Collision Detection Using Quad-Tree Data Structure

Optimizing collision detection

Abhinav Sagar Follow
Jun 6 · 4 min read

Photo by Uriel Soberanes on Unsplash

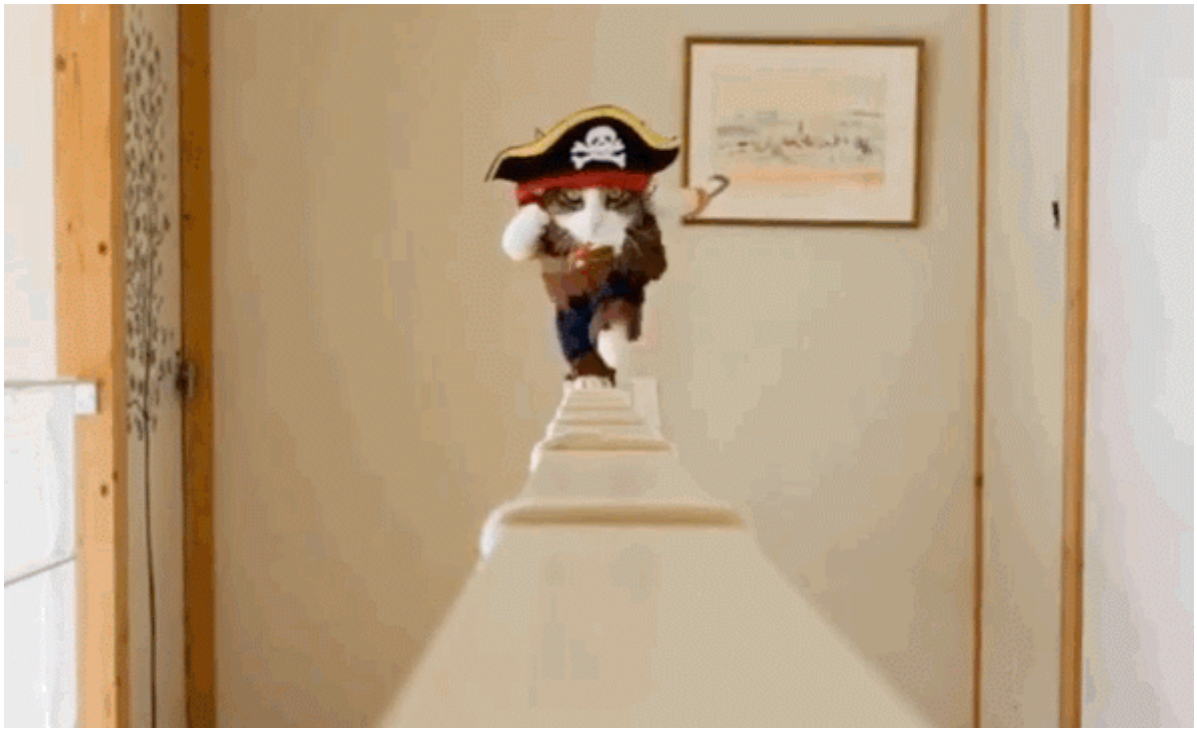
Many games require the use of collision detection algorithms to determine when two objects have collided, but these algorithms are often expensive operations

and can greatly slow down a game. **Collision detection** is an essential part of most video games. Both in 2D and 3D games, detecting when two objects have collided is important as poor collision detection can lead to some very interesting results. However, collision detection is also a very expensive operation. Let's say there are 100 objects that need to be checked for collision. Comparing each pair of objects requires 10,000 operations — that's a lot of checks!

One way to speed things up is to reduce the number of checks that have to be made. Two objects that are at opposite ends of the screen can not possibly collide, so there is no need to check for a collision between them. This is where a **quadtree** comes into play.

QuadTree

QuadTree is a data structure in which every internal node has exactly 4 children, as is evident from the prefix 'Quad'. Quadtrees are used to partition space into 4 quadrants or regions. Partitioning occurs when the number of points in a quadrant is more than 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 quadtree contains some special spacial 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 quadtree to the leaf level.



