac{2}{p}.$$

PROOF. We first derive a lower bound on the range of the ratios. From the definition of the cell expansion factor p, we know that given an object o with MBHR(o,b,r) the smallest quadtree cell c of width w with which o can be associated so that o's centroid lies in the non-expanded portion of c arises when the centroids of b and c coincide, and moreover the cell c' resulting from the expansion of c (i.e., having width $(1+p)w$) is just large enough to contain b of width $2r$ (see Figure 3(a)). This leads to the following inequality:

$$(1+p)w \geq 2r \tag{1}$$

and can be rewritten as

$$\frac{w}{2r} \geq \frac{1}{1+p}. \tag{2}$$

We can use similar reasoning to obtain an upper bound on the range of the ratios, and in the process use a similar construction to that of Ulrich [47] except that for a given cell expansion factor p, Ulrich assumed the existence of a quadtree cell c of width w and was seeking the radius r of the minimum bounding hypercube box b of the smallest object o that could possibly be associated with the expanded cell c, while we are assuming that for a given cell expansion factor p, we are given an object o with minimum bounding hypercube box b of radius r and are seeking the width w of the largest cell c with whose expanded cell b would be associated. We make use of our observation that the centroid of the object o with MBHR(o,b,r) is always required to be contained in the non-expanded portion of the associated quadtree cell.

An alternative way of casting our goal is that we want to find the width of the smallest object o that can have a minimum enclosing expanded quadtree cell c of width w. Doing this enables us to calculate an upper bound on the range of $w/2r$. Given our requirement that the centroid of the object is always in the non-expanded portion of the minimum enclosing expanded quadtree cell c, we find that one of c's corners is coincident with the centroid of o, and that the radius r of b is not too large so that b is too large for the expanded region of c (i.e., an attainable upper bound on r of $pw/2$ as shown in Figure 3(b)), and just large enough so that b does not

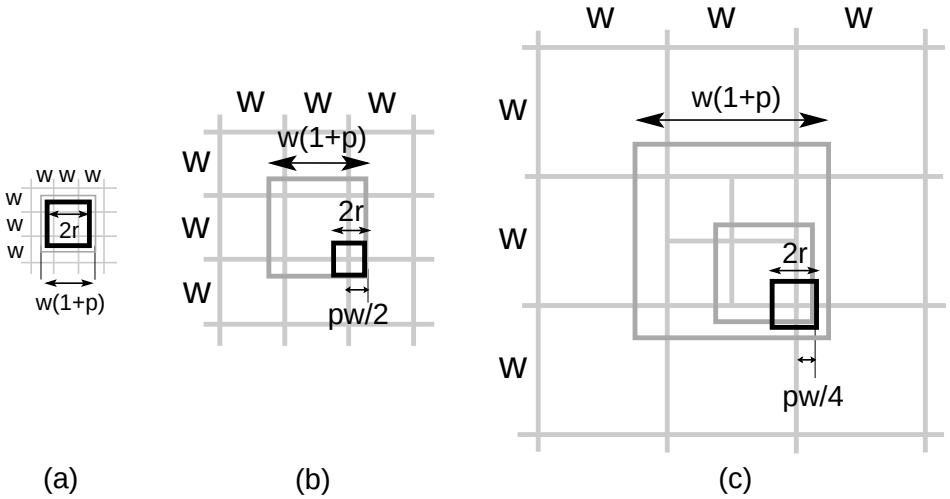


Figure 3: Assuming cell expansion factor p and an examples showing the (a) smallest ratio of the width w of the quadtree cell c associated with b and the width of b which is attained when the centroids of o and c coincide, and the (b) lower and (c) upper bounds on the largest ratio attained when the centroid of o coincides with one of the corners of c. Note that (c) is drawn at a different scale than (b).

fit in the expanded region of one of the subcells of c of width $w/2$ (i.e., an unattainable lower lower bound on r of $pw/4$ as shown in Figure 3(c)). Equivalently, for this particular configuration, we say that $pw/4 = 2^{k-1} = r - \delta' < r \leq 2^k = pw/2$ for some value of k and $\delta' > 0$. Simplifying the notation by letting $\delta' = \delta w/4$, we have $pw/4 = 2^{k-1} = r - \delta w/4 < r \leq 2^k = pw/2$ for some $\delta > 0$. Since the width w of c is the same for all values of r in this range, we point out that c's width relative to that of b is maximized when r takes on the value:

$$r = pw/4 + \delta w/4, \delta > 0. \tag{3}$$

which can be rewritten as:

$$w/2r = \frac{w}{\frac{pw}{2} + \delta\frac{w}{2}}, \delta > 0, \tag{4}$$

$$w/2r = \frac{2}{p+\delta} < \frac{2}{p}, \tag{5}$$

$$w/2r < \frac{2}{p}. \tag{6}$$

Combining relations 2 and 6 yields the range:

$$\frac{1}{1+p} \leq \frac{w}{2r} < \frac{2}{p}. \tag{7}$$

□

We interpret Theorem 3.1 as follows. Without loss of generality, we assume that the quadtree cell corresponding to the root of the loose quadtree has length 2^g, where g is an integer. This enables us to avoid dealing with negative values of k, which is somewhat counter intuitive, as would be the case were we to continue with the unit hypercube assumption. In this case, all cells c in the loose quadtree have width $w = 2^k$, such that $k \leq g$ is an integer. Now, for any given value x, let us define a function $M(x)$ which determines a k such that $2^{k-1} < x \leq 2^k$, and returns the value 2^k. In other words,

$$M(x) = 2^k, \text{ s.t. } 2^{k-1} < x \leq 2^k. \tag{8}$$

Moreover, we also have that

$$1 \leq \frac{M(x)}{x} < 2. \tag{9}$$

The rationale behind the function $M(x)$ is that it *quantizes* x to the next higher power of 2 unless it is already a power of 2. To explain the utility of $M(x)$ from a geometric point of view, consider an input object R with a minimum bounding hypercube box of radius r. We have that $M(r)$ is the radius of the smallest quadtree cell (i.e., half the width) that can potentially contain R. We now derive the minimum and maximum possible ratios of $w/2r$ in terms of $M(.)$. Let us assume that $2r$ is a power of 2 which means that the minimum bounding box is a quadtree cell (i.e., $M(r) = r$). The number of levels of the loose quadtree spanned by the range $[1/(p+1), 2/p)$ is upper-bounded by the number of integers of the form 2^k, where k is an integer, and $2^k/2r$ is contained in the range $[1/(p+1), 2/p)$. That is, we have just shown that the number of levels spanned by the range in relation 7 cannot exceed V, which is given by Lemma 3.1 below.

LEMMA 3.1. *The number of levels in the loose quadtree at which the expanded minimum quadtree cell of the object could possibly lie is upper bounded by V, where*

$$V = \log_2(M(2/p)) - \log_2(M(1/(p+1))). \tag{10}$$

Now, let us make some observations on the possible ranges of relative cell widths on the basis of relations 7 and 10. First, for the degenerate case of the MX-CIF quadtree, in which case no expansion takes place (i.e., $p = 0$), we have an unbounded upper bound on the range of values and a lower bound of 1. As p increases towards 1, the range of values decreases. For example, for $p = 1/4$, we have a range of relative cell widths $[4/5, 8)$. This means that the relative cell widths of the set of possible quadtree cells containing a given input rectangle R with a minimum bounding hypercube box of radius r lie between $[M(4/5) = 1, M(8) = 8) = \{1, 2, 4\}$. In other words, the quadtree cells containing R in the loose quadtree can be of radius $M(r)$, $2M(r)$, and $4M(r)$ (i.e., half the width). In fact, these radii hold for all values of p such that $1/4 \leq p < 1/2$.

