GESTION PARA EL RECONOCIMIENTO DE COLISIONES ENTRE DRONES POLINIZADORES

Laura María Giraldo Castrillón
Universidad Eafit
Colombia
lmgiraldo1@eafit.edu.co

Andrés Felipe Oquendo Usma
Universidad Eafit
Colombia
afoquendou@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

RESUMEN
En un futuro cercano nos encontremos con la gran problemática de la desaparición de abejas y existe la preocupación latente de que no haya nadie que haga su importante labor, se ha visto la necesidad de crear drones que puedan, en alguna medida, suplantarlas. Nuestro objetivo, es encontrar una forma eficiente y efectiva de que los drones no colisionen. Algunos problemas similares que se encuentran para la detección de colisiones son algoritmos como el spatial hashing, quadtree y el árbol AABB. La solución que aquí se presenta es en base a la división del espacio mediante el uso de un árbol que estará dividido en 8 nodos (octree) donde por medio de herramientas matemáticas se ha calculado donde existe la mayor probabilidad de colisión entre los drones por lo tanto los resultados están dados a un ajuste muy preciso donde el porcentaje de error es mínimo puesto que se permite que en cada nodo del octree se evalúen todos los drones que están cerca de sí. En conclusión, esta estructura de datos es bastante eficiente ya que nos provee información de millones de objetos en un tiempo promedio menor a 1 minuto y además los resultados son bastante confiables.

Palabras clave
Algoritmo, eficiencia, colisiones, abejas artificiales.

Palabras clave de la clasificación de la ACM

Information systems---Data management systems---Database design and models-- -Graph-based database models;300.

1. INTRODUCCIÓN
Los avances de la industrialización y de la tecnología han sido un factor que ha impactado especialmente el medio ambiente, actualmente una de las mayores problemáticas es la desaparición paulatina de distintas especies. El enfoque de este trabajo va centrado a dar una solución a como el aumento de los pesticidas y de los depredadores naturales que han reducido en gran porcentaje la población de abejas ya que esto puede llegar a ser catastrófico puesto que la polinización es un proceso sumamente importante no solo para poder obtener alimentos para el consumo humano sino también el de otros animales [5].

2. PROBLEMA
¿Qué sucedería si las abejas desaparecen? Gracias a la tecnología podríamos pensar en suplantar esta especie por medio de drones, de hecho, hay unos primeros desarrollos frente a esto. El problema central de este trabajo es dar una posible solución para el reconocimiento de colisiones que deben tener estos drones.

3. TRABAJOS RELACIONADOS

3.1 Spatial Hashing
Se necesita renderizar un juego, para esto es necesario conocer las posiciones de cada elemento en el plano que estamos utilizando, usar una sola matriz se hace un proceso muy lento. ¿Cuál es una posible solución para optimizar el renderizado?

El "Spatial hashing" es una forma de indexar objetos. Funciona con espacios 2D y 3D, una tabla de hash y una función hash. Consiste en convertir el espacio 2D o 3D a una tabla de hash en donde se indexará cada objeto gracias a la función hash. Esto es especialmente útil para estudiar las colisiones de dos o más objetos. Los objetos con el mismo índice hacen referencia a una posible colisión y descarta a los demás objetos indexados en posiciones diferentes de la tabla. Esto siempre y cuando las celdas de la tabla tengan un tamaño adecuado, ni muy grandes ocupando memoria que no se utiliza, ni muy pequeñas ocupando tiempo de ejecución al tener que examinar cada objeto como si no estuviéramos utilizando este método [6].

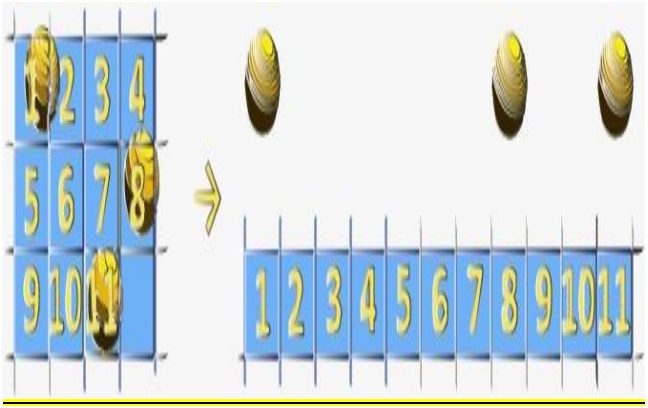


Gráfico 1: Spatial Hashing

3.2 AABB Tree Collision Detection
El "AABB Tree Collision Detection" los AABB (Axis Aligned Bounding Boxes) son cajas con coordenadas, las cuales tienen sus ejes alineados y pueden definirse por 2 puntos ya sea en 2D o 3D. El árbol AABB, permite organizar e indexar los AABB (serían las hojas), dentro de otros AABB más grandes (ramas), y las raíces que pueden ser las ramas o las hojas [2].

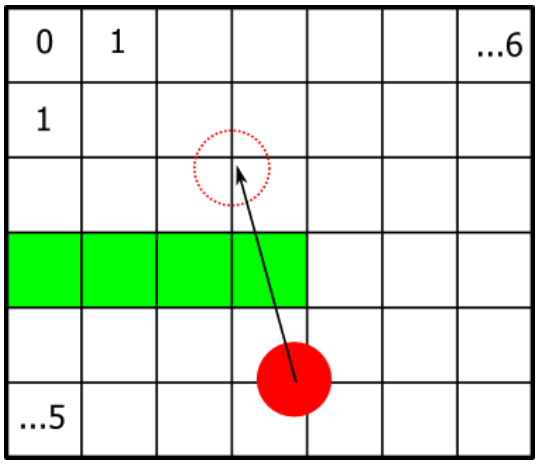


Gráfico 2: AABB Tree Collision Detection

3.3 Quad Tree

Tenemos dos puntos azules en la esquina superior izquierda y uno en la derecha, necesitamos saber cuáles puntos harán colisión, es claro que la más próxima es entre los puntos de la izquierda por lo tanto podríamos descartar aquellos que se encuentran más lejos y se hacen obviamente innecesario comprobar si se chocaran o no [1]. ¿Cuál sería el algoritmo más eficiente en este caso?

Quad tree se basa en la descomposición recursiva del espacio, por lo tanto, este algoritmo se dio en solución a la necesidad de guardar los datos que se insertaban con valores idénticos lo que traducido a la gestión de colisiones es identificar cuando un objeto esta simultáneamente cerca a otro, comúnmente se usa para representar puntos, áreas, curvas, superficies y volúmenes. Se trata de una detección eficiente de la colisión en n dimensiones. Este tipo de estructura de datos va muy de la mano con el concepto conocido en toda la programación divide y conquistaras [7].

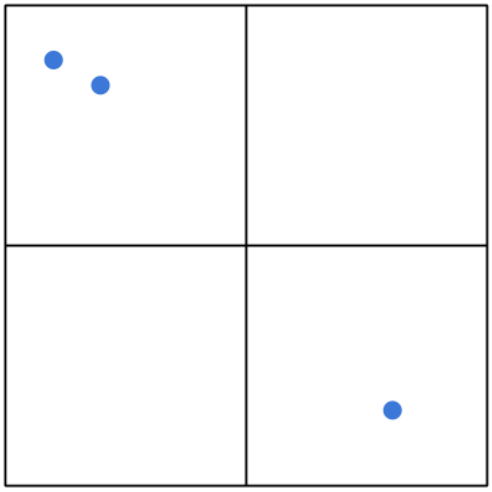


Gráfico 3: Quadtree: colisiones de los puntos azules

3.4 Octree

Almacenar objetos que se encuentran en un espacio tridimensional. Se dan las respectivas coordenadas en x, y, z y se pide decir en donde se encuentra cada objeto.

Octree funciona dividiendo el espacio en ocho y tomando como base un nodo o raíz que pueden ser infinitos puntos en el plano, se utiliza comúnmente para la detección de colisiones en espacios tridimensionales. El octree es una generalización del quad tree en la tercera dimensión. Por lo tanto, funciona perfectamente a este problema ya que al dividir el espacio en subregiones es más rápido a la hora de dar respuesta de la ubicación de lo que estamos buscando [8].

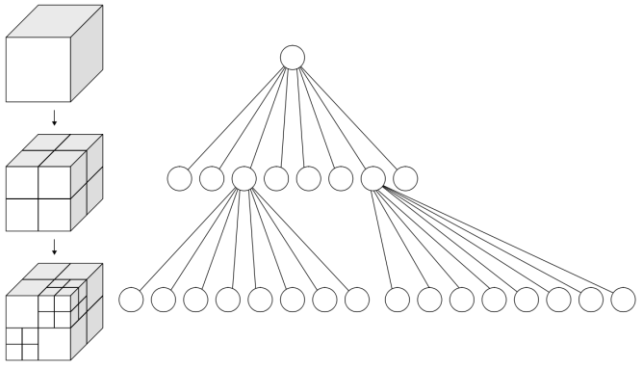
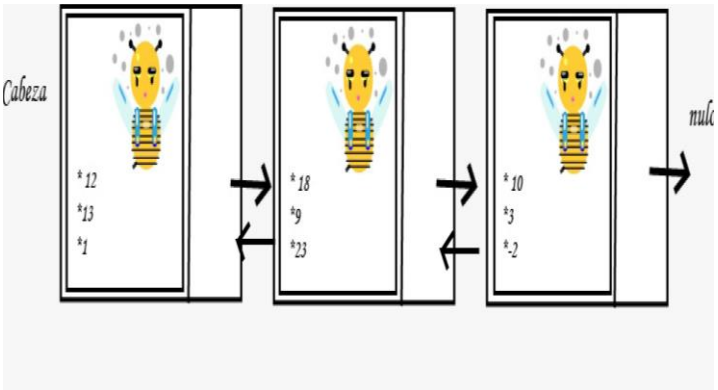


Gráfico 4: Octree

4. DETECCIÓN DE COLISIONES CON LISTAS ENLAZADAS (primera estructura de datos diseñada)

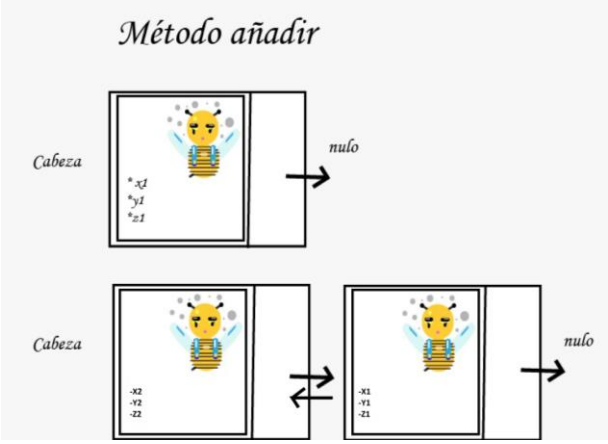
4.1 Lista doblemente enlazada

Se utilizará la estructura de lista enlazada de las librerías de java, la cual contendrá las posiciones de las abejas que tienen riesgo de colisión.



Gráfica 1: Lista Doblemente enlazada que contiene las posiciones x, y, z de cada abeja.

4.1 Operaciones de la estructura de datos



Gráfica 2: Imagen de una operación de agregar en la lista doblemente enlazada

4.2 Criterios de diseño de la estructura de datos

Inicialmente se tenía un código con implementación de arreglos, pero para operaciones como añadir o borrar la complejidad en el peor de los casos era $O(n^3)$. Usando listas enlazadas se pueden conseguir una complejidad de $O(n^2)$ porque sus métodos en el peor de los casos son de O(n).

4.2 Criterios de diseño de la estructura de datos

Implementamos la lista enlazada ya que a pesar de que consume un poco más de espacio, mejora la complejidad de la primera solución de $O(n^3)$ a $O(n^2)$, ya que el añadir a un array list (que era la estructura que implementaba la primera solución) tenía complejidad O(n) la cual se sumaba a dos ciclos anidados, en cambio añadir a una lista doblemente enlazada tiene complejidad O(1).

4.3 Análisis de Complejidad

Se reporta solo el metodo add, ya que es el unico utiliza de linked list.

Método	Complejidad
Add (Añadir)	O(1)

Tabla 1: Tabla con la complejidad de las operaciones principales de el linked list.

4.4 Tiempos de Ejecución

Se toman los tiempos de ejecución 100 veces y se haya la media entre estos para cada conjunto de datos.

	Conjunto de datos 1	Conjunto de datos 2
Colisiones	1ms	1341574ms

Tabla 2: Tiempos de ejecución promedio de la estructura de datos con diferentes conjuntos de datos.

4.5 Memoria

Se toma la memoria utiliza 100 veces y se haya la media entre estos para cada conjunto de datos.

	Conjunto de datos 1	Conjunto de datos 2
Colisiones	1MB	55MB

Tabla 3: Consumo de memoria promedio de la estructura de datos con diferentes conjuntos de datos.

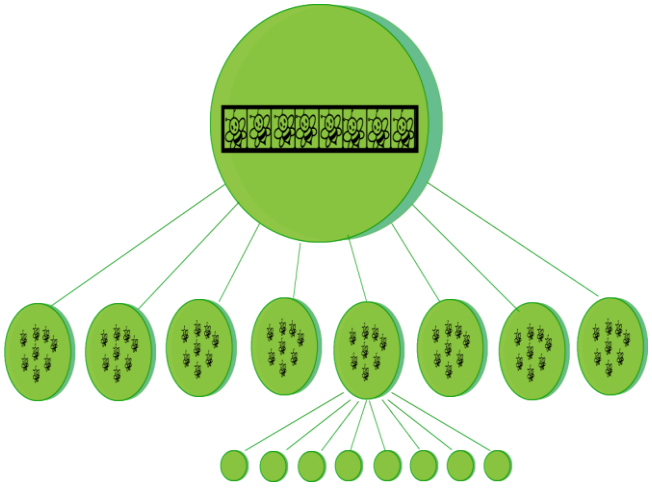
4.6 Análisis de los resultados

	ArrayList	LinkedList
Tiempo de 10 abejas	1ms	1ms
Tiempo de 100000 abejas	1067620ms	1341574ms
Memoria de 10 abejas	1MB	1MB
Memoria de 100000 abejas	54MB	55MB

Table 4: Análisis de los resultados obtenidos con la implementación de la estructura de datos vs la implementación anterior.

El tiempo promedio de ejecución de la linked list es mayor a el tiempo promedio del array list, lo cual va en contra de la teoría y además, el linked list consume más memoria.

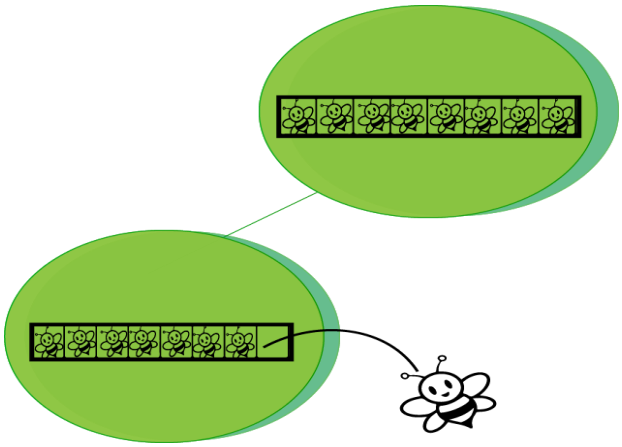
5. DETECCIÓN DE COLISIONES CON OCTREE (implementación final)



Gráfica 3: Octree como estructura general.

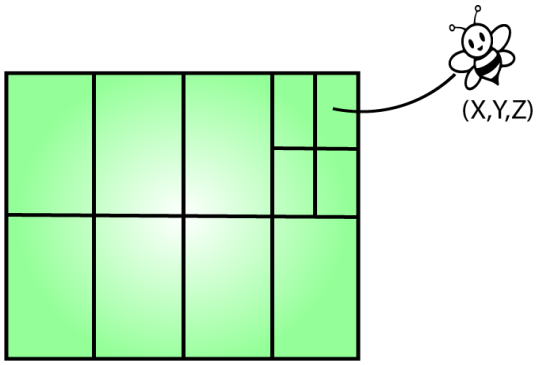
5.1 Operaciones de la estructura de datos

La operación **insertar** añade a un nodo que no esté lleno una nueva abeja.



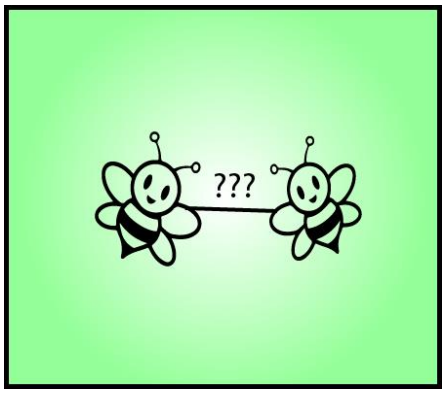
Gráfica 4: Imagen de una operación de insertar de una implementación de octree.

La operación **dónde** realiza los intervalos de tamaño que tendrá cada nodo hijo.



Gráfica 5: Imagen de una where en la implementación de octree.

La operación **Colisión** detecta sí dos abejas de un mismo nodo están a una distancia menor a 100 metros.



Gráfica 6: Imagen de una operación de colision de una implementación de octree.

5.2 Criterios de diseño de la estructura de datos

Se diseñó la estructura de datos como un octree, de tal forma que se creen subgrupos de las abejas totales y que las comparaciones se hagan dentro de cada subgrupo y no entren todas las abejas del conjunto de datos. Es bastante atractiva la complejidad de este algoritmo puesto es el número de abejas por la altura del árbol que es $\log_8 n$, en notación O grande seria O(n*logn)), en comparación con la primera implementación del código usando listas enlazadas con complejidad $O(n^2)$, lo cual mejora considerablemente y donde el tiempo era en promedio de 1.341,574 segundos mientras que usando el octree tenemos para una misma lectura de 100.000 abejas un tiempo de 11.551 segundos lo que permite optimizar el trabajo con bases de datos bastante grandes.

5.3 Análisis de la Complejidad

Metodo	Complejidad
insert	O(log(n))
createChildren	O(1)
where	O(1)
colision	O(n*log(n))

Tabla 5: Tabla para reportar la complejidad

5.4 Tiempos de Ejecución

Se toman los tiempos de ejecución 100 veces y se haya la media entre estos para cada conjunto de datos.

	10 abejas	100 abejas	1000 abejas	10000 abejas	100000 abejas
Colisiones	0.6ms	1ms	3ms	60ms	11551ms

Tabla 6: Tiempos de ejecución de las operaciones de la estructura de datos con diferentes conjuntos de datos

5.5 Memoria

Se menciona la memoria que consume el programa para los conjuntos de datos:

	10 abejas	100 abejas	1000 abejas	10000 abejas	100000 abejas
Colisiones	1MB	1MB	2MB	12MB	60MB

Tabla 7: Consumo de memoria de la estructura de datos con diferentes conjuntos de datos

5.6 Análisis de los resultados

detectarColisiones	ArrayList	LinkedList	Octree
Tiempo 10 abejas	1ms	1ms	0.6ms
Tiempo 1000 abejas	7ms	6ms	3ms
Tiempo 10000 abejas	1091ms	1296ms	60ms
Tiempo 100000 abejas	1067620ms	1341574ms	11551ms
Memoria 10 abejas	1MB	1MB	1MB
Memoria 1000 abejas	2MB	2MB	2MB
Memoria 10000 abejas	13MB	13MB	12MB
Memoria 100000 abejas	54MB	55MB	60MB

Tabla 8: Tabla de valores durante la ejecución

En los resultados anteriores se hace evidente que la estructura de datos que se ofrece como solución es mucho más rápida y entre más datos esta diferencia se hace más notable. Lo que deja entre ver que la complejidad pasa del linked list que es $O(n^2)$ a el octree que es $O(n*log(n))$ donde n es el número de abejas y el log(n) la altura del árbol.

6. CONCLUSIONES

Lo más relevante en el reporte es la explicación detallada del problema, el seguimiento de las mejoras al algoritmo inicial y la explicación de por qué implementamos el octree. Los resultados fueron bastante buenos ya que coinciden con las verdaderas colisiones.

En la solución final optimizamos el tiempo obteniendo un algoritmo muy eficiente en este aspecto y preciso en las colisiones que encontró, pero tiene un margen de error dado que no tiene en cuenta colisiones con regiones vecinas.

En una posible continuación del proyecto la idea es adecuarlo a lecturas de datos más grandes ya que se encontraban problemas al evaluar el dataSet de un millón.

6.1 Trabajos futuros

Nos gustaría encontrar una estructura de datos aún más eficiente en tiempo con una complejidad de O(n) ya que la idea es que el algoritmo pueda utilizarse para diversos problemas de colisiones incluso si tienen una base de datos mayor al millón.

AGRADECIMIENTOS

Esta investigación fue soportada parcialmente por La Asociación Nacional de Empresarios de Colombia.

REFERENCIAS

[1] Adrigm. Teoría de colisiones 2D: QuadTree, 2013. Retrieved August 26, 2018, from genbeta: https://www.genbeta.com/desarrollo/teoria-de-colisiones-2d-quadtree

[2] James. Introductory Guide to AABB Tree Collision Detection, 2017. Retrieved August 26, 2018, from azure from the trenches: https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/

[3] James, M. Quadtrees and Octrees, 2018. Retrieved August 26, 2018, from i-programmer: https://www.i-programmer.info/programming/theory/1679-quadtrees-and-octrees.html?start=1

[4] MacDonald, T. Spatial hashing, 2009. Retrieved August 26, 2018, from gamedev: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/spatial-hashing-r2697/

[5] Rebato, C. Por qué las abejas están muriendo y por qué debería importarte, 2015. Retrieved August 26, 2018, from gizmodo: https://es.gizmodo.com/por-que-las-abejas-estan-muriendo-y-por-que-deberia-imp-1717190711

[6] Shebata, O. Redesign Your Display List With Spatial Hashes, 2016. Retrieved August 26, 2018, from game development: https://gamedevelopment.tutsplus.com/tutorials/redesign-your-display-list-with-spatial-hashes--cms-27586

[7] Wikipedia. Quadtree, (N.D). Retrieved August 26, 2018, from wikipedia: https://en.wikipedia.org/wiki/Quadtree

[8] Wikipedia. Octree, (N.D). Retrieved August 26, 2018, from wikipedia: https://es.wikipedia.org/wiki/Octree