UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Curso : Estructura de Datos Avanzados

Trabajo presentado por

HERRERA COOPER, MIGUEL ALEXANDER

Índice

26 de septiembre de 2019

1. Introducción

Uno de los principales problemas del sistema de colisiones es que en cada interacción tenemos que comprobar si cada objeto está colisionando con el resto de objetos de nuestro juego.

Por ejemplo si tenemos 100 objetos en nuestro juegos nos daría 100 x 100 = 10000 comprobaciones de colisión en en cada bucle de nuestro juego, lo que es una barbaridad para 100 objetos.

Imagina que tenemos un escenario como el siguiente, donde los puntos azules son objetos de nuestro mundo.

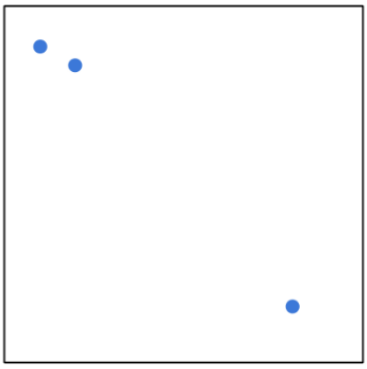

Figura 1: Ejemplo Grafico de QuadTree con 3 puntos

Digamos que los objetos se mueven a una velocidad de unos 10px por segundo. Quizás necesitemos entonces comparar si los objetos de la esquina superior izquierda colisionan entre ellos, pero es totalmente innecesario comprobar si están ambos colisionando con el objeto de la esquina inferior derecha, entonces ¿Cómo lo hacemos para descartar las colisiones que obviamente sabemos que no se van a prouducir? La solución es el clásico divide y vencerás.

2. Quad Tree

2.1. Definición

Los Quadtrees son un tipo de estructura de datos en el que cada nodo tiene a su vez cuatro nodos hijos, sería algo similar a un árbol binario, pero en vez de dos ramas con cuatro ramas.

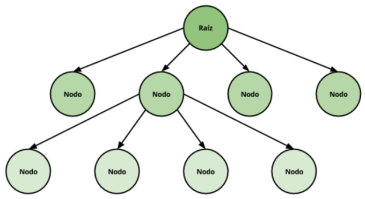

Figura 2: Quad Tree

Esto se puede usar para separar nuestro escenario en cuatro áreas iguales recursivamente, vamos a ver el Quadtree de la siguiente manera.

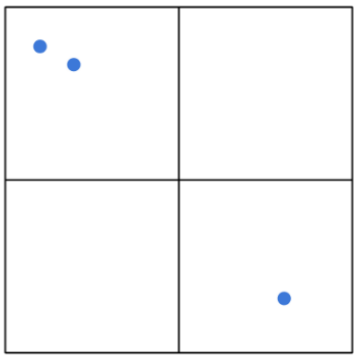

Figura 3: Ejemplo Grafico de QuadTree con 3 puntos

Ahora tendríamos un nodo padre que representa el escenario completo y cuatro nodos hijo que lo subdividen en cuatro partes iguales. Ahora lo que debemos hacer es decirle a nuestro objeto algo como: comprueba sólo las colisiones con los objetos que están en tu cuadrante.
Con esto ya tendríamos nuestros dos objetos superiores en un único cuadrante y evitaríamos comprobar colisiones con el objeto de la esquina inferior derecha.

2.2. Añadiendo Proundidad

Ahora imaginemos que añadimos unos cuantos objetos más a nuestro mundo, tal que nos queda así.

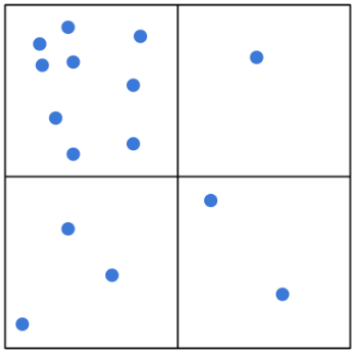

Figura 4: QuadTree con más objetos

Está claro que el cuadrante superior izquierdo empieza a tener demasiados objetos y se empiezan a hacer demasiado costoso comprobar todas las colisiones que suceden en él. Así que la solución es subdividir a su vez este cuadrante en otro nodo Quadtree.

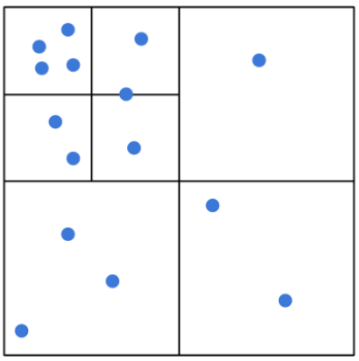

Figura 5: QuadTree con más objetos

De esta manera volvemos a tener un número pequeño de nodos en cada cuadrante que se hace mucho más manejable.

2.3. ¿A partir de cuantos objetos subdivido un cuadrante?

- Esta pregunta dependerá del tamaño del mapa y de los objetos, pero sería recomendable no tener mas de 10 objetos por cuadrante que yo son 10 x 10 = 100 comprobaciones.
- Lo mismo con el número de subdivisiones, sería bueno establecer un número máximo en el que un Quadtree se pueda subdividir en otro Quadtree, como antes todo dependerá del tamaño del mapa de nuestro juego, pero a partir de 5 niveles ya puede empezar a ser excesivo.

2.4. Caso especial

Vamos a fijarnos en el objeto marcado en rojo de nuestra representación:

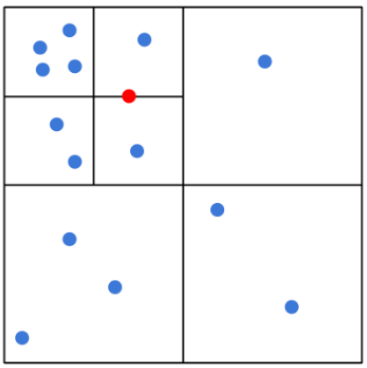

Figura 6: QuadTree con más objetos

- Estos objetos que quedan sobre las líneas no los metemos en ninguno de los cuadrantes hijos sino que lo dejamos en el cuadrante padre.
- Por tanto cada cuadrante deberá comprobar si sus objetos colisionan con los objetos de sus nodos hijos.
- Si un Quadtree que no sea un nodo hoja contiene un objeto podemos decir que dicho objeto se encuentra en una de las líneas que lo subdivide.

3. Actividad

3.1. A

Editemos el archivo quadtree.js y completemos la función query

```
query(range, found) {

    if (!range.intersects(this.boundary)) {
      return found;
    }

    for (let p of this.points) {
      if (range.contains(p)) {
        found.push(p);
      }
    }
    if (this.divided) {
      this.NO.query(range, found);
      this.NE.query(range, found);
      this.SO.query(range, found);
      this.SE.query(range, found);
    }

    return found;
  }
```

3.2. B - C - D

- B. Editemos el archivo sketch.js con el siguiente código. Muestre sus resultados y comente.
- C. En este caso vamos a verificar cuantas veces se consulta un punto en la función query, para esto usaremos la variable global count creada en el paso anterior. Esta variable la incrementaremos en la función query . Muestre sus resultados y comente.
- D. Editemos el archivo sketch.js. En este caso haremos consultas con el mouse. Muestre sus resultados y comente.

```
let count=0;
let qt;
function setup() {
        createCanvas(400,400);

        let boundary =  new Rectangle(200,200,200,200);
        qt = new QuadTree(boundary , 4);

        console.log(qt);
        //
        for (let i = 0; i<300; i++) {
                let p = new Point(Math.random() *400 , Math.random()*400);
                qt.insert(p);
        }

}

function draw(){
        background(0);
        qt.show();

        stroke(0,255,0);
        rectMode(CENTER);
        let range=new Rectangle(mouseX,mouseY,50,50);
        rect(range.x,range.y,range.w*2,range.h*2);
        let points=[];
        qt.query(range,points);
        for(let p of points){
                strokeWeight(4);
                point(p.x,p.y);
        }
}
```

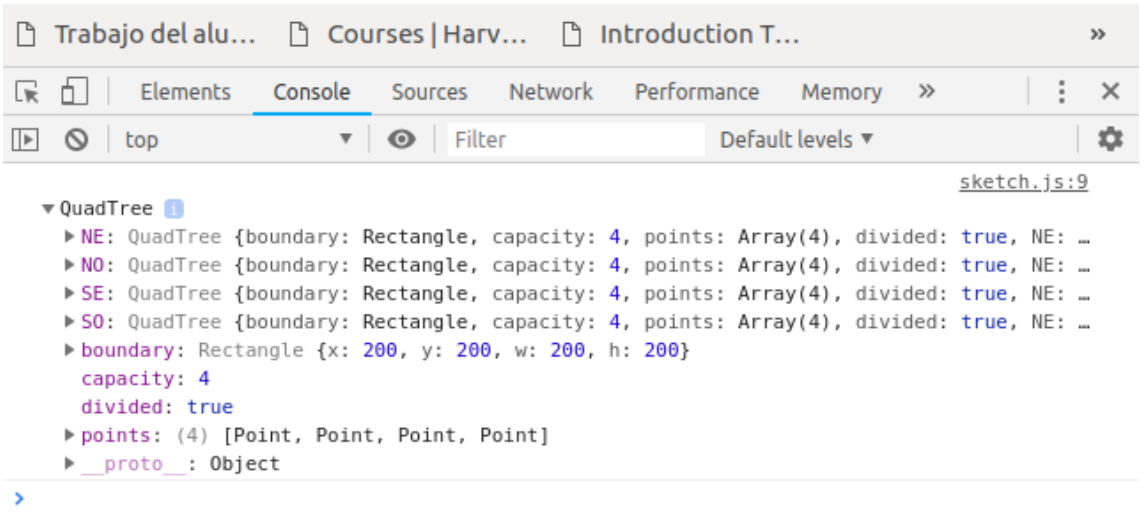

Figura 7: Resultado en la consola

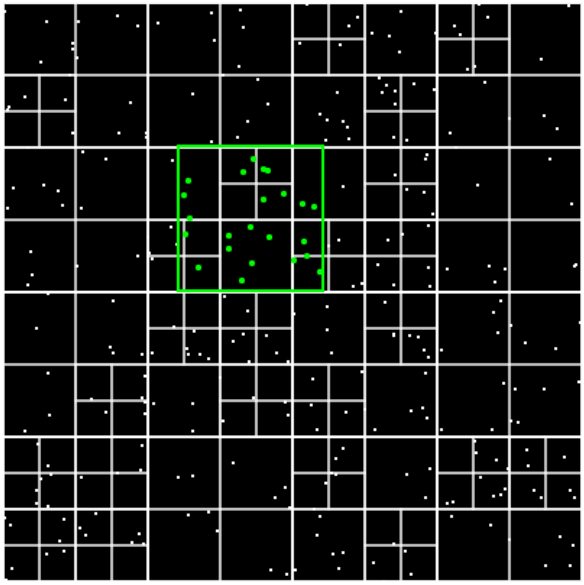

Figura 8: Resultado en el navegador

4. Conclusiones

- La función Query se utiliza para localizar un nodo en el quad dado. También se puede modificar para devolver el nodo más cercano al punto dado.
- Esta función se implementa tomando el punto dado, comparándolo con los límites de los quads secundarios y recurriendo.
- Esta función es O (Log N) donde N es el tamaño de la distancia.