Computación Gráfica

Clase 4 - Estructuras de aceleración

El problema

Queremos maximizar performance.

Los bounding volumes reducen los tests de colisión rayo / mesh a 1 por objeto, en vez de 1 por polígono.

Aún así, una escena con millones de meshes, requiere millones de pruebas de colisión contra sus bounding volumes.

La solución

El código más rápido es aquel que nunca se ejecuta.

Evitar tener que realizar los tests en primer lugar!

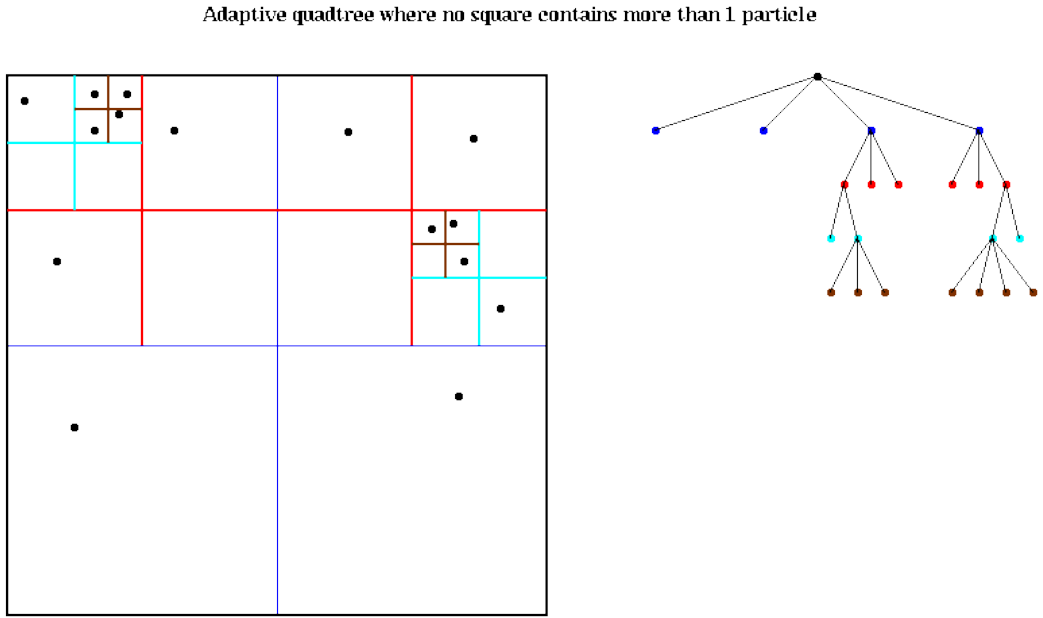
Ya lo vimos en EDA, Divide & Conquer.

Objetivo

Partir la escena de forma de poder descartar partes enteras de la misma sin siquiera observarla.

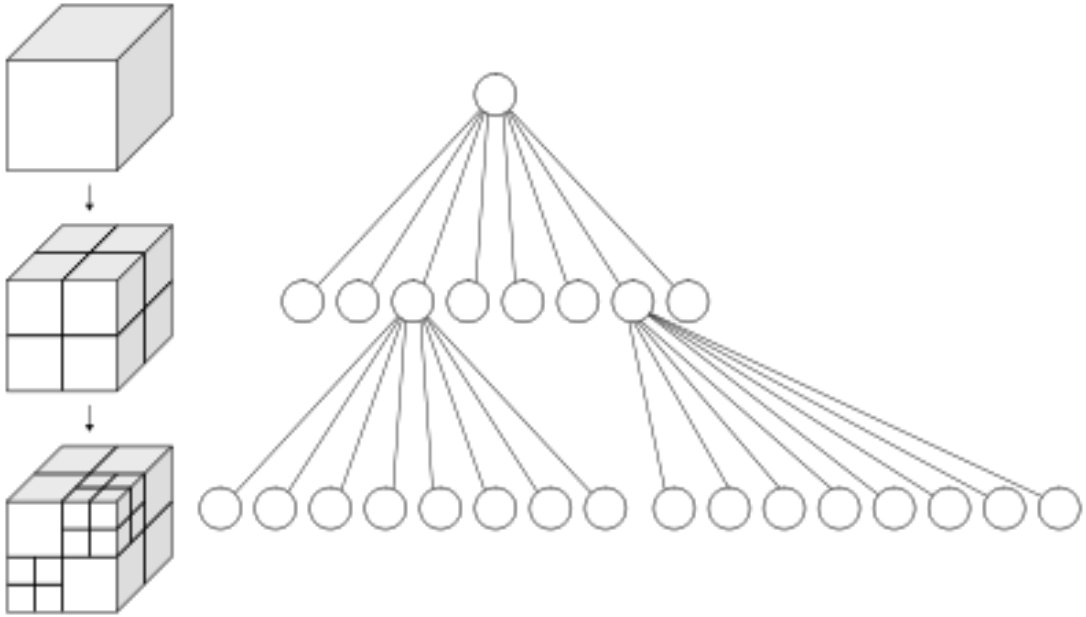
Primer aproximación

En EDA vimos Quadtrees:

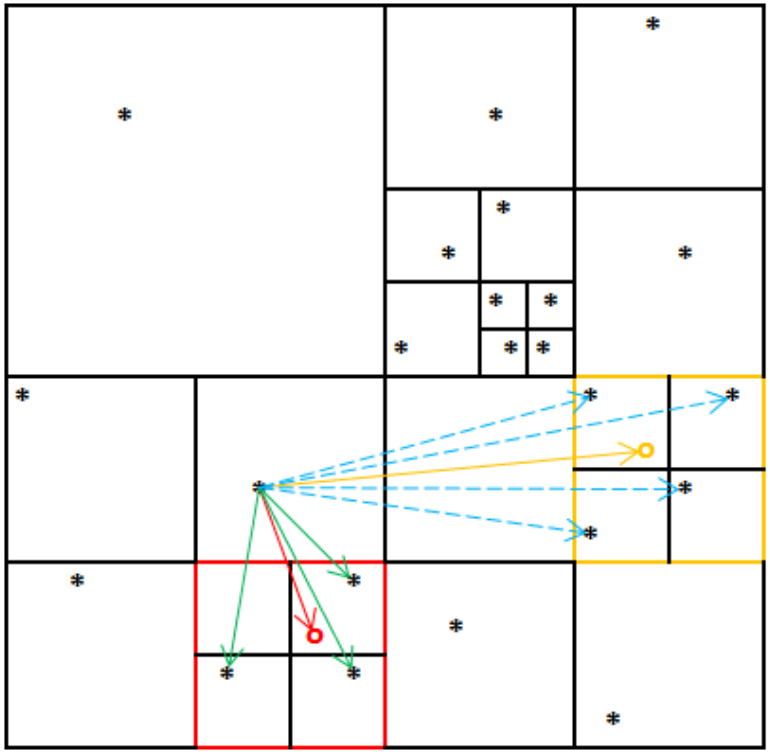


Primer aproximación

En 3D… Octtrees!



Octrees - Transversal



Octtrees

Pros:

- Fácil de implementar

Cons:

- Asumen una subdivisión uniforme de objetos en la escena.
- Es obligatorio definir límites al worldspace (valores máximos y mínimos)

Octrees - Moving on...

Sería ideal, si pudiesemos dividir la escena arbitrariamente para separar uniformemente los objetos, lo que permitiría a priori reducir las comparaciones en $O(\log_m(n))$

En EDA vimos *RB-Trees* para 2 dimensiones, extendido a *k* dimensiones, *k-D-Tree*

KDTrees

Se particiona el espacio usando planos paralelos a los ejes.

Los puntos por los que se realiza esta partición son arbitrarios.

Dado que no hay obligación de elegir el punto medio, no necesito acotar el espacio de la escena.

KDTree - implementación básica

```
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtrees
    var tree_node node;
    node.axis := axis;
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

KDTree

Es la mediana la mejor opción?

- sólo si no hay muchos objetos intersectandola…

Es iterar entre los planos la mejor opción?

- no siempre!

Es tener una profundidad fija la mejor opción?

KDTree - SAH

En 2006, Ingo Wald et al.

“On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)”

Proponen usar “*Surface Area Heuristic*” para elegir como particionar el espacio.

KDTree - SAH

Busca la división que probabilisticamente tenga el menor costo para transversar el árbol.

$$P(V_{sub}|V) = \frac{SA(V_{sub}}{SA(V)}$$

$$Cv(p) = K_T + P(V_i|V)C(V_i) + P(V_d|V)C(V_d)$$

BSP

Binary Space Partitioning

- Generalización de KDTree.
- Particiona el espacio, pero los planos no tienen por qué estar alineados a los ejes.

Usa los planos que contienen polígonos de la escena como candidatos.

BSP

Pros:

- Al tener mayores grados de libertad puede generar mejores resultados.

Cons:

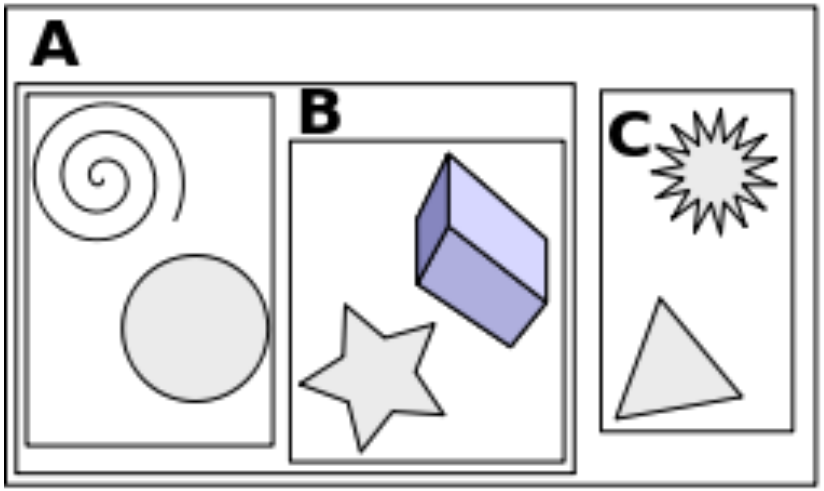
- Su generación es costosa sin una buena heurística para elegir planos.

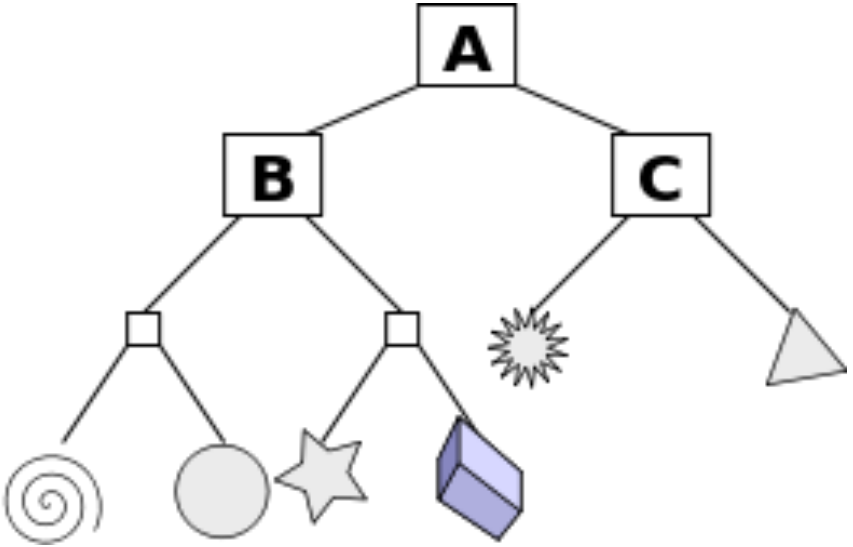
BVH

Bounding Volume Hierarchy

- Se agrupan objetos cercanos en bounding volumes
- Se anidan bounding volumes generando un árbol
- Puede hacerse bottom-up o top-down

BVH





BVH

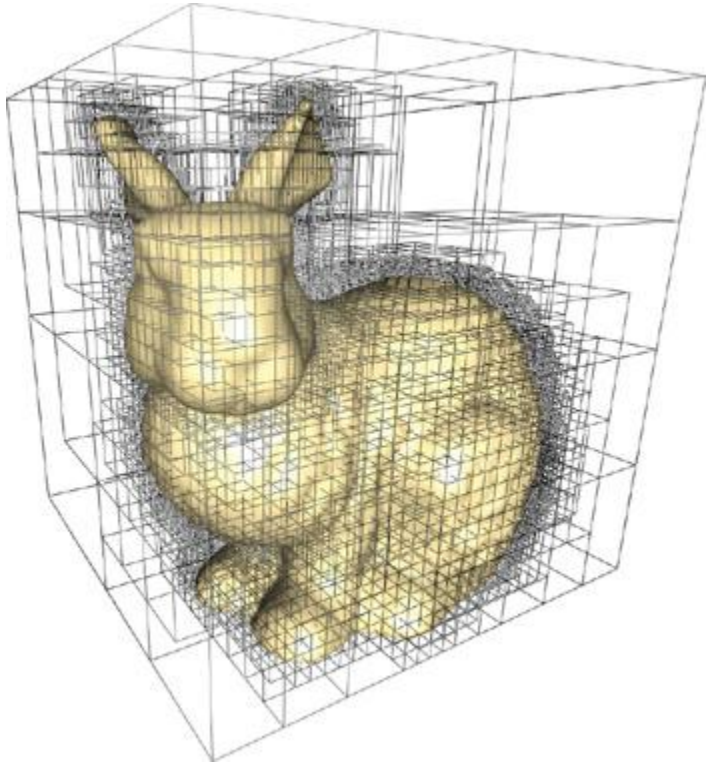
Pros:

- Más sencillo para implementar que BSP
- Se construye ~10X más rápido
- Muy bueno para escenas con movimiento

Cons:

- Su performance es inferior a la de BSP

Aplicación a meshes



Aplicación a meshes

- Se genera un árbol de escena en base a los bounding volumes.
- Los meshes internamente se subdividen con un árbol que contiene los polígonos.

No es necesario usar la misma estructura en ambos niveles.

Otras optimizaciones

El principio de localidad espacial es fundamental para múltiples optimizaciones.

Ya vimos esto aplicado a matrices y vectores.

Ray Packets

- Rayos con dirección muy similar, muy probablemente hagan el mismo recorrido por el árbol.
- Esto reduce los lookups de nodos en memoria.

Ray Packets

Su eficiencia cae cuando los paquetes son demasiado chicos.

- al salir de la cámara es fácil tener paquetes de rayos de recorrido similar de tamaño arbitrario
- al computar efectos de luz es más complejo

Render Buckets

Subdividir la escena con una cuadrícula de tamaño fijo, y dejar que un único thread procese cada área.

Dado que cada área es pequeña, los rayos recorrerán el árbol similarmente, y colisionarán contra el mismo set de objetos.

Render Buckets

Si hay único thread por core, el cache de dicho thread tendrá un mejor nivel de hits.

Evitar saltos

Ya hablamos de evitar `for`s evitando arrays donde sea posible.

Ray Packets no es la excepción.

Armar implementaciones de distintos tamaños.

Evitar saltos

También es bueno evitar `if`s, como vimos al hablar de `lerp`.

En KDTree, podríamos tener 3 implementaciones de nodo específicas para cada eje.