Environment and tools

1. html
2. javascript

Let's get started with the code. Feel free to clone the project from my github repository **here** to follow along.

We will need two files

1. **project.html** which contains the **html canvas** that is the front end or the user interface.
2. **project.js** which contains the **javascript** code for processing the server side stuff in real-time.

```
 <!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
        <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
            <p>Number of checks required for collision detection count using quad tree:
            <p>Ratio improvement in detecting collisions using quad trees over brute for
            <p></p>
        </div>
        <script src="project.js"></script>
    </body>
</html>
```

project.html hosted with ❤ by GitHub view raw

project.html

Now for the main logic code, we'll need three classes for colliding objects (**balls**), **AABB tree** and **Quad tree** respectively. Let's start with writing our ball class which has a function to check whether a ball collides with any other ball or not.

```
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;
    }
}
```

ball.js hosted with ❤ by GitHub view raw

Now let's write the **AABB tree** class which has two functions. The first one checks whether a ball is located in any of its nodes or not and the second one for checking if two nodes of the tree share the same region.

```
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;
    }
}
```

aabb.js hosted with ❤️ by GitHub view raw

Let's begin writing our main class which has a function for inserting a ball in its node.

```
    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;
```

```
        }
```

insert.js hosted with ❤ by GitHub view raw

The next function is for recursively subdividing the length of the four quadrant in each iteration.

```
    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));
    }
```

subdivide.js hosted with ❤ by GitHub view raw

Now let's define a function for adding all the balls depending on its location to each of its four quadrants.

```
    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;
```

```
19      }
```

query_range.js hosted with ❤ by GitHub view raw

Now let's define a function to make a grid.

```
1       draw(context, drawGrid) {
2           if (drawGrid) {
3               this.drawquadrants(context);
4           }
5           this.drawballs(context);
6       }
```

draw.js hosted with ❤ by GitHub view raw

Similarly, we should make four quadrants also. Note this uses **HTML canvas**.

```
1       drawquadrants(context) {
2           if (this.nw != null) {
3               this.nw.drawquadrants(context);
4               this.ne.drawquadrants(context);
5               this.sw.drawquadrants(context);
6               this.se.drawquadrants(context);
7           } else {
8               context.beginPath();
9               context.rect(this.boundaryAABB.x - this.boundaryAABB.halfLength,
10                          this.boundaryAABB.y - this.boundaryAABB.halfLength,
11                          2 * this.boundaryAABB.halfLength, 2 * this.boundaryAABB.halfLer
12              context.lineWidth = 3;
13              context.strokeStyle = 'black';
14              context.closePath();
15              context.stroke();
16          }
17      }
```

draw_quadrants.js hosted with ❤ by GitHub view raw

Continuing the work, we should also define a function to draw the balls.

```
1       drawballs(context) {
2           if (this.nw != null) {
3               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();
        }
    }
}
```

draw_balls.js hosted with ❤️ by GitHub view raw

Now let's define the main function which combines all the above work. It captures the left click of the user to draw more balls on the screen and right-click to delete the balls added on the previous click.

```
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))
}
```

iterate.js hosted with ❤️ by GitHub view raw

Finally, let's write a piece of code to make sure everything is working as desired.

```
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);
```

final.js hosted with ❤️ by GitHub view raw

Results

Conclusions

Using quad tree we are getting a factor of improvement ~289 using just 166 objects. Quad trees are an interesting area of research in computer science. Some of its possible applications are molecular visualization for estimating its properties, studying chemical reactions, numerical simulation, particle swarm optimization, etc. In gaming, collision detection can be an expensive operation and can slow down the performance of your game. Quadtrees are one way you can help speed up collision detection and keep your game running at top speeds.

Before You Go

The corresponding source code can be found here.

abhinavsagar/collision-detection-using-quad-tree

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

References

Collision detection

Collision detection is the computational problem of detecting the intersection of two or more objects. While collision...

en.wikipedia.org

Quadtree

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the...

en.wikipedia.org

Happy reading, happy learning and happy coding.

And don't forget to 👏 if you enjoyed this article 🙂.

JavaScript Collision Detection Quadtree Data Structures