Optimized search engine

Xavier Olivenza Busquets

What is a search engine for?

As the name implies, the search engine searches for certain types of elements in an area of the space, sticking to a rule or rules, such as "search for all elements found at a radius X of this point" , "Search for the element closest to this point", ... Starting from these rules of searching, there are different ways to sort the space to perform the search.

Video games where it is used or can be used

For example, in a game like Age of Empires, if you tell a villager to "cut down trees to get wood," the villager from the position where he is will look for the nearest tree and when he finds it with the search engine, a pathfinding algorithm will be performed to go to the tree. Once it is finished, if the order has not been revoked, you can search for a nearby tree on a radius with the search engine and retrace the pathfinding to this one, if it does not find any, you can tell it to look for the nearest building and shelter, again with the search engine, but instead of looking for tree entities, look for building entities.

Another example may be tower defense games. There are enemies that are advancing in a route, and the towers of defense look for with the engine, enemies that are in range of attack, if so, they attack. Once you have eliminated the target, look for another enemy in range, if there is one, attack, if not, keep doing checks on each frame or every X frames in the search for another enemy.

First choice of how to look for something in space, brute force

If we want to find a type of entity of the world in a limited area/range, the brute force method would make us check if each entity of that type is within the area/range. Therefore, if we have 200 types of that entity, we will have to do 200 checks. Therefore, brute force is slow, since you have to do as many checks as entities are evaluated.

What happens if we group and order the entities?, so we will only have to check how many groups are within the area/range, depends on how we group and order, we can significantly reduce the number of checks. That is what is achieved with space partitioning algorithms.

Space partitioning

Space partitioning in geometry is the process of dividing a space into two or more subsets which do not overlap, if we start with the space A and subdivide it in the subsets B and C, B does not intersect with C and vice versa. With this type of division, any point can be identified and located in a particular region. For the space partitioning in its subsets, planes are used or in larger dimensions, hyperplanes.

Space partitioning algorithms tend to be hierarchical, that is, they use trees to sort themselves out. The root node of the tree is divided into an X number of children and recursively each child can be re-divided, thereby generating a tree.

Most common partition data structures:

· BSP Trees

· Quadtrees

· Octrees

· K-dimensional trees

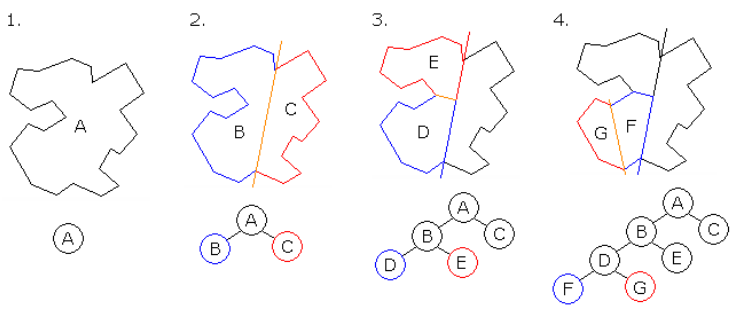
· R-Trees

Binary Space Partitioning (BSP)

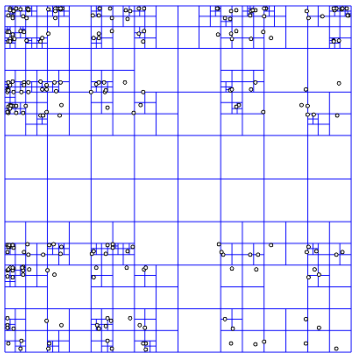
It is a process that can be seen as a generalization of other spatial partitioning structures such as k-d trees and quadtrees. The recursive division of the BSP tree space generates a tree data structure that always cuts each node into two parts, an action that is performed based on requirements that limit how the node is divided. The planes that divide the space can have any orientation, they are not necessarily aligned with the axes.

