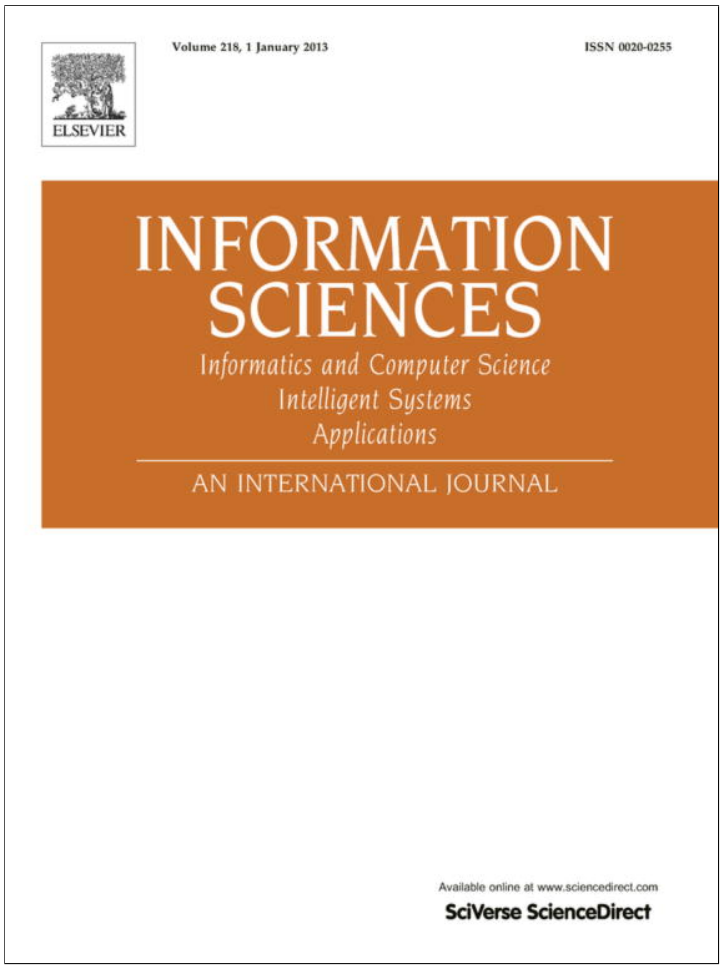

Information Sciences 218 (2013) 133–145

Contents lists available at SciVerse ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Dynamic point-region quadtrees for particle simulations

Oğuzcan Oğuz [1], Funda Durupınar, Uğur Güdükbay *

Department of Computer Engineering, Bilkent University, Bilkent, 06800 Ankara, Turkey

ARTICLE INFO

Article history:
Received 15 October 2010
Received in revised form 13 January 2012
Accepted 23 June 2012
Available online 2 July 2012

Keywords:
Quadtree
Point-region quadtree
Octree
Particle simulations
Crowd simulation
Continuum dynamics

ABSTRACT

We propose an algorithm for dynamically updating point-region (PR) quadtrees. Our algorithm is optimized for simultaneous update of data points comprising a quadtree. The intended application area focuses on simulating continuum phenomena, such as crowds, fluids, and smoke. We minimize the number of tree updates by making use of small changes in the positions of data points. We compare the efficiency of the proposed algorithm with two other approaches for updating a quadtree. One of these techniques creates the tree from scratch at each time-step. The second technique subsequently deletes a data point from the tree and reinserts it in its updated position. We achieve significant performance gains with our method in both cases.

1. Introduction

Simulating continuum phenomena has been widely studied by computer graphics researchers. Any phenomenon that can be modeled as a flow, such as fluid, smoke, and crowds, is a challenging, yet inspiring area for the computer graphics society. One of the major simulation techniques is grid-based (Eulerian) simulation method. In grid-based simulation methods, the simulated phenomena are made up of particles. Simulation is performed by computing the flow fields on the whole simulation space and, for each step, updating the particle positions according to the flow rate where the particles belong. In order to compute flow fields, the simulation state (particle positions, velocities, mass) needs to be represented throughout the simulation space. In order to do this, the simulation space has to be discretized. Hence, numerous ways to store the flow dynamics data tailored for specific purposes have been developed. The most intuitive way is to use a uniform grid (not necessarily orthogonal) placed on top of the simulation space. In the case of orthogonal uniform grids, state data can be stored on cell centers, on the corners of cells and on the edges, such as in the MAC-style grid arrangement [8]. Uniform grids have many advantages including their ease of implementation and fast access times. Furthermore, as mounting a uniform grid is the most natural and intuitive way of discretizing a given space, most of the simulation techniques are designed to work on uniform grids, especially on orthogonal ones. On the other hand, uniform grids have fixed resolution everywhere; thus, the resolution cannot adapt to the desired simulation accuracy in different regions of the simulation space. Different resolution requirements can be determined by the density of particles, flow dynamics and static structures such as obstacles within a region.

Other spatial data structures can be used as alternatives to uniform grids, such as hierarchical data structures including quadtrees and octrees. As a matter of interest, to represent the 2D simulation space, quadtrees offer adaptive space decomposition. Every cell in a quadtree is square shaped and, can be decomposed into its four quadrants if necessary. This regular decomposition has the advantage of easy implementation. A sample quadtree containing a number of particles can be seen in Fig. 1. The positions of the particles in Fig. 1a are updated in Fig. 1b and the quadtree is restructured accordingly.

* Corresponding author. Tel.: +90 312 290 1386; fax: +90 312 266 4047.
E-mail addresses: o.oguz@utwente.nl (O. Oğuz), fundad@cs.bilkent.edu.tr (F. Durupınar), gudukbay@cs.bilkent.edu.tr (U. Güdükbay).
[1] Present address: Faculty EEMCS, Department of Electrical Engineering, Control Engineering, University of Twente, 7500 AE Enschede, The Netherlands.

0020-0255/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2012.06.028

We propose an algorithm for dynamically updating point-region (PR) quadtrees [14]. Point-region quadtrees offer adaptive discretization of the space according to the density of data points. For a PR quadtree discretization, simulation state (positions, velocities, mass) can be stored in the leaf nodes of the quadtree. Leaf nodes cover the entire simulation space with varying degrees of resolution. Data points stored in the quadtree are all dynamic and their positions are updated according to their velocities at each time-step. Thus, the update algorithm is optimized complying with the simultaneous update of points. The proposed approach could be used in grid-based (Eulerian) particle simulations, such as fluid, smoke and crowd simulations. Compared to uniform grids, adaptive refinement of the simulation space would provide not only better accuracy in dense regions but also higher performance gain if there are sparse regions, which is crucial for systems with thousands of particles. In addition, our method can easily be generalized to octrees. In that case, using uniform grids is not feasible in terms of space cost. Worst-case space bounds are $O(n^3)$ for 3D uniform grids and $O(n^2)$ for 2D uniform grids. However, worst case space analysis of PR quadtrees yields $O(n)$ upper bound, where n is the number of data points [13].

The organization of the paper is as follows: Section 2 discusses related work based on different data structures for discretizing the simulation space. Section 3 introduces the proposed quadtree update algorithm. Section 4 explains the experiments, discusses their implications and presents theoretical analysis of the algorithm. Finally, Section 5 gives conclusions.

2. Related work

Representation of spatial data and spatial subdivision techniques have been widely studied as a result of their relevance to areas such as image processing [19,22], computer graphics [15], databases [2,6,11], and computational geometry [3]. The data structures related to spatial data representation include quadtrees, octrees, and bounding volume hierarchies [14,16,17]. These structures provide a way to index the space into discrete sections. Since the related methods are compact and they depend on the nature of the data, they provide space and time efficiency. Furthermore, they facilitate operations such as search and update.

The most intuitive way to represent spatial data is to use a uniform grid. However, uniform grids are ideal only when data is uniformly distributed, which is rare in practice. Under many circumstances, using uniform grids results in many redundant sparse cells. Depending on the application type, huge amounts of space can be required. Adaptive data structures, on the other hand, are efficient in terms of space. However, operations such as neighbor finding, insertion, deletion, and querying, are not as efficient in terms of time cost with adaptive structures.

Quadtrees are variable resolution data structures based on regular decomposition [5]. There are many different decomposition schemes to store different data types: points, lines, regions, rectangles, surfaces, volumes and higher dimensions including time. PR quadtrees provide regular decomposition for point representations [14]. The PR quadtree is based on recursive partitioning of a bounded planar region into four equal-sized quadrants. Decomposition occurs whenever a node contains more than one point. The minimum distance between any pair of points determines the depth of a PR quadtree. In order to prevent excessive depth or because of the specific requirements of the target application, buckets with a predefined capacity can be used or a cut-off depth can be defined. Bucket capacity determines the maximum number of points that a node can contain.

Point quadtrees [14] are similar to PR quadtrees; however, instead of dividing the space regularly, they perform the division always on a data point. Since data points are stored at the upper nodes rather than the leaves, point quadtrees often have fewer nodes. However, deletion of a point can be expensive since it requires reinserting all the points into the subtree that is rooted at the deleted node [14].

Region quadtrees [14] repetitively subdivide a region until a homogeneous region is obtained or the maximum refinement level is reached. The rationale for using region quadtrees is to save execution time. Space requirements of a region quadtree double as the resolution doubles [17]. Another data structure for efficient storage and fast query times is the skip quadtree [4], which combines the best features of region quadtrees and skip lists. Skip quadtrees are built on top of

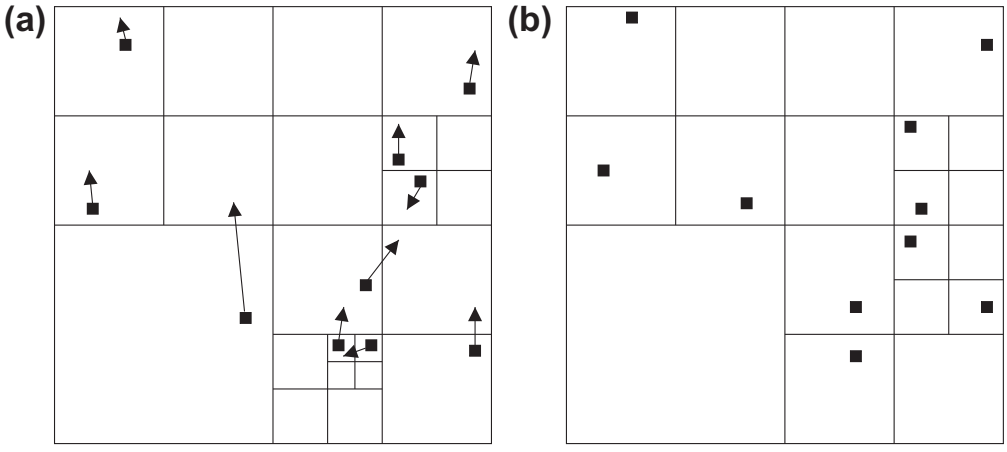

Fig. 1. A quadtree (a) before, and (b) after an update operation.

compressed quadtrees [23], which make use of the concept of interesting squares. In order to be "interesting", a square needs to be either the root of the quadtree or should contain at least two non-empty quadrants. Compressed quadtrees are hard to dynamize [4]. Skip quadtrees, on the other hand, are considered dynamic data structures since they allow fast insertion and deletion. Despite their efficiency, skip quadtrees are not suitable for simulating continuum phenomena. Since they are based on compressed quadtrees, they do not span the whole region, but only the regions that contain data points. However, simulation of continuum phenomena requires a representation of the entire simulation space. For instance, computing a potential field over a region requires information from all over the simulation space, which is not possible with the upper levels of the skip quadtree.

There is one notable study for attribute indexing in database context where a quadtree structure is dynamically updated [20]. Both this study and the skip quadtrees work on dynamic point sets. However, in our context, "dynamic" means "moving"; i.e., all the data points are updated at each iteration. In that sense, the kinetic PR quadtree described by Winder [21] has a similar context to ours as it deals with points moving over time. The project animates the PR quadtree based on the movements of data points. It uses a priority queue to sort the points and a hash table to keep version numbers for each point. These data structures are used to modify the PR quadree in chronological order. To update the node of a data point, the point is removed from the PR quadtree and reinserted using the velocity vector of the point. The quadtree is updated using the standard delete and insert methods as opposed to our update scheme. Our update scheme could be used in conjunction with kinetic PR quadtree to update the quadtree efficiently.

There are studies that use tree structures for particle and fluid simulations. One of these studies is performed by Hegeman et al. [9], who use hierarchical bounding volume (HBV) trees on GPU to accelerate nearest neighbor queries. The authors based their simulation on Smoothed Particle Hydrodynamics (SPH) model of Mueller et al. [12], which is a particle-based (Lagrangian) simulation method. In contrast to Hageman et al. [9], we aim to contribute to grid-based (Eulerian) simulation methods. In this respect, PR-quadtrees, different from HBV trees, fully discretize the simulation space and the leaf cells represent the grid cells required for grid-based simulations.

Shi and Yu [18] use incomplete octrees for smoke simulation. They also assign an internal uniform grid to every filled node for application-specific purposes. In our case, we can use incomplete quadtrees as well. The only necessary change would be on the data structure that holds the nodes (e.g., using a matrix instead of an array).

3. Method

We use a dynamic PR quadtree as the basic data structure for our method. Considering continuum simulation applications, particles define a density field. Quadtree refinement due to static structures, such as obstacles in the environment, is computed only once, whereas refinement due to dynamic particles is adjusted on the go. We focus on quadtree updates within the scope of this paper. Due to the regular updates of particle positions, coarsening and refinement operations on the quadtree occur frequently. Therefore, these operations should be optimized.

3.1. Quadtree structuring and update for moving point data

The intended application areas require simultaneous update of data points at each simulation step. Insertion of the points can happen anywhere, whereas deletion operations can only happen on the borders of the simulation space when a point moves out of the bounding space. The most frequently occurring process among moving, insertion and deletion is the move operation, so we focus on the optimization of move operations. Moving a data point can be realized by a deletion followed by an insertion. However, this would be costly due to the nature of the intended move operation. At the end of a move, if a data point is in a different cell, it would be mostly in one of the adjacent cells; a complete in-depth traversal required by the insertion operation would, most of the time, be redundant. The two extreme cases of traversal that a point needs to perform during a move operation are depicted in Fig. 2. In the figure, point 'a' needs to get up to the root node and then down to the leaf node containing it. This would be more costly than an insertion. Point 'b', on the other hand, only needs to get one level up and then one level down. All other possible moves are in between these two extreme cases with a bias towards the less costly one.

At each time-step, when a point moves to the corresponding leaf node, the quadtree structure may violate the requirements. After all the positions of the points are updated, some of the leaf nodes may have bucket overflow and/or there may be some nodes with all their children with empty buckets. For the intended simulation applications, positions and velocities of all the points are updated synchronously at each time-step. This property proposes that the restructuring of the tree should be performed not for each point one-by-one but once for the whole tree after all the points are updated. This idea is the core of our quadtree update algorithm. After a single data point is moved, if it gets to a new leaf node then the new containing leaf node may require refinement in case there are other points in the same leaf node. The old containing leaf node may need coarsening in case the node and all of its siblings are empty or containing just a single point. The overall update procedure executed at each time-step is given in Algorithm 1.

The move procedure involves a *CheckBottomUp ()* procedure to check whether a bottom-up search or a top-down search is more efficient while moving a point. *CheckBottomUp ()* procedure inspects if a moving point will visit a set of

low-level nodes while searching for its new container node. This can be done in $O(1)$ time by checking if the line segment formed by the old and new position of the point crosses any axes of the predefined set of levels. After the check is performed, the move procedure picks one of the search methods by taking the current level of the node that contains the point into account.

The pseudo-code of the refine procedure called within the step method is given in Algorithm 3. The refine procedure continuously refines the nodes until the maximum refinement level is reached or no node containing more than one point is left. The refine procedure may trigger a coarsening process if the new containing leaf node and its siblings store just a single point, or the maximum resolution allowed for the quadtree is not sufficient for differentiating two or more particles. The coarsen method (Algorithm 2) works similarly, from bottom to top this time.

The key point in these algorithms is that only the necessary nodes are restructured. A leaf node that is either a new container or an old container is updated. Any node that is not a new or old container is not affected unless one of its siblings requires an update. This ensures that each node is restructured at most once within a time-step.

In the algorithms, *GetPointCount ()* procedure returns the number of points in a given quadtree node. *SubDivide ()* procedure divides a leaf node into four child nodes, whereas *DeleteChildren ()* deletes all the four children of a given node. *InsertPoint (p)* inserts a point p into the point list of a node, and *DeletePoint (p)* removes p from the list of points of a node. Finally, *GetContainingChildIndex (p)* returns the index of the smallest child node containing point *p*.

The generalization of the proposed approach to octrees is straightforward. The step and move methods remain the same. In the coarsen and refine methods, instead of the four children of a node, we must consider the eight children of a node of the octree, simply replacing three with seven.

Algorithm 1. STEP

```
 //Move data points
foreach p ∈ Points do
    p.pos ⟵ p.pos+p.vel;
    p.MOVE();

 //Refine new containers of data points
foreach p ∈ Points do
    if p.isMoved then
        p.newContainer. REFINE();

 //Coarsen old containers of data points
foreach p ∈ Points do
    if p.isMoved and p.oldContainer.isLeaf then
        p.oldContainer.parent.COARSEN();
```

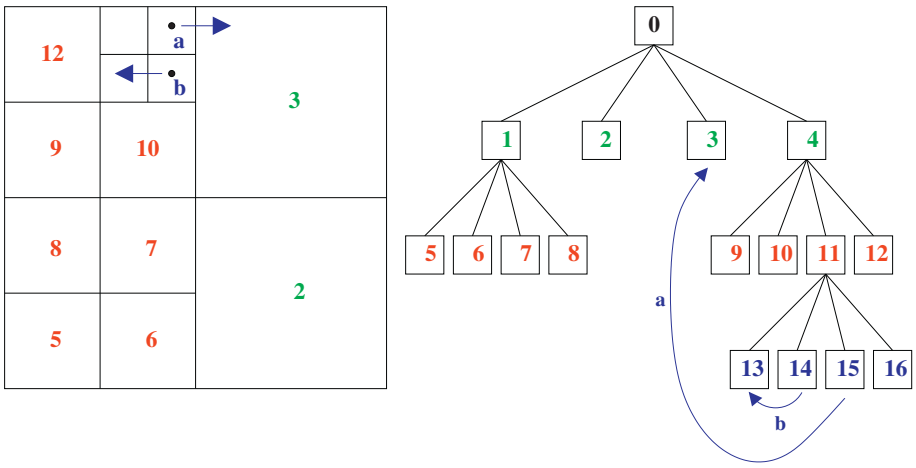

Fig. 2. Traversal of two points a and b.

Algorithm 2. COARSEN

```
if isLeaf then
    return;
if !(children[0].isLeaf and children[1].isLeaf and children[2].isLeaf and children[3].isLeaf) then
    return;
pointCnt ⟵ Σ_{i=0}^{3} children[i].GetPointCount();
if pointCnt = 1 then
    for i ⟵ 0 to 3 do
        if children[i].GetPointCount() ≠ 0 then
            childIndex ⟵ i;
    InsertPoint(children[childIndex].GetPoint());
    children[childIndex].ClearPoints();
if pointCnt = 0 then
    isLeaf ⟵ true;
    DeleteChildren();
    parent.COARSEN();
```

Algorithm 3. REFINE

```
if GetPointCount() > 1 then
    SubDivide();
    foreach p ∈ Points do
        children[GetContainingChildIndex(p)].InsertPoint(p);
        p.newContainer ⟵ children[GetContainingChildIndex(p)];
    ClearPoints();
    for i ⟵ 0 to 3 do
        children[i].REFINE();
else
    parent.COARSEN();
```

Algorithm 4. MOVE

```
input: Point p
if CheckBottomUp() then
    temp ⟵ p.oldContainer;
     //Find the smallest containing ancestor
    while !temp.Contains(p) do
        temp ⟵ temp.parent;
        if temp.level = 0 then
            break;
else
    temp ⟵ root;
 //Now delve into the leaf
while true do
    if temp.isLeaf then
        break;
    else
        temp ⟵ temp.children[temp.GetContainingChildIndex(p)];
p.newContainer ⟵ temp;
if p.newContainer ≠ p.oldContainer then
    p.oldContainer.DeletePoint(p);
    p.newContainer.InsertPoint(p);
    p.isMoved ⟵ true;
```

4. Performance experiments

4.1. Experimental analysis

In this study, we propose a technique for optimizing the update mechanism of a dynamic quadtree. We analyze the efficiency of our algorithm by comparing it with two other quadtree update techniques, which are the most intuitive and commonly used ones. We achieve performance improvement over these methods.

One of these methods is constructing the quadtree from scratch at each time-step. The construction is performed in a top-down manner. The space is divided into four quadrants and each quadrant is recursively divided until a sub-quadrant contains no more than the allowed number of agents [7]. The other method is based on consecutively deleting each point from the quadtree and then reinserting it to the corresponding node after updating its position [17]. Reinsertion starts from the root; thus, this is also a strictly top-down approach in contrast to our method, which may use bottom-up traversal.

We perform tests on particle groups initiated by uniform random distributions and Gaussian distributions, consisting of different number of points. For the target application areas, such as Eulerian simulations of fluids, smoke or crowds, single Gaussian distributions accurately represent the distribution of particles in a simulation space. We perform various tests for groups of particles initiated by single 2D Gaussian distributions with varying parameters. Covariance matrices and mean vectors of the Gaussians are randomized, keeping the two standard deviations of the Gaussians inside the simulation area. The resulting Gaussian-distributed particle groups have various elliptical shapes and are placed into various locations on the simulation space, which is discretized by the quadtree.

In our tests, the main parameter that the speed-up is investigated along is the number of particles. Simulation area for all the tests is constant. For a particular test instance, particle positions are randomly initiated in the simulation area according to the distribution and the number of particles. Then the test is performed for a number of update steps. At each update step, particles are moved with their assigned velocities; and the quadtree structure is updated by one of the three update methods. Execution times of the update operations and the number of levels traversed by the particles in the update operations are recorded.

At an update step, the amount of change in the tree structure affects the cost of the update operation. We perform various tests tuned for changing the tree structure in different amounts per update step. The main factors influencing the amount of change in the tree structure are the speeds of particles and the distances between them. We introduce a speed coefficient parameter in order to control the amount of change within a test instance. The speed coefficient is a fixed value expressing the speed of particles with respect to the average distance between two particles. The average distance is approximately computed by using the density of the particles. Tests with Gaussian distributions consider the area of the ellipses formed by two standard deviations of the Gaussians in order to compute the density of the particles, whereas tests with uniform random distributions take the whole simulation space into account.

In the experiments, velocities of the particles are given random directions. Speeds of the particles are scaled according to the speed coefficient. By controlling the speed coefficient parameter, we investigate the effect of the scaled speed of the particles on the efficiency and speed-up of our algorithm.

We pay attention to generate exactly the same test conditions while testing each of the three methods under comparison. The tests are performed on an idle computer with minimal background tasks and with scripts to minimize cache effects and obtain objective execution times.

In our tests, we compare the execution times of quadtree update operations of the three methods. We also examine how much improvement our method provides in terms of the number of levels that particles traverse during the update operations. A particle traverses levels when it is moved from a node to another node in a different level. This can happen in three cases: (1) moving the particle in the tree to find its new position; (2) refining (subdividing) nodes with more than the predefined number (bucket size) of particles; (3) coarsening (deleting the children of) nodes whose children are all leaf nodes and do not have more than the predefined number of particles in total. Thus, level traversal is the key operation that reflects the asymptotic complexity (see Section 4.2). We perform various tests with different combinations of the parameters (see Table 1). Here, we present and discuss the speed-up results of the tests for two particular values of the speed coefficient parameter. The results for other values of speed coefficient can be found in Tables 4 and 5 in Appendix A.

Figs. 3 and 4 show the results obtained in the tests with two particular speed coefficient values, 0.1 and 4, respectively. Figs. 3a and 4a depict the execution time speed-up obtained with our method for uniform random and Gaussian distributions. Figs. 3b and 4b depict the improvements obtained with our method in terms of the average number of levels that a particle traverses in an update operation.

For a relatively low speed coefficient value (0.1), Fig. 3a indicates that our method provides about twice and four times better execution time performance over the delete/insert method and the from-scratch method, respectively. The speed-up values are consistent for uniform random and Gaussian distributions.

For a relatively high speed coefficient (4), Fig. 4a depicts the execution time speed-up obtained with our method. For both of the tested distributions, our method provides improvement over the other two methods. However, there is less improvement than the simulations with the lower speed coefficient (0.1). In addition, the results are almost the same for the delete/insert method and the from-scratch method in contrast to the low speed coefficient case, where the delete/insert method outperforms the from-scratch method. Lower speed-up is due to the higher speed coefficient, as our method is optimized

Table 1
Parameters used in the experiments

Parameter	Values
Method	From Scratch, Delete/Insert, Dynamic
Distribution of particles	Uniform Random, Gaussian
Speed coefficient	0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 4, 8, 16
Particle count	100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200
Number of iterations	10 Gaussians with 10 updates each or a uniform random distribution with 50 updates each

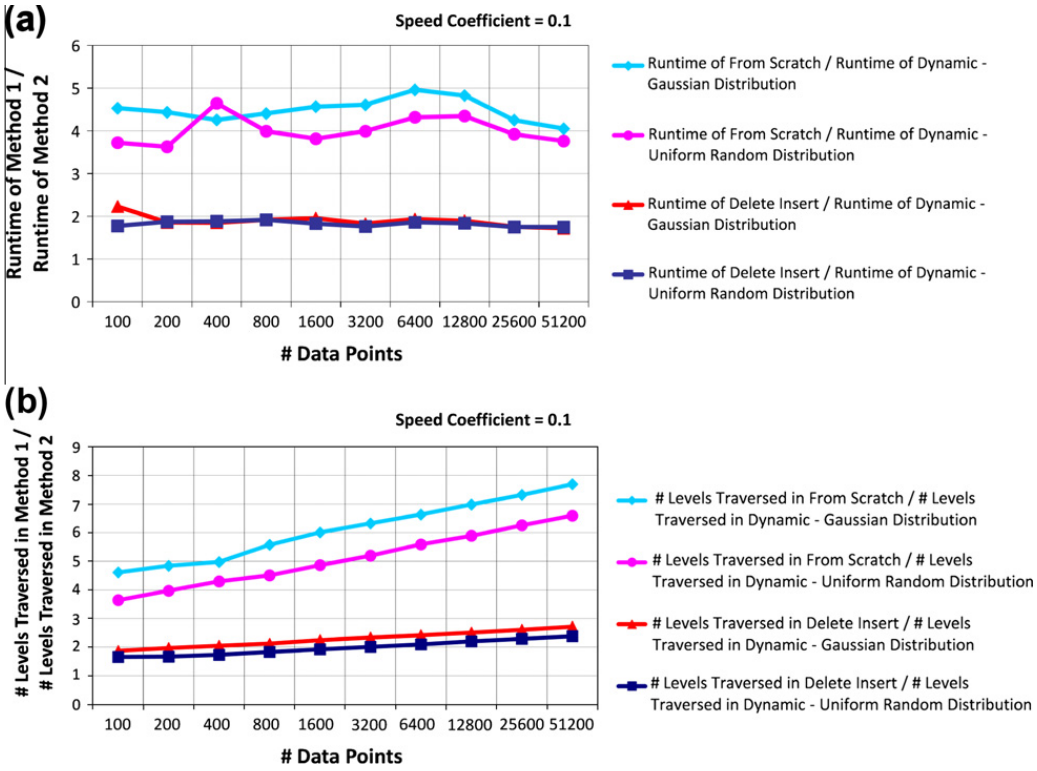

Fig. 3. Comparison of our method with other methods in terms of (a) runtime, and (b) average number of levels traversed (speed coefficient = 0.1).

for lower speed coefficients, with a slow change in the tree structure. This is a characteristic of the simulations of continuum phenomena, where the shape of the continuum body does not show abrupt changes. The reason that the delete/insert method and the from-scratch method perform similarly is that with an increasing amount of change in the tree structure, the cost of creating the tree from scratch stays unaffected, whereas the delete/insert method has to handle higher amount of change for every insert operation, causing performance reduction.

For the lower speed coefficient (0.1), Fig. 3b depicts the improvement obtained with our method in terms of the average number of levels that a particle traverses in an update operation. For both of the tested distributions, particles traverse at least twice as fewer levels with our method than the other two methods. The amount of improvement increases linearly with the number of particles and it is always higher for creating the tree from scratch. The linear increase in the depicted ratios is due to the increasing tree depth. In both the delete/insert method and the from-scratch method, particles have to go through higher number of traversals in a deeper tree. In contrast, since our method optimizes tree restructuring operations, it is affected less by the depth of the tree.

For the higher speed coefficient (4), Fig. 4b depicts the comparison of the three methods based on the average number of levels traversed by a particle in an update step. In the tests with smaller number of particles, our method performs the update operations with higher number of level traversals compared to the other methods. However, as the number of particles increases, our method performs the update operations with a slightly fewer number of level traversals except for the from-scratch method on uniform random distribution. Similar to the results of the lower speed coefficient, higher particle count results in deeper trees; thus a linear increase in the depicted ratios is observed. Compared to the delete/insert method, the from-scratch method has smaller values for the average number of traversals. This is again caused by the high speed coefficient, which results in increased amount of change in the tree structure per update step.

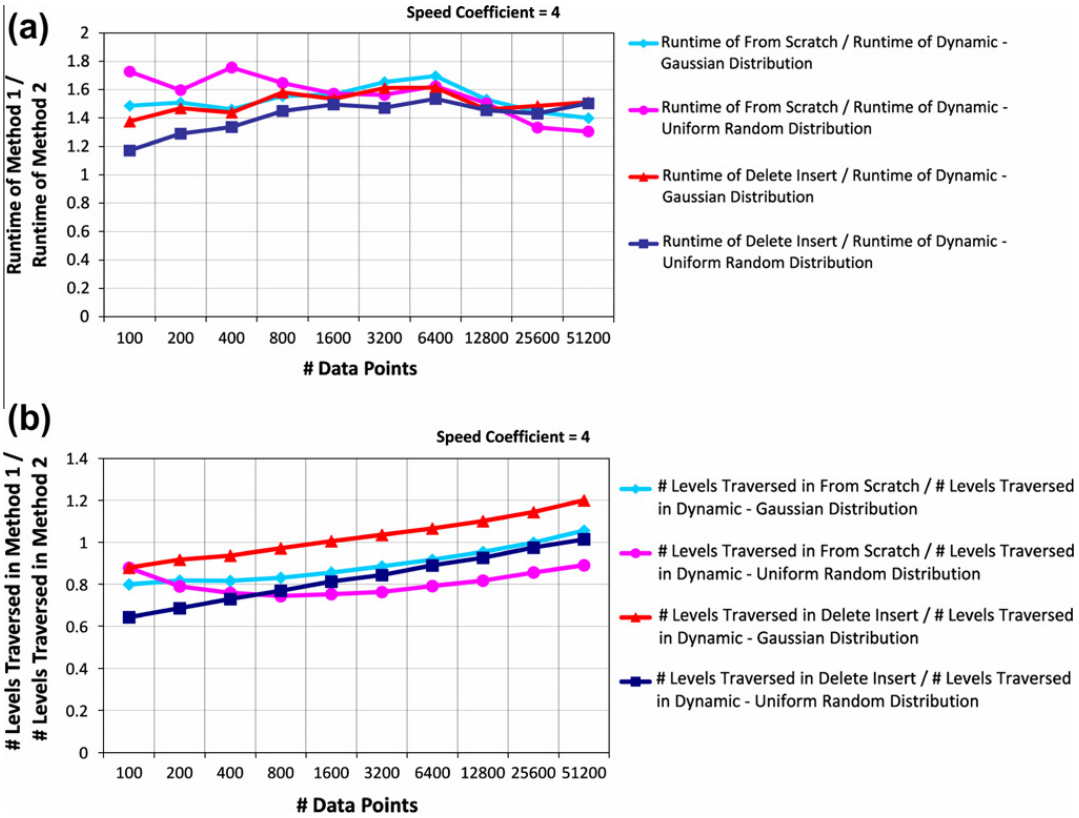

Fig. 4. Comparison of our method with other methods in terms of (a) runtime, and (b) average number of levels traversed (speed coefficient = 4).

The ratios in Fig. 3a are almost constant, whereas ratios in Fig. 3b show a linear increase with the particle count, although the initial ratio values are mostly consistent among the two figures. A similar variation can also be observed between Fig. 4a and b. We can also observe in Fig. 4a and b that although the runtime plots show positive speed-up values in all cases, the level traversal ratios are less than one in some instances. Such differences emerge since level traversal counts reflect asymptotic complexity, but they do not exactly measure the execution times. Still, level traversal plots show that the ratios of the average number of traversals measured with the delete/insert method and the from-scratch method to the values acquired with our method are larger than one or slightly smaller than one, and these ratios increase linearly with the particle count. This is a significant evidence that our update algorithm has better asymptotic complexity than the other approaches (see Section 4.2.3).

One of the reasons for better asymptotic complexity is that our method makes use of the overlap on the tree updates within an iteration. Since the tree is updated after moving each point, we can avoid redundant updates. For instance, a point p can replace another point q. In the delete/insert method, the leaf node containing q has to be coarsened first, and then refined. However, our method does not perform any coarsening or refining operations in this case. Furthermore, our method performs an adaptive search for the new containing node of the points moved; it can search the tree bottom-up or top-down.

Mean and standard deviation values for the tests with speed coefficient values 0.1 and 4 are presented in Tables 2 and 3, respectively. In order to test the statistical significance, we perform two-tailed, paired Student's t-test between the results of our method and the other methods for each number of data points and distributions. The results indicate that our findings are statistically significant (cf. Tables 6 and 7 in Appendix B). There are two cases with significance values greater than the chosen threshold, 0.05; but since the two distributions that are compared against have very close mean values for those instances, this finding does not contradict with the purpose of the significance tests.

4.2. Theoretical analysis

We theoretically analyze the performances of the quadtree update techniques compared within this study. We analyze best-case, average-case and worst-case time performance for our method, successive deletion/insertions and constructing the tree from scratch.

The complexity analysis is based on the number of levels traversed by each particle as a structural measure. Level traversal of a particle is a fundamental operation for all of the three update algorithms. Moving a particle in the tree, refining (subdividing) a leaf node and coarsening an internal node that is a direct parent of leaf nodes all involve the level traversal operation. Thus we can conclude that level traversals dominate all the operations on the tree.

4.2.1. Best-case running time performance analysis

For our method, best case is achieved when we do not need to update the tree after each move. This is possible either if the points remain within the same node or move to the siblings of their current node. Thus, they move at most one level up and down at each iteration. For instance, such a condition can be achieved if data points are distributed uniformly when the whole tree structure looks like a uniform grid. The resulting tree will be balanced and the expected depth will be $log_4(n)$. Since the tree does not need to be updated, running time of the best case will be dominated by the update of the data points. The best-case running time will be $O(n)$, where n is the number of data points.

The same conditions also hold for the delete/insert method; i.e., best case is achieved when the points remain within the same node. In that case, it is sufficient to traverse the list of points only once, which leads to $O(n)$ time complexity.

Creating the tree from scratch is different from these two methods. The construction time is $O((d+1)n)$, where d is the depth of the quadtree and n is the number of data points. [7]. In the best case, the depth of the tree will be minimum, which is achieved if the tree is balanced such as in the uniform distribution instance. Then, the construction time will be $O(nlog(n))$.

4.2.2. Worst-case running time performance analysis

The worst case for both our method and the delete/insert method occurs when each point has to move up to the root and then down to the leaf nodes. The running time is dominated by the movement of data points. The depth of the tree will be a function of the side length of the simulation space and the closest distance between any two points. Thus, worst-case running time will be $O((d+1)n)$, where d is the depth of the quadtree and n is the number of data points.The worst-case complexity for creating the tree from scratch is also $O((d+1)n)$.

The depth d of a quadtree is computed according to the height lemma as:

$$d \leqslant log(s/c) + 3/2, \tag{1}$$

where s is the side length of the initial square containing all data points, and c is the smallest distance between any two points. So, as the points get closer to each other, the upper bound of running time will increase for all the three cases.

4.2.3. Average-case running time performance analysis

Regarding the average-case complexity, the costs of all the three methods for moving points depend on the structure of the quadtree during the move action. In the literature, there have been successful efforts to analyze the node distribution of a PR quadtree related to the complexity of update methods [1,10]. The proposed methods are able to compute the expected complexity, node distribution of the quadtree and average occupancy of the nodes when the points are drawn from a known and possibly non-uniform distribution. However, in the case of moving points, using either of the three methods,

Table 2
Mean and standard deviation values of (a) runtime of each method in seconds, and (b) average number of levels traversed in each method (speed coefficient = 0.1)

Method	Number of points									
	100	200	400	800	1600	3200	6400	12800	25600	51200
Panel a:										
Dynamic-Random Uniform Mean	0.114	0.225	0.412	0.907	1.922	3.700	7.107	14.941	34.297	73.908
Dynamic-Random Uniform StdDev	0.024	0.029	0.048	0.099	0.263	0.537	1.220	0.849	0.917	1.486
Dynamic-Gaussian Mean	0.122	0.233	0.477	0.887	1.711	3.488	6.710	14.250	33.581	73.140
Dynamic-Gaussian StdDev	0.025	0.030	0.113	0.118	0.196	0.366	0.356	0.659	2.046	4.028
Delete Insert-Random Uniform Mean	0.203	0.421	0.776	1.740	3.516	6.524	13.226	27.465	60.075	129.207
Delete Insert-Random Uniform StdDev	0.058	0.067	0.099	0.223	0.378	0.412	1.110	1.080	1.652	2.691
Delete Insert-Gaussian Mean	0.273	0.433	0.883	1.702	3.351	6.375	12.983	26.996	58.850	125.890
Delete Insert-Gaussian StdDev	0.072	0.079	0.131	0.216	0.336	0.409	0.789	1.291	3.981	7.013
From Scratch-Random Uniform Mean	0.425	0.816	1.916	3.621	7.335	14.782	30.699	64.951	134.504	278.014
From Scratch-Random Uniform StdDev	0.035	0.065	0.150	0.256	0.658	0.482	0.665	1.673	1.780	3.275
From Scratch-Gaussian Mean	0.554	1.034	2.030	3.910	7.815	16.079	33.286	68.798	142.783	296.332
From Scratch-Gaussian StdDev	0.151	0.218	0.178	0.275	0.246	0.352	0.689	0.951	1.974	3.808
Panel b:										
Dynamic-Random Uniform Mean	1.165	1.190	1.215	1.274	1.284	1.299	1.295	1.314	1.317	1.326
Dynamic-Random Uniform StdDev	0.280	0.193	0.111	0.086	0.060	0.040	0.036	0.020	0.017	0.015
Dynamic-Gaussian Mean	1.272	1.316	1.397	1.309	1.318	1.328	1.338	1.329	1.344	1.350
Dynamic-Gaussian StdDev	0.305	0.176	0.161	0.145	0.112	0.090	0.079	0.056	0.105	0.083
Delete Insert-Random Uniform Mean	1.931	1.988	2.105	2.337	2.471	2.611	2.716	2.893	3.017	3.160
Delete Insert-Random Uniform StdDev	0.455	0.312	0.184	0.146	0.127	0.076	0.064	0.037	0.033	0.027
Delete Insert-Gaussian Mean	2.377	2.595	2.867	2.778	2.959	3.107	3.236	3.331	3.503	3.665
Delete Insert-Gaussian StdDev	0.498	0.375	0.346	0.270	0.256	0.215	0.203	0.145	0.293	0.247
From Scratch-Random Uniform Mean	4.243	4.725	5.219	5.743	6.243	6.745	7.234	7.739	8.238	8.739
From Scratch-Random Uniform StdDev	0.115	0.080	0.045	0.030	0.026	0.017	0.012	0.009	0.006	0.005
From Scratch-Gaussian Mean	5.863	6.371	6.947	7.297	7.918	8.399	8.872	9.275	9.833	10.380
From Scratch-Gaussian StdDev	0.182	0.165	0.204	0.120	0.208	0.148	0.160	0.114	0.195	0.208

Table 3
Mean and standard deviation values of (a) runtime of each method in seconds, and (b) average number of levels traversed in each method (speed coefficient = 4).

Method	Number of points									
	100	200	400	800	1600	3200	6400	12800	25600	51200
Panel a:										
Dynamic-Random Uniform Mean	0.249	0.564	1.141	2.369	4.737	9.572	19.121	43.331	101.257	215.649
Dynamic-Random Uniform StdDev	0.042	0.067	0.108	0.174	0.432	1.144	0.818	1.358	2.184	4.745
Dynamic-Gaussian Mean	0.321	0.647	1.331	2.447	4.882	9.610	19.530	45.442	99.722	213.634
Dynamic-Gaussian StdDev	0.039	0.070	0.115	0.232	0.804	0.401	0.644	3.126	2.387	5.043
Delete Insert-Random Uniform Mean	0.291	0.728	1.526	3.435	7.086	14.100	29.373	63.097	145.057	324.514
Delete Insert-Random Uniform StdDev	0.038	0.076	0.136	0.282	0.724	0.676	0.998	1.752	3.691	6.204
Delete Insert-Gaussian Mean	0.443	0.951	1.916	3.869	7.496	15.495	31.543	66.391	148.218	322.919
Delete Insert-Gaussian StdDev	0.056	0.144	0.170	0.566	0.365	0.562	0.836	1.228	2.823	8.152
From Scratch-Random Uniform Mean	0.430	0.901	2.005	3.903	7.452	14.974	31.077	65.228	135.108	281.564
From Scratch-Random Uniform StdDev	0.025	0.055	0.253	0.512	0.869	0.647	0.485	1.030	1.563	3.175
From Scratch-Gaussian Mean	0.478	0.977	1.945	3.807	7.644	15.896	33.133	69.538	143.891	299.224
From Scratch-Gaussian StdDev	0.037	0.115	0.142	0.442	0.272	0.507	0.658	1.189	1.590	3.544
Panel b:										
Dynamic-Random Uniform Mean	4.770	5.981	6.890	7.715	8.281	8.826	9.132	9.465	9.621	9.808
Dynamic-Random Uniform StdDev	0.484	0.261	0.207	0.158	0.093	0.070	0.048	0.025	0.022	0.017
Dynamic-Gaussian Mean	5.731	6.395	7.188	7.776	8.462	8.914	9.305	9.509	9.718	9.767
Dynamic-Gaussian StdDev	1.158	1.177	1.383	1.121	0.859	0.669	0.474	0.297	0.255	0.195
Delete Insert-Random Uniform Mean	3.069	4.106	5.031	5.938	6.734	7.451	8.132	8.768	9.378	9.946
Delete Insert-Random Uniform StdDev	0.294	0.194	0.212	0.118	0.077	0.048	0.038	0.019	0.016	0.011
Delete Insert-Gaussian Mean	5.040	5.866	6.732	7.555	8.506	9.232	9.916	10.468	11.113	11.715
Delete Insert-Gaussian StdDev	0.688	0.663	0.769	0.632	0.512	0.370	0.283	0.197	0.200	0.209
From Scratch-Random Uniform Mean	4.190	4.727	5.232	5.743	6.237	6.738	7.234	7.739	8.237	8.739
From Scratch-Random Uniform StdDev	0.100	0.071	0.061	0.040	0.023	0.019	0.014	0.009	0.007	0.005
From Scratch-Gaussian Mean	4.581	5.233	5.870	6.462	7.240	7.891	8.533	9.071	9.703	10.308
From Scratch-Gaussian StdDev	0.349	0.387	0.395	0.382	0.377	0.334	0.265	0.186	0.196	0.205

the cost of moving all the points is also highly dependent on the change in the structure of the quadtree after each move. A point can move to any level of the tree during the simulation. Thus, there can be quite different scenarios cost-wise, with exactly the same sequence of quadtree structures before and after moving the points for each time-step. So, it seems virtually impossible to formally model and analyze the complexity of our method. There could be some efforts by making assumptions about the quadtree structure and putting restrictions to moving points (on direction and distance). However, this approach is avoided since it may only be valid for a limited subset of targeted applications specific to some domains. Instead, we perform a rough analysis of the average case considering that a position update for a point can cause it to move to any depth of the tree with equal probability $\frac{1}{d}$, where d is the depth of the tree. We compute the average number of traversals for a point as:

$$\left(\frac{1}{d}\right)\times 1+\left(\frac{1}{d}\right)\times 2+\cdots+\left(\frac{1}{d}\right)\times d=\frac{(d+1)}{2}. \tag{2}$$

The average case would then be $O(dn)$. This computation is actually the same for the other two methods. The order of all techniques is the same; however, the average number of levels traversed in our method is smaller as supported by the experiments. Fig. 3b depicts comparisons of the average number of levels traversed for the delete/insert method and the from-scratch method with our method for a low speed coefficient (0.1). Similarly, Fig. 4b depicts comparisons of the average number of traversals for a higher speed coefficient (4). Both Figs. 3b and 4b show that, compared to the other two methods, our method tends to have lower number of traversals per particle in an update step with increasing particle count.

5. Conclusion

We propose an algorithm to dynamically update PR quadtrees. The proposed algorithm adaptively subdivides the simulation space depending on the motion of data points. Our technique is optimized considering the simultaneous update of data points for the intended application areas, such as grid based simulations of crowd, fluids and smoke.

We analyze the methods both theoretically and experimentally. Experiments indicate that our method results in important advantages over other obvious methods of updating a quadtree. Creating a quadtree from scratch in order to update it is the first technique that comes to mind. Another common update method is deleting a point and then reinserting it when its position changes. The experiments we conducted show that our method outperforms these techniques. Our update scheme can easily be extended to octrees as well.

Acknowledgments

This work is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. EEE-AG 104E029. We are grateful to Rana Nelson for proofreading and suggestions.

Appendix A. Performance comparison tables

This appendix provides comparisons of our method with the other two approaches (the insert/delete and from-scratch methods) in terms of runtime (Table 4) and the number of levels traversed (Table 5) for different parameter values.

Table 4
Comparison of our method with other methods in terms of runtimes (in seconds). The first column gives speed coefficient values. The parameters in the second row correspond to *Distribution* (G: *Gaussian Distribution*, UR: *Uniform Random Distribution*). The parameters in the third row correspond to *Method* (Dy: *Dynamic Method*, DI: *Delete/Insert Method*, FS: *From-Scratch Method*).

Parameters			Number of points									
Speed coeff.	Distr.	$Method_1/Method_2$	100	200	400	800	1600	3200	6400	12800	25600	51200
0.01	UR	DI/Dy	1.668	1.721	1.537	1.522	1.693	1.375	1.359	1.546	1.412	1.382
0.01	UR	FS/Dy	12.523	12.881	13.385	14.335	13.554	12.870	13.066	15.485	13.327	12.954
0.01	G	DI/Dy	1.345	1.500	1.493	1.410	1.391	1.426	1.459	1.493	1.432	1.409
0.01	G	FS/Dy	13.306	13.616	14.343	11.928	13.469	14.893	15.638	15.696	14.782	13.954
0.02	UR	DI/Dy	1.719	1.549	1.606	1.694	1.653	1.551	1.438	1.552	1.570	1.555
0.02	UR	FS/Dy	8.792	10.218	10.469	9.738	9.120	8.902	9.418	10.686	10.271	9.848
0.02	G	DI/Dy	1.710	1.709	1.738	1.620	1.607	1.610	1.651	1.624	1.576	1.537
0.02	G	FS/Dy	10.082	10.743	10.841	10.242	10.429	11.357	12.204	12.127	10.950	10.530
0.05	UR	DI/Dy	1.609	1.545	1.623	1.937	1.779	1.850	1.860	1.805	1.708	1.692
0.05	UR	FS/Dy	5.968	5.756	5.783	6.338	5.654	6.177	6.752	6.878	6.051	5.860
0.05	G	DI/Dy	1.801	1.852	1.913	1.808	1.783	1.834	1.865	1.829	1.724	1.676
0.05	G	FS/Dy	7.177	6.524	6.778	6.051	6.496	7.075	7.530	7.449	6.594	6.307
0.1	UR	DI/Dy	1.777	1.872	1.883	1.918	1.829	1.763	1.861	1.838	1.752	1.748
0.1	UR	FS/Dy	3.723	3.628	4.648	3.992	3.817	3.995	4.320	4.347	3.922	3.762
0.1	G	DI/Dy	2.229	1.857	1.851	1.918	1.958	1.828	1.935	1.895	1.752	1.721
0.1	G	FS/Dy	4.532	4.437	4.257	4.407	4.567	4.610	4.961	4.828	4.252	4.052
0.2	UR	DI/Dy	1.877	1.867	1.904	2.037	1.817	1.958	1.798	1.838	1.731	1.756
0.2	UR	FS/Dy	2.699	2.860	2.890	3.150	2.703	2.897	2.846	2.909	2.573	2.485
0.2	G	DI/Dy	1.991	1.902	1.881	1.785	2.005	1.944	1.946	1.868	1.743	1.753
0.2	G	FS/Dy	3.429	3.032	2.887	2.975	3.205	3.279	3.360	3.196	2.840	2.739
0.5	UR	DI/Dy	1.632	1.801	1.834	1.843	1.715	1.757	1.845	1.708	1.636	1.707
0.5	UR	FS/Dy	1.435	1.671	1.921	1.892	1.852	1.867	2.046	1.866	1.682	1.629
0.5	G	DI/Dy	1.787	1.848	1.751	1.798	1.800	1.861	1.850	1.720	1.663	1.680
0.5	G	FS/Dy	2.139	2.265	1.844	2.031	2.106	2.244	2.247	2.088	1.886	1.830
1	UR	DI/Dy	1.752	1.593	1.794	1.581	1.595	1.582	1.695	1.558	1.545	1.546
1	UR	FS/Dy	1.471	1.578	1.771	1.538	1.555	1.583	1.709	1.556	1.419	1.359
1	G	DI/Dy	1.713	1.633	1.594	1.652	1.649	1.740	1.704	1.572	1.556	1.581
1	G	FS/Dy	1.945	1.843	1.680	1.736	1.745	1.888	1.880	1.715	1.579	1.535
2	UR	DI/Dy	1.414	1.535	1.512	1.541	1.575	1.597	1.581	1.477	1.456	1.499
2	UR	FS/Dy	1.637	1.555	1.556	1.543	1.605	1.556	1.628	1.488	1.342	1.304
2	G	DI/Dy	1.532	1.758	1.770	1.646	1.626	1.674	1.656	1.545	1.511	1.534
2	G	FS/Dy	1.664	1.827	1.681	1.654	1.660	1.731	1.770	1.621	1.486	1.437
4	UR	DI/Dy	1.172	1.291	1.337	1.450	1.496	1.473	1.536	1.456	1.433	1.505
4	UR	FS/Dy	1.728	1.598	1.757	1.648	1.573	1.564	1.625	1.505	1.334	1.306
4	G	DI/Dy	1.377	1.470	1.439	1.581	1.535	1.612	1.615	1.461	1.486	1.512
4	G	FS/Dy	1.487	1.510	1.461	1.556	1.566	1.654	1.697	1.530	1.443	1.401
8	UR	DI/Dy	0.544	0.871	1.167	1.153	1.450	1.457	1.505	1.440	1.409	1.459
8	UR	FS/Dy	2.800	2.048	2.025	1.590	1.761	1.617	1.708	1.547	1.356	1.306
8	G	DI/Dy	1.135	1.273	1.242	1.431	1.439	1.550	1.557	1.471	1.473	1.498
8	G	FS/Dy	1.333	1.462	1.323	1.478	1.493	1.605	1.658	1.532	1.418	1.376
16	UR	DI/Dy	0.773	0.738	0.971	1.153	1.293	1.354	1.400	1.372	1.360	1.423
16	UR	FS/Dy	12.375	11.290	3.791	2.519	1.996	1.837	1.813	1.627	1.398	1.341
16	G	DI/Dy	1.004	1.353	1.271	1.327	1.314	1.407	1.476	1.440	1.436	1.475
16	G	FS/Dy	1.902	1.687	1.608	1.528	1.460	1.560	1.606	1.507	1.386	1.346

Table 5
Comparison of our method with other methods in terms of the average number of levels traversed. The first column gives speed coefficient values. The parameters in the second row correspond to *Distribution* (G: *Gaussian Distribution*, UR: *Uniform Random Distribution*). The parameters in the third row correspond to *Method* (Dy: *Dynamic Method*, DI: *Delete/Insert Method*, FS: *From-Scratch Method*).

Parameters			Number of points									
Speed coeff.	Distr.	$Method_1/Method_2$	100	200	400	800	1600	3200	6400	12800	25600	51200
0.01	UR	DI/Dy	1.783	1.865	1.833	1.956	2.133	2.098	2.254	2.334	2.450	2.536
0.01	UR	FS/Dy	33.421	36.854	33.575	38.126	40.492	43.908	46.881	49.962	52.693	55.494
0.01	G	DI/Dy	2.017	2.191	2.183	2.317	2.368	2.509	2.573	2.664	2.789	2.881
0.01	G	FS/Dy	34.396	40.822	40.939	44.296	48.554	52.559	54.770	58.434	61.116	64.184
0.02	UR	DI/Dy	1.855	1.755	1.765	1.963	2.048	2.103	2.242	2.301	2.412	2.511
0.02	UR	FS/Dy	14.853	18.885	17.773	19.404	20.566	22.588	23.868	25.651	26.961	28.537
0.02	G	DI/Dy	2.051	2.147	2.189	2.294	2.371	2.485	2.567	2.643	2.755	2.861
0.02	G	FS/Dy	18.206	19.479	21.165	23.159	25.196	27.199	28.191	30.058	31.345	33.027
0.05	UR	DI/Dy	1.778	1.676	1.761	1.927	1.981	2.071	2.162	2.257	2.353	2.453
0.05	UR	FS/Dy	6.417	7.374	7.983	8.218	8.877	9.562	10.274	10.857	11.474	12.127
0.05	G	DI/Dy	1.963	2.075	2.149	2.198	2.316	2.406	2.495	2.578	2.688	2.797
0.05	G	FS/Dy	8.182	8.794	8.981	10.036	10.909	11.556	12.132	12.788	13.394	14.115
0.1	UR	DI/Dy	1.657	1.670	1.733	1.834	1.924	2.011	2.098	2.202	2.291	2.384
0.1	UR	FS/Dy	3.641	3.969	4.297	4.507	4.861	5.193	5.587	5.890	6.256	6.593
0.1	G	DI/Dy	1.869	1.972	2.052	2.122	2.245	2.338	2.419	2.507	2.607	2.716
0.1	G	FS/Dy	4.610	4.841	4.973	5.574	6.007	6.323	6.632	6.981	7.318	7.692
0.2	UR	DI/Dy	1.519	1.548	1.648	1.735	1.821	1.910	1.998	2.091	2.179	2.278
0.2	UR	FS/Dy	2.121	2.297	2.474	2.607	2.792	2.997	3.188	3.384	3.582	3.778
0.2	G	DI/Dy	1.750	1.846	1.918	1.983	2.116	2.211	2.291	2.383	2.472	2.578
0.2	G	FS/Dy	2.702	2.831	2.898	3.221	3.469	3.640	3.831	4.027	4.217	4.424
0.5	UR	DI/Dy	1.279	1.341	1.404	1.492	1.560	1.649	1.718	1.818	1.887	1.988
0.5	UR	FS/Dy	1.169	1.259	1.347	1.427	1.526	1.614	1.721	1.819	1.928	2.029
0.5	G	DI/Dy	1.501	1.564	1.632	1.707	1.815	1.891	1.977	2.041	2.136	2.214
0.5	G	FS/Dy	1.544	1.610	1.632	1.770	1.891	1.991	2.079	2.192	2.282	2.404
1	UR	DI/Dy	1.026	1.077	1.135	1.202	1.261	1.328	1.390	1.466	1.529	1.608
1	UR	FS/Dy	0.884	0.914	0.972	1.018	1.084	1.135	1.210	1.267	1.349	1.410
1	G	DI/Dy	1.286	1.323	1.347	1.415	1.487	1.551	1.606	1.674	1.740	1.816
1	G	FS/Dy	1.146	1.178	1.181	1.258	1.336	1.401	1.459	1.535	1.600	1.686
2	UR	DI/Dy	0.803	0.835	0.891	0.937	0.991	1.032	1.092	1.136	1.199	1.246
2	UR	FS/Dy	0.796	0.776	0.800	0.822	0.858	0.891	0.940	0.980	1.036	1.083
2	G	DI/Dy	1.070	1.101	1.110	1.159	1.209	1.248	1.287	1.338	1.389	1.454
2	G	FS/Dy	0.930	0.947	0.933	0.980	1.034	1.077	1.120	1.171	1.225	1.292
4	UR	DI/Dy	0.643	0.687	0.730	0.770	0.813	0.844	0.891	0.926	0.975	1.014
4	UR	FS/Dy	0.879	0.790	0.759	0.744	0.753	0.763	0.792	0.818	0.856	0.891
4	G	DI/Dy	0.879	0.917	0.937	0.972	1.005	1.036	1.066	1.101	1.144	1.200
4	G	FS/Dy	0.799	0.818	0.817	0.831	0.856	0.885	0.917	0.954	0.998	1.055
8	UR	DI/Dy	0.486	0.534	0.612	0.651	0.692	0.720	0.757	0.787	0.826	0.859
8	UR	FS/Dy	2.237	1.100	0.886	0.779	0.740	0.718	0.721	0.728	0.750	0.772
8	G	DI/Dy	0.712	0.758	0.783	0.823	0.861	0.888	0.911	0.940	0.976	1.027
8	G	FS/Dy	0.775	0.759	0.742	0.761	0.765	0.774	0.794	0.817	0.850	0.901
16	UR	DI/Dy	0.330	0.423	0.518	0.529	0.590	0.623	0.662	0.687	0.721	0.749
16	UR	FS/Dy	2.120	2.620	2.102	1.097	0.882	0.772	0.727	0.702	0.701	0.706
16	G	DI/Dy	0.565	0.623	0.656	0.696	0.742	0.772	0.796	0.821	0.853	0.900
16	G	FS/Dy	0.971	0.826	0.756	0.730	0.720	0.729	0.728	0.740	0.751	0.791

Table 6
Two-tailed, paired Student's *t*-test significance results for the runtime comparison of our method with the other two methods for (a) speed coefficient = 0.1 and (b) speed coefficient = 4.

Method	Number of points									
	100	200	400	800	1600	3200	6400	12800	25600	51200
Panel a:										
From Scratch/Dynamic Random	7E−49	1.6E−48	2E−48	9.3E−54	8.3E−44	6.3E−59	5.5E−63	2.6E−71	2.6E−84	1.4E−91
Delete Insert/Dynamic Random	1.3E−17	3.9E−31	7.4E−35	8.4E−31	1E−29	1.3E−33	1.9E−30	2.8E−49	2E−60	4.3E−72
From Scratch/Dynamic Gaussian	1.7E−50	4.3E−59	7.3E−87	6E−105	7E−138	2E−144	4E−161	8E−170	3E−171	9E−174
Delete Insert/Dynamic Gaussian	1.8E−42	5.5E−49	5.2E−46	6E−69	3.6E−68	4.8E−80	2E−102	4E−103	1E−103	1E−107
Panel b:										
From Scratch/Dynamic Random	1.9E−34	4.9E−32	6.8E−27	3.2E−26	8.1E−23	5.9E−31	1.8E−57	1.1E−59	2.6E−61	1.2E−59
Delete Insert/Dynamic Random	2.3E−09	3.3E−16	4.8E−20	1E−28	4.6E−23	1.5E−29	6.9E−48	1.1E−49	2E−56	4.7E−75
From Scratch/Dynamic Gaussian	9.5E−61	5.2E−43	9.8E−60	2E−48	3E−56	6E−100	2E−118	1.3E−84	4E−125	5E−116
Delete Insert/Dynamic Gaussian	2.4E−43	5.4E−35	2.3E−58	3.8E−43	2E−51	9.2E−97	5E−106	3.6E−82	3E−125	9E−134

Table 7
Two-tailed, paired Student's *t*-test significance results for the comparison of our method with the other two methods in terms of the average number of levels traversed for (a) speed coefficient = 0.1 and (b) speed coefficient = 4.

Method	Number of points									
	100	200	400	800	1600	3200	6400	12800	25600	51200
Panel a:										
From Scratch/Dynamic Random	1.3E−54	1.1E−64	4.8E−77	1.1E−86	1.6E−96	4.2E−107	1.9E−112	1.4E−124	9.2E−130	1.7E−134
Delete Insert/Dynamic Random	1.5E−20	8.2E−31	2.7E−41	4.3E−51	1.4E−55	4.0E−68	1.5E−74	2.5E−86	8.2E−94	2.0E−98
From Scratch/Dynamic Gaussian	1.7E−119	3.4E−140	2.7E−148	6.9E−153	5.7E−157	1.4E−177	1.7E−176	2.1E−182	9.1E−176	1.9E−169
Delete Insert/Dynamic Gaussian	5.2E−57	2.4E−70	3.0E−78	6.2E−98	2.2E−100	2.7E−112	1.6E−116	1.3E−132	1.5E−106	3.1E−115
Panel b:										
From Scratch/Dynamic Random	5.3E−11	4.8E−35	1.6E−45	1.8E−55	5.2E−67	3.6E−74	4.1E−80	1.8E−91	5.2E−89	4.9E−91
Delete Insert/Dynamic Random	6.5E−29	5.0E−41	1.5E−43	5.7E−52	1.4E−58	7.4E−65	1.2E−67	2.5E−69	1.2E−53	8.9E−46
From Scratch/Dynamic Gaussian	2.0E−22	3.6E−24	1.2E−22	9.2E−31	1.2E−41	3.4E−46	8.1E−48	1.7E−38	0.47	3.4E−47
Delete Insert/Dynamic Gaussian	2.4E−17	6.0E−14	2.8E−10	0.0001	0.29	1.2E−15	1.1E−41	8.3E−72	2.0E−85	8.8E−100

Appendix B. Significance tables

This appendix provides two-tailed, paired Student's *t*-test significance results for the comparison of our method with the other two approaches (the insert/delete and from-scratch methods) in terms of runtimes (Table 6) and the average number of levels traversed (Table 7).

References

[1] C.-H. Ang, H. Samet, Node distribution in a PR quadtree, in: Proceedings of the First Symposium on Design and Implementation of Large Spatial Databases, Santa Barbara, CA, 1990, pp. 233–252.
[2] H. Cao, S. Wang, L. Li, Location dependent query in a mobile environment, Information Sciences 154 (1–2) (August 2003) 71–83.
[3] M. de Berg, M. Van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry, Algorithms and Applications, second ed., Springer-Verlag, 2000.
[4] D. Eppstein, M.T. Goodrich, J.Z. Sun, The skip quadtree: a simple dynamic data structure for multidimensional data, in: Proceedings of the 21st Annual Symposium on Computational Geometry, Pisa, Italy, June 2005, pp. 296–305.
[5] R.A. Finkel, J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, Acta Informatica 4 (1) (1974) 1–9.
[6] D.H. Francis, S. Madria, C. Sabharwal, A scalable constraint-based Q-hash indexing for moving objects, Information Sciences 178 (6) (2008) 1442–1460.
[7] E. Langetepe, G. Zachmann, Geometric Data Structures for Computer Graphics, A.K. Peters/CRC Press, 2006.
[8] F. Harlow, J. Welch, Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface, Physics of Fluids 8 (12) (1965) 2182–2189.
[9] K. Hegeman, N.A. Carr, G.S.P. Miller, Particle-based fluid simulation on the GPU, in: V.N. Alexandrov et al. (Eds.), Proceedings of the International Conference on Computational Science (ICCS), Lecture Notes in Computer Science, vol. 3994, Part IV, 2006, pp. 228–235.
[10] B.G. Mobasseri, A generalized solution to the quadtree expected complexity problem, Pattern Recognition Letters 16 (5) (1995) 443–456.
[11] M. Morse, J.M. Patel, W.I. Grosky, Efficient continuous skyline computation, Information Sciences 177 (17) (2007) 3411–3437.
[12] M. Mueller, D. Charypar, M. Gross, Particle-based fluid simulation for interactive applications, in: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003, pp. 154–159.
[13] S.V. Pemmaraju, C.A. Shaffer, Analysis of the worst case space complexity of a PR quadtree, Information Processing Letters 49 (5) (1994) 263–267.
[14] H. Samet, The quadtree and related hierarchical data structures, ACM Computing Surveys 16 (2) (1984) 187–260.
[15] Hanan Samet, Implementing ray tracing with octrees and neighbor finding, Computers & Graphics 13 (4) (1989) 445–460.
[16] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley Reading, MA, USA, 1990.
[17] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan-Kaufmann, San Fransisco, CA, USA, 2006.
[18] Y. Yu, L. Shi, Visual smoke simulation with adaptive octree refinement, in: Proceedings of IASTED International Conference on Computer Graphics and Imaging, Kauai, Hawaii, USA, August 2004, pp. 13–19.
[19] M. Shneier, Path-length distances for quadtrees, Information Sciences 23 (1) (1981) 49–67.
[20] J. Tayeb, O. Ulusoy, O. Wolfson, A quadtree-based dynamic attribute indexing method, The Computer Journal 41 (3) (1998) 185–200.
[21] R.K. Winder, The kinetic PR quadtree, 2000. <http://www.cs.umd.edu/mount/Indep/Ransom/index.htm>.
[22] W.-T. Wong, F.Y. Shih, T.-F. Su, Thinning algorithms based on quadtree and octree representations, Information Sciences 176 (10) (2006) 1379–1394.
[23] J.R. Woodwark, Compressed quad trees, The Computer Journal 27 (3) (1984) 225–229.