Improving the Performance of Region Quadtrees

Facharbeit im Nebenfach Informatik Supervised by: Prof. Dr. Michael Böhlen, Anton Dignös

> Marius Wolfensberger Zürich, Schweiz 08-915-795

Department of Informatics - Database Technology Binzmühlstrasse 14 CH-8050 Zurich

 $02.\,04.\,2013$

Contents

List of Figures					
List of Tables					
Al	stract	1			
1	Introduction	2			
	1.1 Background of Quad- and Octrees	. 2			
	1.2 Scope of the Study	. 2			
2	Definition of Quadtrees	4			
3	Decomposition Methods	6			
	3.1 Coverage-Based Splitting	. 6			
	3.2 Density-based Splitting	. 7			
4	Quadtree Variants	9			
	4.1 Regular Quadtree	. 9			
	4.1.1 MX-CIF Quadtree	. 9			
	4.1.2 Disadvantages of MX-CIF Quadtrees	. 11			
	4.2 Loose Quadtree \ldots	. 12			
	4.3 Comparison	. 14			
5	Conclusion	17			
Re	ferences	18			

List of Figures

1	The Example of a simple Region Quadtree	4
2	Block Decomposition induced by an MX-CIF Quadtree	7
3	Overlapping Polygons with density-based Splitting	8
4	Regular Quadtree Example	10
5	Large Accumulation of Points in a Quadtree	10
6	Disadvantage of Regular Quadtrees	11
7	Loose Quadtree Example	12
8	Different Factors for k \hdots	13
9	The Problem of longer Verlet Lists	14
10	Octree / Loose Octree Comparison	15
11	Histogramm of an Octree / Loose Octree Comparison	15

List of Tables

1	The average Number of Neighbours per Sphere for a completed			
	Packing of 10'000 Spheres	16		

Abstract

The worldwide dispersal of spatial databases and higher requirements regarding the speed of these systems have led to the demand of new techniques of indexing spatial data for faster data access.

This study will focus on a way to increase the speed of spatial indexing, introducing quadtrees as a popular and widespread data structure for the partitioning of objects and identifies its difficulties with the positioning of objects in the tree. A comparison with the lesser known loose quadtree shows that this data structure is able to overcome the disadvantages of the regular quadtree.

To improve the efficiency of the partitioning performed by a quadtree, alternative ways of halting the decomposition process have been defined, such as coverage-based splitting which restricts the number of quadtree blocks that contain an object or density-based splitting that restricts the number of objects which can be covered by a quadtree block. This study discusses the advantages of these methods and analyses difficulties implementing these decomposition rules.

1 Introduction

1.1 Background of Quad- and Octrees

During the recent decades the availability of spatial data has greatly expanded. With the use of spatial databases such as GIS¹ in sensitive areas like security systems and control centres, the requirements levied on the performance of these systems have increased dramatically. In this context, extensive research has been conducted on designing efficient data structures to permit a high performance spatial searching for instance using range queries. Therefore quadtrees for two-dimensional data and octrees as their counterpart for three dimensions have been introduced. [Kothuri et al., 2002]

Quadtrees are hierarchical data structures based on the principle of recursive decomposition of the environment embedding each object into blocks, until a predefined condition is satisfied. [Samet, 1984; Samet, 2005]

However, ordinary quadtrees have a few disadvantages which are reducing the effectiveness of the partitioning. A key problem consists in the strong spatial dependency of the objects when they are placed into a node². In order to counter this prejudice, several new variants have been introduced such as the loose quadtree, which enlarges the nodes' bounding box [Ulrich, 2000].

Quadtrees need well defined decomposition methods to determine when to halt for an efficient partitioning. Therefore several decomposition methods have been defined, such as density-based splitting that restricts the number of objects that are covered by a quadtree block, or coverage-based splitting that delimits the number of quadtree blocks that contain an object.

1.2 Scope of the Study

The demand for such new quadtree-variants and decomposition methods as well as the innovative character of this subject lead to the following research questions:

• What are the advantages and disadvantages of the different decomposition methods presented in this thesis?

¹Geographic Information Systems

 $^{^{2}}$ Each node represents a square block in the quadtree, called the node's bounding box covering a part of the indexed space

• Do loose quadtrees generate added value in spatial indexing compared to regular quadtrees?

The goal of this work is to give an overview about the regional quadtree and to introduce different state-of-the-art quadtree variants with their advantages and disadvantages. It will be outlined in which situations the loose quadtree outperforms the regular quadtree. This work may serve as a basis for decision-making whether the loose quadtree provides a good alternative in a given application.

This thesis is organized as follows: Section 2 gives the definition of the term quadtree. Section 3 sets its focus on the different decomposition methods, giving an introduction of how to improve the efficiency of this data structure. Section 4 illustrates the strengths and weaknesses of regular quadtrees, introducing the loose quadtree as an alternative which is able to overcome these drawbacks. These insights will be used then in the conclusion to demonstrate the acquired cognitions.

The theories and conclusions presented in this thesis are equally valid for quadtrees as well as octrees, unless otherwise stated.

2 Definition of Quadtrees

The term quadtree is "used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space" [Samet, 1984]. Quadtrees are a two-dimensional tree data structure invented by Finkel and Bentley in 1974 originally designed to sort spatial data [Finkel, 1974]. There are many variants of quadtrees used in numerous application areas [Samet, 2005]. This work will focus on regional quadtrees which are hereinafter referred to as quadtrees.

Region Quadtrees are based on a recursive subdivision of a given region into four equal-sized quadrants at each level. The region is recursively decomposed into four rectangular blocks until all blocks are occupied by an object or are empty (i.e. every leaf node belongs to a region or not). Every node of the tree corresponds to a quadrant in the dataset. Each inner (non-leaf) node has four child-nodes where each one represents a quarter of its parent node (referred to with geographic direction: NW, NE, SW, SE) and all quadrants in the same level have the same size [Samet, 1984/ Fischer, 2012]. The leaf nodes correspond to the blocks, which do not require further subdivision. [Samet, 1984; Samet, 2005]



Figure 1: A given region (a), its binary array representation (b), the block decomposition of the region (c) and its quadtree representation (d) [Samet, 1984]

Figure 1 shows an example of a region in Figure (1a) which is represented by

a quadtree in Figure (1d). Figure 1(b) shows its binary array with the size $2^3 * 2^3$. The 1's represent the elements that are in the region and the 0's the elements outside of it. The resulting block decomposition is shown in Figure 1(c). Note the different sizes of the blocks which contain a part of the region. Figure 1(d) shows the region represented as a quadtree. The quadtree has a degree of 4, which means 3 subdivisions are necessary to represent the region [Samet, 1984]. Take into consideration that this example is kept very simple, as there is only one object to be represented and the region boundaries coincide exactly with the quadtree block extents.

3 Decomposition Methods

In quadtrees as shown in Figure 1 the decomposition process is halting, whenever a block is completely contained in an object or a set of objects. However, since the boundaries of the objects don't need to coincide in regular data sets, this rule can't be satisfied in general. Thus alternative ways to halt the decomposition process are required. Therefore two methods are introduced in this section for an efficient partitioning [Samet, 2005]:

- 1. Coverage-based splitting Restrict the number of quadtree blocks that contain an object.
- 2. *Density-based splitting* Restrict the number of objects that are covered by a block or part of a block.

3.1 Coverage-Based Splitting

A conventional method to implement coverage-based splitting is to set the number of blocks that can cover an object to T = 1. In concrete terms this means, that an object can only be part of 1 block. Usually this is the smallest possible block that may contain an object. The *MX-CIF* quadtree decomposes the region into four evenly divided blocks, so that each object is associated with its minimum enclosing quadtree block [Samet, 2005]. The MX-CIF quadtree will be described in more detail in Section 4.1.1.

A disadvantage when restricting the blocks that can cover an object to 1 is that the blocks tend to be rather large on average, as objects that straddle the first split partition will automatically be placed in the root [Samet, 2005]. As the region shown in Figure 1 only consists of one polygon it would not have been split at all using the coverage-based splitting method. Instead the quadtree would have represented the whole region using only the root.



Figure 2: A collection of rectangles and the block decomposition induced by its MX-CIF quadtree (a) and its tree representation (b) [Samet, 2005]

Figure 2 shows an example of an MX-CIF quadtree using coverage-based splitting. More than one object is associated per node, for instance the objects 3, 4 and 5 are all part of block E, yet all objects are only part of one block. Notice that for example object 2 on level 0 (the root) is smaller than other objects appearing on the second level, due to its position. The MX-CIF quadtree and its drawbacks will be discussed in further detail in chapter 4.1.2.

3.2 Density-based Splitting

Density-based splitting is defined, that a block is decomposed whenever it covers more than $T(T \ge 1)$ objects (or parts of them). The case T = 1means that the decomposition will halt whenever each block contains one or zero objects. Such rules are known as *bucketlike* decompositions. Notice that this decomposition rule does not split the block if it contains more than T objects as such a rule would be difficult to satisfy, since blocks usually contain portions of many objects rather than many entire objects. [Samet, 2005]

Density-based splitting works well when the polygons are disjoint. However, when the dataset contains overlapping or adjacent polygons, density-based splitting can lead to problems. Overlapping can occur for example if you calculate a buffer zone³ around a polygon dataset or in traffic networks.

³Area around objects with a defined distance



Figure 3: An example of overlapping polygons where density-based splitting using T = 2 will never halt. The block is represented by the external blue square, whereas the three squares inside the block represent the objects. [Samet, 2005]

Figure 3 shows an overlapping polygon configuration that will never halt. Assuming T = 2, the block containing the three polygons is not split when inserting the third polygon. The situation that all the blocks will be part of T or fewer polygons will never occur. [Samet, 2005]

In geographic information systems the most common case of polygon data are adjacent polygons e.g. land coverage. Using density-based splitting with adjacent polygons the case that the decomposition will not halt due to overlapping polygons is not possible anymore. However, infinite decompositions are still possible if a map has a vertex where more than T polygons are incident. There are several ways to overcome this problem, for example to decompose the block just once if it is part of more than T polygons using a PMR quadtree which has a tree structure that is sensitive to the order in which the polygons are inserted. This solution is not discussed here in detail. [Samet, 2005]

4 Quadtree Variants

As mentioned in section 2, there are numerous variants of quadtrees. Most of them are particularly suited for a specific field of operation as for example spatial indexing, image-encoding and many more [Samet, 2005]. This chapter introduces two variants of region quadtrees and gives a comparison.

4.1 Regular Quadtree

4.1.1 MX-CIF Quadtree

In an MX-CIF quadtree, all objects associated to a specific node, must be contained completely within the node's bounding box. It decomposes the underlying space into four equally sized blocks at each level, so each object is associated to the smallest possible block that can entirely contain the object, i.e., the *minimum enclosing quadtree block* [Samet, 2005]. An object can only be part of one single cell. The quadrant on level 0 is divided into four quadrants with side length s_1

$$s_1 := s_0/2$$

on level 1 where s_0 corresponds to the side length at level 0 [Raschdorf et al., 2009]. In this quadtree the side length is static and can be generalized as:

$$L(depth) = W/2^{depth}$$

where W is the side length of the given region and *depth* is the number of levels by which a node is separated from the root. The root node itself has depth = 0 [Ulrich, 2000]

Generally an MX-CIF quadtree is generated with this algorithm [Fischer, 2012]:

- 1. Begin with a cell which contains all points (bounding box).
- 2. Divide the cell into four squares with the same size and distribute the objects amongst the squares.
- 3. Divide them until you have a maximum of k points in a square or a maximum number of levels.



Figure 4: An example of a regular quadtree with point data [Burad, 2006]

Insertion of an object into an MX-CIF quadtree is done by traversing the tree. Beginning with the root, the insertion algorithm checks whether the object fits into one of its four child nodes without intersecting another node. If this holds, the object descends the tree to the corresponding child node. Otherwise the object is inserted in the present parent node. This guarantees that all objects are placed as deep as possible into the tree. [Raschdorf et al., 2009]

The quadtree encloses its object completely so none of the adjacent nodes have to be searched for overlapping objects [Raschdorf et al., 2009]. Figure 5 shows that large accumulations of objects only have a limited influence on the overall structure of the quadtree. [Fischer, 2012]



Figure 5: A large accumulation of points in a quadtree [Fischer, 2012]

4.1.2 Disadvantages of MX-CIF Quadtrees

The MX-CIF quadtree is very simple to implement and usually provides a good performance, especially with well distributed values [Samet, 1984]. However, this quadtree has a great disadvantage: An object intersecting the boundaries at level 0 will be placed automatically into the root node no matter its size. Figure 6 shows that objects of the exact same size may be stored on different levels depending on their positions. Thereby they reduce the efficiency of the spatial operations, due to a reduction in the ability to decrease the times the objects have to be taken into consideration when traversing the tree. [Samet, 2008; Ulrich, 2000; Raschdorf et al., 2009].



Figure 6: All of the objects have the same size, yet they are on different levels (marked grey/black), due to their position on the partitioning line. [Ulrich, 2000]

There are various ways to approach the problem that objects with the same diameter can be placed on different levels [Ulrich, 2000; Samet, 2008]:

- Split the objects on the partitioning planes and classify these individually thereby receiving additional polygons which need to be connected using extra storage.
- Reference the object by more than one node, so they are referenced by child nodes on either side of a node. This increases the administrative effort as the algorithm needs to handle redundancy since the number of references to an object is replicated.

Both approaches have their disadvantages and lead to a more complex implementation. Therefore, in the next section another quadtree structure is introduced that addresses these disadvantages by expanding the bounding area, i.e., the loose quadtree.

4.2 Loose Quadtree

Not every data may fit as good as the point data from Figure 4 into a quadtree node where it is intrinsic, that objects do not cross the boundary lines of the nodes. For objects with a spatial extension other solutions are required. The loose quadtree and the cover fieldtree method adjust the nodes bounding area by "loosening" the boundary, but leaving the hierarchy and the centre of the nodes as they are [Ulrich, 2000; Samet, 2009]. Cover fieldtrees were developed to represent spatial objects in geographic information systems, whereas loose quadtrees are used in the context of game programming applications. This work uses these expressions interchangeably.

In the conventional quadtree the edge has a length L, whereas in the loose quadtree the edge has the length kL, where k is a factor > 1. The formula for the edge length of a loose quadtree is

$$L(depth) = k * W/(2^{depth})$$

where k determines how much the edges are expanded. This node expansion causes the objects to be placed much deeper into the tree compared to MX-CIF quadtrees.



Figure 7: An example of a loose quadtree [Fischer, 2012]

Figure 7 shows an example of a loose quadtree. In a regular quadtree the polygon would have been placed in the parent node despite the small size. However, here the south-east cell is "loosen" to fit the entire polygon. The bounding node is still a square. As we can see, with loose bounding squares smaller objects will fit within deeper levels of the tree, making the size of an object more important for its level than its position.

The main challenge using loose quadtrees is to find a good factor for k. If the factor is too small the root node can be overfilled with small poorly partitioned objects and will suffer the same problems as MX-CIF quadtrees.

Using a factor too large will result in excessively loose bounding lengths and therefore will have too much overlapping between nodes on one level. [Ulrich, 2000 / Samet, 2005]. This leads to longer so-called *Verlet* lists $L_v(\sigma)$, containing all the neighbours of an object σ in a node v that have outer boundaries overlapping the outer boundary of v. [Raschdorf et al., 2009].

Ulrich suggests the factor k=2 as a "useful all-around value" [Ulrich, 2000] which is widely accepted as the ideal value [Samet, 2009]. With k=2 an object with diameter d necessarily fits into the nodes of level l with an inner boundary length of

$$d \le s_l^{(in)} := s_0/2^l$$

where $s_l^{(in)}$ is the side length of the inner boundary at level l and s_0 is the side length of the root. At the same time the factor k=2 is not causing too much overlap, therefore preventing the quadtree from having a long Verlet lists of neighbours [Raschdorf et al., 2009].



Figure 8: Different factors for k. The blue square shows the node's bounding box. Left: regular quadtree with k = 1, the object is placed in the root. Middle: loose quadtree with factor k = 2, the object is placed in the NWchild node of the root. Right: loose quadtree with factor k = 3.5, the object is allocated in level 3 [Examples generated with Brabec et al., 2012]

Figure 8 shows the effects of factor k. Without "loosening" the boundaries as shown in the left picture, the object is placed in the root as it crosses the coordinate plane of the first level. In the middle image, which shows the "ideal" value k = 2, the node is placed in the north-west-node of the root and overlaps only one other node (north-east). In the right image with a factor k = 3.5 the object is allocated very deep in the tree and overlaps 5 other nodes.



Figure 9: An example to show the enlargement of Verlet lists with a bigger factor k [Example generated with Brabec et al., 2012]

Figure 9 shows the drawback if k is too large. Given a spatial query, which is searching for objects in the north-east node, the regular quadtree only searches in one node. However if you expand the bounding box with k=2(indicated with the blue line), the query needs to check all 4 nodes whether they contain objects in the north-east node. The higher the factor k is chosen, the longer are the resulting Verlet list, which means more nodes have to be checked.

4.3 Comparison

Figure 10 is an excellent example to show the differences using regular and loose octrees. 50'000 spheres of different sizes were inserted into a regular octree and an octree with loose boundaries. The darkness of the color specifies the level in the quadtree. The darker the color, the lower the level (black spheres are placed in the root node). The boundary lines are easily visible in the regular octree as the spheres crossing a low level coordinate plane are placed on a lower level. Regarding the loose octree, the black lines have disappeared. The spheres with bigger sizes are on a lower level than the other spheres and the overall average level is much higher.



Figure 10: A comparison of the dense packing of 50'000 spheres with a regular octree (left) and a loose octree (right). The darker the color, the lower the level [Raschdorf et al., 2009]

Raschdorf compared the performance of these different structures comparing samples of 10'000 spheres using monodisperse⁴ and polydisperse⁵ packings. The histograms in Figure 11 show the distribution of the spheres among the levels of the regular and the loose octree.



Figure 11: Distribution of spheres among octree and loose octree levels in the final packings, averaged over 10 simulation runs. [Raschdorf et al., 2009]

The results are very distinct and show a similar pattern as in Figure 10. While the objects from the original octree are mainly distributed on the first few levels in both test cases, the objects in the loose variant reside mainly within the last levels. In the monodisperse test with the loose variant there are no spheres on the levels 0-3 as their diameter is smaller than the length of the inner boundary. Therefore it is apparent that the main allocation in

⁴A collection of objects with the same size

⁵A collection of objects with an inconsistent size

the regular octree is based on the position, while the loose octree allocates the spheres predominant according to their size [Raschdorf et al., 2009].

method	av. no. of neighbours		
	monodisperse	polydisperse	
octree	2027.6	1 188.1	
loose octree	51.7	31.2	

Table 1: The average number of neighbours per sphere for a completed packing of 10'000 spheres, [Raschdorf et al., 2009]

Table 1 shows that compared to the regular octree the loose octree has about 40 times less neighbours in average to check on possible overlap with both kinds of packings, making it the only suitable choice in these cases.

However, Ulrich stated, that the loose variant is not always a guaranteed improvement of the performance. His approach was, to compare the performance of typical quadtree applications such as frustum culling⁶ and collision detection with a regular and a loose octree. He mentions that it depends mainly on the circumstances whether such a modification is necessary. Especially for situations with a large number of interactions and dynamic objects he observed differences in the performance which underline the advantages of the loose modification. On the other hand in the test with frustum culling the loose variant returned fewer possible visible objects, yet had to check more nodes leading to a balance in performance differences. [Ulrich, 2000].

Despite of the many disadvantages, regular quadtrees do have one advantage over their loose counterpart which was shown in Figure 9. Their nodes cover exactly the same region as their descendants, while loose quadtrees nodes overlap their siblings on the same level, therefore most space is covered by more than one node. However, the advantages of the loose quadtree outweigh its drawbacks. Most of the objects are on a deeper level ensuring much more precise results with shorter Verlet lists [Raschdorf et al., 2009].

⁶Line-of-sight analysis

5 Conclusion

Quadtrees have found use in many diverse applications such as geographic information systems and game applications for instance in order to detect collision, spatial indexing and line-of-sight analysis. This study discussed the disadvantages of common MX-CIF quadtrees in combination with line and polygon objects. It was established, that the main disadvantage consists in the allocation of objects in the tree, which depends more on the object's position than on its size since small objects crossing a certain coordinate plane are automatically inserted in the corresponding level, decreasing the overall performance.

This work has shown that the loose quadtree resolves this drawback and therefore provides added value without compromising important properties of the MX-CIF quadtree. It can be applied not only for monodisperse, but for polydisperse packings too [Raschdorf et al., 2009], which is an important factor regarding geographic information systems as line and polygon data of real world objects always have a polydisperse structure. The factor k=2is widely accepted as a reasonable compromise between poorly partitioned objects and long Verlet lists.

The presented decomposition methods are useful enhancements to the quadtree systems to improve the efficiency and the functionality. Coverage-Based splitting for example leads us to the MX-CIF quadtree and its loose variant. Nevertheless, both methods must be used with caution as coverage-based splitting can lead to large blocks and an imprudent use of density-based splitting can create situations where the decomposition will never halt.

Since the demands in performance in game applications and geographic information systems continuously rises, these modifications provide a powerful alternative to the regular variant. However, whether it is appropriate to use the loose modification or a presented splitting method must be determined on a case by case basis. In situations where large numbers of interacting and dynamic objects occur, the loose quadtree is a rather good option to increase the overall performance, making it a powerful alternative to the classic quadtree. [Ulrich, 2000]

References

Brabec, F., Samet, H. "Cover fieldtree and loose quadtree" (demo). http://donar.umiacs.umd.edu/quadtree/rectangles/loosequad.html, (2012). Viewed: 31.03.2013.

Burad, A. "Advanced data structures" http://multimediadb.blogspot.ch/ 2006/03/advanced-data-structures.html, (2006). Viewed: 06.03.2013.

Finkel, R., Bentley, J. "Quad trees a data structure for retrieval on composite keys." *Acta informatica* 4.1: 1-9, (1974).

Fischer, M. "Algorithmen für hochkomplexe Virtuelle Szenen." Heinz Nixdorf Institut, Universität Paderborn, (2012).

Kothuri, R., Ravadi, S., Abugov, D. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data." Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, (2002).

Raschdorf, S., and M. Kolonko. "Loose octree: a data structure for the simulation of polydisperse particle packings." (2009).

Samet, H. "The quadtree and related hierarchical data structures." ACM Computing Surveys (CSUR) 16.2: 187-260, (1984).

Samet, H. "Foundations of multidimensional and metric data structures." Morgan Kaufmann, (2005).

Samet, H. "A sorting approach to indexing spatial data." International Journal of Shape Modeling 14.1: 15-37, (2008).

Ulrich, T. "Loose octrees." Game Programming Gems 1: 434-442, (2000).