Motor de búsqueda optimizado

Xavier Olivenza Busquets

¿Para qué sirve un motor de búsqueda?

Como su nombre indica, el motor de búsqueda, busca cierto tipo de elementos en una zona del espacio, ciñéndose a una/s regla/s como, por ejemplo: "busca todos los elementos que se encuentran a un radio X de este punto", "busca el elemento más cercano a este punto", … Partiendo de estas reglas de que buscar, hay distintas maneras de ordenar el espacio para realizar la búsqueda.

Videojuegos dónde se utiliza o se puede utilizar

Por ejemplo, en un juego como Age of Empires, si le dices a un aldeano "corta árboles para conseguir madera", el aldeano desde la posición dónde está, busca el árbol más próximo y cuando lo encuentra con el motor de búsqueda, traza un algoritmo de *pathfinding* hasta dicho árbol. Una vez ha acabado, si la orden no ha sido revocada, puede volver a buscar un árbol cercano en un radio con el motor de búsqueda y volver a trazar el *pathfinding* hasta éste, si no encuentra ninguno, se le puede decir que busque la edificación más cercana y se cobije, otra vez con el motor de búsqueda, pero en vez de buscar entidades árbol, busca entidades edificio.

Otro ejemplo pueden ser los juegos *tower defense*. Se generan enemigos, que van avanzando en un recorrido, y las torres de defensa, buscan con el motor los enemigos que están a rango de ataque, si es así, atacan. Una vez eliminado el objetivo, busca a ver si hay otro enemigo en rango, si hay uno, procede a atacar, si no, sigue haciendo comprobaciones a cada fotograma o cada X fotogramas en búsqueda de otro enemigo.

Primera opción de cómo buscar algo en el espacio, fuerza bruta

Si lo que queremos es buscar un tipo de entidad del mundo en un área/rango limitado, el método de fuerza bruta nos haría comprobar si cada una de las entidades de ese tipo está o no dentro del área/rango. Por lo tanto, si tenemos 200 tipos de esa entidad, tendremos que hacer 200 comprobaciones. Por lo tanto, la fuerza bruta es lenta, ya que se tienen que hacer tantas comprobaciones como entidades se evalúen.

¿Qué pasa si agrupamos y ordenamos las entidades?, pues que solo tendremos que comprobar cuantos grupos están dentro del área/rango, depende de cómo agrupemos y ordenemos, podemos reducir significativamente el número de comprobaciones. Eso es lo que se consigue con los algoritmos de partición del espacio.

Partición del espacio

La partición del espacio en geometría es el proceso de dividir un espacio en dos o más subconjuntos los cuales no se superponen, es decir si partimos A en los subconjuntos B y C, B no interseca con C y viceversa. Con este tipo de división, cualquier punto puede ser identificado y ubicado en una región concreta. Para la partición del espacio en sus subregiones, se utilizan planos o en dimensiones mayores, hiperplanos.

Los algoritmos de partición del espacio acostumbran a ser jerárquicos, es decir, utilizan árboles para ordenarse. El nodo raíz del árbol, se divide en un numero X de hijos y recursivamente cada hijo se puede volver a dividir, con lo que se genera un árbol.

Estructuras de datos de partición del espacio más comunes:

· Árboles BSP

· Quadtrees

· Octrees

· Árboles k-dimensionales

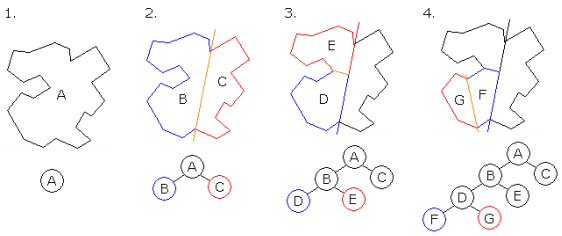
· Árboles R

Partición del espacio binaria

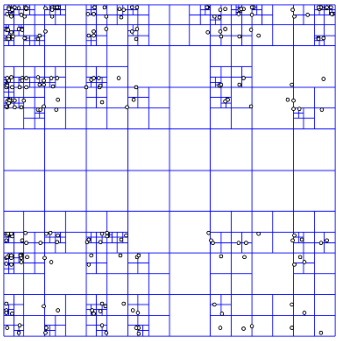
Es un proceso que puede ser visto como una generalización de otras estructuras de particionamiento espacial como los árboles k-d y quadtrees. La división recursiva del espacio del árbol BSP, genera una estructura de datos en árbol que siempre corta cada nodo en dos partes, acción que se realiza en base a unos requerimientos sobre los cuales se limita cuando se divide el nodo. Los planos que realizan las divisiones pueden tener cualquier orientación, no necesariamente están alineados con los ejes.

El particionamiento binario del espacio surgió de la necesidad de los gráficos de computadora de dibujar rápidamente las escenas tridimensionales compuestas de polígonos. Aplicando árboles BSP se obtiene un método rápido de clasificación de polígonos con respecto a un punto de vista dado. Una desventaja de la partición de espacio binario es que la generación de un árbol BSP puede llevar mucho tiempo. Por lo general, se realiza por lo tanto una vez sobre geometría estática, como un paso de pre-cálculo, antes de renderizar u otras operaciones en tiempo real en una escena. El costo de construir un árbol BSP hace que sea difícil e ineficiente implementar directamente objetos en movimiento en un árbol.

Los árboles BSP se pueden utilizar en juegos 3D, como *shooters* en primera persona. Algunos de los motores de videojuegos 3D que utilizan árboles BSP son el Doom Engine (id Tech 1) y Quake Engine y sus descendientes.



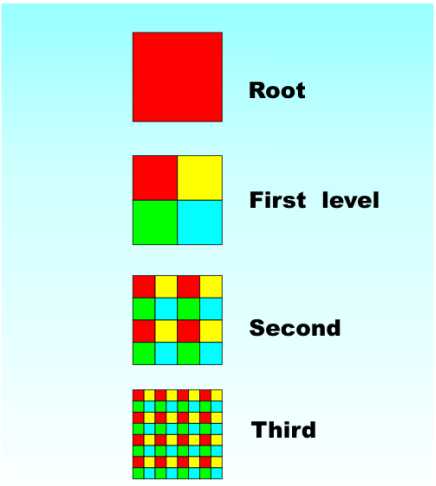
Quadtree



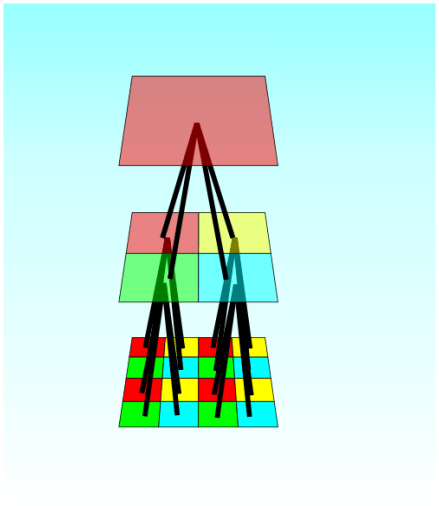
Partiendo de la idea de la partición del espacio y que la partición binaria del espacio divide cada nodo del árbol en dos, debería ser fácil deducir el funcionamiento básico de un quadtree, ésta estructura de datos en árbol divide recursivamente el espacio cortando cada nodo en cuatro subconjuntos que no se superponen, normalmente cuadrados o rectángulos. Generalmente se utilizan en mundos 2D.

Cada nodo tiene una capacidad máxima, cuando se llena, se divide en los subconjuntos pertinentes que se irán llenando, hasta volver a repetir el proceso.

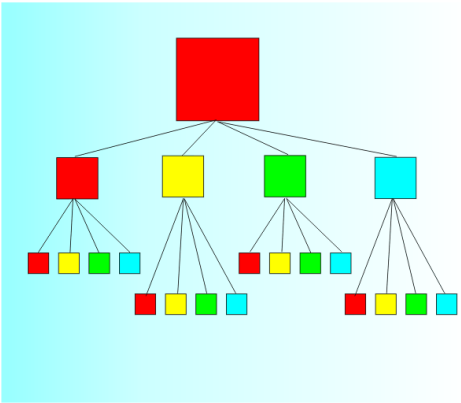
Subdivisiones de un quadtree:



Árbol 3D de un quadtree:

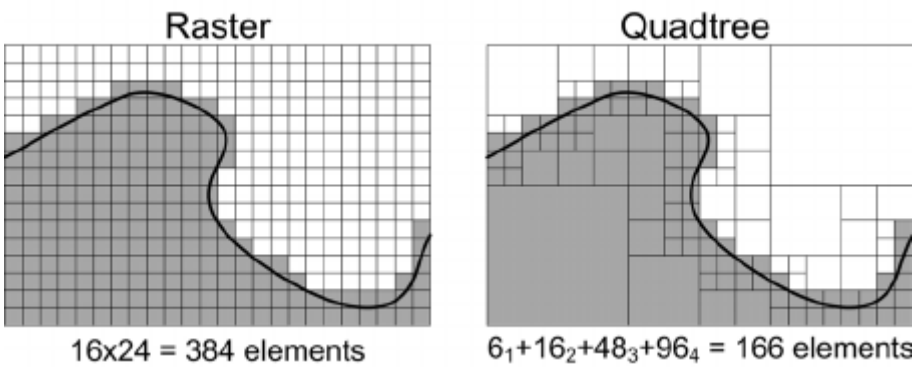


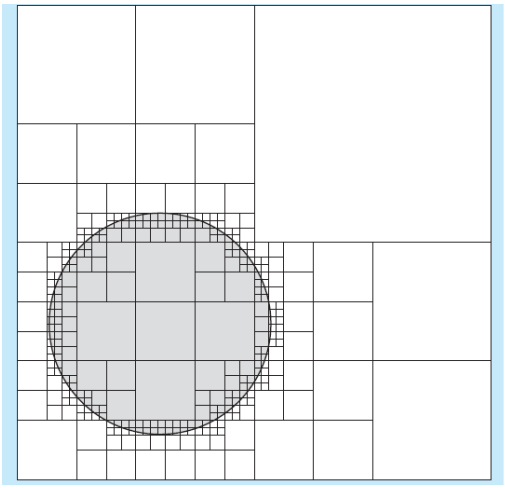
Árbol 2D de un quadtree:



Cuatro usos comunes de quadtrees

·Procesamiento de imágenes





·Bitmap y su representación comprimida de quadtree

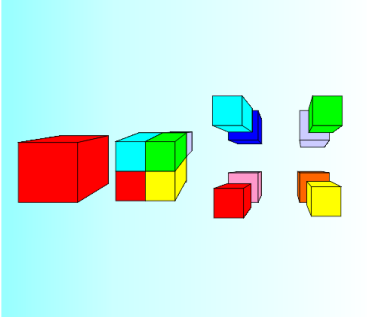
· Detección eficiente de colisiones en dos dimensiones

·Se puede utilizar para optimizar el *camera culling*. Una aproximación en mapas por *tiles*, se podría tener todo el mapa como el nodo raíz y partir el espacio añadiendo cada *tile* al árbol, entonces se puede optimizar que se pinta y que no, dependiendo de si está en pantalla.

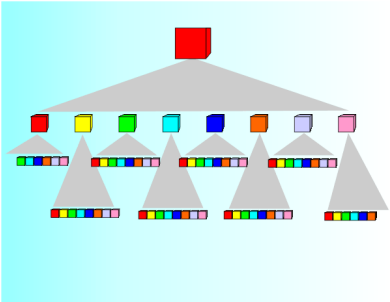
Octree

Partiendo del quadtree debería ser fácil deducir el octree. El quadtree divide cada nodo en cuatro hijos y el octree en ocho, pero en esto hay un matiz, el quadtree se utiliza más en 2D y el octree es su análogo en 3D, por lo que es utilizado en gráficos 3D y motores de videojuegos. El octree divide el paralelepípedo del nodo raíz en cuatro sub paralelepípedos contenidos en el volumen del nodo raíz y sin incluirse unos a otros. Esta división tiene distintos criterios dependiendo de que almacene el octree, o de cómo se haya hecho la implementación.

