Spatial Search Data Structures

Corso di dottorato: Geometric Mesh Processing

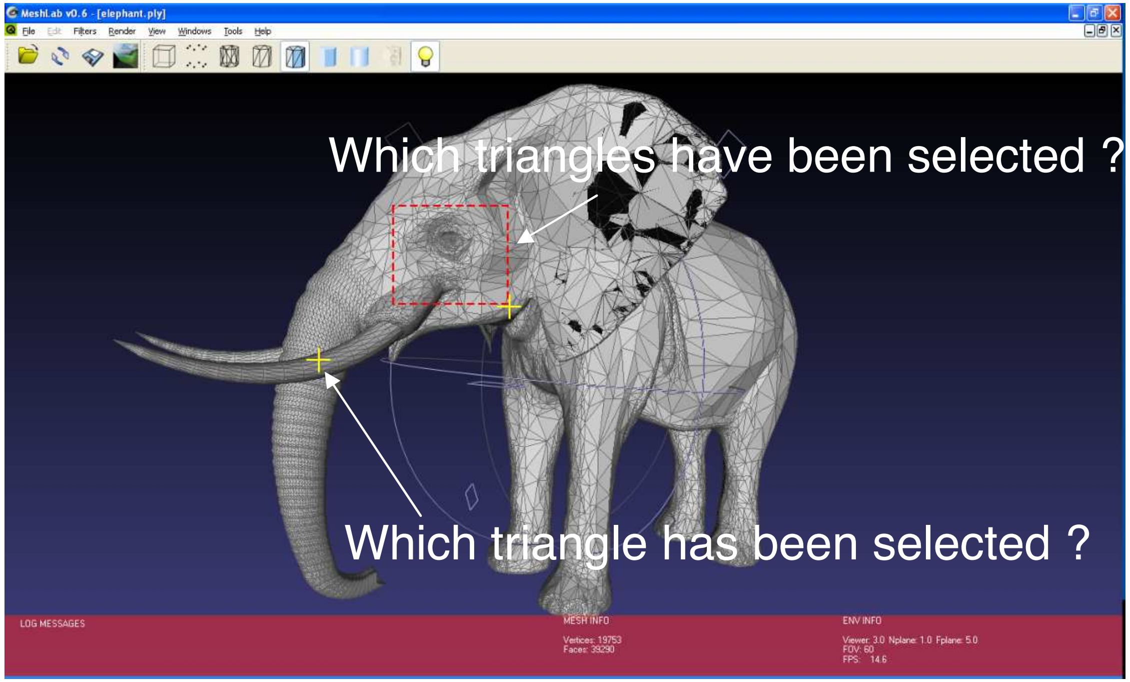
Fabio Ganovelli
fabio.ganovelli@isti.cnr.it

Problem statement

- Let *m* be a mesh:
  - Which is the mesh element closest to a given point *p*?
  - Which are the elements inside a given region?
  - Which elements are intersected by a given ray *r?*
- Let *m'* be another mesh:
  - Do *m* and *m'* intersect? If so, where?