For $p = 1/2$, there are just two possible relative cell widths corresponding to $[M(2/3) = 1, M(4) = 4) = \{1, 2\}$. In other words, the associated quadtree cells of R can be either the quadtree cell of radius $M(r)$ or of radius $2M(r)$. These radii hold for all values of p such that $1/2 \leq p < 1$. For $p = 1$, there are also just two possible relative cell widths corresponding to $[M(1/2) = 1/2, M(2) = 2) = \{1/2, 1\}$. In other words, the associated quadtree cells of R can be either the quadtree cell

of radius $M(r)$, or can be of radius half of $M(r)$. These radii hold for all values of p such that $1 \leq p < 2$. As p increases beyond 1, the number of possible ratios of relative cell widths oscillates between one and two. In particular, for $\lfloor p \rfloor = 2^k - 1$, where $k \geq 1$ is an integer, the ratio $w/2r$ takes on two values $[M(1/2^k) = 2^{-k}, M(2/(2^k - 1)) = 2^{2-k})$, while for all other values of p (i.e., $2^k \leq p < 2^{k+1} - 1$, where $k \geq 1$ is an integer), $w/2r$ takes on just one value $M(1/2^k) = 2^{-k}$.

4. INSERTION IN A LOOSE QUADTREE

In this section we show how Theorem 3.1 and Lemma 3.1 can be used to derive a simple $O(1)$ time object insertion algorithm for the loose quadtree. We first give an example of the algorithm using $p = 1/4$. From Theorem 3.1, we have that the quadtree cells containing a given input rectangle object o with a minimum bounding hypercube box of radius r can be associated with one of three possible cells of radius $M(r)$, $2M(r)$, and $4M(r)$. The insertion algorithm proceeds as follows. We first find a cell b of radius $M(r)$, such that it contains the centroid of o. This can be done in $O(1)$ time by noting that $M(r) = 2^{\lceil \log_2 r \rceil}$. At this point, we have that either b, the parent of b (say b') of radius $2M(r)$, or the parent of b' (say b") of radius $4M(r)$ contains o and we insert o in the smallest one whose expanded region contains o.

The actual insertion algorithm is given by procedure LOOSEQUADTREEINSERT below. It uses Lemma 3.1 to determine the number of quadtree cells and their corresponding sizes that are to be checked in the loop in lines 8–22 to determine the minimum-sized quadtree cell that is to contain the object to be inserted. The algorithm does not assume that the loose quadtree is represented as a tree structure with out degree 4 (8 for a loose octree in three dimensions). Instead, it assumes the use of a pointerless quadtree representation (e.g., [14, 37]) that just keeps track of the leaf nodes (i.e., cells) of the loose quadtree which are represented using, for example, a number, termed a *locational code* (referred to as the Morton Representation [28] in Section 6), that uniquely identifies each leaf node. This number can be formed by concatenating the size of the cell, say i for a cell of width 2^i, with a number j resulting from interleaving the binary representations of the coordinate values of a predefined corner such as the lower-left corner assuming that the origin of the underlying space is at the lower-left corner (e.g., (a, b) in two dimensions) so that i is at the right of j. The collection of these numbers can be represented using any access structure including binary search trees, balanced binary search trees, B-trees, etc. although our implementation in the experimental setup in Section 6 uses a B-tree. Thus the role of LOOSEQUADTREEINSERT is simply to create records for the loose quadtree which consist of the locational code and a reference to the object so that we can differentiate between objects that are associated with the same leaf node (i.e., cell) of the loose quadtree. In this case, the cell is replicated in the access structure.

```
1  pointer loose_quadtree_block procedure LOOSE-
   QUADTREEINSERT(P, O)
2  /* Given a loose quadtree with expansion factor p, create and
       return a loose quadtree record for object o which contains
       the object and its locational code. Object o is represented by
       a record of type object having the fields XCENT, YCENT,
       and MBBRADIUS corresponding to the x and y coordi-
       nate values of o's centroid, and the radius of o's minimum
       bounding hypercube box. The function M(r) returns the
       integer 2^k such that 2^(k-1) < r ≤ 2^k. The locational code
       is obtained by applying bit interleaving to the binary repre-
       sentations of x_low and y_low, the x and y coordinate values
       of the lower-left corner of the loose quadtree cell b of width
       w which contains o and concatenating it to the depth of b
       (i.e., log2(w)) and its value is a pointer to object o. If sev-
       eral objects are associated with the same cell of the loose
       quadtree, then the cell is replicated. These replicated loose
       quadtree cells are differentiated by virtue of the objects that
       are associated with them. The actual loose quadtree record
       for the cell including its locational code is constructed by
       procedure FORMBLOCK (not given here). Note the use of
       "÷" to denote integer division, "/" to denote real division,
       and ↑ to denote exponentiation. */
3  value real p
4  value object o
5  real r
6  integer i, w, x_low, y_low
7  r ← MBBRADIUS(o)
8  for i ← log2(M(1/(p + 1))) step 1
9      until log2(M(2/p)) − 1 do
10   /* Calculate width of smallest possible cell b containing o
       */
11   w ← (2 ↑ (i + 1)) * M(r)
12   /* Determine b's lower-left corner (x_low, y_low) */
13   x_low ← (XCENT(o) ÷ w) * w
14   y_low ← (YCENT(o) ÷ w) * w
15   /* Determine if b's expanded region contains o */
16   if x_low − p * w/2 ≤ XCENT(o) − r and
17      XCENT(o) + r ≤ x_low + (1 + p/2) * w and
18      y_low − p * w/2 ≤ YCENT(o) − r and
19      YCENT(o) + r ≤ y_low + (1 + p/2) * w
20      then exit_for_loop
21   endif
22 enddo
23 return(FORMBLOCK(x_low, y_low, w, o))
```

5. DISCUSSION

The calculations of the possible containing quadtree cell widths for $p = 1/4$, $p = 1/2$, and $p = 1$ lead to the observation that as p takes larger values (even for p as small as $1/4$), the loose quadtree treats the input objects as if they are points and it is their centroid that determines their associated quadtree cell, while their size and the value of the cell expansion factor determine the size of their associated quadtree cell. Actually, the above statement must be tempered a bit. In particular, although it implies that the position of object o is not a factor in the determination of the width w of the expanded quadtree cell c with which o's MBH(o,b) is associated, this is not quite true as the existence of a range of values for the ratio $w/2r$ of the widths of c and b is a direct result of the variation in the position of o along with that of the value of p. However, as we showed above, for values of $p \geq 1/2$, the values of the ratio of the widths of c and b take on at most two values which differ by one where, in the case of $p \geq 1$, the only reason for the two possible ratio values is the fact that at times p takes on a value which is one less than a power of 2.

