Spatial Indexing for Location-Aware Systems

Robert K. Harle
Computer Laboratory
University of Cambridge
Cambridge, UK
Email: Robert.Harle@cl.cam.ac.uk

***Abstract*—As location systems provide increasingly fine-grained locations for mobile entities, location-aware systems that react appropriately and autonomously to location events will be in demand. Although much research has been devoted to indexing schemes for very large scale GIS applications (where the systems are predominantly query-based), comparably little attention has been given to the use of spatial indexing within location-aware systems leveraging local positioning systems (predominantly event-based).**

This paper reviews the notion of spatial indexing (the representation of a tracked user's space to facilitate spatial event generation based on incoming locations) for event-based systems. It establishes the principles and requirements of a wide-area spatial indexer, reviews the R-tree, Quadtree and RQ-tree methods proposed before for indoor location-awareness, and significantly adapts the latter to meet the requirements.

The proposed indexing method uses a combination of R-trees (for high level spatial information, down to structural level) and Quadtrees (for lower level representation of objects). It proposes the use of linear Quadtrees to exploit the ease of direct node movement rather than the re-rasterisation of polygons used in systems to date. This approach is compared in simulation with other approaches.

I. Introduction

With a continuing increase in the uptake of mobile computing devices and ever-advancing sensor technologies, interest in large-scale context-aware systems is fast growing. Of particular interest in this paper is the subset of these systems that exploit location information to derive context. The most common of such systems are the so-called Location-Based Services (LBS), closely linked to Geographical Information Systems (GIS). These tend to be motivated by the wide availability of GPS positioning and supply facilities such as "where am I" (local maps and directions) and "where is my nearest..." (spatial queries).

In general a location-aware system has two primary components—a location-determination system [5] and a computational model of the space inhabited by tracked entities. The locations provided by the former have very limited use without the context provided by the latter. For LBS applications, the computational model often amounts to little more than a street-map. Indoors, however, it may be advantageous to model the world in greater detail. Here, the context may be as simple as inferring a meaningful room name from a co-ordinate location or as complex as evaluating the optimal lighting, cooling or security status for a space.

Much of the LBS applications envisaged can be viewed as application pull—the model is one of many devices and tracked entities, each making relatively infrequent demands on the LBS. Users themselves tend to be responsible for initiating queries when they know the LBS can help them. Conversely, in an indoor environment, it has proven more useful for a location-aware service to avoid user query initiation. Here, information push is more appropriate since indoor environments feature many more densely-packed objects of potential interaction, higher location update rates, and more local machinery that can be autonomously controlled. For example, a system that automatically illuminates a user's path based on their location data should not require the user or their digital agent to provide regular inputs to the system beyond reporting position if necessary.

Thus, indoor location-aware systems tend to be predominantly *event-based*, using a push model where no explicit query is generated by the user. Instead, the user (or more usually their agent) expects to be informed of spatial interactions automatically. A simple example is a 'friend alert' application, where users are automatically notified if they are physically near to any of their friends. This could be achieved by each user's device continually querying the location of *all* possible friends and computing proximity. However, it would be more efficient to have a notionally centralised system that monitors spatial proximities and *informs* the user's devices when a friend is nearby. This would be an event-based system.

Given such an event-based system, then, the question of how to generate the required location events efficiently naturally arises. For this, a *spatial indexer* is used. This is simply a process that monitors for spatial interactions and generates the events.

The efficiency of a spatial indexer is particularly important within an indoor environment. Here, objects and spaces are frequently encountered and proximity may be highly transient. AT&T Research Laboratories in Cambridge pioneered the idea of *programming with space*. This introduced the notion of user-defined spaces ("zones") which were uniquely labelled. Zones are simply 2D polygons anchored to some chosen point in space. The paradigm proved to be highly effective, and was used to enable building-wide location-awareness in conjunction with a fine-grained tracking system for personnel and objects. As an example of the basic concept, a virtual zone might be attached to computer terminals. When a user interacts with a terminal zone (by entering it), this is a signal which implies physical proximity to the terminal. A very similar technique to this was used at AT&T research to automatically

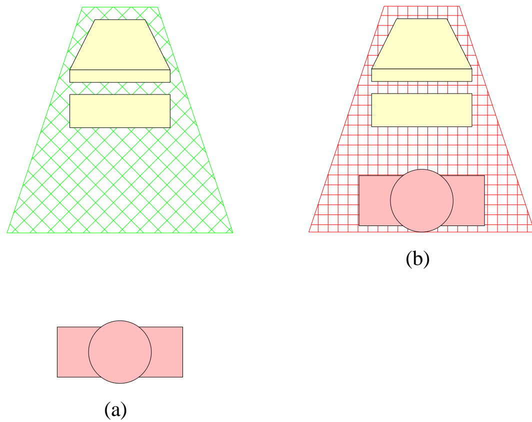

Fig. 1. The *programming with space* metaphor. Objects are assigned local virtual zones (illustrated in crossed hatch) and spatial events result from their interactions (interacting zones shown in square hatch). In this example a user enters the interaction zone of a computer.

and dynamically customise a terminal to an approaching user (providing automatic hotdesking) [3]. Figure 1 illustrates the concept.

The general lack of fine-grained tracking systems to date has meant that surprisingly little attention has been given to event-based spatial indexers. However, with commercial indoor tracking systems now generally available [1] it is likely that they will be in demand. In order to address this, this paper seeks to develop the notion of spatial indexing for location-aware systems which use the programming with space metaphor. It documents the requirements of such a spatial indexer, reviews previous attempts to meet them, and proposes and evaluates a new approach.

II. Requirements for a Spatial Indexer

The requirements listed here are based on day-to-day experiences with an indoor location system, accrued over many years at both AT&T Research Cambridge and the University of Cambridge, UK. A key event type has been room changes (i.e. "Alice has entered room 100"). Such events must not be ambiguous (it is not helpful to be reported as being in two rooms simultaneously) and so it is useful to have a point location for users and to test for room containment using a point-in-polygon test with the room bounds. This is simple to implement and, in principle, could be used for general spatial interaction. This is more appropriate for outdoor, LBS applications where the physical extents of the user are orders of magnitude smaller than the event-generating zones. However, when the user's extent is comparable to the zone size (as is the case indoors) it is necessary to associate a zone relative to the user's point location. Monitoring in this *zone-based* model requires constantly watching for interactions between multiple polygons (overlap, containment, etc), rather than point-in-polygon tests. It is this model of interactions that is addressed herein.

An oft overlooked issue with the pure zone-based model is how zones should interact with physical boundaries—in general there will be spaces that should be cropped to local physical boundaries, such as walls. This cropping recognises that mobile entities on the other side of the boundary may be within a given euclidean distance, but physical access will require a different route. For example, a user would rather the nearest telephone *in their office* rang for them, rather than the nearest telephone, which may happen to be on the other side of the wall in an inaccessible office!

Based on experience, there are four key requirements for a spatial indexer in an event-based location-aware system:

1) The indexer must generate appropriate room or region change events based on point location data;
2) The indexer must generate events describing ingress, egress, and overlap of interaction zones;
3) The indexer must support "nearest neighbour" queries for point data;
4) The indexer must account for immutable physical boundaries (walls, etc) when generating certain spatial events, whilst still supporting cross-boundary spatial interactions.

III. Related Work

Location-based indexing has received much attention in the space of GIS. Researchers have developed a variety of tree structures, including B-trees, R-trees, Quadtrees, D-trees [11], etc. Rigaux provides a good summary in [7] and many details can be found in Samet's books [8], [9]. The general consensus is that no one structure is globally optimal, leading to hybrid structures such as the QR-tree [6] which uses a Quadtree in main memory linking to disk-based R-trees housing GIS data. In general the work in the field of GIS has concentrated on processing specific location queries efficiently on very large scales. The primary location mechanism has been GPS and the queries often user-initiated (e.g. "where's my nearest...", "where am I").

When applying the ideas indoors, the research field is less developed. Research has tended to concentrate on creating a GPS-like location system and less on how it might be used. The general lack of deployed fine-grained indoor location systems means there are few examples of event-based indoor indexing systems in existence. However, the general GIS techniques have been adapted. Here we summarise the most important techniques and briefly review the adaptations.