- A spatial search data structure helps to answer efficiently to these questions

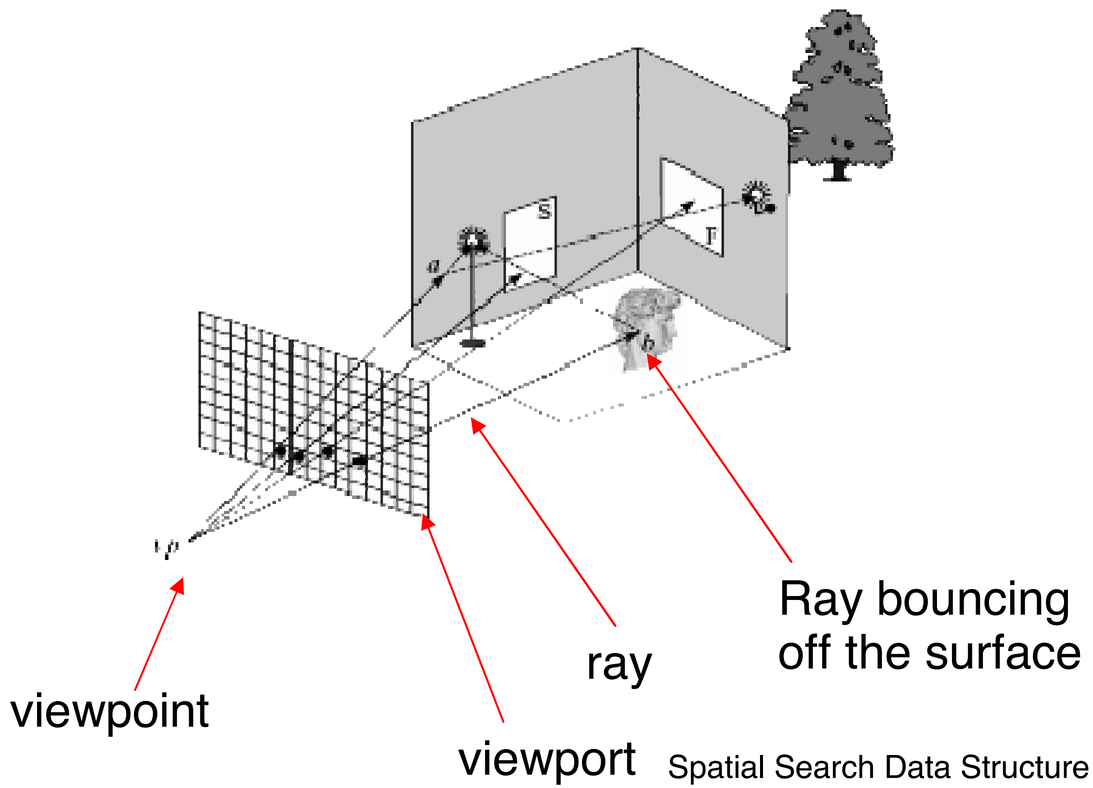
Motivations



- Picking on a point
- Selecting a region

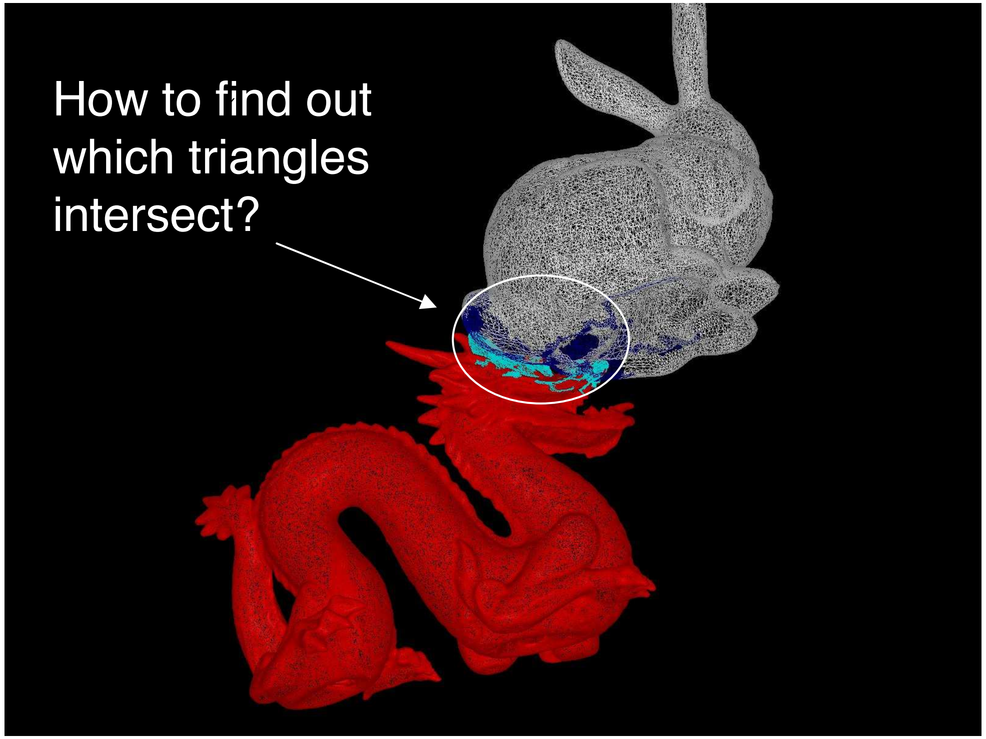
Motivations^{cntd}

- Ray tracing: shoot a ray for each pixel, see what it hits, possibly recur, compute pixel color
- Involves plenty of ray-objects intersections



Motivations$^{cntd^{cntd}}$

- Collision detection: in dynamic scenes, moving objects can collide.

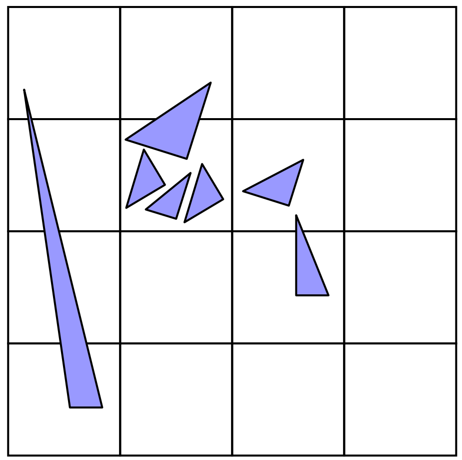


Motivations$^{cntd^{cntd^{cntd}}}$

- Without any spatial search data structure the solutions to these problems require $O(n)$ time, where n is the numbers of primitives ($O(n^2)$ for the collision detection)
- Spatial data structure can make it (average) constant
  - ..or average logarithmic

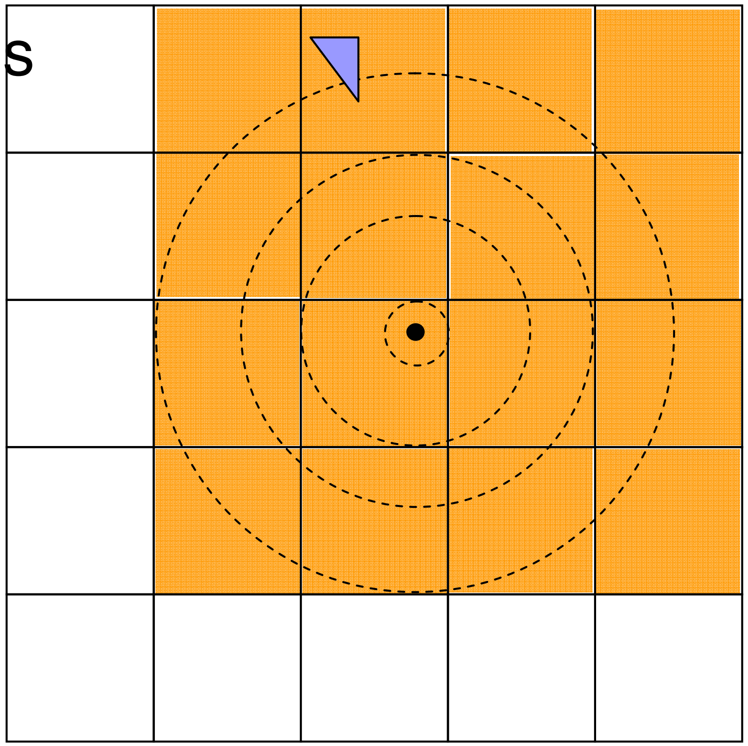
Uniform Grid (1/4)

- **Description**: the space including the object is partitioned in cubic cells; each cell contains references to “primitives” (i.e. triangles)
- **Construction**.
  Primitives are assigned to:
  - The cell containing their feature point (e.g. barycenter or one of their vertices)
  - The cell**s** spanned by the primitives



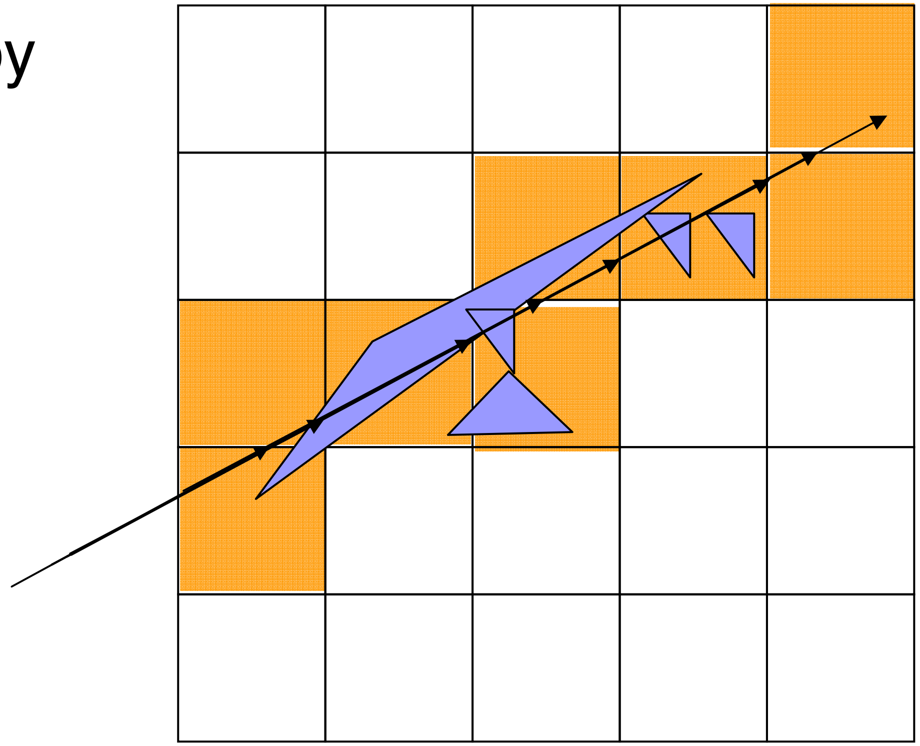
Uniform Grid (2/4)

- **Closest element** (to point p):
  - Start from the cell containing p
  - Check for primitives inside growing spheres centered at p
  - At each step the ray increases to the border of visited cells
- **Cost.**
  - Worst: O(#cells+n)
  - Average; O(1)



Uniform Grid (3/4)

- **Intersection with a ray**:
  - Find all the cells intersected by the ray
  - For each intersected cell, test the intersection with the primitives referred in that cell
  - Avoid multiple testing by flagging primitives that have been tested (*mailboxing*)
- **Cost:**
  - Worst: $O(\#cells + n)$
  - Aver: $O(\sqrt[d]{\#cells} + \sqrt[d]{n})$

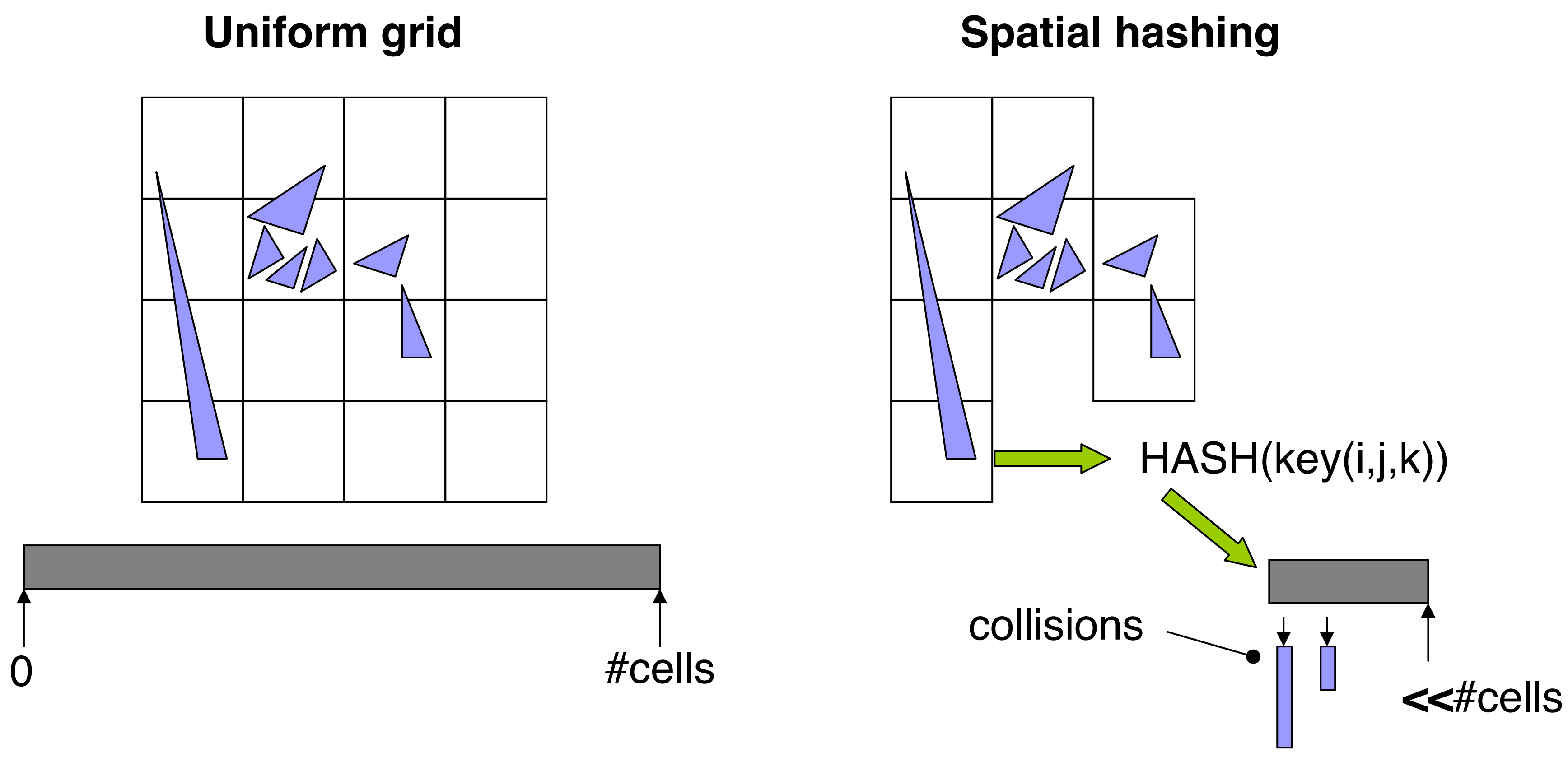


Uniform Grid (4/4)

- **Memory occupation:** $O(\#cells + n)$
- **Pros:**
  - Easy to implement
  - Fast query
- **Cons**:
  - Memory consuming
  - Performance **very** sensitive to distribution of the primitives.