Binary space partitioning arose from the need for computer graphics to quickly draw three-dimensional scenes composed of polygons. Applying BSP trees yields a rapid method of classifying polygons with respect to a given point of view. A disadvantage of the binary space partition is that generating a BSP tree can take a long time. It is therefore usually performed once on static geometry, such as a pre-calculation step, before rendering or other real-time operations in a scene. The cost of building a BSP tree makes it difficult and inefficient to directly implement moving objects in a tree.

BSP trees can be used in 3D games, such as first person shooters. Some of the 3D video game engines that use BSP trees are the Doom Engine (id Tech 1) and Quake Engine and their descendants.



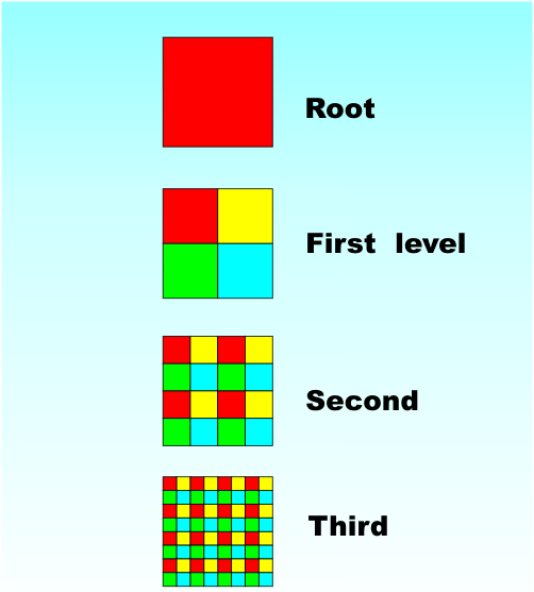
Quadtree



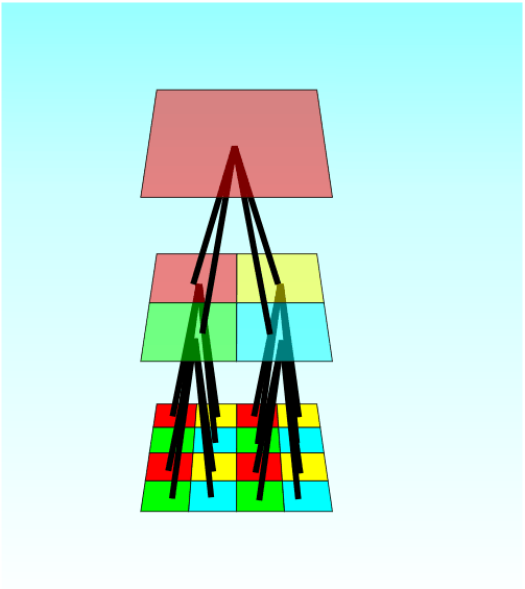
Starting from the idea of space partition and that the binary partition of space divides each node of the tree into two, it should be easy to deduce the basic operation of a quadtree, this tree data structure recursively divides the space by cutting each node into four subsets that do not overlap, usually squares or rectangles. Generally used in 2D worlds.

Each node has a maximum capacity, when it is filled, it is divided into four more subsets these will be filled, until maximum capacity is reached and the process is repeated again.

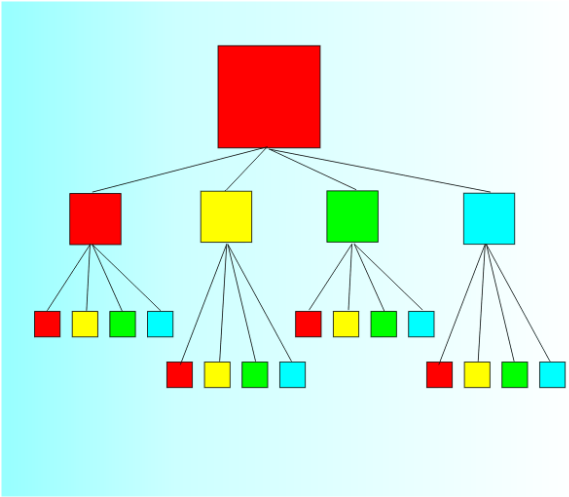
Quadtree subdivisions:



Quadtree 3D tree:

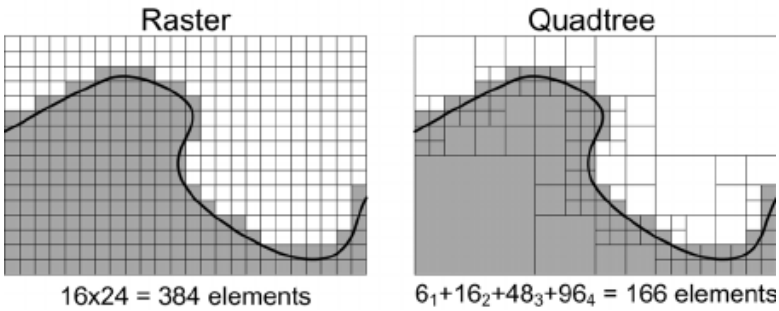


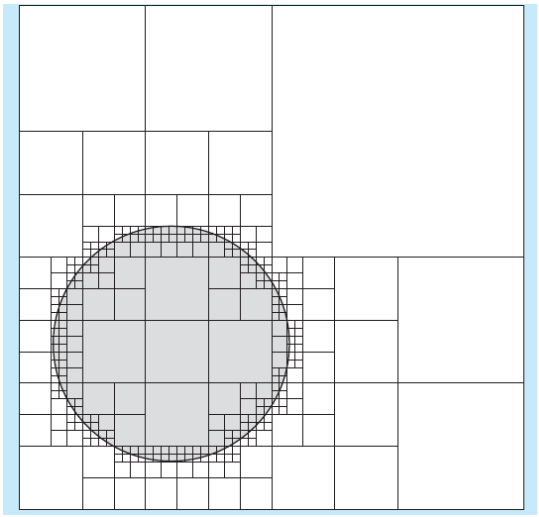
Quadtree 2D tree:



Four common uses of quadtrees

·Image processing





· Bitmap and its quadtree compressed representation

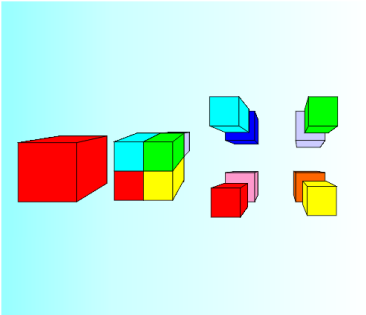
· Efficient collision detection in two dimensions

· Can be used to optimize the camera culling. An approximation to the maps by tiles, you could have the entire map as the root node and divide the space by adding each of the tiles to the tree, and you can optimize what is painted and what not, depending on whether it is on screen.

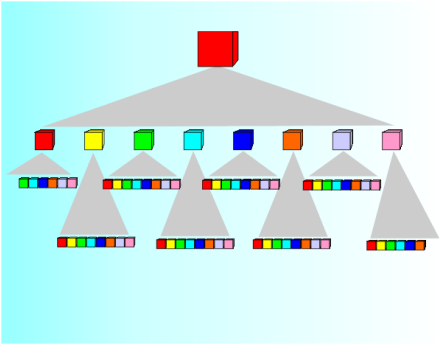
Octree

From the quadtree it should be easy to deduce the octree. The quadtree divides each node into four children and octree into eight, but in this there is a nuance, the quadtree is used more in 2D and octree is its analogue in 3D, so it is used in 3D graphics and video game engines. The octree divides the parallelepiped of the root node into four sub-parallelepipeds contained in the volume of the root node and not included in each other. This division has different criteria depending on whether you store in the octree, or how the implementation was done.

Octree subdivisions:

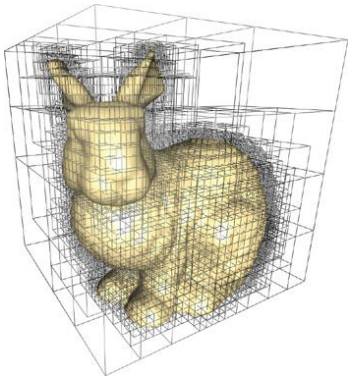


Octree 2D tree:



Four common uses of octrees

· 3D Graphics



· Efficient collision detection in three dimensions

· Culling / Hidden Surface Determination (HSR) / Occlusion Culling (OC) / Visible Surface Determination (VSD)

·Finite element analysis

Why are these algorithms faster than brute force?

The speed of these algorithms is given by their organization in tree of the spatial partitions. When calling the method that is in charge of finding the entities of the space that are within a certain area/range, the first thing to do is to look at whether or not the node is inside the area/range, if it is not, it aborts the whole branch of subdivisions. Since the node is not in area/range, neither will be its subdivisions nor its content, so many unnecessary checks are saved. If the node is above the area/range, it looks at whether its content is or not, and then its subdivisions are analysed, looking at whether or not each subnode is within the area/range.

In summary, the way forward in the analysis of the tree is based on looking if the node is in the area/range, if it is, it continues to go into the tree and see if the contents of the node is within the area/range, if no, cut off all that branch.

How does my quadtree implementation work

My Quadtree only accepts iPoint (points), it is up to you to modify it so that it accepts other things, or to make a template, thus you have to touch code and thus you learn how the code works, reason why it will be easier to you to modify things if you want.
In this section I explain how to use the quadtree I have implemented, just by taking the SDLQuadtree.h/.cpp and putting it in your project, I do not explain the code inside these files, for more information, enter the files, are full of comments next to each line of code to be able to understand what each thing does.
In the code provided what I explain below on how to use my SDLQuadtree, is in the files j1Scene.h/.cpp, also with some comments to identify the different zones and that is done there

In the .h of the module where the Quadtree is going to be implemented, save the following:

`Quadtree* Point_quadtree = nullptr;` Pointer to the Quadtree that we will use
`SDL_Rect Quadtree_area_search;` Area where we will look for entities

In the .cpp file:

In the function that initializes the module, in my case Start (), create the new Quadtree, save it in the pointer, and the search area.

```
Point_quadtree = new Quadtree(Quadtree_area);
Quadtree_area_search = { 500,200,100,200 };
```

Add the entities that will handle, in my case in the same Start () after creating the Quadtree.

```
Point_quadtree->Insert(&iPoint( x,y ));
```

When you need to do the search, in my case in Update () when you press "S", simply perform the search using the pointer, and passing to CollectCandidates a vector to store all the entities in range and the search area.

```
Point_quadtree->CollectCandidates->(Points_in_range_quadtree_search,
Quadtree_area_search);
```

Links

Space Partitioning

http://gameprogrammingpatterns.com/spatial-partition.html

Quadtree implementations

https://github.com/Lectem/YAQ

http://codereview.stackexchange.com/questions/143955/quadtree-c-implementation

Octree

http://www.gamasutra.com/view/feature/131625/octree_partitioning_techniques.php

Quadtree and Octree

http://www.i-programmer.info/programming/theory/1679-quadtrees-and-octrees.html

K-D Tree Implementation links

https://rosettacode.org/wiki/K-d_tree

https://github.com/orangejulius/kdtree

https://github.com/gvd/kdtree

K-D Tree VS Quadtree

http://stackoverflow.com/questions/13487953/difference-between-quadtree-and-kd-tree

http://gamedev.stackexchange.com/questions/87138/fully-dynamic-kd-tree-vs-quadtree

Quadtrees and Hilbert Curves

http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves