Quadtrees for Boids algorithm

For a personal project I want to recreate Boids algorithm, I decided to start on quadtrees.

Quadtrees is a special partitioning algorithm that greatly reduces the amount of checks between different nodes from – if its every node to each other checking every node has a time complexity of O(N^2) where implementing a quadtree has a general time complexity of O(N×Log(N) + N) which is much better.

Quadtrees is a recursive algorithm where it takes the nodes. Finds where they are in the space and depending on how many nodes are allowed in the area it will either leave it as is or split the space into quarters and checks again. The tree is one node that splits into 4 which in turn splits in to 4 more and continues however long is needed. This reduces the searching time by querying the tree for an area, then the program moves down the tree by location until it gets the area or areas that intersects with the query and returns the nodes within. If all of the nodes need checked and returned, for example if all of the nodes are in the same place, this then reduces the efficiency of the algorithm back down to O(N^2).

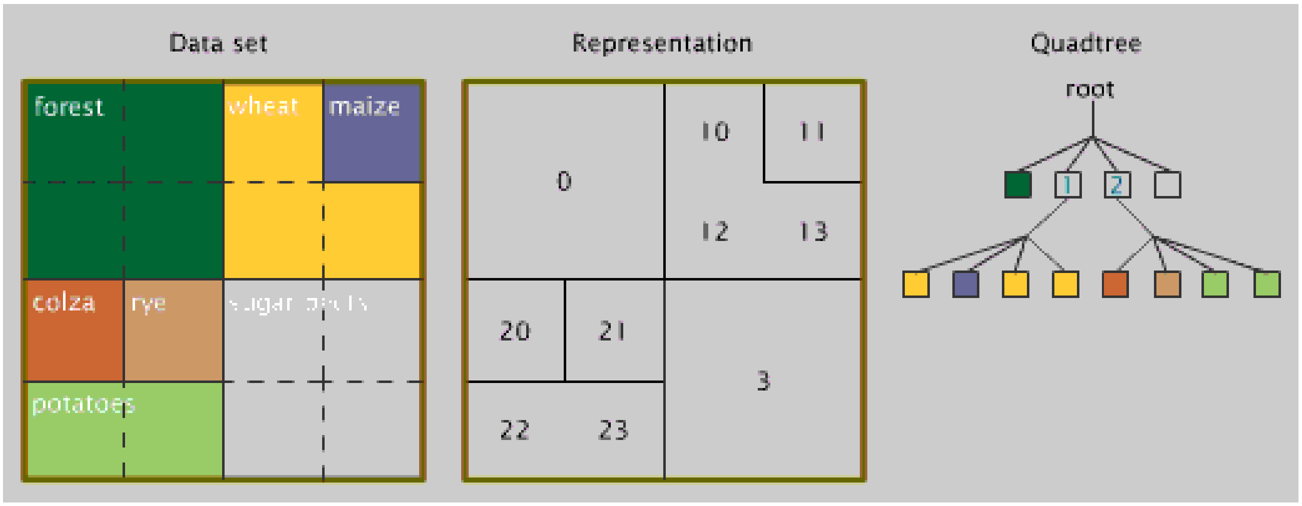
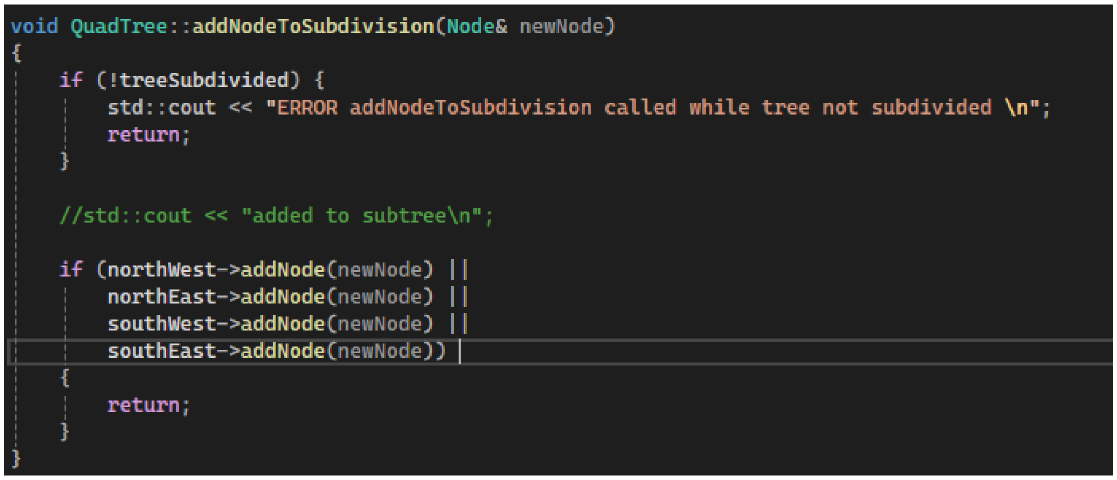
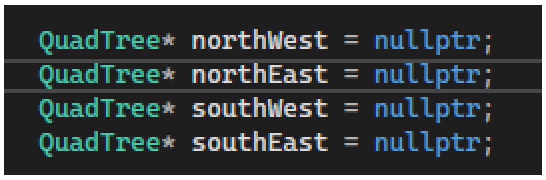


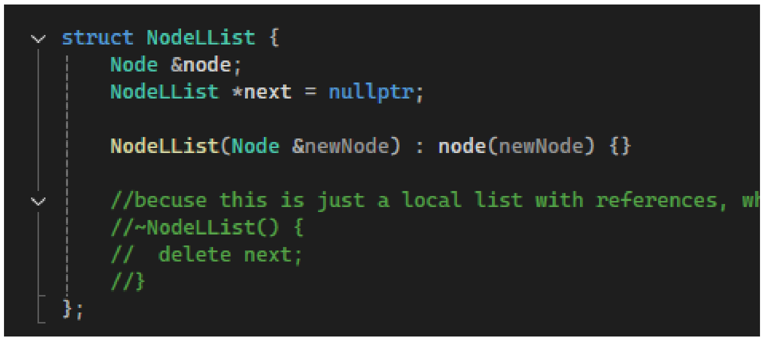
Image from http://www.gitta.info/SpatPartitio/en/html/RegDecomp_learningObject3.html

I implemented it by functions and recursion initializing the object and setting the boundaries, we then create the recursive structure which bases the general structure of the octree class, where when the tree is split at one point, it creates four more octree objects.

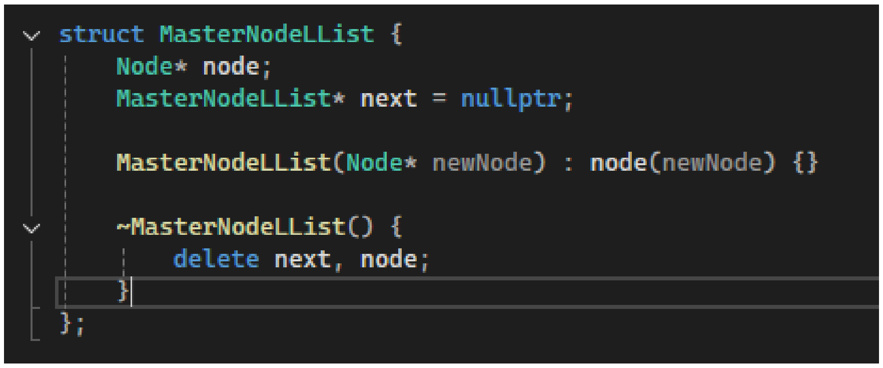

```cpp
void QuadTree::addNodeToSubdivision(Node& newNode)
{
    if (!treeSubdivided) {
        std::cout << "ERROR addNodeToSubdivision called while tree not subdivided \n";
        return;
    }

    //std::cout << "added to subtree\n";

    if (northWest->addNode(newNode) ||
        northEast->addNode(newNode) ||
        southWest->addNode(newNode) ||
        southEast->addNode(newNode))
    {
        return;
    }
}
```


```cpp
QuadTree* northWest = nullptr;
QuadTree* northEast = nullptr;
QuadTree* southWest = nullptr;
QuadTree* southEast = nullptr;
```

I created the ability to hold nodes, nodes are only held in the lowest most tree node, and they are held in a linked list structure.


```cpp
struct NodeLList {
    Node &node;
    NodeLList *next = nullptr;

    NodeLList(Node &newNode) : node(newNode) {}

    //becuse this is just a local list with references, wh
    //~NodeLList() {
    //  delete next;
    //}
};
```

Nodes are created and kept track of in a master list outside of the tree, they cannot be held in the tree because the tree is rebuilt every frame.

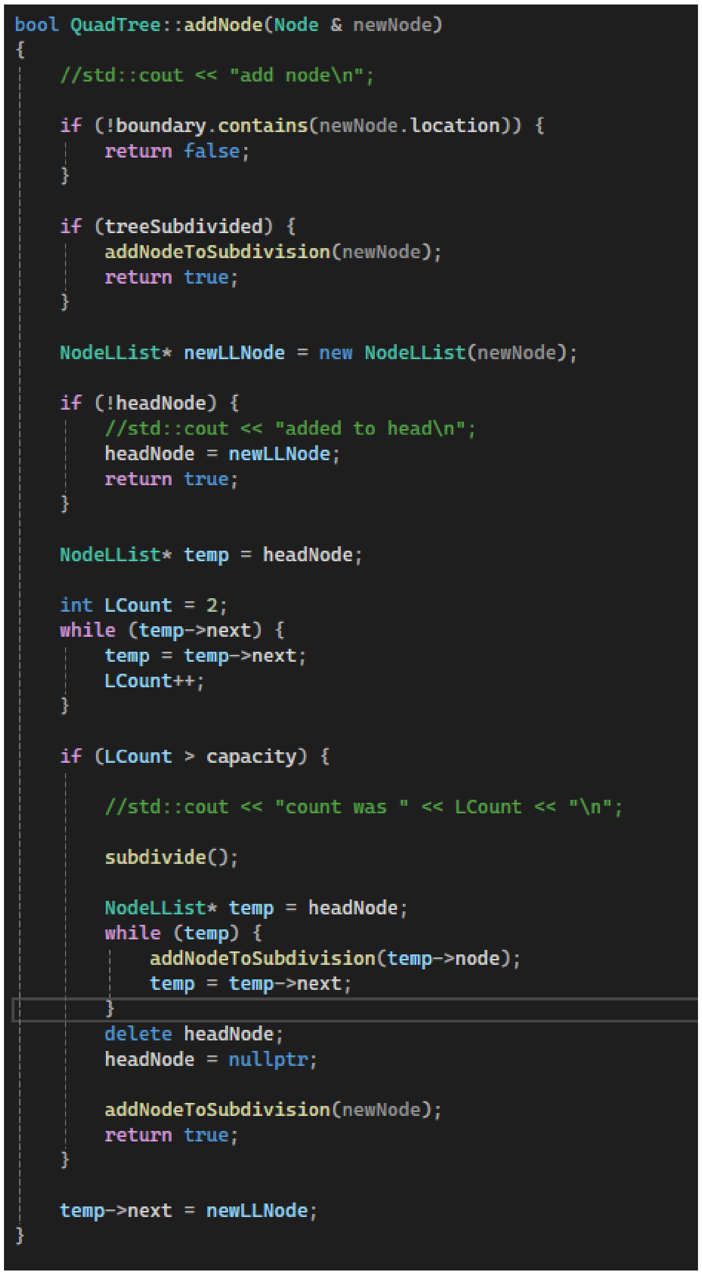

```cpp
struct MasterNodeLList {
    Node* node;
    MasterNodeLList* next = nullptr;

    MasterNodeLList(Node* newNode) : node(newNode) {}

    ~MasterNodeLList() {
        delete next, node;
    }
};
```

We now need to create the ability to add nodes, which will also include some of the subdivision logic.

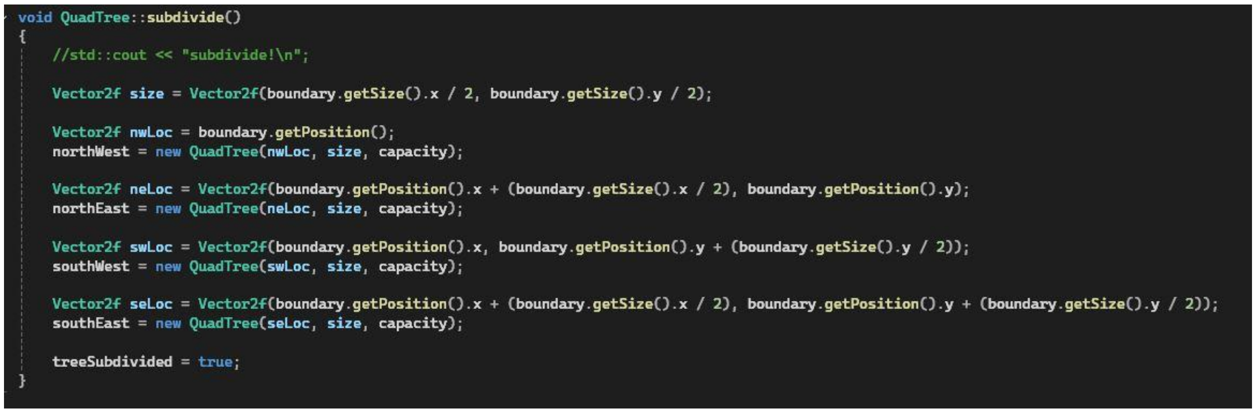
It first checks if it's within the bounds, then checks if it is subdivided from that point, if so move down the tree. There is also a return here because on the way back out of the recursion that is the end of the action. If it is the end of the tree add it to the list, to the head node or the next node of the linked list. If it has hit the capacity, then subdivide and add it to the subdivision.

The node is originally added to the main list to keep track of it then added to the quadtree.

I have also been practicing separating out if statements and implementing early outs for readability and efficiancy.

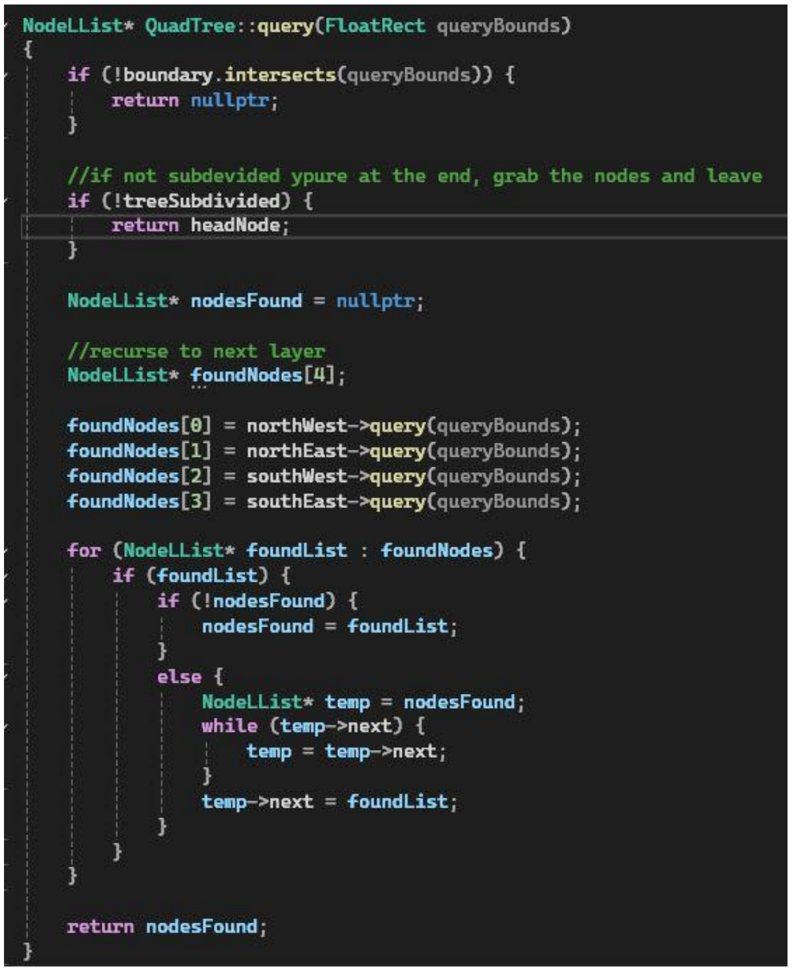


```cpp
bool QuadTree::addNode(Node & newNode)
{
    //std::cout << "add node\n";

    if (!boundary.contains(newNode.location)) {
        return false;
    }

    if (treeSubdivided) {
        addNodeToSubdivision(newNode);
        return true;
    }

    NodeLList* newLLNode = new NodeLList(newNode);

    if (!headNode) {
        //std::cout << "added to head\n";
        headNode = newLLNode;
        return true;
    }

    NodeLList* temp = headNode;

    int LCount = 2;
    while (temp->next) {
        temp = temp->next;
        LCount++;
    }

    if (LCount > capacity) {

        //std::cout << "count was " << LCount << "\n";

        subdivide();

        NodeLList* temp = headNode;
        while (temp) {
            addNodeToSubdivision(temp->node);
            temp = temp->next;
        }
        delete headNode;
        headNode = nullptr;

        addNodeToSubdivision(newNode);
        return true;
    }

    temp->next = newLLNode;
}
```



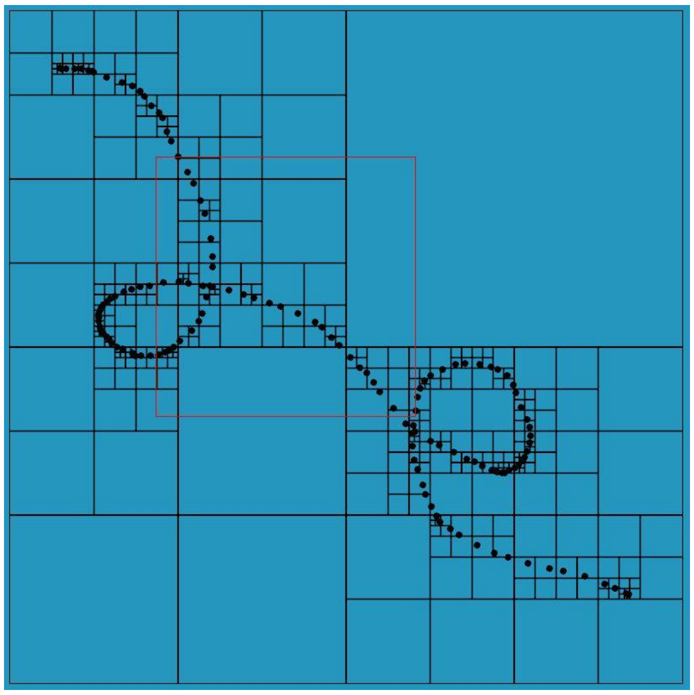
```cpp
void QuadTree::subdivide()
{
    //std::cout << "subdivide!\n";

    Vector2f size = Vector2f(boundary.getSize().x / 2, boundary.getSize().y / 2);

    Vector2f nwLoc = boundary.getPosition();
    northWest = new QuadTree(nwLoc, size, capacity);

    Vector2f neLoc = Vector2f(boundary.getPosition().x + (boundary.getSize().x / 2), boundary.getPosition().y);
    northEast = new QuadTree(neLoc, size, capacity);

    Vector2f swLoc = Vector2f(boundary.getPosition().x, boundary.getPosition().y + (boundary.getSize().y / 2));
    southWest = new QuadTree(swLoc, size, capacity);

    Vector2f seLoc = Vector2f(boundary.getPosition().x + (boundary.getSize().x / 2), boundary.getPosition().y + (boundary.getSize().y / 2));
    southEast = new QuadTree(seLoc, size, capacity);

    treeSubdivided = true;
}
```

Now the tree needs to be able to be queried. This involves having an area and the quadtree checks if the area intersects the current piece of the tree and if it does, it then moves into the next four subtrees and does the same thing, if it is at the end, it returns the nodes within. It also checks all four parts, which return a list of nodes, we add the nodes found together and then return that, until it is returned to the original query.



```cpp
NodeLList* QuadTree::query(FloatRect queryBounds)
{
    if (!boundary.intersects(queryBounds)) {
        return nullptr;
    }

    //if not subdevided ypure at the end, grab the nodes and leave
    if (!treeSubdivided) {
        return headNode;
    }

    NodeLList* nodesFound = nullptr;

    //recurse to next layer
    NodeLList* foundNodes[4];

    foundNodes[0] = northWest->query(queryBounds);
    foundNodes[1] = northEast->query(queryBounds);
    foundNodes[2] = southWest->query(queryBounds);
    foundNodes[3] = southEast->query(queryBounds);

    for (NodeLList* foundList : foundNodes) {
        if (foundList) {
            if (!nodesFound) {
                nodesFound = foundList;
            }
            else {
                NodeLList* temp = nodesFound;
                while (temp->next) {
                    temp = temp->next;
                }
                temp->next = foundList;
            }
        }
    }

    return nodesFound;
}
```

I have a simple function to add new nodes with the mouse, this lets me test the system properly. I also have functions to render all the parts of the oct tree and query, where all of this will be invisible in the full boids game.



Reflection

I created this project to improve my understanding of algorithms and to create something I find fascinating. If I was to do it again I would use better variable names where node is generally used as a piece of the tree, more comments and use less pointers where I can because there are parts of the code causing data leaks and I now understand that pointers are generally slower because they are accessing the heap instead of the stack, and with an algorithm like this the point of it is efficiency and speed. One area I want to improve is very much this one and I intend to do so by restarting and making octrees which are the 3D version of quadtrees.