Spatial Acceleration Structure
Quadtrees

Munteanu Mihai

June 2022

Abstract

When it comes to the navigation of a robot or game AI, real-time search is largely used to plan the next motions. In order to approach something we must first determine where that something is and/or whether it is visible or not. For example, in a RTS (real time strategy) game, where there are many entities on the screen and beyond it, we would like to determine how many and which of them are actually on the screen, in real-time, to only render and update the visible ones, gaining performance. This paper addresses real-time search of entities, presenting a faster way, in most cases, using a spatial acceleration structure called quad tree, and comparing it to a standard linear approach.

1 Introduction

[1]Quadtrees are a structure that encapsulate and divides a two-dimensional space into adaptable cells. Just like a binary tree, the quadtree is a tree structure where every non-leaf node has four children. In other words, [2]they partition a 2D space by recursively sub-dividing it into 4 quadrants or regions. This regions may have arbitrary shapes or simply be a square. Quadtrees may be classified according to the type of data represented such as areas, points, lines and curves. Figure 1.1 represents a set of 2D points mapped within different levels of the quadtree. The quadtree we're going to implement is represented by areas, respectively rectangles and it's going to store other rectangles, just like in figure 1.2, the green rectangle is an object stored within the quadtree. Each and every rectangle the quadtree divides into will be treated as a quadtree on it's own.

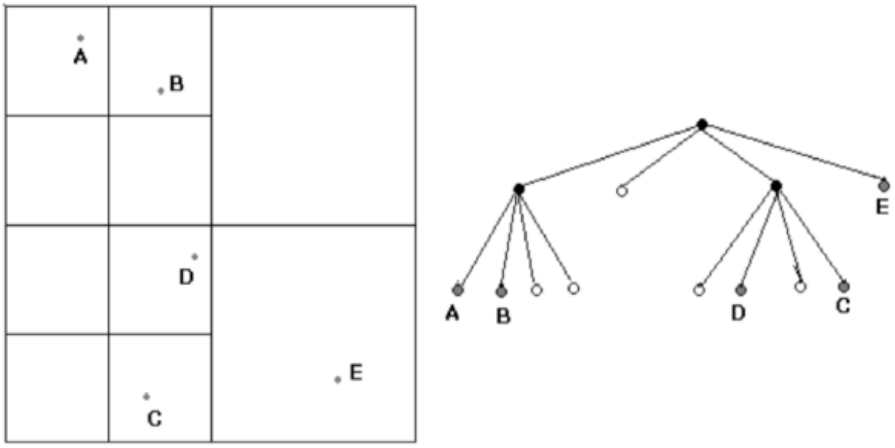

Figure 1.1: Data points stored within a Quadtree

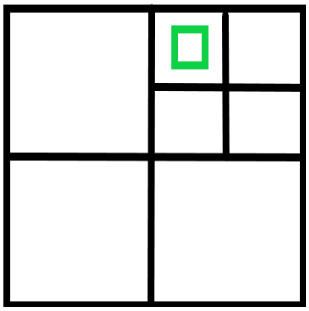

Figure 1.2: Rectangle stored within the Quadtree

2 Quadtree implementation

[3]We will first discuss about the guts of the quadtree. Because the quadtree is represented by areas, it'll be best to define a separate data type called rectangle, that will hold the position of the rectangle, represented by the top left x and y coordinates, and the area, for this we can just hold the width and height of the rectangle. The first field of the quadtree is going to be the depth, which will be initialized with 0, we'll also define a separte macro/constant, outside the quadtree, arbitrary initialized based to our needs, which will hold the maximum depth the quadtree is allowed to go, we will need this for the insert function. The next field is the root of the quadtree, which is also a rectangle, if we take a look back to the figure 1.2, we will define it's root as being the first (largest) rectangle. Next, an array of 4 elements holding the 4 rectangles the actual quadtree divides into called rChilds, followed by another array of quadtrees, called pChilds, representing the actual childs of the quadtree. And the last one will be a container in which will hold the objects inserted in the quadtree.

2.1 Insert function

As we said before, the item we're going to insert into the quadtree must have an area and a position of it's own, or we can pass them as parameters. First we will recursively check if any of the 4 childs wholly contains the item's area, if affirmative, we'll check if the max depth has been reached, then we'll make sure that the childs exists, if not we will create them, and after that we'll just recursively call the insert function on the child that passed this checks and then exit the function. If in any call of the insert function all 4 children failed to pass the first condition, that means that the item belongs to the current quadtree (the quadtree on which the function is called), so right after the for loop we will insert the item into the quadtree's container. To be mentioned, only one of the calls will get pass through the for loop and actually insert the item, this happens when the item's area overlaps the area of the quadtree's children or when the maxium depth has been reached, just like it happens in figure 2.

Pseudocode:

Algorithm 1 Insert

```
for i ← 0; i < 4; i ← i + 1 do
    if rChilds[i].contains(itemSize) then
        if depth + 1 < MaxDEPTH then
            if !pChilds[i] then
                pChilds[i].create
            end if
            pChild[i].insert(item, itemSize)
            return
        end if
    end if
end for
pItems.push_back(item)
```

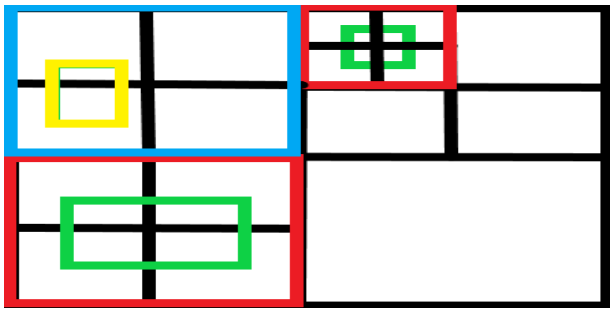

Figure 2.1: The green and yellow rectangles belong to the red, respectively blue quadtrees

2.2 Items function

This function will return a list of all the items stored in the quadtree. It works by just adding the items of the quadtree in a list and then recursively add its children's items into the same list, so, perhaps we should pass a reference to that list.
Pseudocode:

Algorithm 2 items(itemList)

```
for item in pItems do
    itemList.push_back(item)
end for
for i ← 0; i < 4; i ← i + 1 do
    if pChild[i] exists then
        pChild[i].items(itemList)
    end if
end for
```

2.3 Search function

This is the main function, it gets as input an area and it returns a list containing all the items stored in that area. In the beginning it will check if the area of the items in the quadtree overlap with the given area, if true, we'll add the item to the list, next for its children, we'll check if the area wholly contains the area of the child, if true, we call the items function on the child, else we'll check if the child overlaps with given area, if so, the search function is called recursively on the child. When the child overlaps with the area we are in a case similar to the one ilustrated in figure 2.2, where the items within the red area are visible.
Pseudocode:

Algorithm 3 search(rArea, itemList)

```
for item in pItems do
    if rArea.overlaps(item.area) then
        itemList.push_back(item)
    end if
end for
for i ← 0; i < 4; i ← i + 1 do
    if pChild[i] exists then
        if rArea.contains(rChild[i] then
            pChild[i].items(itemList)
        end if
        if rChild[i].overlaps(rArea) then
            pChild[i].search(rArea, itemList)
        end if
    end if
end for
```

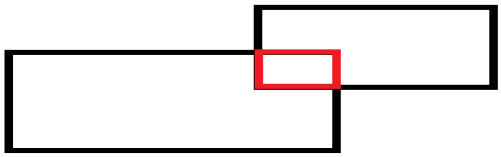

Figure 2.2: Rectangle overlap

3 Comparison with the linear approach

Let's imagine having lots of object on a screen, around a million, and we would like to determine how many of them are visible on the screen. With a linear approach we would simply iterate over the elements and check if it's visible or not, on the other hand, with a quadtree we'll just call the search function passing the area of the screen and it will return a list of all the item contained within that area. The quadtree really excels, offering much better performance than the linear method when the number of items visible on the screen is small. For example, when we have under 500/1000000 items on the screen, the linear approach offer a time that counts in milliseconds, whereas the quadtree's time is measured in microseconds. The more items we have on the screen the smaller the difference between linear's and quadtree's time gets,

moreover when it exceeds a certain number of items quadtree's performance gets worse than the linear one, from the test i've done, after 70000 elements the linear approach gets better, but we're almost never gonna have so many items at once on the screen, so quadtree's speed is indeed impressive.

[3]However, things are actually not bright as they seem. You probably noticed that we're copying things around quite a bit in the search functions, this is not a problem when it comes a smaller objects like those we presented the implementation with, but usually we're going to have objects with lots of properties and they might get a little bit heavy and harder to copy, so it's not ideal. Also, in real cases the visible objects will have fields that needs to be updated often, with a simple, basic container that wouldn't be a problem, we'd just iterate over it and do the operations, but with the quadtree we must first execute quite a complicated set of recursive searches and copy the items into a list and then deliver that list, iterate it and do what we have to do. This means doing transformations and operations on the quadtree is extraordinary extremely costly. We can actually solve a pretty much completely this problems by creating some sort of wrapper for the quadtree, treating it just like a container. We define another data type that uses the quadtree and has a separate basic, simple container that has a begin() an and an end(). Instead of storing the items in the quadtree, we'll store them in the container and insert pointers to the items from the container, along with the area, in the quadtree. Pointers are relatively small objects which are very easy to copy, and now we can iterate very easy through all the items, the only downside is that the search functions return a list of pointers to the container, which is still not as bad as before and it's acceptable because we gain so much in return.

4 Conclusion

Quadtrees are a very interesting and efficient data structure. We've presented an area represented quadtree, implemented recursively and compared it's powers to a basic linear approach of a search.

References

[1] Blog, website educative.io 6/23/2022
https://www.educative.io/answers/what-is-a-quadtree-how-is-it-used-in-location-based-services

[2] ProQuest thesis, document preview 6/23/2022
https://www.proquest.com/openview/f503ab1b6e76b0d103fb56015df07a13/1?pq-origsite=gscholarcbl=18750

[3] Youtube video, author: javidx9, title: "Quirky Quad Trees Part1: Static Spatial Acceleration 6/23/2022
https://www.youtube.com/watch?v=ASAowY6yJIIt=2219s