At this point, it is appropriate to ask what value p should one use. The answer must bear in mind that as p gets large, the radii (i.e., half the width) of the associated expanded quadtree cells get larger and thus they overlap adjacent quadtree cells of half the radius for $p = 1$ and of equal radius for $p = 2$, and even greater radii as p increases further. On the other hand, as p approaches 0, the radii of the quadtree cells associated with object o are increasingly dependent on the position of the centroid of object o and can get disproportionately large independent of the radius of o's minimum bounding hypercube box. The cardinality of the set of possible values of these radii is minimized at 1 when $p \geq 2$ with the exception of $p = 2^k - 1$ for integer values of k in which case the cardinality of the set is 2 corresponding to radius values 2^i and 2^{i+1} for

some integer i. Clearly, there is no point in letting p get larger than 2 in which case the radius of the associated quadtree cell is pre-determined and depends solely on the value of the radius of o's minimum bounding hypercube box.

Thus we remain with the range $1/2 \leq p < 2$ for which the cardinality of the set of possible values of the radii of the quadtree cells is 2 corresponding to radius values 2^i and 2^{i+1} for some integer i. Our rationale for choosing p in this range is that the expanded quadtree cells are not so large as is the case for $p = 2$ and hence the extent of the overlap with adjacent quadtree cells is reduced, while the burden of having two possible radii for the quadtree cells is not great. Of course, procedure LOOSEQUADTREEINSERT in Section 4 is not as simple for $p = 1$ as it is for $p = 2$, in which case there is no need for the loop in lines 8–22. Nevertheless, for $p = 1$, the loop in lines 8–22 need only be executed twice, which is still quite simple. Ulrich [47] lets $p = 1$, while results of our experiments described in Section 6 make a case for choosing p to be infinitesimally smaller than 1. It is important to observe that all of the results that we have described hold for loose quadtrees of arbitrary dimension (e.g., three dimensions such as the loose octree) as they are all formulated in terms of the radii of the quadtree cells.

Algorithms that make use of the loose quadtree are simplified by our observation that the centroid of object o (actually of o's MBHR(o,b,r) is always contained in the non-expanded portion of the quadtree cell c with which o is associated. However, there are scenarios where users may wish to violate this property. For example, for certain values of r and p, r may be sufficiently small so that both the centroid of o lies in the expanded portion of c and o still fits in the expanded cell c. This situation is desirable when users want to move o as much as possible without having to associate it with another quadtree cell just because o's centroid is no longer in the non-expanded region of c. Interestingly, this modification does not change the ranges of relative cell widths as the example in Figure 3(c) still corresponds to the largest value of the ratio. The difference is that now the motion of the object so that the centroid of o is also in the expanded portion of c does not result in the association of o with another cell as long as o lies entirely in the expanded portion of c. Of course, this complicates subsequent searches (as well as delete operations), as now instead of just looking for a cell whose non-expanded portion contains the centroid of o, we must examine all possible cells whose expanded cells can contain o. Notice that in essence, we have transformed the search problem from one involving points (i.e., centroids of the objects) to one involving regions (i.e., the minimum bounding hypercube boxes of the objects).

6. EXPERIMENTAL EVALUATION

Experiments were run on a Linux (2.6.18) quad 1.86 GHz Xeon server with four gigabyte of RAM. The algorithms were implemented using GNU C++. The experiments studied the behavior of loose quadtrees in an environment where the objects are in motion. Our experimental setup consisted of a large collection of rectangle objects. For most, but not all, of our experiments, we used random rectangle data obtained by generating their centroid and extents at random, which is equivalent to the method used by Ulrich [47]. Each object (i.e, rectangle) in the collection is associated with its minimum enclosing quadtree cell (actually minimum enclosing expanded quadtree cell), which is represented by its bit-interleaved Morton representation [28]. The Morton representation is indexed using a B-tree index, which is referred to as a *linear* quadtree [14, 37].

In our setup, we use a non-spatial index (e.g., array, B-tree, Hash) to index the input objects by their identifier and a spatial index (i.e., loose quadtree in our case represented as a linear quadtree) to index the current positions of the objects. As an object's position changes, we first update its current position using the non-spatial index. This operation is fairly quick as updating the position of the object requires no modification to the non-spatial index itself. Next, we must update the spatial index which poses a real computational bottleneck as even small changes in the position of the object result in an update to the index. We remedy this problem to a limited extent by representing an object by the quadtree cell with which it is associated (not necessarily containing it as the loose quadtree permits objects to be associated with smaller cells). This means that the index does not store the exact geometry of the object. However, given that we know that the ratio of the sizes of the object's minimum enclosing expanded quadtree cell and of the object is bounded by a small value which is a function of p, we are in some sense implicitly recording the geometry of the object in the index. Moreover, the Morton representation that is stored in the B-tree contains a reference to the actual object, which is stored in an array and also indexed by the non-spatial index in order to facilitate quick updates, when necessary.

In this respect, the loose quadtree is distinguished from all other spatial indices, such as an R-tree, that explicitly store the positions of objects (i.e., rectangles). This means that when the position of an object changes, in the case of an R-tree and related spatial indices, we would have to always update the indices as they depend on the minimum bounding hypercube boxes of the objects which have changed, while in the case of the loose quadtrees, we only need to update the index if if the object is associated with a different quadtree cell. This property of the loose quadtree makes it attractive for serving as a spatial index for moving object applications. In contrast, as we pointed out, updates in spatial indices such as the R-tree, as well as other related spatiotemporal indices, will often require a complete rebuild step when the position of the object changes, which is quite complicated. Nevertheless, for the sake of completeness, we provide a comparison of comparison of loose quadtree with a suitable implementation of a R-tree in Section 7.

We ran a number of experiments to test the sensitivity of the loose quadtree to the motion of the objects that it stores. We used a collection of one million randomly generated rectangles in a two-dimensional space, which were stored in a B-tree based loose quadtree index. Our implementation of the B-tree is single threaded with a node size of 8 kb that can store up to 64 objects per node. Furthermore, we cache 10% of the nodes in an in-memory cache. The non-spatial index is an in-memory array indexed by the object identifier. For this set of experiments we chose a disk-based data structure such as a B-tree to index the objects instead of an in-memory spatial data structure. The B-tree is better than the pointer-based quadtree in the sense that it provides access to each quadtree cell (node) in the tree using its locational code in constant time (i.e., time proportional to the height of the B-tree, which we view as a constant) whereas in the pointer-based quadtree our access time is proportional to the base 2 logarithm of the width (i.e., maximum depth) of the underlying embedding space.

We let the expansion factor p vary between 0 and 5. Recall that for the case $p = 0$, the loose quadtree corresponds to an MX-CIF quadtree. We first built an index for all the objects in a loose quadtree for a given p. Next, we translated the objects in order to mimic a moving object application. If the translations resulted in an object being associated with a different quadtree cell, then we updated the index, which involves deleting an entry from the B-tree index and adding a new entry corresponding to the minimum expanded quadtree cell containing the object after the translation. We tabulated the number of objects for which the index needed to be updated. We controlled the motion of the objects using a value s, denoting the maximum translation of the object across a single dimension. For example, suppose that s is 5%, then all of the rectangles are translated across each of the dimensions by a value that is at most 5% of its side length across each of the dimensions.

In order to provide a better understanding of the effect of motion on the loose quadtree index, we distinguish between two types of

motion, namely *uniform* and *fixed* translations. In the case of a uniform translation, the motion is controlled by a random variable, which is bounded by s. In other words, all the objects are subjected to different translations, where the translation across any dimension is less than s. In the case of a fixed translation, all of the objects are translated by a fixed value (i.e., s) across each of the dimensions, which basically represents the worst case scenario (in terms of the maximum amount of motion) of any moving object application.

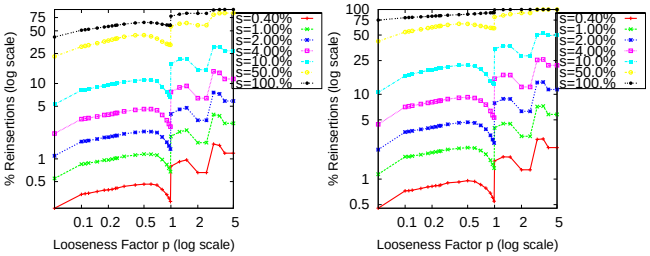


Figure 4: Reinsertion rates for two-dimensional rectangle input for varying values of p and s with δ = 10, for a) uniform and b) fixed translations.

Finally, we also varied the sizes of the input rectangle objects by using the value δ, which denotes the ratio of the largest side length of a rectangle object in the data set to the smallest side length of a rectangle object in the data set. It should be clear that a large value of δ means a large range of rectangle object sizes, while a small value of δ means that the rectangle objects are more or less of the same size. All of the experiments whose results we present varied one or more of these variables to showcase the utility of loose quadtrees for moving object applications. Note that this is a far more extensive experimental evaluation than the one conducted by Ulrich [47] who only studied the loose quadtree's behavior for culling operations and only compared the $p = 0$ and $p = 1$ cases.

Our first experiment considered the case of one million two-dimensional rectangle objects for values of p ranging between 0 and 5, and values of s ranging between 0.40% and 100%. The value of δ was kept constant at 10. Figure 4 shows the percentage of objects that required reinsertion as a function of p, while the different curves in the plot show the behavior of the loose quadtree index for different values of s. As expected, the percentage of objects that require reinsertion increases as s increases. From the figure, we observe that this percentage increases with p with a precipitous drop at $p = 0.999$ where the results are comparable to $p = 0$. Figure 5 provides a more vivid illustration of the comparability of the results for $p = 0.999$ with those for $p = 0$ by showing the result of normalizing the reinsertion rates for the different values of p and s vis-a-vis those for $p = 0.999$. Here we see that the reinsertion rate for $p = 0.999$ is superior to the other values of p for all reasonable values of s (i.e., less than s <50%). It is interesting to note that for all values of s, for small values of p, this percentage increases with p with a local maximum at around $p = 0.5$, at which time it has a precipitous drop at $p = 0.999$ where the results are comparable to $p = 0$, and then increases sharply for $p = 1$, and continues to increase, but at a lesser rate, as p continues to increase (i.e., p >1). This phenomenon is explained in greater detail below.

The percentage of objects requiring reinsertion is relatively low at $p = 0$ since the range of the values of the side lengths of the minimum enclosing quadtree cells is large as is also the value of the maximum side length. This means that objects often have a large area in which to move without requiring reinsertion. As p increases, we observe that the range of values of the side lengths of the minimum enclosing expanded quadtree cells become increasingly smaller, which means that the area in which the objects can move without requiring reinsertion gets smaller. Figure 6 illustrates this observation using an example object with minimum bounding hypercube box of radius r. In particular, we can see that for values of p in the range $[0.25, 0.5)$, the side length of the minimum enclosing expanded quadtree cell is either 2, 4, or 8 times the radius of the minimum bounding hypercube box of the object (first row of Figure 6) ; while for values of p in the ranges $[0.5, 1)$, the side length of the minimum enclosing expanded quadtree cell is either 2 or 4 times the radius of the minimum bounding hypercube box of the object (second row of Figure 6); and for values of p in the ranges $[1, 2)$ the side length of the minimum enclosing expanded quadtree cell is either 1 or 2 times the radius of the minimum bounding hypercube box of the object (third row of Figure 6). Notice that the percentage of objects requiring reinsertion starts to decrease at $p = 0.5$ with a minimum at $p = 0.999$. This is because at $p = 0.5$ there are only two choices for the size of the minimum enclosing expanded quadtree cell, having eliminated the cell with side length 8 times the radius of the minimum bounding hypercube box of the object. Moreover, the number of objects associated with the elim-

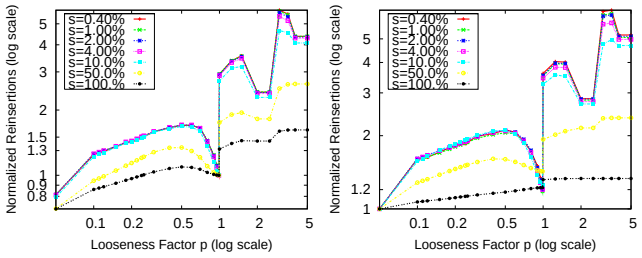


Figure 5: Reinsertions for two-dimensional rectangle input for varying values of p and s with δ = 10, normalized with the reinsertion rate for p = 0.999 for a) uniform and b) fixed translations.

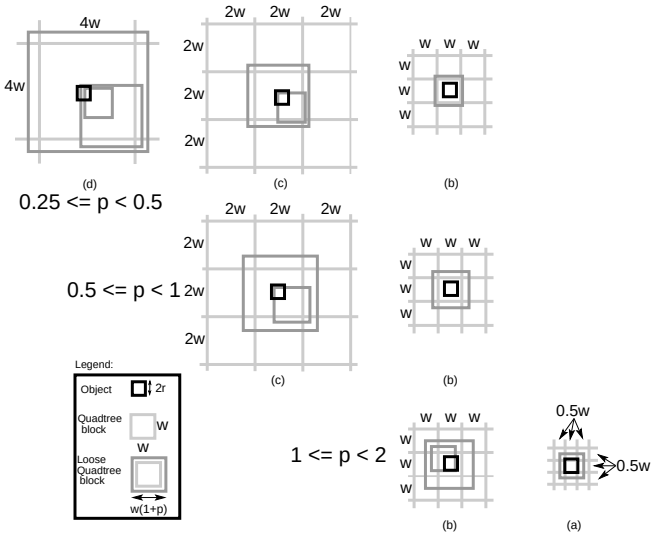
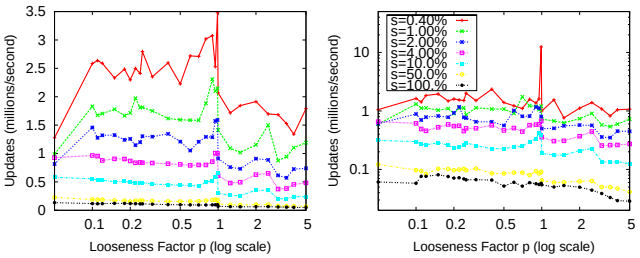


Figure 6: Illustration of the variation of the relative sizes of the minimum enclosing expanded quadtree cells and of the minimum bounding hypercube boxes of the objects with respect to different ranges of values of p: (first row) 0.25≤p<0.5, (second row) 0.5≤p<1, and (third row) 1≤p<2.

