Point Region Quadtree
Requirements & Design Document

Jens Åkerblom

University of Helsinki
Faculty of Science

Course
Joint Project Group: Data Structure Project

Instructor
Henning Lübbers

Spring 2011

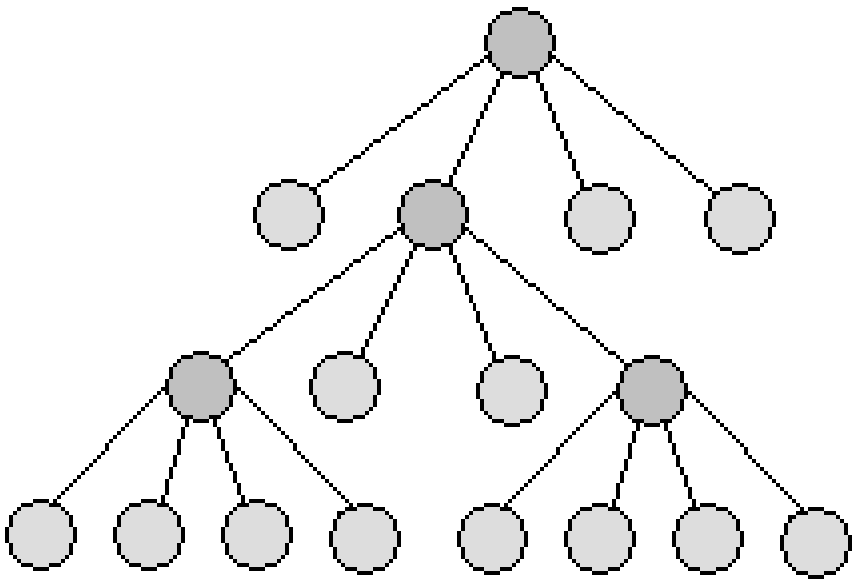
Contents

1 Introduction

This is the requirements and design document for my Point Region Quadtree. This is a part of the course "Joint Project Group: Data Structure Project" supervised by Henning Lübbers in spring 2011.

This document will describe all parts of quadtree that are of special interest, that is the fundamentals of the tree and how it reacts to insertions, extractions, removal and updating of points.

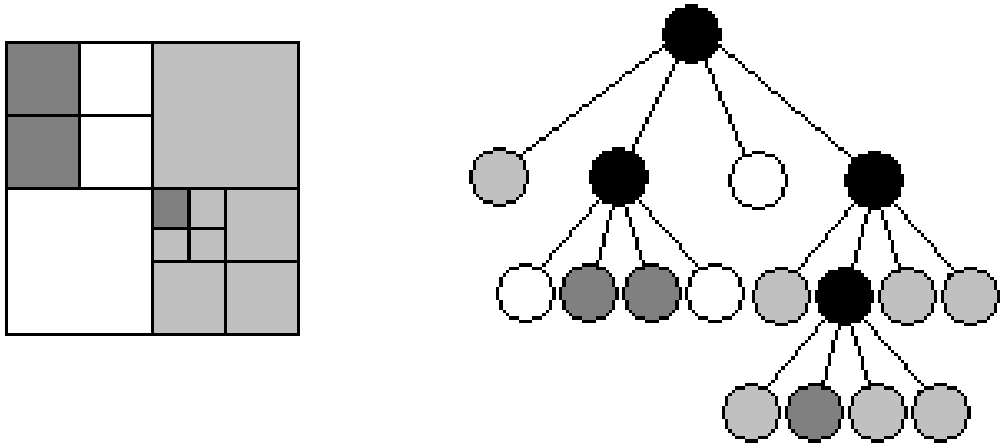
2 Quadtree



A quadtree is a tree where each internal node has exactly four children. The image above describes a generic quadtree, the internal nodes are the dark gray colored ones and the leaves are bright gray.

A quadtree can be used to partition a rectangular 2 dimensional region into smaller sub regions. These quadtrees are called Region Quadtrees.