Subdivisiones de un octree:

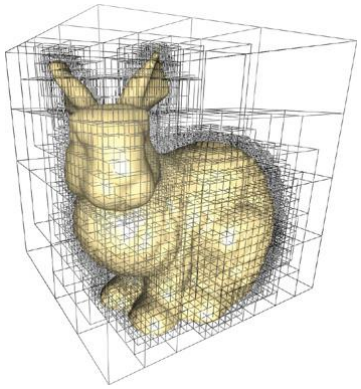


Árbol 2D de un octree:



Cuatro usos comunes de octrees

·Gráficos 3D



·Detección eficiente de colisiones en tres dimensiones

·Frustum Culling/Hidden Surface Determination (Hidden Surface Removal (HSR)) / Occlusion Culling (OC) / Visible Surface Determination (VSD)

·Análisis de elementos finitos

Cómo funciona mi implementación de quadtree

Mi Quadtree sólo acepta iPoint(puntos), queda a vuestro cargo modificarlo para que acepte otras cosas, o templatizarlo, así tenéis que tocar código sí o sí y así aprendéis cómo va el código, por lo que os será más fácil modificar cosas si quereis.

En este apartado explico cómo utilizar el quadtree que he implementado, sólo cogiendo el SDLQuadtree.h/.cpp y poniéndolo en vuestro proyecto, no explico el código de dentro de estos archivos, para más información, entrad en los archivos, están llenos de comentarios al lado de cada línea de código para poder comprender que hace cada cosa.

En el código proporcionado lo que seguidamente explico de cómo utilizar mi SDLQuadtree, está en los archivos j1Scene.h/.cpp, también con algunos comentarios para identificar las distintas zonas y que se hace allí.

En el .h del módulo dónde se vaya a implemetnar el Quadtree, guardar lo siguiente:

`Quadtree* Point_quadtree = nullptr;` Puntero al Quadtree que utilizaremos
`SDL_Rect Quadtree_area_search;` Es el área dónde/rango buscaremos las entidades

En el .cpp:
En la función que inicialice el módulo, en mi caso Start(), crear el nuevo Quadtree, guardarlo en el puntero, y el área de búsqueda.

```
Point_quadtree = new Quadtree(Quadtree_area);
Quadtree_area_search = { 500,200,100,200 };
```

Añadir las entidades que va a manejar, en mi caso en el mismo Start() después de crear el Quadtree.

```
Point_quadtree->Insert(&iPoint( x,y ));
```

Cuando se requiera hacer la búsqueda, en mi caso en el Update() cuando se presione "S", simplemente, realizar la búsqueda utilizando el puntero, y pasando a CollectCandidates un vector para que almacene todas las entidades en rango y el área de búsqueda.

```
Point_quadtree->CollectCandidates->(Points_in_range_quadtree_search,
Quadtree_area_search);
```

Links

Space Partitioning

http://gameprogrammingpatterns.com/spatial-partition.html

Quadtree (implementaciones)

https://github.com/Lectem/YAQ

http://codereview.stackexchange.com/questions/143955/quadtree-c-implementation

Octree

http://www.gamasutra.com/view/feature/131625/octree_partitioning_techniques.php

K-D Tree (implementaciones)

https://rosettacode.org/wiki/K-d_tree

https://github.com/orangejulius/kdtree

https://github.com/gvd/kdtree

K-D Tree VS Quadtree

http://www.i-programmer.info/programming/theory/1679-quadtrees-and-octrees.html

http://stackoverflow.com/questions/13487953/difference-between-quadtree-and-kd-tree

http://gamedev.stackexchange.com/questions/87138/fully-dynamic-kd-tree-vs-quadtree

Quadtrees y Hilbert Curves

http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves