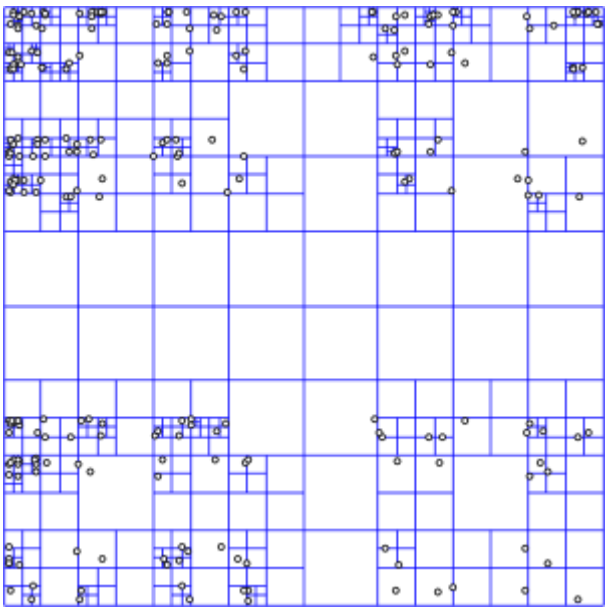
Quadtree

A **quadtree** is a tree data structure in which each internal node has exactly four children. Quadtrees are the two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The data associated with a leaf cell varies by application, but the leaf cell represents a "unit of interesting spatial information".

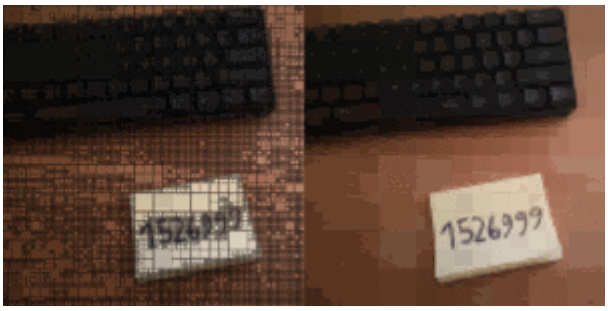
The subdivided regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a *Q-tree*. All forms of quadtrees share some common features:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the quadtree.

A **tree-pyramid** (**T-pyramid**) is a "complete" tree; every node of the T-pyramid has four child nodes except leaf nodes; all leaves are on the same level, the level that corresponds to individual pixels in the image. The data in a tree-pyramid can be stored compactly in an array as an implicit data structure similar to the way a complete binary tree can be stored compactly in an array.[1]



A point quadtree with point data. Bucket capacity 1.



Quadtree compression of an image step by step

Contents

Types

Quadtrees may be classified according to the type of data they represent, including areas, points, lines and curves. Quadtrees may also be classified by whether the shape of the tree is independent of the order in which data is processed. The following are common types of quadtrees.

Region quadtree

The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has exactly four children, or has no children (a leaf node). The height of quadtrees that follow this decomposition strategy (i.e. subdividing subquadrants as long as there is interesting data in the subquadrant for which more refinement is desired) is sensitive to and dependent on the spatial distribution of interesting areas in the space being decomposed. The region quadtree is a type of trie.

A region quadtree with a depth of n may be used to represent an image consisting of $2^n \times 2^n$ pixels, where each pixel value is 0 or 1. The root node represents the entire image region. If the pixels in any region are not entirely 0s or 1s, it is subdivided. In this application, each leaf node represents a block of pixels that are all 0s or all 1s. Note the potential savings in terms of space when these trees are used for storing images; images often have many regions of considerable size that have the same colour value throughout. Rather than store a big 2-D array of every pixel in the image, a quadtree can capture the same information potentially many divisive levels higher than the pixel-resolution sized cells that we would otherwise require. The tree resolution and overall size is bounded by the pixel and image sizes.

A region quadtree may also be used as a variable resolution representation of a data field. For example, the temperatures in an area may be stored as a quadtree, with each leaf node storing the average temperature over the subregion it represents.

If a region quadtree is used to represent a set of point data (such as the latitude and longitude of a set of cities), regions are subdivided until each leaf contains at most a single point.

Point quadtree

The point quadtree[2] is an adaptation of a binary tree used to represent two-dimensional point data. It shares the features of all quadtrees but is a true tree as the center of a subdivision is always on a point. It is often very efficient in comparing two-dimensional, ordered data points, usually operating in O(log n) time. Point quadtrees are worth mentioning for completeness, but they have been surpassed by *k*-d trees as tools for generalized binary search.[3]

Point quadtrees are constructed as follows. Given the next point to insert, we find the cell in which it lies and add it to the tree. The new point is added such that the cell that contains it is divided into quadrants by the vertical and horizontal lines that run through the point. Consequently, cells are rectangular but not necessarily square. In these trees, each node contains one of the input points.

Since the division of the plane is decided by the order of point-insertion, the tree's height is sensitive to and dependent on insertion order. Inserting in a "bad" order can lead to a tree of height linear in the number of input points (at which point it becomes a linked-list). If the point-set is static, pre-processing can be done to create a tree of balanced height.

Node structure for a point quadtree

A node of a point quadtree is similar to a node of a binary tree, with the major difference being that it has four pointers (one for each quadrant) instead of two ("left" and "right") as in an ordinary binary tree. Also a key is usually decomposed into two parts, referring to x and y coordinates. Therefore, a node contains the following information:

- four pointers: quad['NW'], quad['NE'], quad['SW'], and quad['SE']
- point; which in turn contains:
  - key; usually expressed as x, y coordinates
  - value; for example a name

Point-region (PR) quadtree

Point-region (PR) quadtrees[4][5] are very similar to region quadtrees. The difference is the type of information stored about the cells. In a region quadtree, a uniform value is stored that applies to the entire area of the cell of a leaf. The cells of a PR quadtree, however, store a list of points that exist within the cell of a leaf. As mentioned previously, for trees following this decomposition strategy the height depends on the spatial distribution of the points. Like the point quadtree, the PR quadtree may also have a linear height when given a "bad" set.

Edge quadtree

Edge quadtrees[6][7] (much like PM quadtrees) are used to store lines rather than points. Curves are approximated by subdividing cells to a very fine resolution, specifically until there is a single line segment per cell. Near corners/vertices, edge quadtrees will continue dividing until they reach their maximum level of decomposition. This can result in extremely unbalanced trees which may defeat the purpose of indexing.

Polygonal map (PM) quadtree

The polygonal map quadtree (or PM Quadtree) is a variation of quadtree which is used to store collections of polygons that may be degenerate (meaning that they have isolated vertices or edges).[8] [9] A big difference between PM quadtrees and edge quadtrees is that the cell under consideration is not subdivided if the segments meet at a vertex in the cell.

There are three main classes of PM Quadtrees, which vary depending on what information they store within each black node. PM3 quadtrees can store any amount of non-intersecting edges and at most one point. PM2 quadtrees are the same as PM3 quadtrees except that all edges must share the same end point. Finally PM1 quadtrees are similar to PM2, but black nodes can contain a point and its edges or just a set of edges that share a point, but you cannot have a point and a set of edges that do not contain the point.

Compressed quadtrees

This section summarizes a subsection from a book by Har-Peled.[10]

If we were to store every node corresponding to a subdivided cell, we may end up storing a lot of empty nodes. We can cut down on the size of such sparse trees by only storing subtrees whose leaves have interesting data (i.e. "important subtrees"). We can actually cut down on the size even further. When we only keep important subtrees, the pruning process may leave long paths in the tree where the intermediate nodes have degree two (a link to one parent and one

child). It turns out that we only need to store the node u at the beginning of this path (and associate some meta-data with it to represent the removed nodes) and attach the subtree rooted at its end to u. It is still possible for these compressed trees to have a linear height when given "bad" input points.

Although we trim a lot of the tree when we perform this compression, it is still possible to achieve logarithmic-time search, insertion, and deletion by taking advantage of *Z*-order curves. The *Z*-order curve maps each cell of the full quadtree (and hence even the compressed quadtree) in $O(1)$ time to a one-dimensional line (and maps it back in $O(1)$ time too), creating a total order on the elements. Therefore, we can store the quadtree in a data structure for ordered sets (in which we store the nodes of the tree). We must state a reasonable assumption before we continue: we assume that given two real numbers $\alpha, \beta \in [0, 1)$ expressed as binary, we can compute in $O(1)$ time the index of the first bit in which they differ. We also assume that we can compute in $O(1)$ time the lowest common ancestor of two points/cells in the quadtree and establish their relative *Z*-ordering, and we can compute the floor function in $O(1)$ time. With these assumptions, point location of a given point q (i.e. determining the cell that would contain q), insertion, and deletion operations can all be performed in $O(\log n)$ time (i.e. the time it takes to do a search in the underlying ordered set data structure).

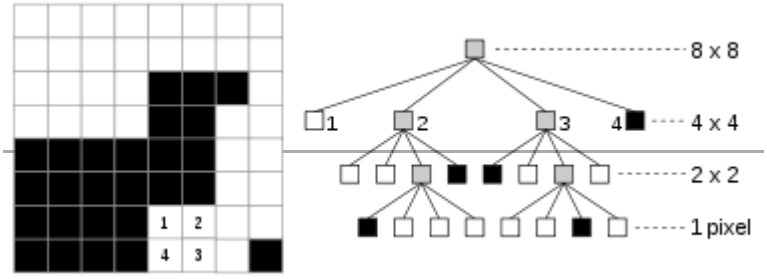
To perform a point location for q (i.e. find its cell in the compressed tree):

1. Find the existing cell in the compressed tree that comes before q in the *Z*-order. Call this cell v.
2. If $q \in v$, return v.
3. Else, find what would have been the lowest common ancestor of the point q and the cell v in an uncompressed quadtree. Call this ancestor cell u.
4. Find the existing cell in the compressed tree that comes before u in the *Z*-order and return it.

Without going into specific details, to perform insertions and deletions we first do a point location for the thing we want to insert/delete, and then insert/delete it. Care must be taken to reshape the tree as appropriate, creating and removing nodes as needed.

Some common uses of quadtrees

- Image representation



- Image processing
- Mesh generation
- Spatial indexing, point location queries, and range queries
- Efficient collision detection in two dimensions
- View frustum culling of terrain data
- Storing sparse data, such as a formatting information for a spreadsheet[11] or for some matrix calculations
- Solution of multidimensional fields (computational fluid dynamics, electromagnetism)
- Conway's Game of Life simulation program.[12]
- State estimation[13]
- Quadtrees are also used in the area of fractal image analysis
- Maximum disjoint sets

Image processing using quadtrees

Quadtrees, particularly the region quadtree, have lent themselves well to image processing applications. We will limit our discussion to binary image data, though region quadtrees and the image processing operations performed on them are just as suitable for colour images.[3][14]

Image Union/Intersection

One of the advantages of using quadtrees for image manipulation is that the set operations of union and intersection can be done simply and quickly.[3][15][16][17] [18] Given two binary images, the image union (also called *overlay*) produces an image wherein a pixel is black if either of the input images has a black pixel in the same location. That is, a pixel in the output image is white only when the corresponding pixel in *both* input images is white, otherwise the output pixel is black. Rather than do the operation pixel by pixel, we can compute the union more efficiently by leveraging the quadtree's ability to represent multiple pixels with a single node. For the purposes of discussion below, if a subtree contains both black and white pixels we will say that the root of that subtree is coloured grey.

The algorithm works by traversing the two input quadtrees (T_1 and T_2) while building the output quadtree T. Informally, the algorithm is as follows. Consider the nodes $v_1 \in T_1$ and $v_2 \in T_2$ corresponding to the same region in the images.

- If v_1 or v_2 is black, the corresponding node is created in T and is colored black. If only one of them is black and the other is gray, the gray node will contain a subtree underneath. This subtree need not be traversed.
- If $v1$ (respectively, $v2$) is white, $v2$ (respectively, $v1$) and the subtree underneath it (if any) is copied to T .
- If both $v1$ and $v2$ are gray, then the corresponding children of v1 and v2 are considered.

While this algorithm works, it does not by itself guarantee a minimally sized quadtree. For example, consider the result if we were to union a checkerboard (where every tile is a pixel) of size $2^k \times 2^k$ with its complement. The result is a giant black square which should be represented by a quadtree with just the root node (coloured black), but instead the algorithm produces a full 4-ary tree of depth k. To fix this, we perform a bottom-up traversal of the resulting quadtree where we check if the four children nodes have the same colour, in which case we replace their parent with a leaf of the same colour.[3]

The intersection of two images is almost the same algorithm. One way to think about the intersection of the two images is that we are doing a union with respect to the *white* pixels. As such, to perform the intersection we swap the mentions of black and white in the union algorithm.

Connected Component Labelling

Consider two neighbouring black pixels in a binary image. They are *adjacent* if they share a bounding horizontal or vertical edge. In general, two black pixels are *connected* if one can be reached from the other by moving only to adjacent pixels (i.e. there is a path of black pixels between them where each consecutive pair is adjacent). Each maximal set of connected black pixels is a *connected component*. Using the quadtree representation of images, Samet[19] showed how we can find and label these connected components in time proportional to the size of the quadtree.[3][20] This algorithm can also be used for polygon colouring.

The algorithm works in three steps:

- establish the adjacency relationships between black pixels
- process the equivalence relations from the first step to obtain one unique label for each connected component
- label the black pixels with the label associated with their connected component

To simplify the discussion, let us assume the children of a node in the quadtree follow the *Z*-order (SW, NW, SE, NE). Since we can count on this structure, for any cell we know how to navigate the quadtree to find the adjacent cells in the different levels of the hierarchy.

Step one is accomplished with a post-order traversal of the quadtree. For each black leaf v we look at the node or nodes representing cells that are Northern neighbours and Eastern neighbours (i.e. the Northern and Eastern cells that share edges with the cell of v). Since the tree is organized in *Z*-order, we have the invariant that the Southern and Western neighbours have already been taken care of and accounted for. Let the Northern or Eastern neighbour currently under consideration be u. If u represents black pixels:

- If only one of u or v has a label, assign that label to the other cell
- If neither of them have labels, create one and assign it to both of them
- If u and v have different labels, record this label equivalence and move on

Step two can be accomplished using the union-find data structure.[21] We start with each unique label as a separate set. For every equivalence relation noted in the first step, we union the corresponding sets. Afterwards, each distinct remaining set will be associated with a distinct connected component in the image.

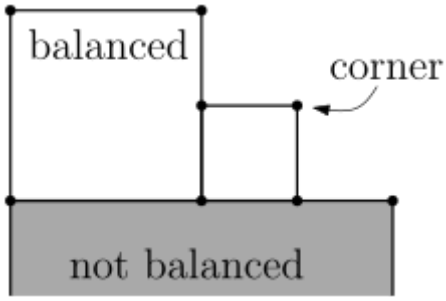
Step three performs another post-order traversal. This time, for each black node v we use the union-find's *find* operation (with the old label of v) to find and assign v its new label (associated with the connected component of which v is part).

Mesh generation using quadtrees

This section summarizes a chapter from a book by Har-Peled and de Berg et al.[22][23]

Mesh generation is essentially the triangulation of a point set for which further processing may be performed. As such, it is desirable for the resulting triangulation to have certain properties (like non-uniformity, triangles that are not "too skinny", large triangles in sparse areas and small triangles in dense ones, etc.) to make further processing quicker and less error-prone. Quadtrees built on the point set can be used to create meshes with these desired properties.

Consider a leaf of the quadtree and its corresponding cell v. We say v is *balanced* (for mesh generation) if the cell's sides are intersected by the corner points of neighbouring cells at most once on each side. This means that the quadtree levels of leaves adjacent to v differ by at most one from the level of v. When this is true for all leaves, we say the whole quadtree is balanced (for mesh generation).



A balanced leaf has at most one corner in a side.

Consider the cell v and the 5×5 neighbourhood of same-sized cells centred at v. We call this neighbourhood the *extended cluster*. We say the quadtree is *well-balanced* if it is balanced, and for every leaf u that contains a point of the point set, its extended cluster is also in the quadtree and the extended cluster contains no other point of the point set.

Creating the mesh is done as follows:

1. Build a quadtree on the input points.
2. Ensure the quadtree is balanced. For every leaf, if there is a neighbour that is too large, subdivide the neighbour. This is repeated until the tree is balanced. We also make sure that for a leaf with a point in it, the nodes for each leaf's extended cluster are in the tree.
3. For every leaf node v that contains a point, if the extended cluster contains another point, we further subdivide the tree and rebalance as necessary. If we needed to subdivide, for each child u of v we ensure the nodes of u's extended cluster are in the tree (and re-balance as required).
4. Repeat the previous step until the tree is well-balanced.
5. Transform the quadtree into a triangulation.

We consider the corner points of the tree cells as vertices in our triangulation. Before the transformation step we have a bunch of boxes with points in some of them. The transformation step is done in the following manner: for each point, warp the closest corner of its cell to meet it and triangulate the resulting four quadrangles to make "nice"

triangles (the interested reader is referred to chapter 12 of Har-Peled[22] for more details on what makes "nice" triangles).

The remaining squares are triangulated according to some simple rules. For each regular square (no points within and no corner points in its sides), introduce the diagonal. Note that due to the way in which we separated points with the well-balancing property, no square with a corner intersecting a side is one that was warped. As such, we can triangulate squares with intersecting corners as follows. If there is one intersected side, the square becomes three triangles by adding the long diagonals connecting the intersection with opposite corners. If there are four intersected sides, we split the square in half by adding an edge between two of the four intersections, and then connect these two endpoints to the remaining two intersection points. For the other squares, we introduce a point in the middle and connect it to all four corners of the square as well as each intersection point.

At the end of it all, we have a nice triangulated mesh of our point set built from a quadtree.

Pseudo code

The following pseudo code shows one means of implementing a quadtree which handles only points. There are other approaches available.

Prerequisites

It is assumed these structures are used.

```
// Simple coordinate object to represent points and vectors
struct XY
{
  float x;
  float y;

  function __construct(float _x, float _y) {...}
}

// Axis-aligned bounding box with half dimension and center
struct AABB
{
  XY center;
  float halfDimension;

  function __construct(XY center, float halfDimension) {...}
  function containsPoint(XY point) {...}
  function intersectsAABB(AABB other) {...}
}
```

QuadTree class

This class represents both one quad tree and the node where it is rooted.

```
class QuadTree
{
  // Arbitrary constant to indicate how many elements can be stored in this quad tree node
  constant int QT_NODE_CAPACITY = 4;

  // Axis-aligned bounding box stored as a center with half-dimensions
  // to represent the boundaries of this quad tree
  AABB boundary;

  // Points in this quad tree node
  Array of XY [size = QT_NODE_CAPACITY] points;

  // Children
  QuadTree* northWest;
  QuadTree* northEast;
  QuadTree* southWest;
```

```
  QuadTree* southEast;

  // Methods
  function __construct(AABB _boundary) {...}
  function insert(XY p) {...}
  function subdivide() {...} // create four children that fully divide this quad into four quads of equal area
  function queryRange(AABB range) {...}
}
```

Insertion

The following method inserts a point into the appropriate quad of a quadtree, splitting if necessary.

```
class QuadTree
{
  ...

  // Insert a point into the QuadTree
  function insert(XY p)
  {
    // Ignore objects that do not belong in this quad tree
    if (!boundary.containsPoint(p))
      return false; // object cannot be added

    // If there is space in this quad tree, add the object here
    if (points.size < QT_NODE_CAPACITY)
    {
      points.append(p);
      return true;
    }

    // Otherwise, subdivide and then add the point to whichever node will accept it
    if (northWest == null)
      subdivide();

    if (northWest->insert(p)) return true;
    if (northEast->insert(p)) return true;
    if (southWest->insert(p)) return true;
    if (southEast->insert(p)) return true;

    // Otherwise, the point cannot be inserted for some unknown reason (this should never happen)
    return false;
  }
}
```

Query range

The following method finds all points contained within a range.

```
class QuadTree
{
  ...

  // Find all points that appear within a range
  function queryRange(AABB range)
  {
    // Prepare an array of results
    Array of XY pointsInRange;

    // Automatically abort if the range does not intersect this quad
    if (!boundary.intersectsAABB(range))
      return pointsInRange; // empty list

    // Check objects at this quad level
    for (int p = 0; p < points.size; p++)
    {
      if (range.containsPoint(points[p]))
        pointsInRange.append(points[p]);
    }

    // Terminate here, if there are no children
    if (northWest == null)
```

```
      return pointsInRange;

    // Otherwise, add the points from the children
    pointsInRange.appendArray(northWest->queryRange(range));
    pointsInRange.appendArray(northEast->queryRange(range));
    pointsInRange.appendArray(southWest->queryRange(range));
    pointsInRange.appendArray(southEast->queryRange(range));

    return pointsInRange;
  }
}
```

See also

- Binary space partitioning
- Kd-tree
- Octree
- R-tree
- UB-tree
- Spatial database
- Subpaving
- Z-order curve

References

Surveys by Aluru[3] and Samet[20][14] give a nice overview of quadtrees.

Notes

1. Milan Sonka, Vaclav Hlavac, Roger Boyle. "Image Processing, Analysis, and Machine Vision" (https://books.google.com/books?id=DcETCgAAQBAJ). 2014. p. 108-109.
2. Finkel, R. A.; Bentley, J. L. (1974). "Quad Trees A Data Structure for Retrieval on Composite Keys". *Acta Informatica*. Springer-Verlag. **4**: 1–9.
3. Aluru, S. (2004). "Quadtrees and octrees". In D. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC. pp. 19–1 — 19–26. ISBN 9781584884354.
4. Orenstein, J. A. (1982). "Multidimensional tries used for associative searching". *Information Processing Letters*. Elsevier. 14 (4): 150–157.
5. Samet, H. (1984). "The quadtree and related hierarchical data structures". *ACM Computing Surveys (CSUR)*. ACM. 16 (2): 187–260.
6. Warnock, J. E. (1969). "A hidden surface algorithm for computer generated halftone pictures". *Computer Science Department, University of Utah*. TR 4-15.
7. Schneier, M. (1981). "Two hierarchical linear feature representations: edge pyramids and edge quadtrees". *Computer Graphics and Image Processing*. Elsevier. 17 (3): 211–224.
8. Hanan Samet and Robert Webber. "Storing a Collection of Polygons Using Quadtrees". *ACM Transactions on Graphics* July 1985: 182-222. *InfoLAB*. Web. 23 March 2012
9. Nelson, R. C.; Samet, H. (1986). "A consistent hierarchical representation for vector data". *ACM SIGGRAPH Computer Graphics*. 20 (4): 197–206.
10. Har-Peled, S. (2011). "Quadtrees - Hierarchical Grids". *Geometric approximation algorithms*. Mathematical Surveys and Monographs Vol. 173, American mathematical society.
11. Sestoft, Peter (2014). *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press. pp. 60–63. ISBN 9780262526647.
12. Tomas G. Rokicki (2006-04-01). "An Algorithm for Compressing Space and Time" (http://www.ddj.com/hpc-high-performance-computing/184406478). Retrieved 2009-05-20.
13. Henning Eberhardt, Vesa Klumpp, Uwe D. Hanebeck, *Density Trees for Efficient Nonlinear State Estimation*, Proceedings of the 13th International Conference on Information Fusion, Edinburgh, United Kingdom, July, 2010.

14. Samet, H. (1989). "Hierarchical spatial data structures". *Symposium on Large Spatial Databases*: 191–212.
15. Hunter, G. M. (1978). *Efficient Computation and Data Structures for Graphics*. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University.
16. Hunter, G. M.; Steiglitz, K. (1979). "Operations on images using quad trees". *IEEE Transactions on Pattern Analysis and Machine Intelligence*. **(2)**: 145–153.
17. Schneier, M. (1981). "Calculations of geometric properties using quadtrees". *Computer Graphics and Image Processing*. 16 (3): 296–302.
18. Mehta, Dinesh (2007). *Handbook of Data Structures and Applications*. Chapman and Hall/CRC Press. p. 397.
19. Samet, H. (1981). "Connected component labeling using quadtrees". *Journal of the ACM (JACM)*. 28 (3): 487–501.
20. Samet, H. (1988). "An overview of quadtrees, octrees, and related hierarchical data structures". In Earnshaw, R. A. *Theoretical Foundations of Computer Graphics and CAD*. Springer-Verlag. pp. 51–68.
21. Tarjan, R. E. (1975). "Efficiency of a good but not linear set union algorithm". *Journal of the ACM (JACM)*. 22 (2): 215–225.
22. Har-Peled, S. (2011). "Good Triangulations and Meshing". *Geometric approximation algorithms*. Mathematical Surveys and Monographs Vol. 173, American mathematical society.
23. de Berg, M.; Cheong, O.; van Kreveld, M.; Overmars, M. H. (2008). "Quadtrees Non-Uniform Mesh Generation". *Computational Geometry Algorithms and Applications* (3rd ed.). Springer-Verlag.

General references

1. Raphael Finkel and J.L. Bentley (1974). "Quad Trees: A Data Structure for Retrieval on Composite Keys". *Acta Informatica*. **4** (1): 1–9. doi:10.1007/BF00288933 (https://doi.org/10.1007%2FBF00288933).
2. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised ed.). Springer-Verlag. ISBN 3-540-65620-0. Chapter 14: Quadtrees: pp. 291–306.
3. Samet, Hanan; Webber, Robert (July 1985). "Storing a Collection of Polygons Using Quadtrees" (http://infolab.usc.edu/csci585/Spring2008/den_ar/p182-samet.pdf) (PDF). Retrieved 23 March 2012.

External links

- A discussion of the Quadtree and an application (http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html)
- Considerable discussion and demonstrations of Spatial Indexing (https://web.archive.org/web/20120204170335/http://homepages.ge.ucl.ac.uk/~mhaklay/java.htm)
- Java Implementation (https://github.com/varunpant/Quadtree/)
- Java tutorial (http://javaprogrammernotes.blogspot.ru/2015/01/why-algorithms-matter-quad-tree-example.html)
- C++ Implementation of a Quadtree used for spatial indexing of triangles (https://sourceforge.net/projects/quadtreedemo/)
- Objective-C implementation of QuadTree used for GPS clustering (http://robots.thoughtbot.com/how-to-handle-large-amounts-of-data-on-maps/)
- SquareLanguage (http://squarelanguage.sourceforge.net/)
- Working demonstration of Quadtree algorithm in Javascript (http://jsfiddle.net/sombra2eternity/vNS7m/)
- MIT licensed Quadtree library in Javascript (https://github.com/sombra2eternity/quadtree)