Explanation to why algo.js is better than the quadTree approach

My Algorithm:

https://github.com/SoumitraAgarwal/Ideas-GSOC-Helikar2015Labs/blob/master/Ideas/algo.js

The approach: First of all, in 3 dimensional space, the algorithm we use is the following:

```
var closestNode =function()
{
    var hollowSphere=new THREE.sphere(material);
    var min=net THREE.geometry(450,450,450);
    var sphereGeometry = new THREE.SphereGeometry(0, 0, 0);
    var sphereMaterial = new THREE.MeshLambertMaterial({color: 0x7777ff});
    var sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
    // position the sphere
    sphere.position.x = mouseX;
    sphere.position.y = mouseY;
    sphere.position.z = mouseZ;
    var max=450;
    var min=0;
    while(hollowSphere.intersect(geometry)!=2)
    {
        hollowsphere.setRadius((min+max)/2);
        if(hollowSphere.intersect(geometry)>2)
        {
            min=(min+max)/2;
        }
        else
        {
            max=(min+max)/2;
        }
    }
    var finalRadius=hollowSphere.getRadius();
}
var intersect= function()
{
    raycaster.setFromCamera( mouse, camera );
    var INTERSECTED=null;
    var intersects = raycaster.intersectObjects( scene.children );
    if ( intersects.length > 0 )
    {
        if ( INTERSECTED != intersects[ 0 ].object )
        {
            if ( INTERSECTED ) INTERSECTED.material.program = programStroke;
            INTERSECTED = intersects[ 0 ].object;
            INTERSECTED.material.program = programFill;
        }
    }
    else
    {
        if ( INTERSECTED ) INTERSECTED.material.program = programStroke;
        INTERSECTED = null;
    }
    return INTERSECTED;
}
```

For each step, we use a binary search like algorithm to create a sphere of varying sizes. Thus for 3-Dimensional and 2 Dimensional space, the number of steps required are a function of log(n) due to its analogy. For the complexity of each step, we study what actually happens in each step. Each step utilizes the function given below. The function named intersect is the major area of concern here.

```
var intersect= function()
{
    raycaster.setFromCamera( mouse, camera );
    var INTERSECTED=null;
    var intersects = raycaster.intersectObjects( scene.children );
    if ( intersects.length > 0 )
    {
        if ( INTERSECTED != intersects[ 0 ].object )
        {
            if ( INTERSECTED ) INTERSECTED.material.program = programStroke;
            INTERSECTED = intersects[ 0 ].object;
            INTERSECTED.material.program = programFill;
        }
    }
    else
    {
        if ( INTERSECTED ) INTERSECTED.material.program = programStroke;
        INTERSECTED = null;
    }
    return INTERSECTED;
}
```

Here we see that it directly gets the number of intersect length. Though the number of steps may change every time depending on n, us limiting the number of steps by using the while condition hollowSphere.intersect(geometry)!=2, we stop the maximum number of checks to 3. As most of the present day machines can use dynamic memory allocation and thus as in three.js we can utilise on the structural hierarchy of bounding boxes., this process runs a max of 3 times. Thus time taken can be written as :(t)

O(1*log(n))<t<O(3*log(n))

Thus, finally what we get is that, the time we require for this to execute is O(c*log(n)) which can be written as O(log(n)).The average time can be calculated by taking all the cases. Let us assume a sphere of thickness dx at a distance x from the centre. The max distance of the closest particle from the

sphere = floor[M-x] , up to the point where x< M, the average being the same for both the latter and earlier parts combined.(M being the max distance).

Thus maximum average =

(Integral from 0 to x)(floor(M-x)/x)< (Integral from 0 to x)((M-x)/x)

=Mlog(n)-M

Thus even the work average case (where all particles are assumed to be at the max distance) is O(log(n))

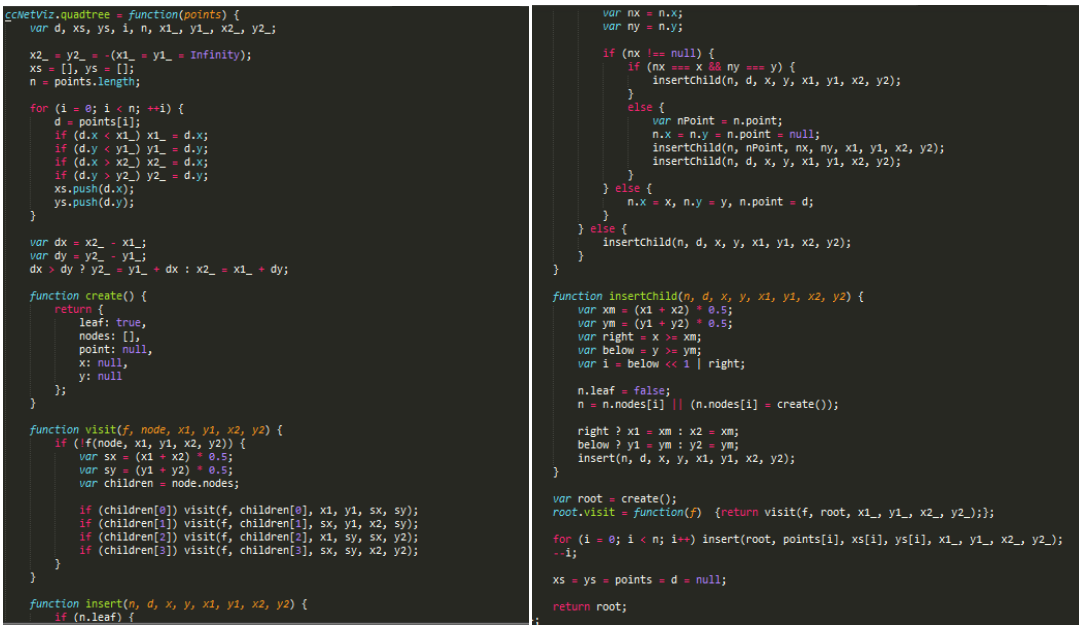
Whereas if all particles are placed at M/2, the average becomes O(log(n/2)).

Thus the approximate average for the function is (O(log(n))+O(log(n/2)))/2=O(log(pow(n,2))/2))/2=O(log(n/2)) which is in most real cases better than O(log(n)) for no leading constants.

THE QuadTree Algorithm:

http://helikarlab.org/svn/ccnetviz/trunk/js/quadTree.js

The approach: What I realise from the approach is that, for the time being, it is meant for 2-Dimenstional structures, which is not an issue at all. The algorithm being:



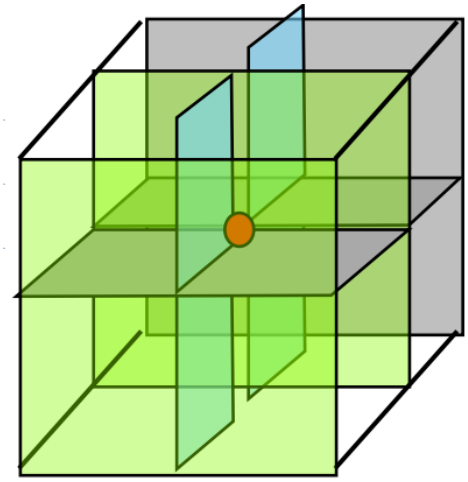
```
ccNetViz.quadtree = function(points) {
    var d, xs, ys, i, n, x1_, y1_, x2_, y2_;

    x2_ = y2_ = -(x1_ = y1_ = Infinity);
    xs = [], ys = [];
    n = points.length;

    for (i = 0; i < n; ++i) {
        d = points[i];
        if (d.x < x1_) x1_ = d.x;
        if (d.y < y1_) y1_ = d.y;
        if (d.x > x2_) x2_ = d.x;
        if (d.y > y2_) y2_ = d.y;
        xs.push(d.x);
        ys.push(d.y);
    }

    var dx = x2_ - x1_;
    var dy = y2_ - y1_;
    dx > dy ? y2_ = y1_ + dx : x2_ = x1_ + dy;

    function create() {
        return {
            leaf: true,
            nodes: [],
            point: null,
            x: null,
            y: null
        };
    }

    function visit(f, node, x1, y1, x2, y2) {
        if (!f(node, x1, y1, x2, y2)) {
            var sx = (x1 + x2) * 0.5;
            var sy = (y1 + y2) * 0.5;
            var children = node.nodes;

            if (children[0]) visit(f, children[0], x1, y1, sx, sy);
            if (children[1]) visit(f, children[1], sx, y1, x2, sy);
            if (children[2]) visit(f, children[2], x1, sy, sx, y2);
            if (children[3]) visit(f, children[3], sx, sy, x2, y2);
        }
    }

    function insert(n, d, x, y, x1, y1, x2, y2) {
        if (n.leaf) {
            var nx = n.x;
            var ny = n.y;

            if (nx !== null) {
                if (nx === x && ny === y) {
                    insertChild(n, d, x, y, x1, y1, x2, y2);
                }
                else {
                    var nPoint = n.point;
                    n.x = n.y = n.point = null;
                    insertChild(n, nPoint, nx, ny, x1, y1, x2, y2);
                    insertChild(n, d, x, y, x1, y1, x2, y2);
                }
            } else {
                n.x = x, n.y = y, n.point = d;
            }
        } else {
            insertChild(n, d, x, y, x1, y1, x2, y2);
        }
    }

    function insertChild(n, d, x, y, x1, y1, x2, y2) {
        var xm = (x1 + x2) * 0.5;
        var ym = (y1 + y2) * 0.5;
        var right = x >= xm;
        var below = y >= ym;
        var i = below << 1 | right;

        n.leaf = false;
        n = n.nodes[i] || (n.nodes[i] = create());

        right ? x1 = xm : x2 = xm;
        below ? y1 = ym : y2 = ym;
        insert(n, d, x, y, x1, y1, x2, y2);
    }

    var root = create();
    root.visit = function(f)  {return visit(f, root, x1_, y1_, x2_, y2_);};

    for (i = 0; i < n; i++) insert(root, points[i], xs[i], ys[i], x1_, y1_, x2_, y2_);
    --i;

    xs = ys = points = d = null;

    return root;
};
```

I would like to elaborate on this by what I know about QuadTree in general,

Given a query point q, find the deepest node in which q lies, by performing binary search on the depth, each time checking whether the node in depth i containing q exists in the tree

Query time: O(logh) where h is the maximal depth in the tree.

This is for a 2 dimensional structure.



Generalizing to large dimensions uses a lot of space. - octatree = QuadTree in 3-D (each node has 8 pointers)

In d dimensions, each node has 2d pointers! d = 20 => nodes will ~ 1 million children

This makes our web application slow and cumbersome. Also regular crashes cannot be prevented then. Though a detailed study may show that the average time for the algorithm in 2-D is O(logh) as well.

Thus our algorithm beats in on space as well as average time complexity. Also it is easier to understand.