Spatial Hashing (1/2)

- The same as uniform grid, except that only non empty cells are allocated

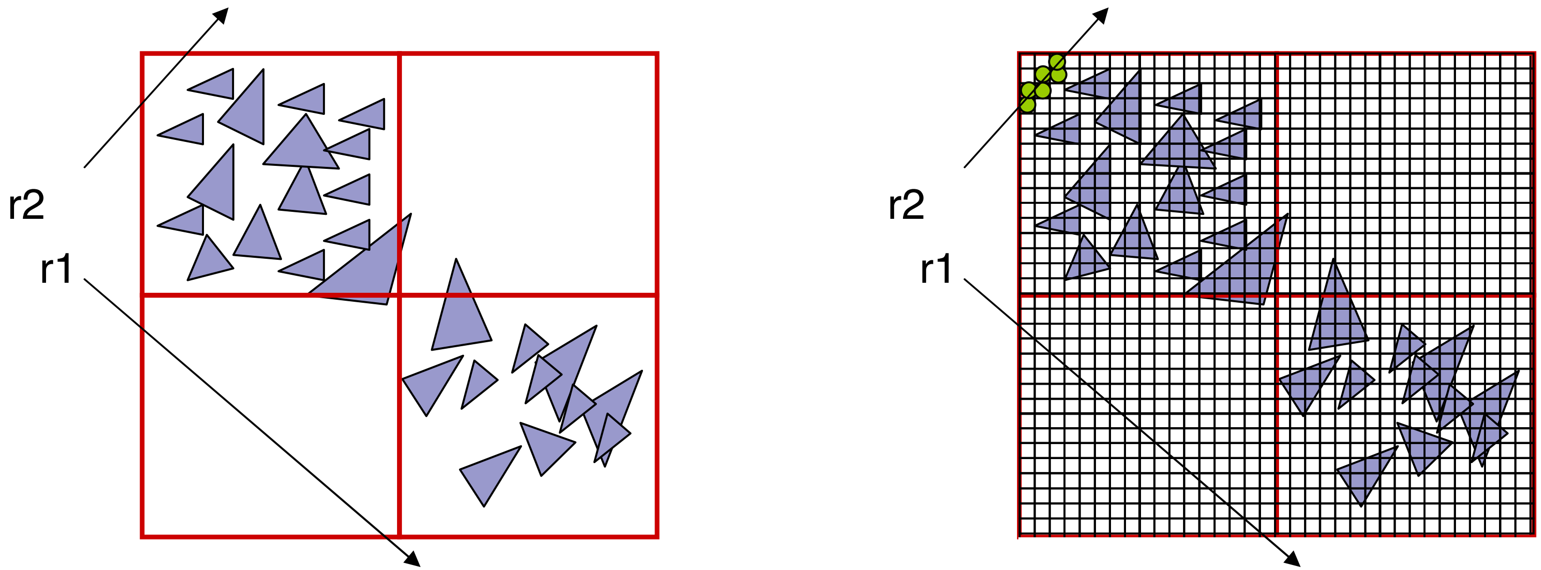


Spatial Hashing (2/2)

- **Cost:** same as UG, except that in worst case the access to a cell is *O(#cells)* because of collisions
- **Memory occupation:**
  - **Worst. :** $O(\#cells)$
  - **Aver. :** $O\left(\left(\frac{\#cells}{Vol}\right)^{\frac{2}{3}} \cdot S\right) \quad \mathrm{S:surface, Vol:Volume}$
- **Pros:**
  - Easy to implement
  - Fast query if **good hashing** is done
  - Less memory consuming
- **Cons**:
  - Performance **very** sensitive to distribution of the primitives.

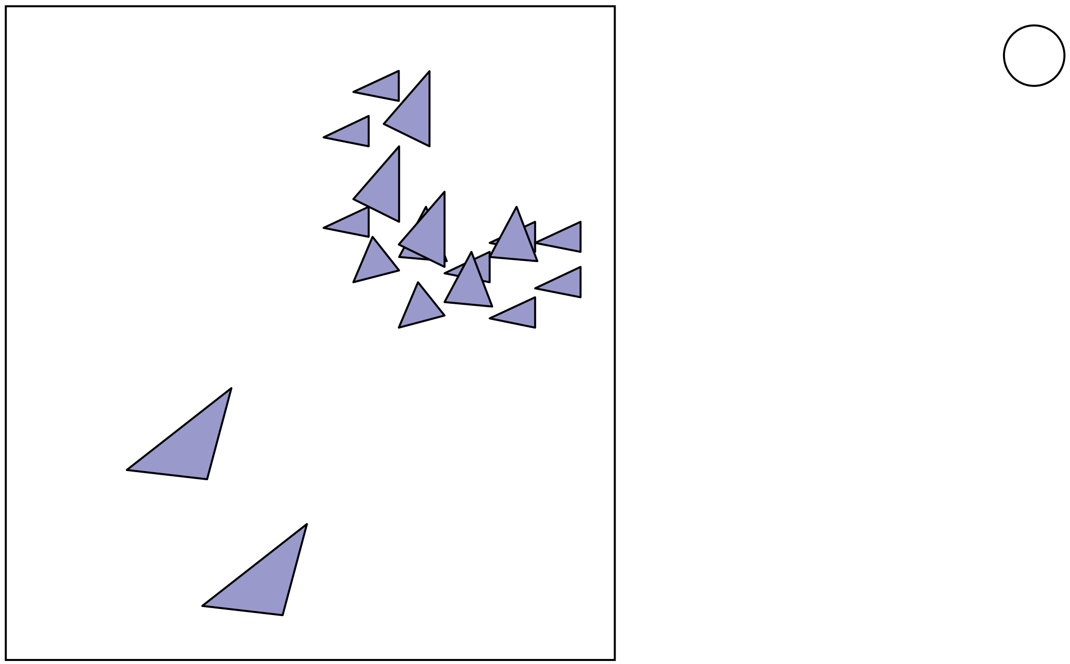
Beyond UG

- Uniform grids are **input insensitive**
- What's the best choice for the example below?



