Understanding spatial partitioning optimization systems such as Oct-trees & Quad-trees

Following American psychological association guidelines

Noaman Khalil

SAE Institute Dubai

Abstract

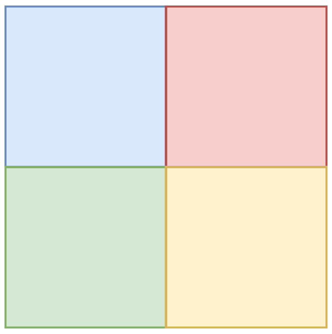
The goal of this paper is to demonstrate the understanding of complex spatial partitioning techniques used in games in order to optimise a theoretical scene with the goal of setting up a theoretical system which would allow for larger and more efficient scenes within large scale games that require large amounts of processing. The paper will also cover what it is along with the comparison between Quadtrees & Octrees alongside some common uses of the data structure within the games industry.

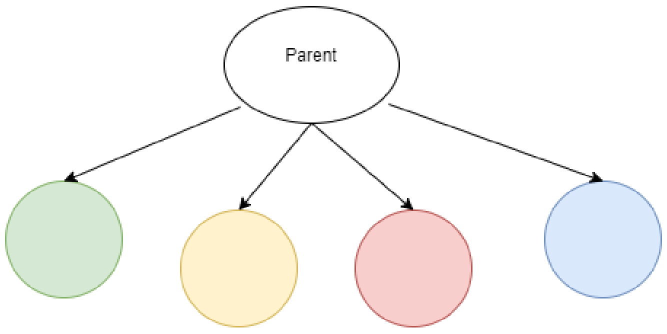
1 – Introduction

Quad trees and octrees are spatial data structures based off tree-based partitioning method that are used in advanced spatial partitioning of a region of space into four or eight equally sized quadrants or octants such as cells .Starting from the root, cells are subdivided into smaller cells under certain conditions, such as when a cell contains an object boundary such as a region in a quad tree or when a cell contains more than a specified number of objects such as point quad tree .Compared to methods that do not partition space or that the partition the space uniformly , quad trees and octrees can reduce the amount of processing required to represent objects such as images and improve the time needed for querying and processing data such as calculations being done in collision detection . (Li/University of Canterbury & Mukundan/University of Canterbury, 2013)

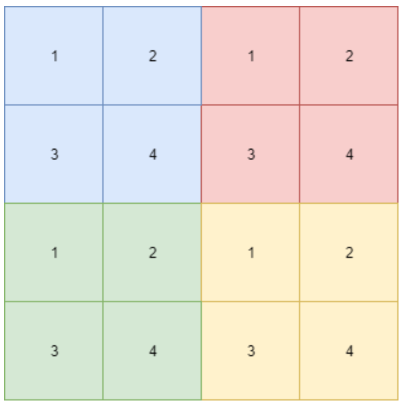
2 –Quadtrees

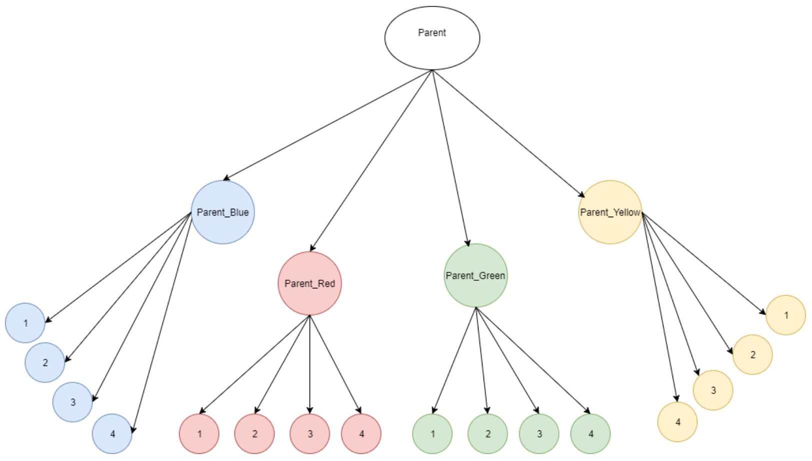
By definition a quad tree is an off tree-based data structure in which each internal node has exactly four children. In a quad tree, each node represents a bounding container covering some part of the space being indexed, with the root node covering the entire area. How a quad tree is structured internally can be demonstrated in the images below.





The representation above is only what the first node will look like, after which the system will sub dived each child into four more nodes and in doing so make them parent nodes which can be further seen in the images below.



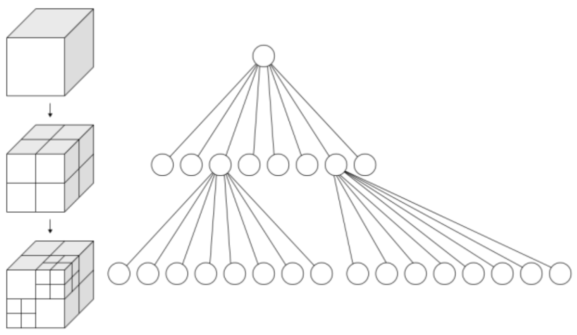


Inserting data into a quad tree is very simple, starting at the root which quadrant your point occupies after which recurs to that node and repeat, until you find a leaf node. Then, add your point to that nodes list of points. If the list exceeds some pre-determined maximum number of elements, split the node, and move the points into the correct sub nodes.

To query a quad tree, starting at the root, examine each child node, and check if it intersects the area being queried for. If it does, recurs into that child node. Whenever you encounter a leaf node examine each entry to see if it intersects with the query area, and return it if it does.

2 –Octrees

Octrees are very similar to quad trees apart from two main differences wherein the first, the octree data structure is designed for a three dimensional space unlike the quad tree data structure which is designed to be used in a two dimensional space. The second is that the Oct tree data structure as the name denotes subdivides the grid into eight children in contrast to the Quadtrees which only does four. The Octree diagram below shows the representation of what it would look like. The subdivisions of the cube are called octants.



(, n.d.)

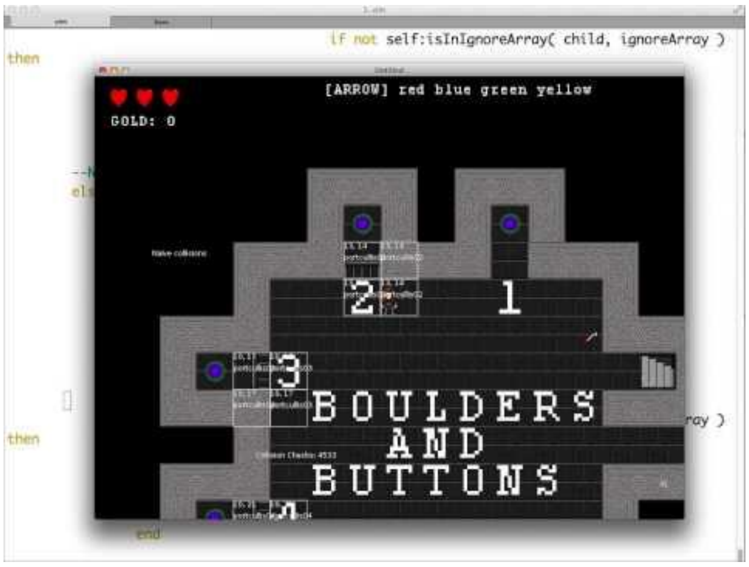
4 – Common uses

The quad tree has some common uses such as image processing, generating meshes, spatial indexing using point /range queries & efficient collision detection in two/ three dimensional space. The Octree structure is commonly used for Collison detection in 3-dimentional games and it is also used to optimize the calculations needed for the collisions , as calculating multiple collisions in a three dimensional space can be affect the games performance and overall user experience of the game .

5- The problem

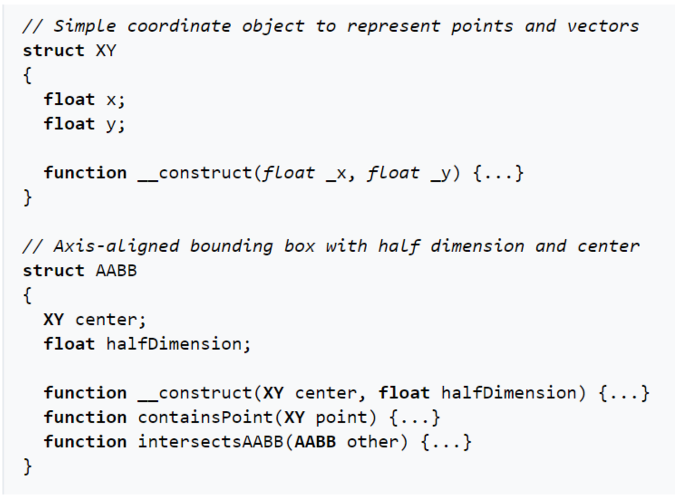
In many games the goal is to run the game at 60 frames per second, but since there are so many collision detections running in the background the framerate can often be much lower than the goal of 60 frames per second minimum. To demonstrate the problem I will be using a real world example wherein the developer's game was constantly doing collision checks per frame which resulted in 4000 or more collision checks per frame, this yielded a frame rate of 30 frames per second on some levels.

After the spatial portioning was implemented the game was getting around 60 frames per second with around 100 collision checks per frame , this essentially made the game run at a higher framerate by reducing the calculations to certain "buckets " as this is the analogy used by the developer to explain spatial portioning . (Hardcrawler, 2014)



```
if not self:isInIgnoreArray( child, ignoreArray )
then
ray )
then
end
```

6-Pseudo code

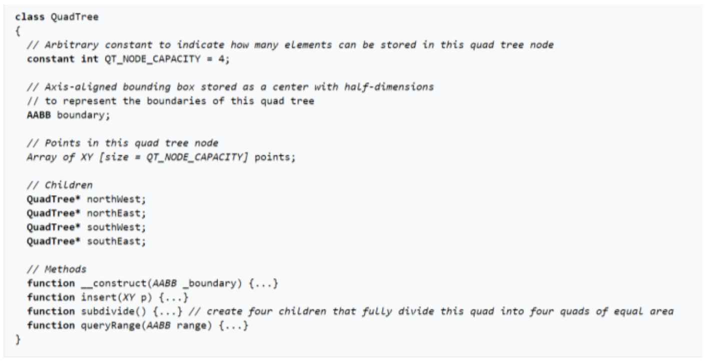


```
// Simple coordinate object to represent points and vectors
struct XY
{
  float x;
  float y;

  function __construct(float _x, float _y) {...}
}

// Axis-aligned bounding box with half dimension and center
struct AABB
{
  XY center;
  float halfDimension;

  function __construct(XY center, float halfDimension) {...}
  function containsPoint(XY point) {...}
  function intersectsAABB(AABB other) {...}
}
```

For the pseudo code to be explained I have taken the example from Wikipedia where in each point is represented by a two dimensional co-ordinate system, it is assumed the following structure is used.

("Quadtree," 2017)

The code also creates a bounding box which is used to create boundaries for the subdivision to occur. The strut AABB is the code to create the bounding box. It is to be noted that the bounding box is the outer area from which the subdivision. The X & Y center is used to determine the lines along which the first horizontal and vertical divisions are made.

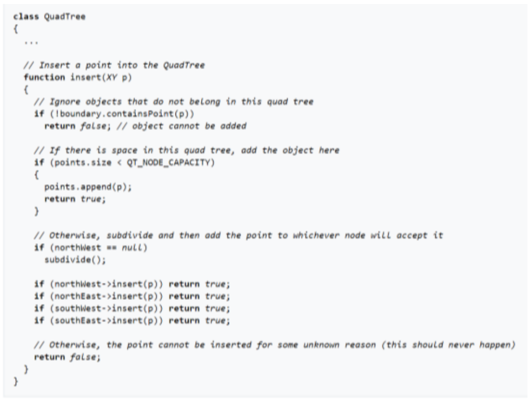


```
class QuadTree
{
  // Arbitrary constant to indicate how many elements can be stored in this quad tree node
  constant int QT_NODE_CAPACITY = 4;

  // Axis-aligned bounding box stored as a center with half-dimensions
  // to represent the boundaries of this quad tree
  AABB boundary;

  // Points in this quad tree node
  Array of XY [size = QT_NODE_CAPACITY] points;

  // Children
  QuadTree* northWest;
  QuadTree* northEast;
  QuadTree* southWest;
  QuadTree* southEast;

  // Methods
  function __construct(AABB _boundary) {...}
  function insert(XY p) {...}
  function subdivide() {...} // create four children that fully divide this quad into four quads of equal area
  function queryRange(AABB range) {...}
}
```

In the example, each node capacity is set to 4 wherein this indicates the how many nodes can be stored in each tree node.

("Quadtree," 2017)

For each points location to be saved there will be a two dimensional array where their X and Y positions will be saved based on the node capacity which in this case will be four points. Each section of the quad tree is then separated into North West, north east, south west & south east quadrants. The quad tree class will be handling the creation of the Axis aligned bounding box, inserting the points stored in the array of positions & the subdivision of each node .

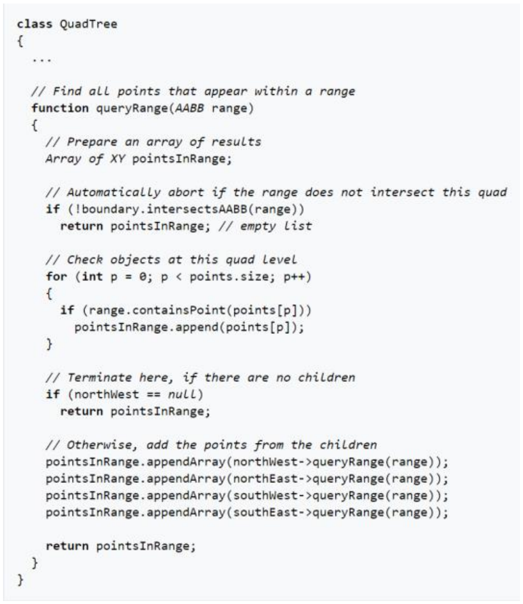


```
class QuadTree
{
  ...

  // Insert a point into the QuadTree
  function insert(XY p)
  {
    // Ignore objects that do not belong in this quad tree
    if (!boundary.containsPoint(p))
      return false; // object cannot be added

    // If there is space in this quad tree, add the object here
    if (points.size < QT_NODE_CAPACITY)
    {
      points.append(p);
      return true;
    }

    // Otherwise, subdivide and then add the point to whichever node will accept it
    if (northWest == null)
      subdivide();

    if (northWest->insert(p)) return true;
    if (northEast->insert(p)) return true;
    if (southWest->insert(p)) return true;
    if (southEast->insert(p)) return true;

    // Otherwise, the point cannot be inserted for some unknown reason (this should never happen)
    return false;
  }
}
```

Next each point is evaluated to check if it belongs in the quad tree or not, it would ignore the objects that do not belong in the quad tree & return false if the object cannot be added.

("Quadtree," 2017)

The if condition will be checking if there is space in the quad tree then it would add the object here and return true else it would subdivide and then add the point to which ever node will accept it , next it would check which quadrant it would belong to & return true . In the rare condition it cannot be inserted into the tree then it would return. This is not a likely case as it should never happen.



```
class QuadTree
{
  ...

  // Find all points that appear within a range
  function queryRange(AABB range)
  {
    // Prepare an array of results
    Array of XY pointsInRange;

    // Automatically abort if the range does not intersect this quad
    if (!boundary.intersectsAABB(range))
      return pointsInRange; // empty list

    // Check objects at this quad level
    for (int p = 0; p < points.size; p++)
    {
      if (range.containsPoint(points[p]))
        pointsInRange.append(points[p]);
    }

    // Terminate here, if there are no children
    if (northWest == null)
      return pointsInRange;

    // Otherwise, add the points from the children
    pointsInRange.appendArray(northWest->queryRange(range));
    pointsInRange.appendArray(northEast->queryRange(range));
    pointsInRange.appendArray(southWest->queryRange(range));
    pointsInRange.appendArray(southEast->queryRange(range));

    return pointsInRange;
  }
}
```

This section finds all the points that appear within a range, after which an array of points in range is created. Next there is a condition to check if the range does not intersect with the quad and will return. Assuming the last step was successful then it would check the objects at the current quad level by looping through each object in the array.

("Quadtree," 2017)

Next it would check if there are no children, if this condition is true then it would terminate the operation and return all the points in range otherwise it will continue and add points from the children of the other quadrants which are within range after which it will return the points in range.

One thing to note is that the operations has a lot of conditions to check if certain parameters are met, this is to reduce the amount of calculations done and makes it more efficient over all

.

References

Retrieved from https://en.wikipedia.org/wiki/Octree://

Hardcrawler. (2014, August 24). *Grinding out Collisions and Space Partitioning* [Video file]. Retrieved from https://www.youtube.com/watch?v=dqPGyWpljUo

Li/University of Canterbury, B., & Mukundan/University of Canterbury, R. (2013). *A Comparative Analysis of Spatial Partitioning Methods for Large-scale, Real-time Crowd Simulation*. Paper presented at 21st International Conference on Computer Graphics, Visualization and Computer Vision 2013. Retrieved from https://otik.uk.zcu.cz/bitstream/11025/10652/1/Li.pdf

Quadtree. (2017, November 13). Retrieved December 6, 2017, from https://en.wikipedia.org/wiki/Quadtree#Region_quadtree