A. *Quadtree Spatial Monitoring*

The SPIRIT middleware ([2]) used a dedicated software process (the "spatial monitor") to monitor for spatial interactions directly. This monitor was based on a pure *Quadtree* dissection of space and used a patented analysis to determine zone interactions [10]. The authors of SPIRIT have commercialised the location-aware system in the form of Ubisense. The basic principle of using a Quadtree persists today.

When using a Quadtree (more strictly a *region Quadtree*) for spatial indexing, each zone is represented as a filled two-dimensional simple polygon. A square area is chosen such as to encompass the entire area of interest (for example a wing

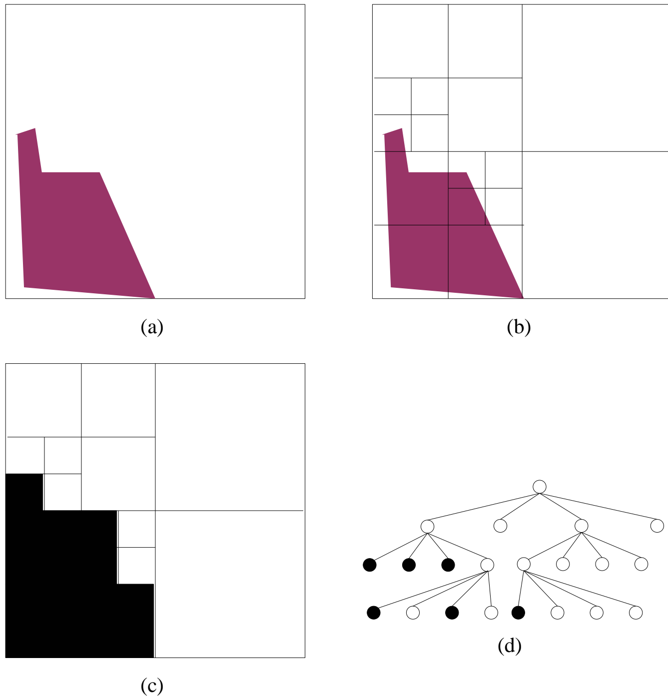

Fig. 2. Quadtree zone approximation. (a) The zone. (b) Recursive division of space according to zone overlap. (c) Filled leaf nodes. (d) Final tree representation.

or floor of a building), which contains the zone. Every node in a Quadtree is associated with a square area, and this particular area associates with the root node. Each node is either a leaf node or has four children, each corresponding to a quarter of their parent node's area. In representing an arbitrary shape, the root node is recursively split until nodes are found that are either fully contained by the shape, fully disjoint from the shape or have reached a predefined maximum tree depth (See Figure 2).

For the purposes of visualisation it may be preferable to imagine a rasterisation of the zone onto a grid of square pixels, where a pixel is a square that corresponds in size to a node at the maximum tree depth, d_{max} (i.e. for a root node of size $s \times s$ a pixel is of size $2^{-d_{max}}s \times 2^{-d_{max}}s$). A Quadtree is then an efficient representation of the resultant grid that saves storage space by merging adjoining pixels of the same type according to certain rules. The storage space requirements are an advantage, along with the general tree structure, which aids search operations. Note that it is difficult to characterise the cost of the Quadtree rasterisation because it depends not just on the number of vertices in the polygon but also on the spatial layout of them.

The advantages of using a Quadtree are that the process is conceptually straightforward, with the basic technique being well-established. Additionally, the implicit tree structure is appropriate for spatial searching. The primary disadvantage centres around the ease with which the tree can become unbalanced or skewed. Ideally spatial zones would be uniformly distributed across the area covered by the tree root. However, this is very unlikely to occur and the resultant unbalanced tree hinders efficient operations. For this reason, a popular alternative is the R-tree.

B. R-tree Spatial Indexing

An R-tree can be viewed as a height-balanced tree [9] where each node represents a rectangular area that completely contains the nodes below it. Rectangles may overlap. Their use in spatial indexing amounts to indexing the bounding boxes of each polygon and referencing the original shape from the associated node. The insertion procedure is effectively a mini-optimisation process that reforms all but the leaf nodes in order to height-balance the tree. There are many variants of the R-tree and their use is widespread in systems such as Oracle SPATIAL.

It is important to realise that an R-tree operates only on rectangular regions. Thus the use of the bounding boxes of zones rather than the zones themselves when creating the tree. As a spatial indexer, the R-tree is used to identify candidate pairs of zones with overlapping bounding boxes. Further tests must be performed on the associated zone polygons to determine whether interaction actually occurs. R-trees are fast and efficient for general spatial searching, but the additional overhead of checking polygonal boundaries can be high when there are many zones.

In [4] the authors of ASMod (a limited event-based spatial indexer) propose a combined R-tree/Quadtree data structure that they refer to as an RQ-tree. This uses an R-tree exactly as described above. If the space is irregular, however, it associates a Quadtree with the rectangle to further approximate the true boundary and speed up operations on it. Details on the method are very sparse, but ASMod seems to be designed on an assumption of very few, relatively large zones. As a result it just assumes point-locations of mobile objects, and associates an entire Quadtree to each zone. Whilst the core of this idea is exploited herein, ASMod as described in [4] is not appropriate as a general spatial indexer due to these design decisions.

IV. Developing a New Spatial Indexing System

In previous spatial indexing systems for location-awareness, little attention has been given to the types of spatial zones which are possible. Here two primary categories are defined:

- *Free zones*. These zones are completely unrestricted in where they lie or the physical boundaries they cross.
- *Associated zones*. These zones are associated with objects and must be clipped against the physical boundaries of the containment unit they are in (see Figure 3).

Whatever the spatial data structure in use, it should be apparent that it is more efficient to create a single structure with information about multiple zones rather than an individual structure for each zone. For a classic R-tree this is a given, since it deals with bounding boxes, each associated with a different zone. For a Quadtree, however, this is not a standard requirement. Quadtrees are traditionally associated with binary image arrays, and nodes are either *black* (filled pixels) or *white* (empty pixels). For a location-aware system, the first major

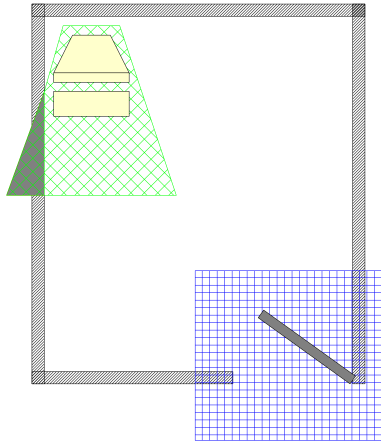

Fig. 3. Free (square hatch) and associated (crossed hatch) zones. Associated zones must be clipped to lie within the physical extents (the clipped portion is shaded dark grey).

extension to this is to associate multiple labels with a given node.

Thus we can extend the notion of the RQ-tree by implementing three key changes:

- Quadtree nodes support multiple zone labels;
- R-tree nodes map to the subset of zones that correspond to immutable physical containers (generally rooms), and nothing smaller. Point locations can then be used to identify the physical container of interest and the relevant Quadtree.
- Free zones are stored in a separate RQ-tree, where no clipping will occur.

Using R-tree nodes to map rooms and nothing smaller is particularly efficient when the boundaries are rectangular (as many rooms are for ease of tessellation). Inevitably, however, not all rooms will be perfectly rectangular, meaning that room determination will be ambiguous where the bounding boxes of two or more rooms overlap. This can be solved either by asserting that R-tree nodes may not overlap (i.e. multiple nodes may be needed per room), or by also representing the room as a zone within the associated Quadtree. In the latter case, when multiple R-tree nodes are returned for a given point location, the Quadtree representation can be used to determine which is correct. In the majority of environments this would be a rare occurrence.

Thus this adapted RQ-tree meets the requirements of a spatial indexer for the real world. In implementation, however, there may be optimisations to be made. The remainder of this paper assumes the adapted RQ-tree approach just defined and concentrates on trying to optimise the Quadtree structure contained by the R-tree nodes. The R-tree itself is not discussed in detail for reasons of brevity,[1]

A. Quadtree Representations

We adopt the terminology used to describe Quadtree nodes used by Samet. Each node has a *depth* that increases monotonically along a tree branch (the root node has depth 0, it's children have depth 1, etc). Equivalently we speak of a node's *level*, which is defined as $l = d_{max} - d$, where d is the node depth and d_{max} is the maximum depth of the tree.

Broadly speaking, Quadtree representations can be divided into pointer-based and linear. The former has a node that contains five pointers to other nodes as its fundamental data container. Four of these pointers point to its children (if it has any) and the remainder points up the tree to its parent. In addition, each node must store a list of zone identifiers that record any zones associated with it. Implementing such a Quadtree is simple, and traversing it is generally efficient, involving a chain of pointer evaluations. It is a favoured representation for existing spatial indexers. However, movement of a group of nodes within such a tree is complicated since a single node cannot be removed or created without destroying or creating the relevant parent hierarchy. Adding support for multiple zone labels per node only magnifies this problem since it means any node (not just leaf nodes) may be labelled and more tests must be performed to determine which branches to cull and which to create.

An alternative representation is a linear Quadtree, which represents a node by an entry (its *locational code*) in a flat list structure. If this structure associates each entry with a single zone label, support for multiple zones is easily achieved by allowing multiple entries with the same code. In effect, the list is a sparse representation of the tree where only labelled nodes are stored. For example, to represent Figure 2(d), a pointer-based tree would create 21 nodes and link their pointers together to form the tree. A linear tree would store only entries for the six filled nodes, implicitly defining the parent tree nodes. This makes the procedures for insertion and deletion much simpler since they require a single list insertion and deletion, respectively. Contrast this with a pointer-based Quadtree, where deleting a single node is likely to involve restructuring the entire branch if it no longer contains a true leaf node, and insertion involves the creation of the necessary parent node hierarchy.

B. Locational Codes and Movement

An interesting observation is that the majority of operations on the spatial data structures described herein will be associated with the *movement* of tracked entities, rather than the execution of explicit spatial queries. Indeed, much of the effort in indexing for a fine-grained event-based location-aware system is expended on capturing movement which is unlikely to elicit an event. Users are highly dynamic and associated zones must be frequently moved to capture their current position. Typically this is done by deleting the current zone and re-rastering a new, shifted version elsewhere on the tree. It may, however, be faster to operate not on the polygonal zone representation when determining the new representation but on the old Quadtree representation. To understand how this might work, we briefly review the formation of locational codes.

[1]The use of an R-tree throughout is exactly as in established literature.

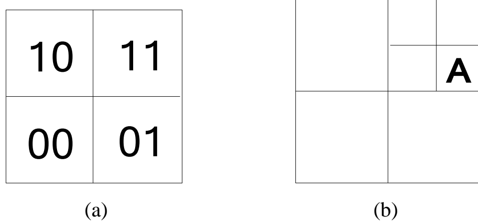

Fig. 4. Locational codes. (a) Quadrant labelling. (b) Node A has code 1101.

A locational code is formed by labelling each quadrant 0, 1, 2, or 3 in a predefined order (e.g. Figure 4(a)). This ordering is chosen because, in the two-bit binary representations of the quadrant number, the first bit corresponds to the row (or y-value) and the second to the column (or x-value). The locational code is simply the concatenation of the binary labels of the quadrants involved. For example, in Figure 4(b), node A lies within quadrant 3 (11 in binary) after one division, and quadrant 1 (01) after the second. Thus the locational code 3,1 concatenates to 1101 in binary. Note that the code is an interleave of the x and y divisions (1101 is 10 interleaved with 11)—when operating on nodes it is often necessary to interleave, deinterleave and reinterleave codes.

Notice also that the length of the locational code as described is dependent upon its depth (a greater code means more concatenations and thus a longer result). It is generally preferable to use an FD linear code, in which every code has the same length as a pixel. This is achieved by zero-padding on the right (so 1101 would be 110100 in a tree of depth three). A node is then identified by a pair {code, depth} (in general a given code refer to up to d_{max} nodes if no depth is specified). This has the distinct advantage that, when numerically sorted by code and sub-sorted by depth in a list, traversing the list traverses the tree depth first.

Returning to the issue of movement, we observe that a zone in a linear FD Quadtree is simply a group of nodes. Thus, if we can move one node, moving a zone is a trivial extension. A sample algorithm is given for translation in the x-direction in Algorithm 1; extension in the y-direction is trivial. Whether it is advantageous to use this algorithm depends on the cost of the splitting and condensing functions (which in turn depend on the zone shape, the maximum tree depth, and the movement vector) compared with the cost of re-rasterising the polygon at the new location.

However, merely moving[2] the nodes is not sufficient for a spatial indexer, which must determine whether the moving zone has egressed from any other zones, or ingressed on others. By the nature of a quadtree, ascending the tree from a given node to the root will visit all containing zones, whilst recursively descending the tree to the leaf nodes will visit all contained zones. A delete-and-insert movement algorithm implicitly descends the tree from the root on the insert operation and this can be leveraged by using the rasterisation process to simultaneously build up the containing zones. It is still necessary to descend the tree from the inserted node. The movement algorithm described above avoids the re-raster and so must perform an additional ascent of the tree compared to the other methods.

As hinted at, any traversals within a linear tree are simplified by the natural ordering of the nodes in a sorted list. Starting at a given node, the tree can be ascended by successively decrementing the depth, d, and zeroing the 2dth and (2d+1)th bits of the code. Descending the tree is simply an iteration down the list starting at the given node and ending when the depth of the current node matches that of the start node.

Moving a zone within a linear tree, then, can proceed in one of two ways:

- Delete-and-insert. A cache of current zone overlaps is maintained. When a zone is deleted the associated nodes are directly deleted and the cache used to determine any egresses this causes. Then the zone polygon is moved and rastered onto the tree whilst maintaining a list of zones associated with any nodes encountered in the process. Finally, the tree is descended from each new node and any encountered zones are recorded in the same list.
- Direct movement. A cache of current zone overlaps is maintained and used to build a list of egresses when a node is deleted. Each node associated with the zone is shifted according to Algorithm 1. The tree is then ascended *and* descended, building up a list of interacting zones.

Both approaches result in two lists that can be cross-referenced to determine the true changes in zone interactions.

C. Performance Comparison

Theoretical comparison of indexing algorithms rarely provides insight into subsequent performance as a spatial indexer. This is because each algorithm typically depends on quantities that are very hard to relate (polygon tests depend on the number of vertices, Quadtrees depend on the number of nodes the polygon makes after rasterisation, R-trees depend on the total number of zones inserted). It is thus better to evaluate algorithms based on simulations of realistic scenarios. In each case the solution was coded using C++, compiled using GNU g++ and executed on an Intel Pentium 4 3.20GHz CPU.

1) Basic Quadtree Movement: To examine the choice of movement algorithm within a Quadtree, a simulated 4m$\times$4m room was used. An 18-sided regular polygon was used to approximate a circular interaction zone of radius 0.4m. Such a zone is representative of the interaction zone that might be assigned to a tracked user. The zone was initially centred on (1,1) and then moved around a square path of side 2m in increments of 0.2m. The mean time for a complete circuit versus the maximum depth of the Quadtree is shown for three different movement algorithms in Figure 5. The algorithms are denoted PB (a pointer-based Quadtree that used delete-then-insert), LID (an FD-linear tree that used delete-then-insert),

[2]Note that throughout this paper, it has been assumed that zones are symmetrical and thus that translation is all that is required. For more complex scenarios, rotation may also be of importance. Fortunately, efficient rotation in a linear tree is possible and the basic principle applies (see Samet [8]).

Algorithm 1 Movement algorithm for a linear node moving in the x-direction

Input:
Integer $node$, locational code of node to move;
Integer l, level of node;
Real x_0, last known x-position of zone;
Real x_1, new x-position of zone;
Real pw, width of a pixel;

Procedure:
Compute number of whole pixels moved ($\lfloor(x_1 - x_0)/pw\rfloor$) and store in T_x;

Find lowest bit position of T_x and store in j;
if $j \geq l$ **then**
  Bit-deinterleave $node$ into $node_x$ and $node_y$ (1010→{11,00});
  Bit-interleave $\{node_x$+T_x, $node_y\}$ and store in $newnode$;
  Insert $newnode$ into the Quadtree list with level l;
  Delete $node$ from the Quadtree list with level l;
else
  Delete $node$ from the Quadtree list with level l;
  Insert the four child nodes of $node$ with level $l-1$;
  Recursively apply this procedure to the four child nodes;
end if

Condense the list structure to ensure each full set of siblings is replaced by a single parent node;

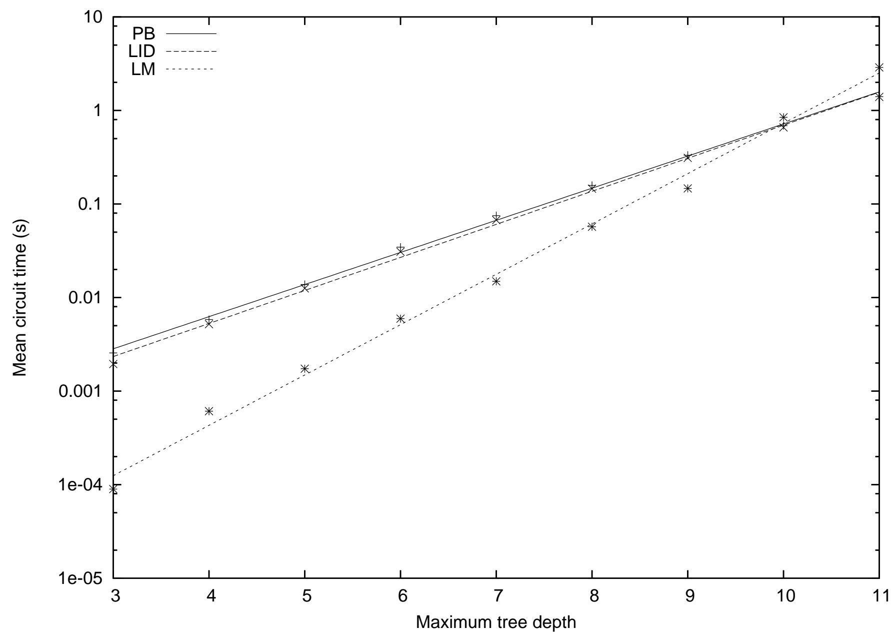

Fig. 5. Variation in mean circuit time with tree depth.

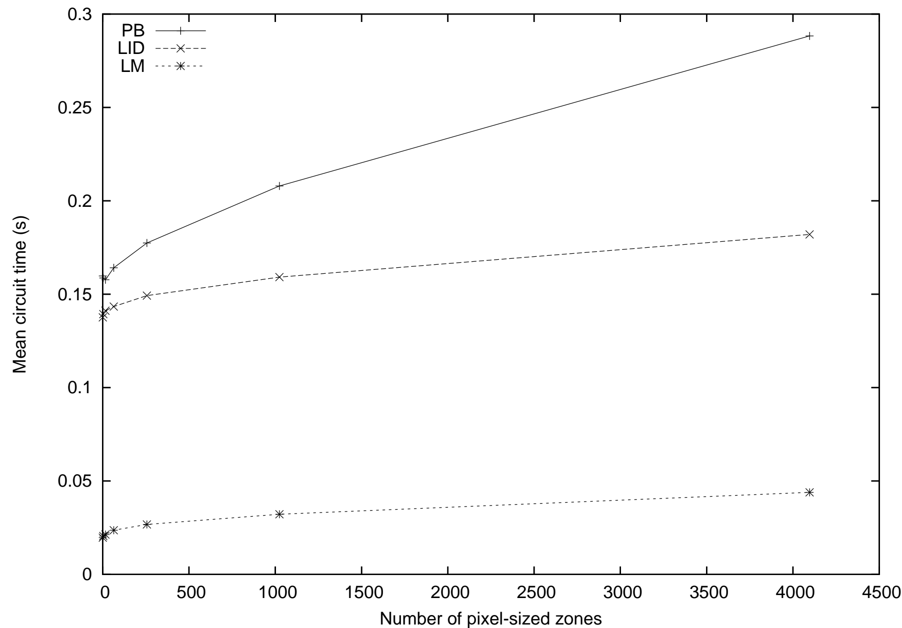

Fig. 6. Variation in mean circuit time with number of inserted nodes.

and LM (an FD-linear tree that used the movement algorithm advocated here).

There is little to choose between the LID and PB schemes, both of which rise exponentially, approximately doubling with each depth increment. This trend derives from the fact that the zone is highly non-rectangular and each division approximately doubles the number of nodes around the perimeter (which make the majority of the constituent nodes in a Quadtree representation) as the raster 'converges' to a true circle. The increased number of nodes leads to a proportional increase in operation time.

The LM scheme has a much lower execution time for small depth, but rises faster as depth increases. The extra cost derives again from the growing number of nodes needed to represent the circle, and the need to recursively divide and rebuild them (moving the same distance on a tree with smaller pixels means it is less likely that nodes will simply move rather than be split/condensed). At a depth of approximately 10 we observe that the LM algorithm offers little benefit over the LID equivalent. This observation vindicates the use of pointer-based trees when using a single Quadtree for indexing very large spaces since a larger total area requires a greater maximum depth to achieve the same pixel size and the resultant linear tree would perform badly using the LM algorithm. However, the approach advocated here has a Quadtree per physical area, each of which requires a smaller maximum depth to achieve the same pixel size (typically a depth of 7 is sufficient), and thus the advantages of a linear tree can be exploited.

2) Multiple Zone Handling: The previous results assumed no zones were installed in the room. In a real monitoring scenario, the presence of zones causes a degradation in performance since there are more spatial events to process. Figure 6 shows the variation in execution time at a given maximum depth of six when a series of pixel-sized interaction zones were distributed uniformly across the space before executing the simulated movement as before. In all cases, the cost of movement is increased; for PB the tree is larger, complicating deletion and insertion and the subsequent search for zone interaction; for LID and LM the number of tree nodes increases in the tree lists and thus list operations are slower. The LM trend mirrors that of LID but with a faster operation time, attributable to the more efficient movement of cells.

V. Further Work

The scheme presented here is unlikely to be the optimal and complete solution to the event-based spatial indexing problem. However, the results are promising and a combination of detailed code optimisation and adapted approach is likely to yield good results. In particular, there has been an implicit assumption that the maximum depth of the Quadtree is immutable. A maximum depth that varies with object type would permit static objects to be well approximated using a high depth, whilst more mobile objects could be approximated using a smaller depth. It may even prove worthwhile varying the depth dynamically according to the apparent mobility of the associated object. It is also interesting to note that nothing prevents a linear Quadtree from using either movement algorithm, dynamically selecting the most appropriate.

VI. Conclusions

This paper has made three primary contributions:

- It has identified and described the requirements for a spatial indexing system in a fine-grained event-based location-aware system;

- It has significantly adapted previous work to create a hybrid R-tree/Quadtree indexing structure that meets these requirements;
- It has described, evaluated, and compared Quadtree representations to better optimise the structure for typical tracking scenarios.

The resultant scheme has been shown to be an improvement on previous attempts, and offers scope for further improvement. The full indexer is intended to appear within a location-aware middleware being developed at Cambridge, UK.

VII. Acknowledgment

The author would like to thank the researchers that contributed to the SPIRIT system developed at AT&T Research Laboratories for their insight into indoor location-awareness. The author is also grateful for the comments received by the reviewers, which have greatly improved this manuscript.

References

[1] Ubisense. http://www.ubisense.net.

[2] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8), August 2001.

[3] N. Adly, P. Steggles, and A. Harter. SPIRIT: a resource database for mobile users. *Proceedings of ACM CHI'97 Workshop on Ubiquitous Computing, Atlanta, Georgia, March*, 1997.

[4] Hongliang Gu, Wei Wei, Wei Wang, Xiaoyong Guo, Jun Liang, Liping Shao, Jifeng Chen, and Na Ye. ASMod: A core model supporting location-aware computing in smart classroom. *International Journal of Computer Science and Network Security*, 6(3A):161–168, 2006.

[5] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, Aug 2001.

[6] Bo Huang and Qiang Wu. A spatial indexing approach for high performance location based services. *The Journal of Navigation*, 2007.

[7] P. Rigaux. *Spatial Databases: With Application to GIS*. Morgan Kaufmann Publishers, 2001.

[8] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[9] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[10] P.J. Steggles. United states patent 6718278. http://www.patentstorm.us/patents/6718278.html, 2000.

[11] Jianliang Xu, Baihua Zheng, Wang-Chien Lee, and Dik Lun Lee. The d-tree: An index structure for planar point queries in location-based wireless services. *IEEE Transactions on Knowledge and Data Engineering*, 16(12), 2004.