Hierarchical Indexing of Space

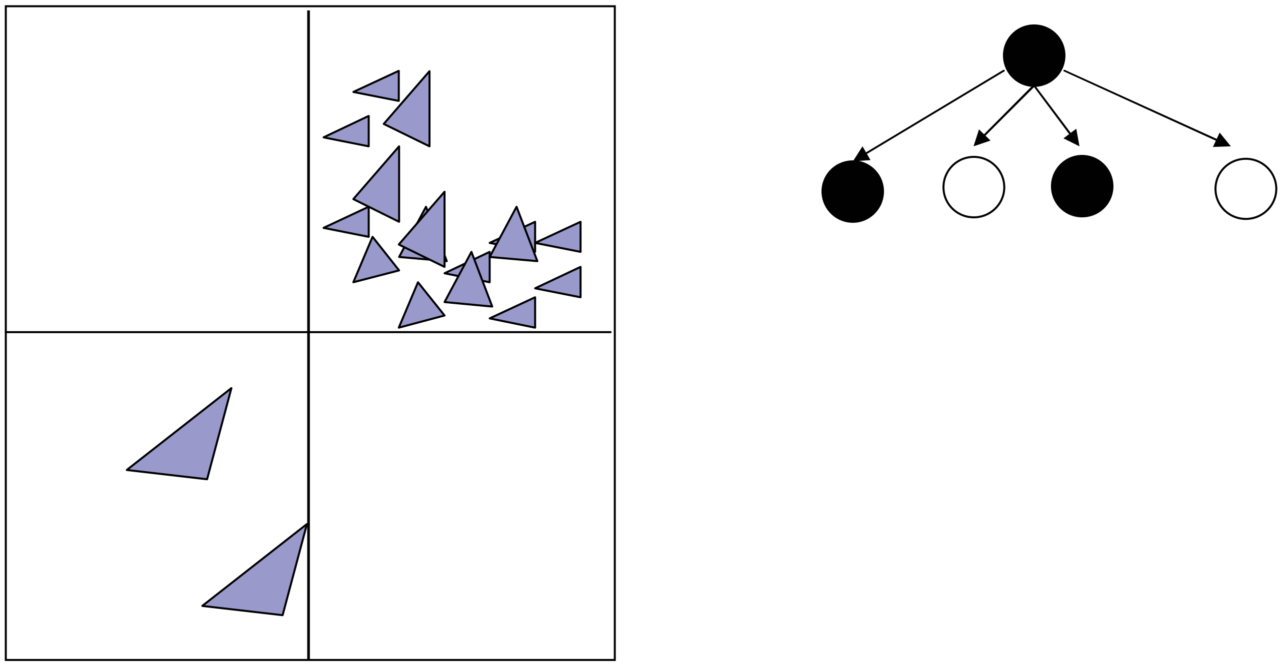
- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ..recursively



Spatial Search Data Structure

Hierarchical Indexing of Space

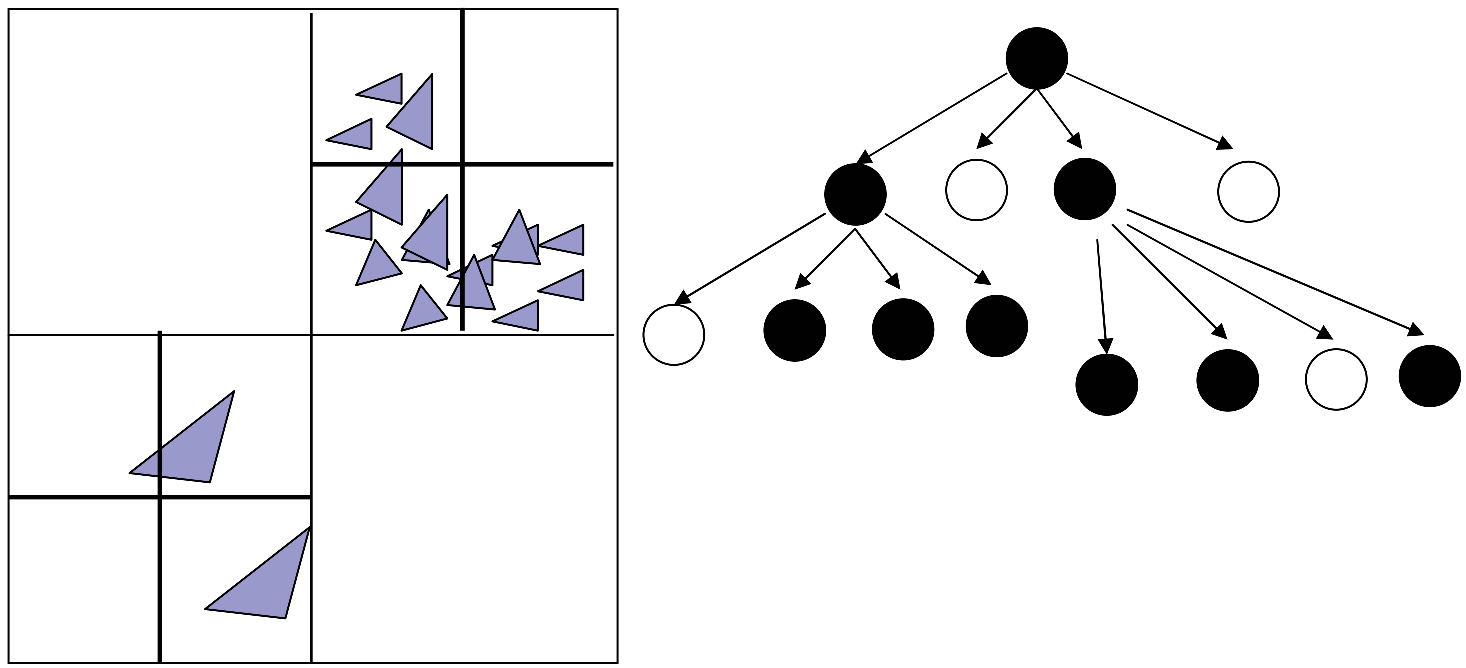
- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ..recursively



Spatial Search Data Structure

Hierarchical Indexing of Space

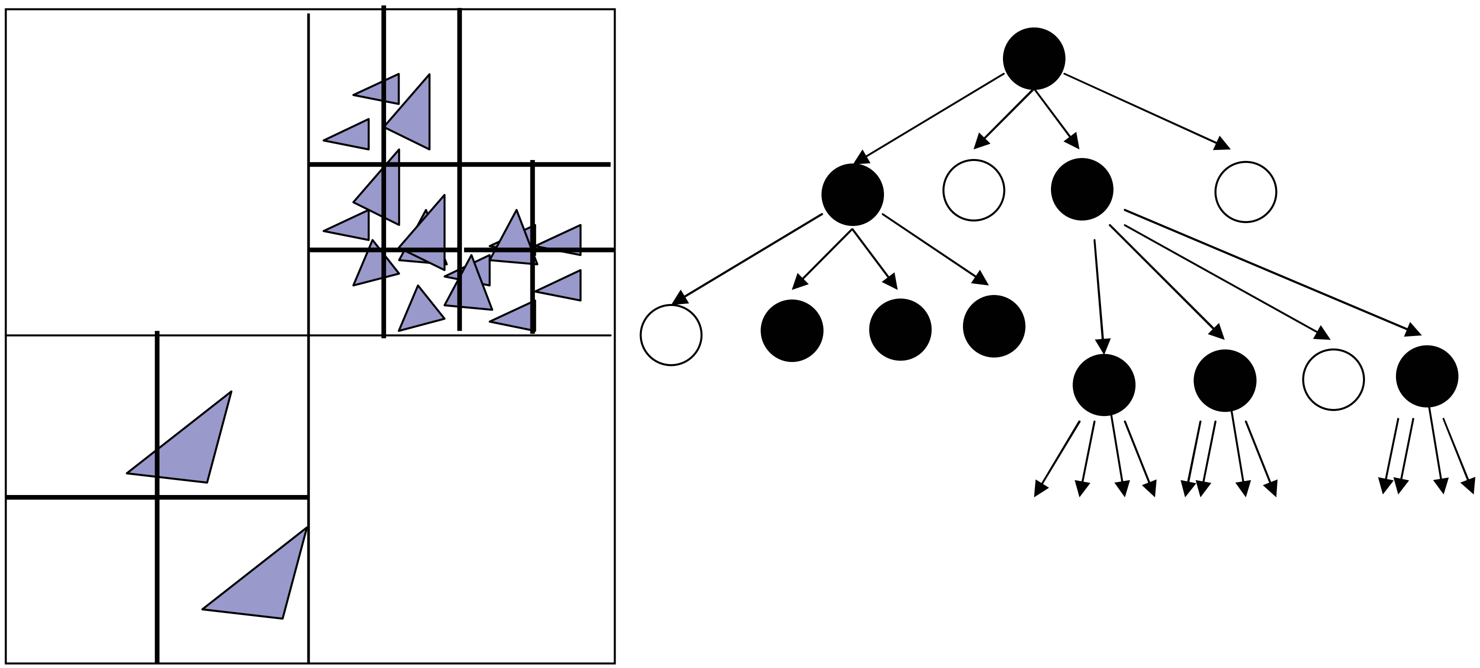
- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ..recursively



Spatial Search Data Structure

Hierarchical Indexing of Space

- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ..recursively



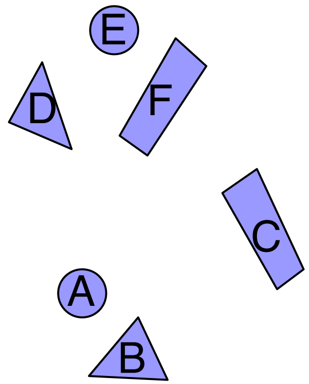
Spatial Search Data Structure

Basic Facts

- The queries correspond to a visit of the tree
  - The complexity is sublinear in the number of nodes (logarithmic)
  - The memory occupation is linear
- A hierarchical data structure is characterized by:
  - Number of children per node
  - Spatial region corresponding to a node

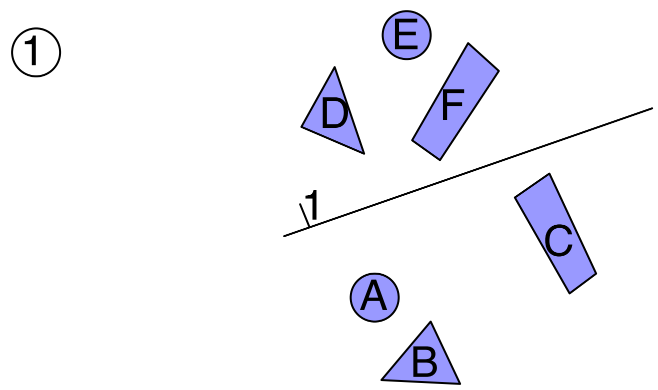
Binary Space Partition-Tree (BSP) (1/3)

- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
  - therefore a node always corresponds to a **convex region**



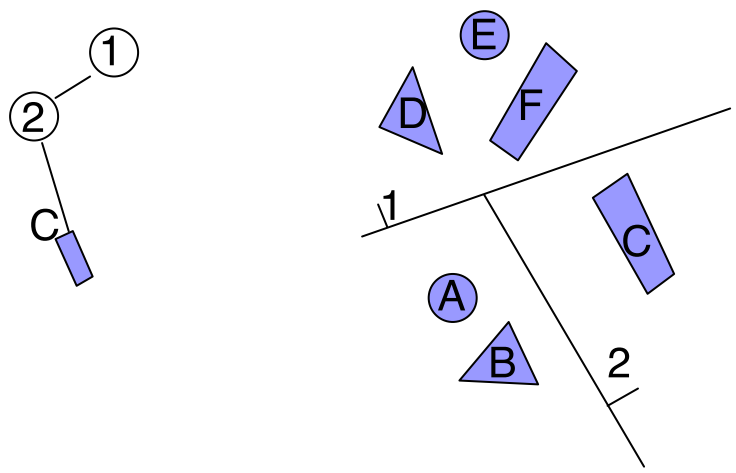
Binary Space Partition-Tree (BSP) (1/3)

- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region



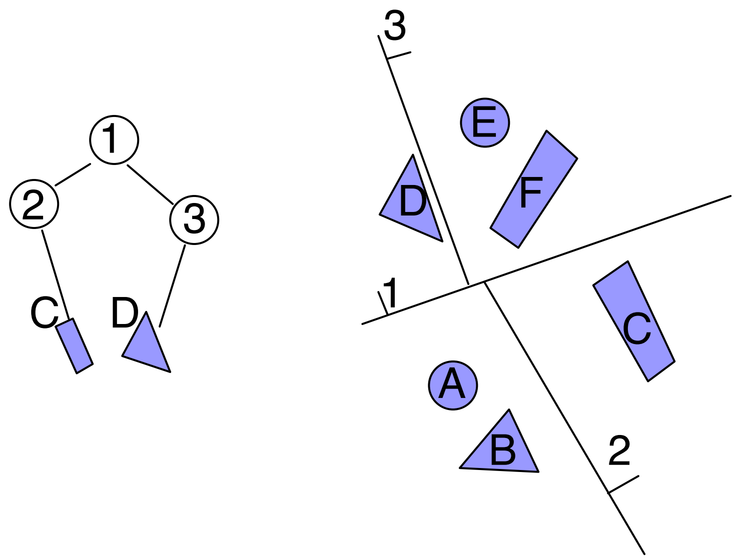
Binary Space Partition-Tree (BSP) (1/3)

- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region



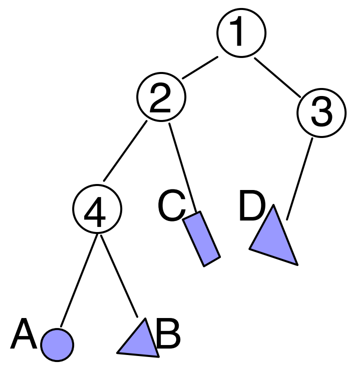
Binary Space Partition-Tree (BSP) (1/3)

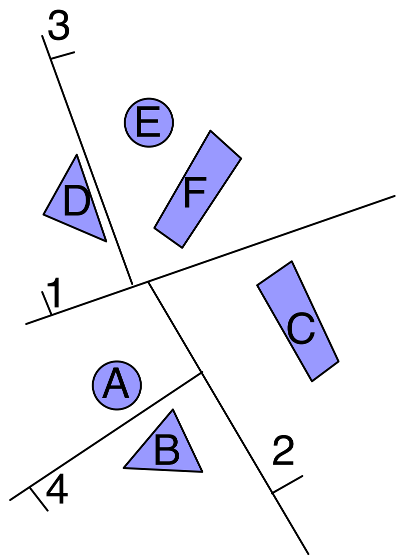
- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region



Binary Space Partition-Tree (BSP) (1/ 3)

- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region

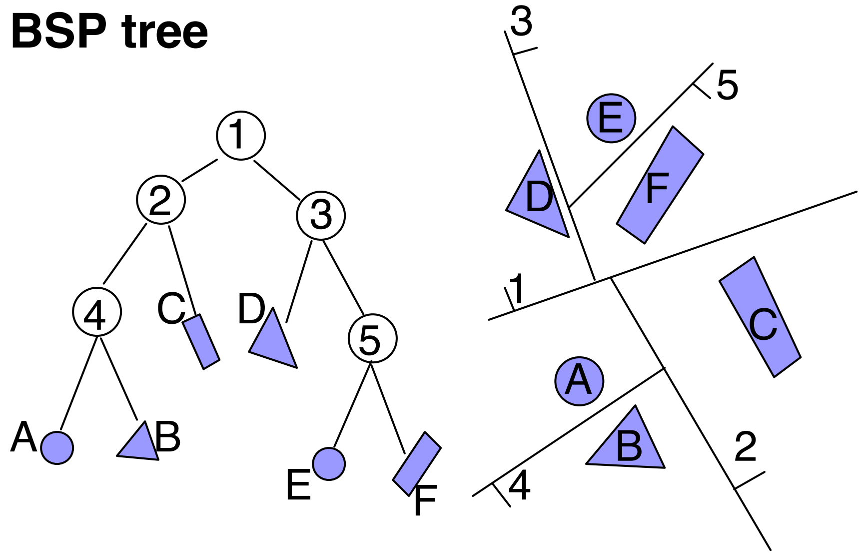




Binary Space Partition-Tree (BSP) (1/ 3)

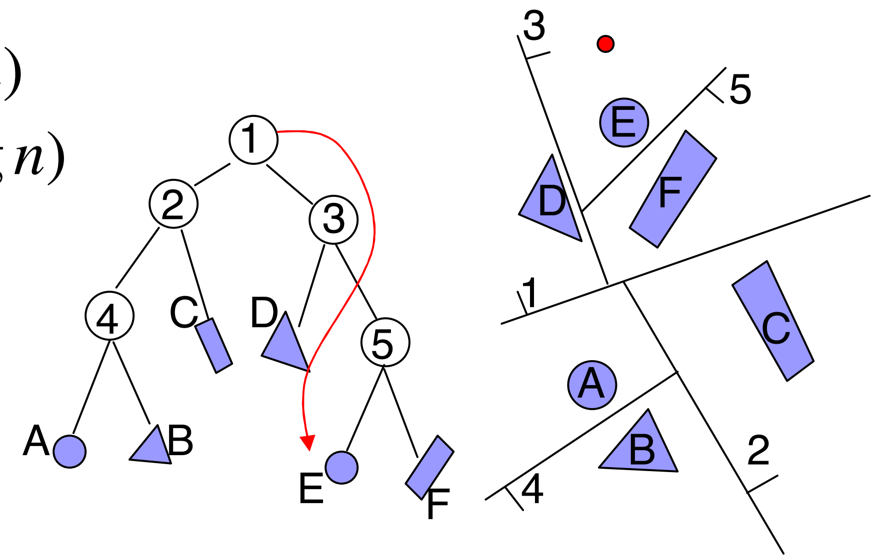
- **Description**:
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region

BSP tree



Binary Space Partition-Tree (BSP) (1/ 3)

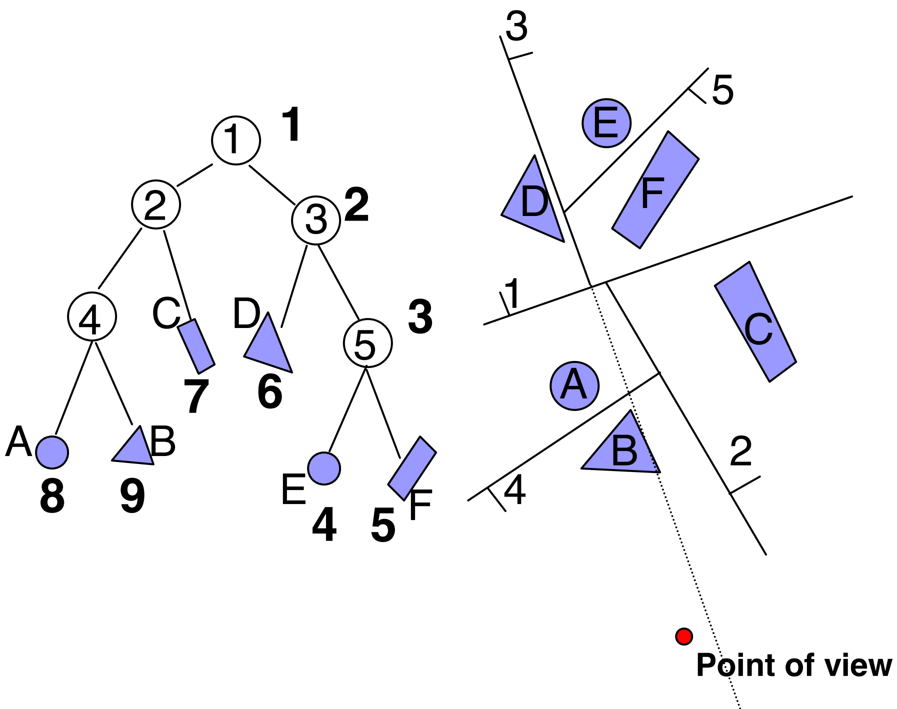
- **Query**: is the point *p* inside a primitive?
  - Starting from the root, move to the child associated with the half space containing the point
  - When in a leaf node, check all the primitives
- Cost:
  - Worst: $O(n)$
  - Aver: $O(\log n)$



BSP tree

BSP-Tree For Rendering

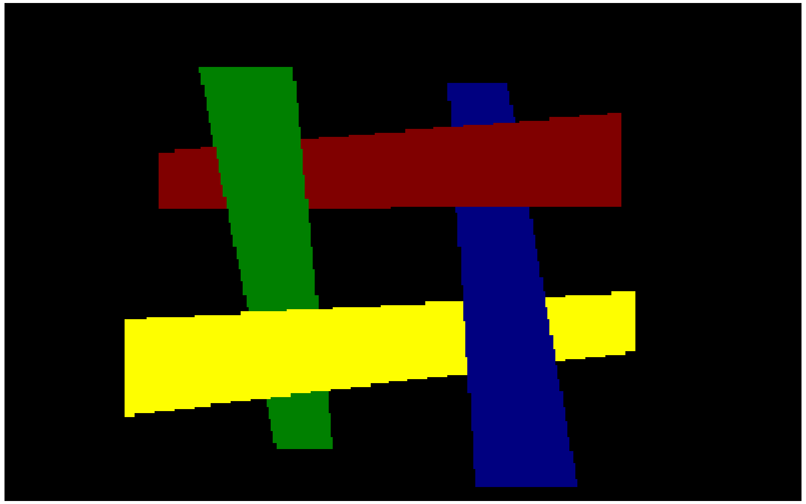
- ordering primitives back-to-front




```
void DrawBackToFront(n,p){
if(IsLeaf(n))
   Draw(n);
if( InNegativeHS(p,n) )
  DrawBackToFront(RightChild(n),p);
else
  DrawBackToFront(LeftChild(n),p);
}
```

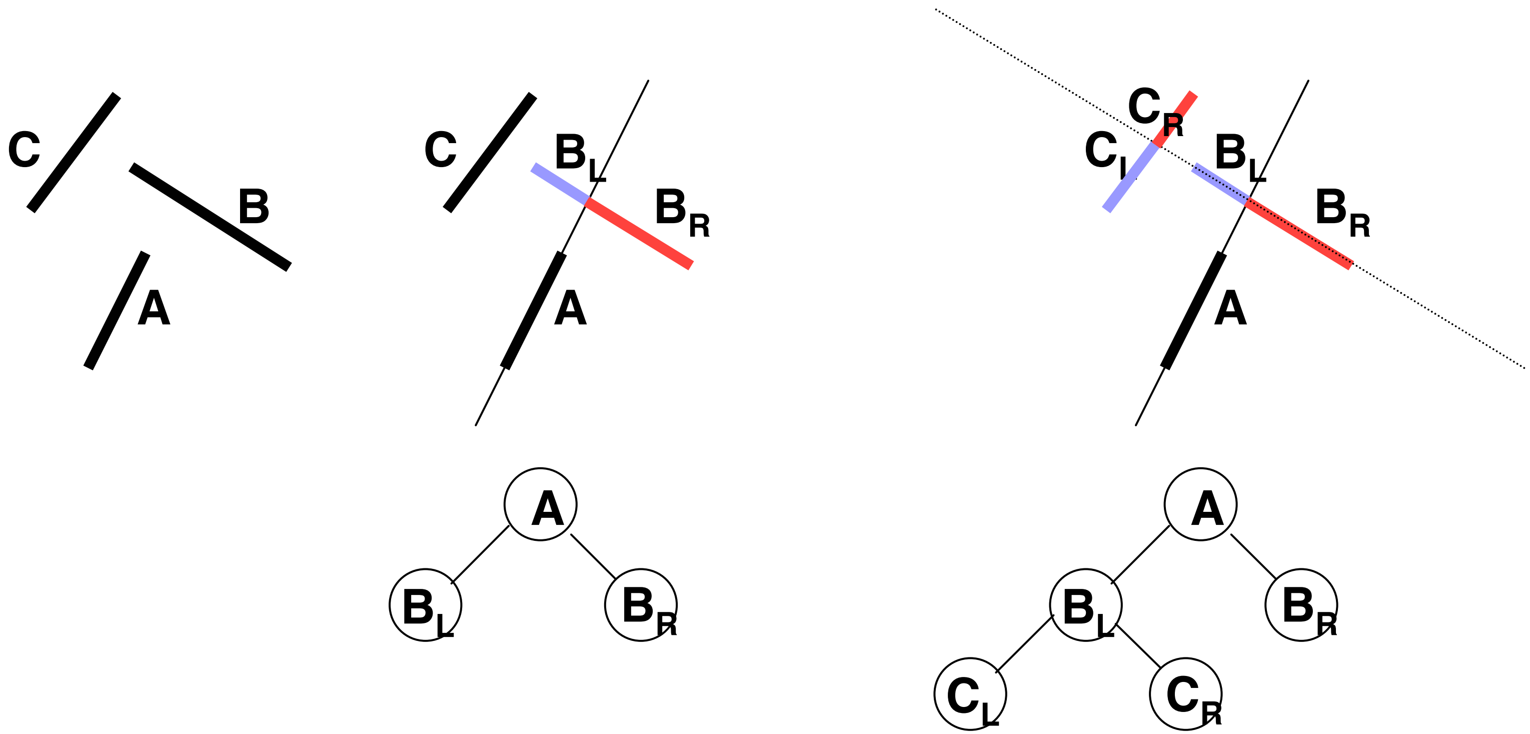
BSP-Tree For Rendering

- **Not so fast:** set of polygons not always separable by a plane



Binary Space Partition-Tree (BSP) (3/3)

- **Auto-partition :**
  - use the extension of primitives as partition planes
  - Store the primitive used for PP in the node



Bulding a BSP-Tree

- Building a BSP-tree requires to choose the partition plane
- Choose the partition plane that:
  - Gives the best balance ?
  - Minimize the number of splits ?
  - …..it depends on the application

- Cost of a BSP-Tree

Cost of visiting $T_{L[r]}$

$$C(T) = 1 + P(T_L)\ C(T_L) + P(T_R)\ C(T_R)$$

Probability that $T_{L[R]}$ is visited if T has been visited

Bulding a BSP-Tree: example

$$C(T) = 1 + P(T_L)\ C(T_L) + P(T_R)\ C(T_R)$$

$$C(T) = 1 + \quad |S_L|^{\alpha} + \quad |S_R|^{\alpha} + \quad \beta\, s$$

$$S_{L[R]} = number\ of\ primitives\ in\ the\ left\ [right]\ subtree$$

$$s \quad = number\ of\ primitives\ split\ by\ the\ chosen\ plane$$

- α, β *used for tuning*
  - