# OPTIMIZING THE COLLISION DETECTION

SLOT->L-5/6

NAME->ABHINAV SAGAR

REGISTRATION NUMBER->16BME0903

(Individual Project)

# ABSTRACT

Collision detection is the computational problem of detecting the intersection of two or more objects. While collision detection is most often associated with its use in video games and other physical simulations, it also has applications in robotics. In addition to determining whether two objects have collided, collision detection systems may also calculate time of impact, and report a contact manifold. Collision detection is one of the most challenging and complex parts of game programming and is the key area where performance is usually lost. To solve this there are a lot of data structures that eliminate unnecessary checks for collisions like quadtrees, octrees, BSP trees, grids etc.

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The data associated with a leaf cell varies by application, but the leaf cell represents a unit of interesting spatial information.

Many games and numerical intensive simulations require the use of collision detection algorithms to detect whether and at what instant of time two objects have collided. But these operations are highly inefficient and slow down the game or the simulation as the case may be. So in this project I have used quad trees as data structure for reducing the number of possible comparisions at a particular instant of time.

# AIM

My project focuses on the optimizing the collision detection using quad trees and AABB trees. As an example let's suppose we have n point-sized objects in some space. We can detect collisions easily using the naive O(n^2) implementation (checking every point against the other). Although this might seem feasible for very small n, the algorithm is very inefficient even for small n = 100 (10^4 comparisions). A more efficient and elegant solution is to use quad trees. The idea is to insert all the points in a quad tree .Since expected time for insert into a quad tree is O(log n), inserting all of the n points costs O(nlogn) time for the expected case. (In theory, height of a quad tree can be unbounded, which can dramatically affect performance. However, this happens only if the distribution of points is very uneven). Once the quad tree is built, we simple traverse the tree using a depth first search and upon hitting a leaf node, use the naive algorithm (since every leaf node can only ever contain a maximum number of points bounded above by the threshold mentioned before). Thus, for n = 100, this algorithm performs much better, as it makes at most 100log(100) = 200 comparisons as opposed to 10^4 comparisons.

# APPLICABILITY

1) Image representation
2) Image processing
3) Mesh generation
4) Spatial indexing
5) Collision detection
6) Maximum disjoint Sets
7) Storing sparse data such as in spreadsheets or in matrix calculations.
8) Frustum culling of terrain data.
9) Connected component Labelling

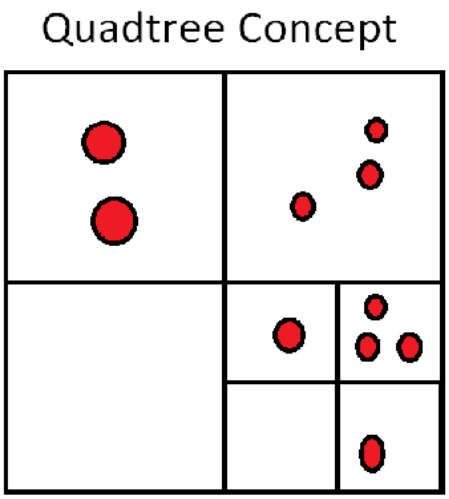
# INTRODUCTION

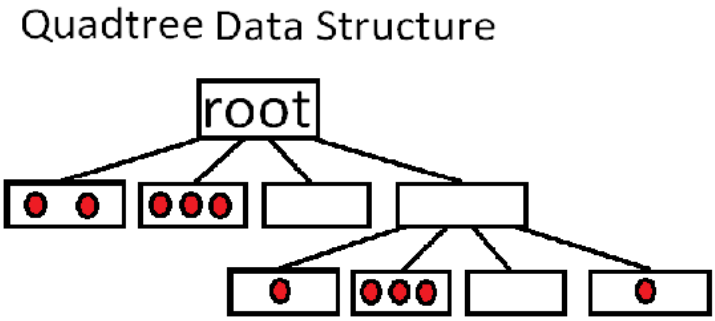
Quad Tree is a data structure in which every internal node has exactly 4 children, as is evident from the prefix 'Quad'. Quad trees are used to partition space into 4 quadrants or regions. Partitioning occurs when the number of points in a quadrant exceeds a certain threshold (usually a small number like 1 or 2). Usually, such partitioning is recursive in nature. Every sub quadrant that gets formed due to the partition is a child of the parent quadrant which was partitioned. Every leaf node of a quad tree contains some special spatial information (for example, coordinates of a point in a region). It is also important to note that internal nodes themselves do not hold spatial information. The spatial information is pushed all the way down the quad tree to the leaf level.

A simple collision detection system. Includes two algorithms-

Brute checks all bodies against all other bodies, and it can be hard on performance and only recommended for 100 or less bodies.

Quadtree data structures test collisions against nearby bodies only. Much more performant than Brute, but takes a bit of extra work to set up.



# LITERATURE REVIEW

Suppose at a particular instant of time there are 100 balls then using the brute force method to detect collisions there would be 10000 collisions between every pairs of balls. But this is highly efficient as in some of the real world problems we have to deal with millions or even more objects. Here objects could refer to agents in simulating artificial intelligence, molecular visualization and inferring chemical properties from atomic collisions ,in 3D graphics rendering etc. Two of the important applications related to my project are-

Finding the nearest neighbor-We have a bunch of points in a space. Rather than asking whether any of them match a given point, someone asks you what the nearest point you have to an arbitrary point among your points. With a quadtree, while searching, you can say, "OK, there's no way anything in this quadrant has any chance of being the nearest neighbor" and eliminate a lot of point comparisons that way.

Hit detection-Let's say you have a bunch of points in a space, like in the maps above. Someone asks you if some arbitrary point *p* is within your bunch of points. How can you find out if you have that point?You could compare every single point you have to *p*, but if you had 1000 points, and none of them were *p*, you'd have to do 1000 comparisons to find that out. Alternatively, you could get very fast lookup by keeping a grid (a 2D array) of booleans for every single possible point in this space. However, if the space these points are on is 1,000,000 x 1,000,000, you need to store 1,000,000,000,000 variables.Or you could set up a quadtree. When you have it search for *p*, it will find out which quadrant it is inside. Then, it will find out what quadrant within that quadrant it is inside. And so forth.

It will only have to do this at most seven times for a 100x100 space(assuming points can only have integer values), even if there are 1000 points in it. For a 1,000,000x1,000,000 space, it's a maximum of 20 times.After it finds its way to that rectangle node, it merely needs to see if any of the four children equal *p*.

# IMPLEMENTATION

For this project I have made use of HTML canvas and javascript. The feature it supports are-

-Balls can be added in the scene using mouse's left click.

-There is a checkbox for displaying and hiding grids as required.

-The following quantities are displayed at every instant of time -count of the number of balls, number of checks required for collision detection count using brute force, number of checks required for collision detection count using quad tree and ratio improvement in detecting collisions using quad trees over brute force.

## SOURCE CODE-

### project.html

```
<!DOCTYPE html>
<html>
   <head>
      <title>DSA Project</title>
      <style>
         .wrapper {
            display: inline-block;
            vertical-align: top;
         }
      </style>
   </head>
   <body>
      <h1 style="text-align:center;font-family:Arial;font-size:34px;letter-spacing:1px">Optimizing The Collision Detection</h1>
```

```
        <div class="wrapper">
            <canvas id="canvas" width="550" height="550" style="border:1px solid black;">
            </canvas>
        </div>
        <div class="wrapper" style="font-family:Arial;font-size:15px">
            <input id="checkbox" type="checkbox">Draw grid<br>
            <p></p>
            <p>Count of balls:</p>
            <div id="ball"></div>
            <p></p>
            <p>Number of checks required for collision detection count using brute force: <div id="bruteforce"></div></p>
            <p>Number of checks required for collision detection count using quad tree: <div id="quadtree"></div></p>
            <p>Ratio improvement in detecting collisions using quad trees over brute force: <div id="ratio"></div></p>
            <p></p>
        </div>
        <script src="project.js"></script>
    </body>
</html>
```

## project.js

```
class ball {
    constructor(x, y, r, velocityX, velocityY) {
        this.x = x;
        this.y = y;
        this.r = r;
        this.isColided = false;
```

```
        this.velocityX = velocityX;
        this.velocityY = velocityY;
    }
    intersects(other) {
        var changeX = this.x - other.x;
        var changeY = this.y - other.y;
        if (Math.sqrt(Math.pow(changeX, 2) + Math.pow(changeY, 2)) <= this.r + other.r) {
            return true;
        }
        return false;
    }
}
class AABB {
    constructor(x, y, halfLength) {
        this.x = x;
        this.y = y;
        this.halfLength = halfLength;
    }
    containsball(ball) {
        if ((ball.x + ball.r >= this.x - this.halfLength) &&
            (ball.x - ball.r <= this.x + this.halfLength) &&
            (ball.y + ball.r >= this.y - this.halfLength) &&
            (ball.y - ball.r <= this.y + this.halfLength)) {
            return true;
        }
        return false;
    }
    intersectsAABB(otherAABB) {
```

```
        if (Math.abs(this.x - otherAABB.x) < this.halfLength + otherAABB.halfLength &&
            Math.abs(this.y - otherAABB.y) < this.halfLength + otherAABB.halfLength) {
            return true;
        }
        return false;
    }
}
class QuadTree {
    constructor(boundaryAABB) {
        this.boundaryAABB = boundaryAABB;
        this.balls = [];

        this.nw = null;
        this.ne = null;
        this.sw = null;
        this.se = null;
    }
    insert(ball) {
        if (!this.boundaryAABB.containsball(ball)) {
            return false;
        }
        if (this.balls.length < QuadTree.size && this.nw == null) {
            this.balls.push(ball);
            return true;
        }
        if (this.nw == null) {
            this.subdivide();
        }
```

```
    if (this.nw.insert(ball)) { return true; };
    if (this.ne.insert(ball)) { return true; };
    if (this.sw.insert(ball)) { return true; };
    if (this.se.insert(ball)) { return true; };
    return false;
}
subdivide() {
    var quarterLength = this.boundaryAABB.halfLength / 2;
    this.nw = new QuadTree(new AABB(this.boundaryAABB.x - quarterLength,
                        this.boundaryAABB.y - quarterLength,quarterLength));
    this.ne = new QuadTree(new AABB(this.boundaryAABB.x + quarterLength,
                        this.boundaryAABB.y - quarterLength,quarterLength));
    this.sw = new QuadTree(new AABB(this.boundaryAABB.x - quarterLength,
                        this.boundaryAABB.y + quarterLength,quarterLength));
    this.se = new QuadTree(new AABB(this.boundaryAABB.x + quarterLength,
                        this.boundaryAABB.y + quarterLength,quarterLength));
}
queryRange(rangeAABB) {
    var foundballs = [];
    if (!this.boundaryAABB.intersectsAABB(rangeAABB)) {
        return foundballs;
    }
    for (let c of this.balls) {
        if (rangeAABB.containsball(c)) {
            foundballs.push(c);
        }
    }
    if (this.nw == null) {
```

```
        return foundballs;
    }
    Array.prototype.push.apply(foundballs, this.nw.queryRange(rangeAABB));
    Array.prototype.push.apply(foundballs, this.ne.queryRange(rangeAABB));
    Array.prototype.push.apply(foundballs, this.sw.queryRange(rangeAABB));
    Array.prototype.push.apply(foundballs, this.se.queryRange(rangeAABB));
    return foundballs;
}
draw(context, drawGrid) {
    if (drawGrid) {
        this.drawquadrants(context);
    }
    this.drawballs(context);
}
drawquadrants(context) {
    if (this.nw != null) {
        this.nw.drawquadrants(context);
        this.ne.drawquadrants(context);
        this.sw.drawquadrants(context);
        this.se.drawquadrants(context);
    } else {
        context.beginPath();
        context.rect(this.boundaryAABB.x - this.boundaryAABB.halfLength,
                this.boundaryAABB.y - this.boundaryAABB.halfLength,
                2 * this.boundaryAABB.halfLength, 2 * this.boundaryAABB.halfLength);
        context.lineWidth = 3;
        context.strokeStyle = 'black';
        context.closePath();
```

```
            context.stroke();
        }
    }
    drawballs(context) {
        if (this.nw != null) {
            this.nw.drawballs(context);
            this.ne.drawballs(context);
            this.sw.drawballs(context);
            this.se.drawballs(context);
        }
        for (let c of this.balls) {
            context.beginPath();
            context.arc(c.x, c.y, c.r, 0, 2 * Math.PI, false);
            if (c.isColided) {
                context.fillStyle = 'blue';
            } else {
                context.fillStyle = 'green';
            }
            context.fill();
            context.lineWidth = 0.1;
            if (c.isColided) {
                context.strokeStyle = 'blue';
            } else {
                context.strokeStyle = 'green';
            }
            context.closePath();
            context.stroke();
        }
```

```
    }
}
function iterate() {
    context.clearRect(0, 0, canvas.width, canvas.height);
    let detections = 0;
    if (mouseIsDown) {
        let velocityX = Math.random() * (100 + 50) + -50;
        let velocityY = Math.random() * (100 + 50) + -50;
        let x = mouseX;
        let y = mouseY;
        let r = 10.0;
        if (x + r > canvas.width) {
            let change = (x + r) - canvas.width;
            x -= change;
        } else if (x - r < 0) {
            let change = Math.abs(x - r);
            x += change;
        }
        if (y + r > canvas.height) {
            let change = (y + r) - canvas.height;
            y -= change;
        } else if (y - r < 0) {
            let change = Math.abs(y - r);
            y += change;
        }
        balls.push(new ball(x, y, r, velocityX, velocityY));
    }
    let quadTree = new QuadTree(boundaryAABB);
```

```
for (let c of balls) {
    c.isColided = false;
    quadTree.insert(c);
}
for (let c of balls) {
    let searchedAABB =  new AABB(c.x, c.y, c.r + 1);
    let foundballs = quadTree.queryRange(searchedAABB);
    for (let fb of foundballs) {
        if (c == fb) {
            continue;
        }
        detections++;
        if (c.intersects(fb)) {
            c.isColided = true;
            fb.isColided = true;
            break;
        }
    }
}
quadTree.draw(context, drawGridCheckbox.checked);
d = new Date();
changeTimeS = (d.getTime() / 1000.0) - lastTimeS;
lastTimeS = d.getTime() / 1000.0;
for (let c of balls) {
    let nextX = c.x + c.velocityX * changeTimeS;
    let nextY = c.y - c.velocityY * changeTimeS;
    if (nextX - c.r <= 0 || nextX + c.r >= canvas.width) {
        c.velocityX *= -1;
```

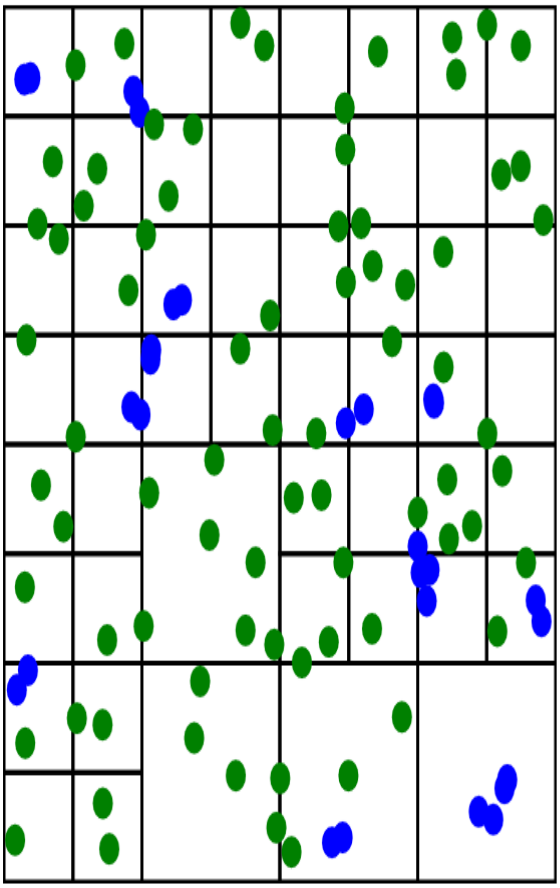
```
            c.x += c.velocityX * changeTimeS;
        } else {
            c.x = nextX;
        }
        if (nextY - c.r <= 0 || nextY + c.r >= canvas.height) {
            c.velocityY *= -1;
            c.y -= c.velocityY * changeTimeS;
        } else {
            c.y = nextY;
        }
    }
    ballCounter.textContent = balls.length;
    detectionCounter.textContent = detections;
    bruteforceCounter.textContent = Math.pow(balls.length, 2);
    ratio.textContent=Math.round((Math.pow(balls.length, 2)-detections)/(detections))
}
QuadTree.size = 3;
var drawGridCheckbox = document.getElementById('checkbox');
drawGridCheckbox.checked = true;
var ballCounter = document.getElementById('ball');
var detectionCounter = document.getElementById('quadtree');
var bruteforceCounter = document.getElementById('bruteforce');
var ratio=document.getElementById('ratio');
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');
var canvasDimension = canvas.height;
var balls = [];
var d = new Date();
```

```
var lastTimeS = d.getTime() / 1000.0;
var changeTimeS = 0;
var mouseIsDown = false;
var mouseX = 0;
var mouseY = 0;
var halfLength = canvasDimension / 2;
var boundaryAABB = new AABB(halfLength, halfLength, halfLength);
canvas.onmousedown = function(e){
    mouseIsDown = true;
}
canvas.onmouseup = function(e){
    mouseIsDown = false;
}
canvas.addEventListener('mousemove', function(evt) {
    var rect = canvas.getBoundingClientRect();
    mouseX = evt.clientX - rect.left;
    mouseY = evt.clientY - rect.top;
    }, false
);
setInterval(iterate, 10);
```

An interactive explanation of qua | literature review on collision dete | DSA Project

File | C:/Users/Win%2010/Desktop/github/dsaproject/project.html

# Optimizing The Collision Detection



Draw grid

Count of balls:

105

Number of checks required for collision detection count using brute force:

11025

Number of checks required for collision detection count using quad tree:

36

Ratio improvement in detecting collisions using quad trees over brute force:

305

Type here to search

ENG IN 10:14 PM 3/18/2019

# MODULE DESCRIPTION

1) There is a class ‘circle’ for representing the balls described with x coordinate, y coordinate of center, radius, iscollided state, velocity in x and y directions. It also has a function ‘intersects’ for checking whether 2 circles collide with each other.

2) The second class is for representing AABB trees with the help of x and y coordinate and half quadrant length. There is a function ‘containsball’ for checking whether a particular quadrant contains ball or not. Another function ‘intersectsAABB’ is for checking if AABB trees structures collides with balls.

3) The third class is for representing quad trees containing variables for list of balls and for checking each of the 4 quadrants contains a ball or not. It has a function ‘insert’ for insering the ball by recursively making use of AABB trees. It also has a function ‘subdivide’ for recursively dividing the dimensions by half and updating the four quadrant values in each iteration. Now we have a ‘queryRange’ function for checking if the dimensions after recursion in above step are correct. We also have ‘draw’, ‘drawquadrants’ and ‘drawballs’ for recursively updating the quadrant dimensions and the number of balls in the appropriate quadrants.

4) Then there is a main function ‘iterate’ which binds all the above classes and functions. It does the following- it iterates the ball ejected on each mouse click and also takes into account the bouncing off effect from the canvas boundaries. Then we are updating all the values by recursively updating the balls and quadrant dimensions.

5) And finally we are initializing all the values ,capturing the mouse co-ordinates and running the ‘iterate’ function after every 10ms.

# RESULTS AND DISCUSSIONS

The results I achieved after the above implementation are as follows-

-For 300 balls in our canvas the number of checks required for collision detection count using brute force are 103684 pair of checks while the number of checks required for collision detection count using quad trees are at the maximum 300 pair of checks. This results in an improvement by a factor of around 350.

Hence it can be seen that quad trees are highly efficient in detecting collisions by reducing the number of inappropriate collisions. For example there is no chance of collision of the balls present in north west corner with the ball present in south east corner. Hence it eliminates that possibility. In a similar manner many collisions are eliminated. In this manner it reduces the load on the number of checks required depending on the application be it numerical simulation or studying chemical properties with the help of atomic collisions. Hence it can be concluded that quad trees are of real practical applications in domains like games, 3D rendering, numerical simulations etc.

# TIME COMLEXITY

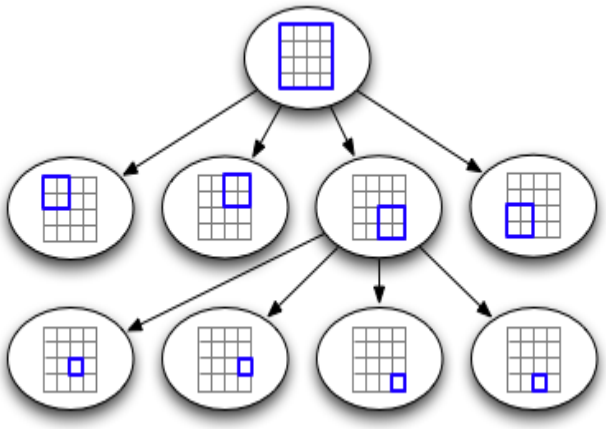
Unbounded in the worst case

The time complexity of this (assuming a uniform distribution of entities): is usually O(n^1.5 log(n)), since the index takes about log(n) comparisons to traverse, there will be about sqrt(n) neighbors to compare, and there are n objects to check. Realistically, though, the number of neighbors is always quite limited, since if a collision does occur, most of the time one of the objects is deleted, or moved away from the collision. thus we get just O(n log(n)).

For reasonable input:

Space: O(n)

Query time: O(n) in the worst case



# REFERENCES


1) Sean Curtis , Rasmus Tamstorf , Dinesh Manocha, Fast Collision Detection for Deformable Models Using Representative-triangles, Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, February 15-17, 2008.
2) Min Tang, Sean Curtis, Sung-Eui Yoon, and Dinesh Manocha, ICCD: Interactive Continuous Collision Detection between Deformable Models Using Connectivity-Based Culling, IEEE Transactions on Visualization and Computer Graphics, 2009, 15( 4):544- 557
3) M. Eitz and G. Lixu. Hierarchical spatial hashing for real-time collision detection. Proc. of IEEE International Conference on Shape Modeling and Applications, pp. 61–70, 2007.
4) R. Mukundan and B. Li. Crowd simulation: Extended oriented bounding boxes for geometry and motion representation. Proc. of the 27th Conference on Image and Vision Computing New Zealand, pp. 121–125, 2012.
5) Boada I, Coll N, Sellares J. The voronoi-quadtree: construc380 tion and visualization. Eurographics 2002 Short Presentation 381 2002;:349–55.
6) M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. Computational Geometry: Algorithms and Applications (3rd edition). Springer, 2008.
7) M. de Berg, H. Haverkort, S. Thite, and L. Toma. Star-quadtrees and guardquadtrees: I/O-efficient indexes for fat triangulations and low-density planar subdivisions. Comput. Geom. Theory Appl. 43:493–513, 2010.
8) D.-J. Kim, L.J. Guibas, and S.Y. Shin. Fast collision detection among multiple moving spheres. IEEE Trans. Vis. Comp. Gr., 4:230–242, 1998.
9) S. Kockara, T. Halic, K. Iqbal, C. Bayrak and R. Rowe. Collision detection: A survey. In Proc. of Systems, Man and Cybernetics, pages 4046–4051, 2007.

10. The Introduction to Algorithm, Thomas H. Cormen, Leiserson, Ronald L. Reivst and Clifford Stein
11. Serviss, B., Seligmann, D.: D.: Escaping the world: high and low resolution in gameing. Multimedia. IEEE 12(4), 4–8 (2005)
12. Gottschalk, S., Lin, M. C., Manocha,: Hierarchical Structure for Rapid Interference Detection. Computer Graphics 30, 171–180 (1996)
13. Moore, M., Wilhelms, J.: Collision detection and response for computer animation. SIGGRAPH Comput. Graph. 22, 289–298 (1988)
14. Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M., Faure, F., Thalmann, M.N., Strasser, W., Volino, P.: Collision detection for deformable objects. Comp. Graph. Forum 24, 119–140 (2005)
15. de Berg, M., Roeloffzen, M., Speckmann, B.: Kinetic convex hulls and Delaunay triangulations in the black-box model. In: Proc. 27th ACM Symp. Comput. Geom., pp. 244–253 (2011)