2.1 Region Quadtree

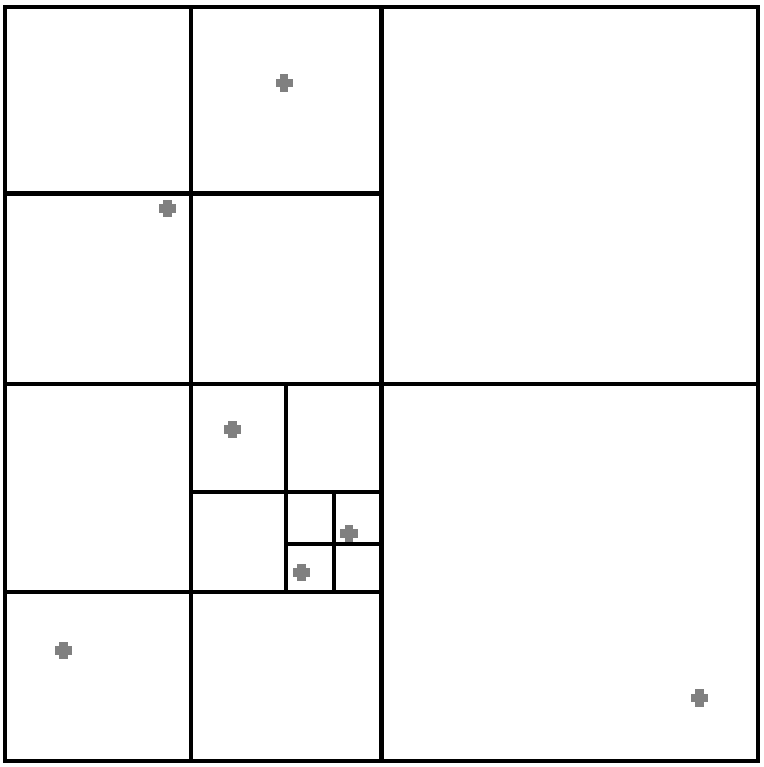


To the left in the image we have a 2 dimensional array (8x8) of colors. In the example use of a region quadtree above, we partition the array by having each subregion subdivided until the node region contains only one color. The interleaved nodes usually don't contain any information about the data, they act only as links to the next depth of the tree.

The subregions of a region are named North East, North West, South West and South East depending on their internal position in the scene, in this example (and in my implementation) this is the ordering of the child nodes.

Note that in the above example, the array position of a color can be determined by its position in the tree. If this is not the case, the tree is called a prefix tree (the data is independent of the position in the tree). This is the case in Point Region Quadtrees.

2.1.1 Point Region Quadtree



A point region quadtree is a region quadtree storing point data (like the above image). The original region will be subdivided enough times to include at most one point in each region, or until a maximum amount of subdivisions have occurred. This way we can have two points "infinitively" close without endless subdivision.

3 Principles of my Point Region Quadtree

The definition of a quadtree is quite generic, it doesn't implicitly require specific rules for subdivisions. Therefore I list the required behavior here in three categories.

- *Use lowest possible resolution.* The tree will contain as large subregions as possible to store at most one point per subregion, up to a specific depth explicitly set at creation.
- *Fast extraction of points in area.* Extracting points from tree, by giving an area or a point, should only be dependent on the size of the branches in the regions inside the area, not the size of the whole tree (as a random tree-traversal method would be).

- *Point consistency.* A point must be saved in a region that contains the point. It is up to the user to notify any changes so the tree can be updated.

From these principles we can derive some algorithms.

4 Algorithms

This section lists the most important algorithms implemented with pseudo-code.

4.1 Locating a point

To locate a point in the tree we can use the third principle. From the root, we traverse to the child having a region containing that point. We continue doing this until we have reached a leaf node, that leaf should contain that point.

In pseudo-code:

```
Quadtree_node findNodeContaining(Point2D p)
    Quadtree_node curNode = root
    while curNode has children do
        for child in curNode do
            if child.pointIsInRegion(p.x, p.y) then
                curNode = child
                break
    if curNode.hasPoint(p) then
        return curNode
    else
        error 'Could not find point'
```

I call this a *directed search* since it will traverse the tree from the root to the searched for node without visiting other branches (as for example a Breadth First Search would do). We see that this method follows the second principle.

4.2 Adding and removing points

Adding and removing points may alter the structure of the tree. When adding a point we might have to subdivide the tree, when removing a point we might have to merge regions (according to the first principle).

To remove a point, with pseudo-code:

```
removePoint(Point2D p)
    Quadtree_node curNode = getLeafAt(p.x, p.y)
    curNode.remove(p)
    while curNode.numberOfPointsInRegion <= 1 do
        curNode = curNode.getParent()
        if curNode == NULL then
            return //Tree is now empty, curNode is parent of root.
        if curNode.numberOfPointsInBranch <= 1 then
            curNode.mergeBranch()
```

The algorithm is quite straight forward; firstly find where the point is saved and get the node holding it, secondly remove the point from bucket and thirdly update the tree starting at current node moving upward (thus evaluating all nodes that are affected by removal).

To add a point, with pseudo-code:

```
addPoint(Point2D p)
    Quadtree_node curNode = getLeafAt(p.x, p.y)
    curNode.add(p)
    stack<Quadtree_node> divideStack
    divideStack.push_back(curNode)
    while divideStack.size() > 0 do
        curNode = divideStack.pop_back()
        if curNode.numberOfPointsInRegion > 1 and
           curNode.depth < maxDepth then
            curNode.subdivide()
            for child in curNode do
                divideStack.push_back(child)
```

Adding a point is a little bit more complex. Since we do not know what child might need further subdividing, we must check them all (in removePoint we only needed to consider parent for merging). Notice that under all circumstances the removePoint and addPoint methods must act as inverses.

4.3 Updating and undirected search

When moving a point, the tree might need to be updated. Since the point's last known position is by the tree unknown it is faster to remove the point from the tree, move the point and then add the point back to the tree. To find the last position, we use a Depth First search.

In pseudo-code:

```
Quadtree_node treeFind(Point2D p)
    stack<Quadtree_node> searchStack
    searchStack.push_back(root)
    while searchStack.size() > 0 do
        Quadtree_node curNode = searchStack.pop_back()
        if not curNode.hasChildren then
            if curNode.hasPoint(p) then
                return curNode
        else
            for child in curNode do
                searchStack.push_back(child)
    error 'Could not find point'
```

To update the point, in pseudo-code:

```
update(Point2D p)
    Quadtree_node curNode = getLeafAt(p.x, p.y)
    if curNode.hasPoint(p) then
        return  //p has not moved out of region, no updating needed
    Quadtree_node oldNode = treeFind(p)
    oldNode.remove(p)
    addPoint(p)  //see above for pseudo-code
    updateBranch(oldNode) //Almost identical code as in removePoint.
```

Notice that we can use addPoint to add the point again, but not removePoint to remove it. This is because removePoint assumes that the tree is consistent.

4.4 Subdividing and merging

Structurally in the tree, subdividing is the process to make a leaf node to an interleaved node, and merging is the inverse.

To subdivide a leaf, create four new nodes and assign them their corresponding subregions. Next read all points saved in the leaf and assign each point to one of the newly created leaves. Finally change the type of the leaf node to be an interleaved node and deallocate the storage that originally contained the points.

The merging algorithm is a little bit more complex, this is because a node isn't merged but rather a (partial) branch is.

The merging is recursive, in pseudo-code:

```
merge(Quadtree_node node)
    if node.isLeaf then
        return
    for child in node do
        if not child.isLeaf then
            merge(child)
    //All children are now leaves.
    PointArray tempStorage
    for child in node do
        tempStorage.append(child.points)
        delete child
    node.isLeaf = true
    node.points = tempStorage
```

The first actual transition from interleaved node to leaf takes place at the first encountered parent to a leaf node. Then we travel backwards with recursion until we hit the caller node.

4.5 Retrieval of points

The Point Region Quadtree is fast at getting points based on a position or area. There are two methods to retrieve data, getContentAt and getContentInArea.

The getContentAt should be used when you have an approximate position of the point. The method uses a directional search to find the leaf node containing the position and then it returns all points saved there.

The getContentInArea should be used when you want all points of a specific area. As with the previous method, this too uses a directional search. This is done by creating a search stack and two lists, one will contain the leaves that are partially inside the area and the other that will contain the leaves completely inside. Then you start traversing the search stack, originally containing only the root, and adding the children of the top of the stack to: the search stack if the child is interleaved and at least partially inside the area, the partial leaf list if the child is a node partially inside the area or to the complete child list if the child is completely inside the area. When the search list is empty we begin sorting out all points from the partial leaf list by checking whether or not the points are inside the area. After that we read the lists to get all points inside the area.

5 Use of Point Region Quadtree

You want to create a GPS map with some locations. Lets say you store the locations as an array of base class Point2D. Most often you will only need to display a small portion of all locations and you don't want to manually check whether or not a location is inside the viewport, since this would require you to check every location (in the world!). In this case a Point Region Quadtree would do the trick, the points are then interpreted as locations and getting the points inside the viewport is quite fast and not dependent on the total amount of points. The algorithms above show all the necessary methods.