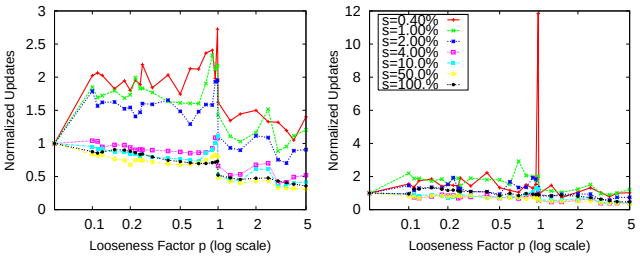
inated cell are relatively small. On the other hand, at $p = 1$, there are again only two choices for the minimum enclosing expanded quadtree cell, but now a large such cell is replaced by one with a quarter of its area thereby greatly limiting the ability of the objects to move without requiring reinsertion. The pattern of increasing percentages requiring reinsertion continues unabated for $p > 1$ as we have increasingly smaller replacement cells with saw-tooth like behavior in the neighborhood of $p = 2^k$.



(a) (b)

Figure 7: Update rates for two-dimensional rectangle objects for varying values of p and s with δ = 10, for a) uniform and b) fixed translations.

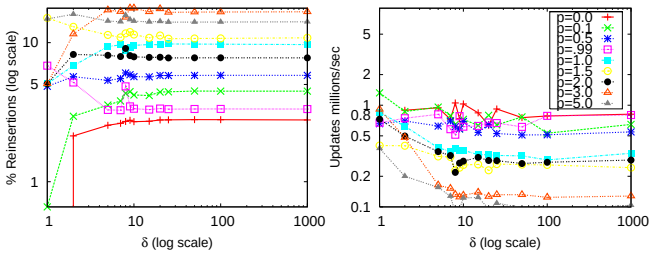
For some data structures, the update process can be sped up by batching the updates using bulk loading methods (e.g., [5, 48, 10, 17]). For example, Dittrich et al. [10] assume that updates can come at fast rates so it may not be good to handle one update at a time. Instead, they use "snapshots", which are basically large static data structures and an update pool, which is a fast data structure stored in the main memory. In this case, updates are collected in the update pool, and once enough of them have been accumulated, they are applied en mass to the snapshot structure to create a new snapshot structure; hence a variant of bulk updates which are cheaper than single updates. In our method, all insertions and deletions are always performed in O(1) time, and thus there is no need for pooling the updates.



(a) (b)

Figure 8: Update rates for two-dimensional rectangle objects for varying values of p and s with δ = 10, normalized with the reinsertion rate for p = 0 for a) uniform and b) fixed translations.

The second set of experiments used the above environment and measured the number of updates in millions/second that can be supported by the loose quadtree data structure since this correlates with updating the spatial index with the new positions of the objects. Recall that an update in a loose quadtree involves deleting an entry from the B-tree index and inserting another entry. The key advantage of the loose quadtree structure is that small motions of objects, most likely, do not require any changes to the index, and if they do, then they are less complex as p becomes increasingly larger than 0 since the range of possible minimum enclosing expanded quadtree cells is much smaller. Figure 7 shows the number of updates in millions/second that a loose quadtree index can support under both the uniform and fixed translation cases. We see that this number decreases as s increases with performance generally peaking at $p = 0.999$, although this is most noticeable for s <50%. We also observe that the number of updates per second for a given value of s does not vary greatly across different value of p with the exception of relatively small values of s (0.40% and 1.0%), for which performance peaks noticeably at $p = 0.999$, especially with respect to $p = 0$. Figure 8 provides a more vivid illustration of this improvement by showing the result of normalizing the number of updates per second for the different values of p and s vis-a-vis those for $p = 0$, where for a fixed translation s =0.40% we see a one order of magnitude improvement. Again, these results are primarily due to the dramatically reduced cost of insertion since the new minimum enclosing expanded quadtree cell can be determined in at most 2 look up operations for $p = 0.999$ versus a significantly higher number for $p = 0$. This figure shows that the improved throughput is observed for most values of p for $s \leq$50%.



(a) (b)

Figure 9: a) Reinsertion and b) update rates in millions/second for two-dimensional rectangle input for varying values of δ and p keeping s=5% with uniform translations.

The third set of experiments examined the effect of varying δ on the performance of the loose quadtree data structure. Recall that δ bounds the ratio of the length of the largest side of a rectangle object in the input data set to the length of the smallest side of a rectangle object in the input data set. In other words, if δ is small, then all the objects in the input are of the same size. If δ is large, then the objects are of different sizes. Figure 9 measures the reinsertion and update rates in millions/second for varying values of δ and p, keeping s fixed at 5% with uniform translations. We can see that both the reinsertion and update rates are relatively independent of δ, which means that the loose quadtree data structure is suitable for handling data sets containing objects of varying sizes.

Finally, we compared the efficacy of loose quadtrees on a real world data set. We used a 2D rectangle data set generated from taking a TIGER road data set of Los Angeles County, CA and then fitting a bounding box around the line segments forming the edges in the road network. This resulted in a data set containing 267K 2D rectangles with a good mix of rectangles of different sizes. We subjected the input rectangles to both uniform and fixed translations as in prior experiments described in Figures 4 and 7. As before, we report both the percentage of the objects that had to be reinserted due to the translation, and the time that it took to perform the reinsertion (i.e., to update the data structure) which we report in terms of the number of updates per second. Figure 10a and 10c shows the percentage of updates (i.e., reinsertions) needed for uniform and fixed translations, and once again it is easy to see that, for $p = 0.999$, very few updates to the data structure are needed. Figure 10b and 10d shows the number of updates per second for uniform and fixed translations, and again it is not surprising that, for $p = 0.999$, the number of updates that can be done per second sky rockets due to the savings in the reinsertion cost.

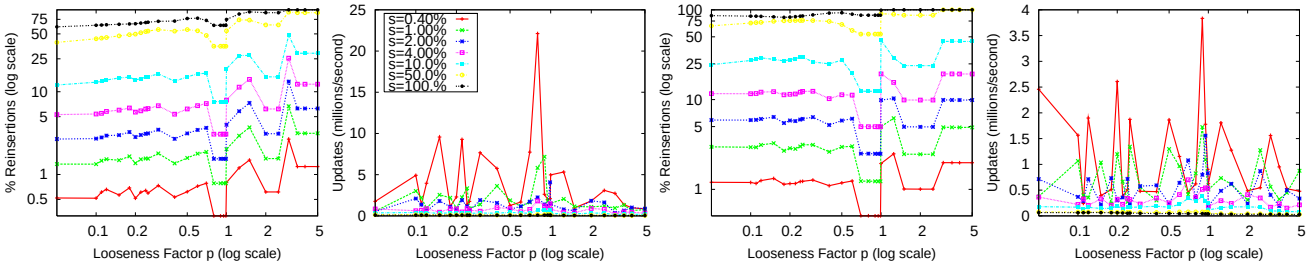


Figure 10: Reinsertions and update rates for Los Angeles County data set for varying values of p and s for a–b) uniform and c–d) fixed translations

7. N-BODY SIMULATION

We also ran a set of experiments designed to replicate common conditions for object dimension, object placement, object movement, and object frequency based on modern video games. The data structure was tested using an N-body simulation [30] which consists of insertion, deletion, update, range, and collision detection queries on the data structure that stores them. We record how much time is needed to fixed number of frames, as well as operation statistics for different numbers of objects and values of p. During each frame, objects are allowed to move, to receive force from external sources to simulate player interaction, and to collide with each other. Every object is processed in a physics engine and the objects are stored using a loose quadtree (with values of the expansion factor p ranging from 0 to 1.0). The physics engine works by processing every object in the simulation every frame. If an object is moving, then the engine performs a range query on the area occupied by the object's bounding hypercube box to return a list of hit objects. Each hit object is then entered into a physics equation and the object's velocities and positions (i.e., trajectories) are updated. At any given time, about 50% of the objects are moving over our entire 1 kilometer squared game universe.

These experiments were run on a Windows 7 enterprise quad core 2.6GHz I7 workstation with eight gigabytes of 1600MHz DDR3 RAM and an Nvidia GT650m discrete GPU. The simulation and related code were compiled using Microsoft Visual Studio 2010's integrated 32 bit compiler. The experiments compared the MX-CIF quadtree ($p = 0$) with variants of the loose quadtree for a few values of p. A pointer-based quadtree implementation was used here in order to simplify the algorithms as the main focus was on evaluating the efficacy of the loose quadtree data structure for the moving object applications such as, but not limited to, video games. In addition, the pointer-based environment enables all execution to occur in main memory which is the environment used in video games.

The rest of this section is organized as follows. Section 7.1 provides more details on the update (i.e., insertion) costs in comparison to the total cost of the N-body simulation. Section 7.2 examines the costs of the different queries involved in the N-body simulation as well as the scalability of the loose quadtree. In order to provide an additional reference point on our performance measures, Section 7.3 reports on a limited comparison with an R-tree data structure in terms of insertion costs and times for a window query as part of the N-body simulation. Extensive modifications were needed to make the R-tree work in this setting due to its inherently poor insertion behavior as outlined in the prologue of this section.

7.1 Insertion and N-Body Simulation Costs

The N-body simulation consists of two-dimensional objects that interact with one another. Each object exerts a force on all the other objects in the scene and consequently each object moves due to the resulting force that is applied on it. In such a simulation, computing

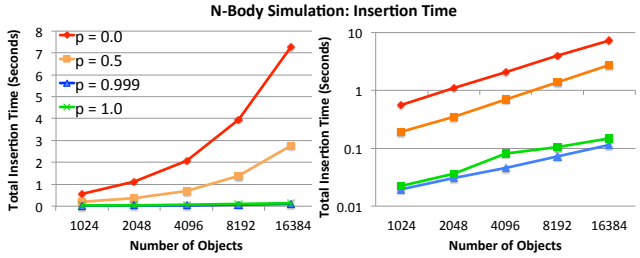


Figure 11: CPU execution time for insertions in the N-body simulation using a normal scale (left) and a log scale (right).

the force interaction between the objects is the single most time-consuming operation. Periodically, a large radial force is exerted on the system. These explosions are designed to simulate common game play cases where movement of many objects occurs in only a subset of the game universe. However, since all the objects move by varying amounts, updating the spatial data structure used to index the objects can be a significant bottleneck. We use this setup to test the update performance of the MX-CIF quadtree and three variants of the loose quadtree having p values of 0.5, 0.999 and 1.0.

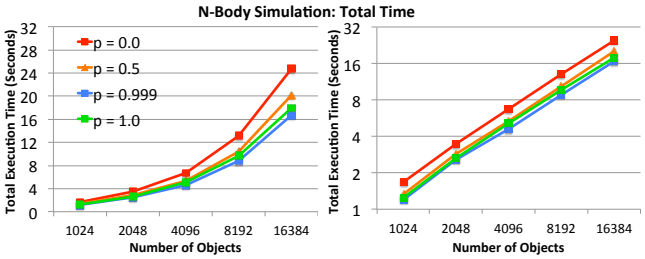


Figure 12: CPU execution time for insertions in the N-body simulation using a normal scale (left) and a log scale (right).

The total execution time is recorded which corresponds to the number of seconds required to perform the simulation for a fixed number of time steps. We record the total time in seconds spent performing insertion operations into the loose quadtree during execution of the N-body simulation. We also record the total number of insertion operations needed (i.e., the number of objects that need to be reinserted after a motion). Figure 11 shows amount of time spent doing insertion using both a normal scale (left of Figure 11) and a log scale (right of Figure 11), while Figure 12 shows the total amount of time spent in the N-body simulation using both a normal scale (left of Figure 12) and a log scale (right of Figure 12). Note

the similar behavior for the four variants with $p = 0.999$ performing the best. In fact, there is a one half order of magnitude difference in the insertion and total N-body simulation execution times between $p = 0$ and $p = 0.999$. It is interesting to note that the difference between the total execution times of the N-body simulation with the use of $p = 0.999$ versus with the use of $p = 0$ is accounted for by the savings in the insertion costs (approximately 7 seconds). Figure 13 shows the relatively large reduction in the number of objects that need reinsertion when using $p = 0.999$ vis-a-vis $p = 0$. It is also interesting to note that all of the log plots show a linear relationship which means that all of the statistics that we collected obey a power law of $y = ax^b$ where a and b are constants with b varying between 1.60 and 1.75. It is clear from this data that the loose quadtree outperforms the MX-CIF quadtree ($p = 0$), and that the value of $p = 0.999$ leads to better performance than $p = 1$. In fact, we see almost two orders of magnitude less insertions for $p = 0.999$ vis-a-vis $p = 0$. In the case of $p = 1.0$ and $p = 0.999$ we see reductions as high as 30% in both the total number of insertions (Figure 13) and time spent in insertions (Figure 11), and as high as 10% in the total execution time of the N-body simulation, which is less as expected since we are looking at the total execution time which involves more operations than insertions, but is still very good.

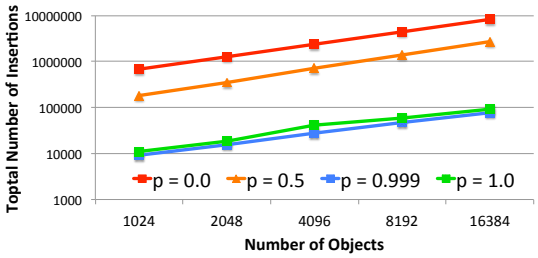


Figure 13: Number of insertions in the N-body simulation using a log scale.

7.2 Query Performance and Scalability

We now examine the performance of the loose quadtree for the queries that form the key geometric operations in an N-body simulation. Our motivation is to show that the good update performance of the loose quadtree (i.e., minimization of reinsertions upon object motion) does not come at the expense of query performance, and that it scales. In order to do this, our first set of experiments measured the time taken to perform a number of different queries as well as recorded the number of blocks and objects in the loose quadtree that are visited by each query. Our second set of experiments examines the performance of the loose quadtree on simulations with varying sizes of objects. In particular, we show that the loose quadtrees can scale to handle really large simulations way beyond the limitations of the MX-CIF quadtree (i.e., a loose quadtree with $p = 0$).

In order to test the query performance of the data structures, we use a different environment than the one in Section 7.1. In particular, we recreate a game environment to simulate a complex 3D scene. In this scenario, the objects do not exert force on one another but move due to gravity. Hence, the bottleneck in this environment is not computing the inter-object forces but rather capturing the interactions between the objects as they collide with one another. Capturing these interactions between the objects requires knowledge of which objects are in close proximity to which other objects in the scene. All objects in the environment are updated dynamically using a physics simulation solver. Furthermore, we also render the scene which requires the application of geometrical operations such as ray tracing, view frustum culling and window queries on the spatial data structures.

We note that the time taken to compute each time step in the simulation is roughly broken down as follows: About 70% of the time is spent on updating the data structure, 20% of the time is spent on performing geometric operations such as frustum culling, ray tracing, window and nearest neighbor search, while the remaining 10% of the time is spent on rendering the scene. Note that rendering is GPU assisted, so that the 10% time only corresponds to the time needed to transfer the objects to the GPU, which makes it relatively independent of the complexity of the scene. Moreover, the time taken to render the scene is also common to all the data structures that we examine in this section.

Below, we examine the time needed to execute a number of queries as part of an N-body simulation with 14,000 dynamic objects. Figure 14a shows the average times in nanoseconds needed to perform each of the queries, while Figure 14b shows the average number of blocks and objects visited while processing the queries. RAY corresponds to ray tracing which is used to find the objects that intersect a single ray (i.e., line) in the scene. FRUS refers to the frustum culling operation which identifies all the objects that are inside the scene being rendered. Geometrically speaking, this operation captures the intersection of a hyperplane with the objects in the simulation. NN refers to a nearest neighbor search that given an object o, find another object that is closest to o. In an N-body simulation, NN is used for collision detection purposes. Finally, WIN-L and WIN-S capture window (region) query operations that obtain a list of objects overlapping with a window query, where WIN-L is a large window search that covers most of the blocks in the data structure, while WIN-S applies a smaller window that only intersects a few blocks.

Figure 14a shows that $p = 0.999$ performs better than $p = 0$ for all query cases. The figure also indicates the percentage improvement resulting from use of $p = 0.999$ over the MX-CIF quadtree ($p = 0$). In particular, the loose quadtree yields an improvement of up to 43% over the MX-CIF quadtree for these common spatial queries. Furthermore, $p = 0.999$ performs better than $p = 1$ for all cases, although the difference is often too small to be discernible in the figure. It is interesting to note from Figure 14b that $p = 0.999$ and 1 both visited more blocks than $p = 0$. This is not surprising as the MX-CIF quadtree is sensitive to the position and size of the objects which results in its association of many objects with the larger-sized blocks in the quadtree thereby ultimately using fewer blocks. On the other hand, the loose quadtree, whether at $p = 0.999$ or $p = 1$, is not sensitive to the position of the objects so it distributes the objects more evenly using up more blocks. However, as can be seen in Figure 14b, the MX-CIF quadtree examines way too many objects resulting in wasted work. On the other hand, the loose quadtree for our values of p (i.e., 0.999 and 1). only examines blocks that have the potential to contain relevant objects but it needs to look at more of them. Even though these two forces negate each other, they still result in the loose quadtree outperforming the MX-CIF quadtree for our values of p.

Notwithstanding the query performance of the loose quadtree, it is important to note that the act of updating objects constitutes the majority of the time of an N-body simulation, which is where the loose quadtree really excels. Figure 15 shows the update cost incurred per frame due to the moving objects in the scene as well as the time needed in milliseconds to render a frame using the loose quadtree with $p = 0.999$ and MX-CIF quadtree for varying sizes of the simulation. The performance of $p = 1$ is slightly worse than $p = 0.999$ but better than the MX-CIF quadtree and is not shown in the Figure. From the figure we see that the loose quadtree can render frames 2–5 times faster than the MX-CIF quadtree. In particular, even at 100,000 objects, the loose quadtree could still sustain about 5 frames per second (210 milliseconds per frame) in contrast to the MX-CIF quadtree which at this size could not even render

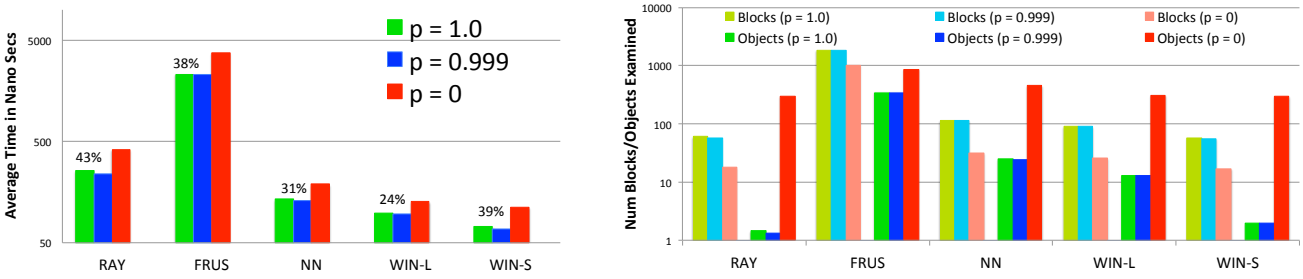


Figure 14: (a) Query times, using a log scale, for operations using the loose quadtree for values of $p = 0$ (the MX-CIF quadtree), $p = 0.999$, and $p = 1$ where $p = 0.999$ performs best as indicated with the percent improvement over the MX-CIF quadtree. (b) The average number of blocks and objects visited by the query

one frame per second (to be precise it took 1.4 seconds per frame). To put the size of the problem in a proper perspective, we note that current commercial games available in the market as of 2012 typically do not involve more 15,000 thousand moving objects [8]. In this sense, we have shown that using loose quadtree makes it possible to scale significantly these simulations vis-a-vis the state of the art. Furthermore, from Figure 15 we see that the time spent needed to update the underlying data structure using an MX-CIF quadtree is at least 3 times more expensive when compared with the time needed when using a loose quadtree with $p \approx 1$.

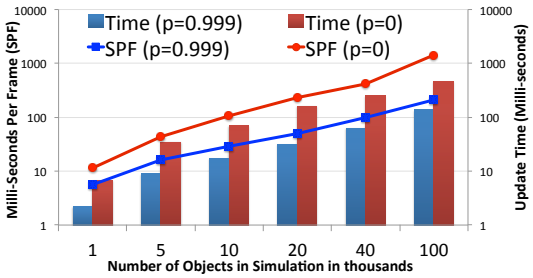


Figure 15: Figure shows the milliseconds to render a frame and the time in milliseconds spent per frame to update the data structure for varying sizes of the simulations

7.3 R-tree

We have justified the use of the loose quadtree which permits p to vary by comparing it with the MX-CIF quadtree which is a loose quadtree with $p = 0$. In order to provide an additional reference point we also present a limited comparison with the R-tree data structure, described in Section 1, which is an object hierarchy that makes use of minimum bounding hypercube boxes somewhat in the same spirit as the MX-CIF quadtree except that the hierarchies of different sets of objects are not in registration. We used the Hilbert R-tree [23] in our comparison as it is easy to construct. However, as expected, we found that updating the position of the objects in the R-tree for each individual object was prohibitively expensive, taking several orders of magnitude time more than the loose quadtree to perform the reinsertions upon execution of an update after a motion thereby making it practically unusable. Therefore, in order to improve the R-tree's performance, we batched up all the objects that required reinsertion and then reinserted them as a single batch rather than one at a time. This approach was found to be faster than rebuilding the structure from scratch. For the simulation with 14,000 objects, the update time for the R-tree structure was 106.99 milliseconds per frame of the simulation. In comparison, the update time for $p = 0.999$ was 22.11 milliseconds, while for the MX-CIF quadtree (i.e., $p = 0$), it was 82.26 milliseconds. So, our modified R-tree structure was still 25% worse than the MX-CIF quadtree but significantly worse (i.e., 5 times worse) than the loose quadtree with $p = 0.999$.

Next, we compared the query performance of the R-tree for the two window queries, WIN-L and WIN-S. Using the R-tree, the average time to execute them was 167 nanoseconds for WIN-L and 142 nanoseconds for WIN-S. In contrast, the query times of WIN-L and WIN-S were 128 and 113 nanoseconds, respectively, for the MX-CIF quadtree and 98 and 73 nanoseconds, respectively, for $p = 0.999$. These results show that the loose quadtree is superior even when compared against an appropriately modified R-tree variant resulting in the loose quadtree being almost twice as fast.

8. CONCLUDING REMARKS

We have shown how to determine for a loose quadtree the maximum possible width w of the minimum enclosing expanded quadtree cell c for an object o with MBHR(o,b,r) and cell expansion factor p. We have also shown that w is independent of the position of o. This property enables determining the cell with which o is associated and can be used, for example, in an algorithm to build a loose quadtree in an environment that deploys a pointerless quadtree. In particular, this independence means that the algorithm requires little or no search and could be used, for example, to populate a spatial database with the latest wave of multiprocessors such as those that make use of GPUs (e.g., [27]). It could also be used in point cloud applications (e.g., [39]). Our experiments (including the N-body simulation which did not rely on a pointerless quadtree) have demonstrated that letting p take on a value infinitesimally smaller than 1 leads to the best results in minimizing the need to update the index when objects are moving thereby increasing the size of the collection (i.e., database) of objects that can be supported. This is in contrast to the conventional use of $p = 1$ [47].

9. REFERENCES

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *CVGIP*, 24(1):1–13, Oct. 1983.

[2] D. J. Abel and J. L. Smith. A data structure and query algorithm for a database of areal entities. *Australian Computer Journal*, 16(4):147–154, Nov. 1984.

[3] A. Aboulnaga and J. F. Naughton. Accurate estimation of the cost of spatial selections. In *ICDE*, pp. 123–134, San Diego, CA, Feb. 2000.

[4] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pp. 265–272, Nashville, TN, Apr. 1990.

[5] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *ALENEX*, LNCS 1619, pp. 328–348, Baltimore, MD, Jan. 1999.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pp. 322–331, Atlantic City, NJ, June 1990.

[7] T. M. Chan. Approximate nearest neighbor queries revisited. In *SCG*, pp. 352–358, Nice, France, June 1997.

[8] D. Collin. Culling the battlefield data oriented design in practice. *Game Developers Conference*, San Francisco, CA, Feb. 2011.

[9] A. J. Demers, J. Gehrke, C. Koch, B. Sowell, and W. M. White. Database research in computer games. In *SIGMOD*, pp. 1011–1014, Providence, RI, June 2009.

[10] J. Dittrich, L. Blunschi, and M. A. Vaz Salles. Indexing moving objects using short-lived throwaway indexes. *SSTD*, LNCS 5644, pp. 189–207, Aalborg, Denmark, July 2009.

[11] C. Esperança and H. Samet. Experience with SAND/Tcl: a scripting tool for spatial databases. *JVLC*, 13(2):229–255, Apr. 2002.

[12] A. U. Frank. Problems of realizing LIS: storage methods for space related data: the fieldtree. TR 71, ETH, Zurich, Switzerland, June 1983.

[13] A. U. Frank and R. Barrera. The Fieldtree: a data structure for geographic information systems. In *SSD*, LNCS 409, pp. 29–44, Santa Barbara, CA, July 1989.

[14] I. Gargantini. An effective way to represent quadtrees. *CACM*, 25(12):905–910, Dec. 1982.

[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pp. 47–57, Boston, June 1984.

[16] E. Haines. Loose octrees for dynamic raytracing. The ray tracing news, Aug. 2001.

[17] G. R. Hjaltason and H. Samet. Speeding up construction of PMR quadtree-based spatial indexes. *VLDBJ*, 11(2): 109–137, Oct. 2002.

[18] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *VLDB*, pp. 606–618, Zurich, Switzerland, Sept. 1995.

[19] A. Hutflesz, H.-W. Six, and P. Widmayer. The R-file: an efficient access structure for proximity queries. In *ICDE*, pp. 372–379, Los Angeles, Feb. 1990.

[20] E. Jacox and H. Samet. Spatial join techniques. *TODS*, 32(1):7, Mar. 2007.

[21] E. Jacox and H. Samet. Metric space similarity joins. *TODS*, 33(2):7, June 2008.

[22] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *VLDB*, pp. 768–779, Toronto, Canada, Sept. 2004.

[23] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *VLDB*, pp. 500–509, Santiago, Chile, Sept. 1994.

[24] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *DAC*, pp. 352–357, Las Vegas, NV, June 1982.

[25] G. Kollios, D. Papadopoulos, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects using dual transformations. *VLDBJ*, 14(2):238–256, Apr. 2005.

[26] S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *ICDE*, pp. 615–622, Heidelberg, Germany, Apr. 2001.

[27] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, pp. 1111–1120, Cancun, Mexico, Apr. 2008.

[28] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. TR, IBM Ltd., Ottawa, Canada, 1966.

[29] T. Nguyen, Z. He, R. Zhang, and P. Ward. Boosting moving object indexing through velocity partitioning. *VLDB*, 5(9):860–871, 2012.

[30] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with CUDA. In *GPU Gems 3*, pp. 677–695. Addison-Wesley, Upper Saddle River, NJ, 2007.

[31] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: an efficient index for predicted trajectories. In *SIGMOD*, pp. 635–646, Paris, France, June 2004.

[32] D. Pfoser and C. S. Jensen. Trajectory indexing using movement constraints. *GeoInformatica*, 9(2):93–115, 2005.

[33] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López. Indexing the positions of continuously moving objects. In *SIGMOD*, pp. 331–342, Dallas, TX, May 2000.

[34] H. Samet. A quadtree medial axis transform. *CACM*, 26(9):680–693, Sept. 1983.

[35] H. Samet. Reconstruction of quadtrees from quadtree medial axis transforms. *CVGIP*, 29(3):311–328, Mar. 1985.

[36] H. Samet. Depth-first k-nearest neighbor finding using the MaxNearestDist estimator. In *IAPR*, pp. 486–491, Mantova, Italy, Sept. 2003.

[37] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[38] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *PR*, 17(6):647–656, November/December 1984.

[39] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *C&G*, 31(2):157–174, Apr. 2007.

[40] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *VLDB*, pp. 16–27, Mumbai (Bombay), India, Sept. 1996.

[41] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, April–June 1990.

[42] D. Šidlauskas, K. A. Ross, C. S. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *Adv. in STD*, LNCS 6849, pp. 186–204. 2011.

[43] D. Šidlauskas, S. Šaltenis, and C. S. Jensen. Parallel main memory indexing for moving-object query and update workloads. In *SIGMOD*, pp. 37–48, Scottsdale, AZ, June 2012.

[44] H.-W. Six and P. Widmayer. Spatial searching in geometric databases. In *ICDE*, pp. 496–503, Los Angeles, Feb. 1988.

[45] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *ICDE*, pp. 254–255, Tokyo, Japan, Apr. 2005.

[46] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *VLDB*, pp. 790–801, Berlin, Germany, Sept. 2003.

[47] T. Ulrich. Loose octrees. In *Game Programming Gems*, pp. 444–453. Charles River Media, Rockland, MA, 2000.

[48] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, pp. 406–415, Athens, Greece, Aug. 1997.

[49] M. L. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDBJ*, 17(3):379–400, Sept. 2008.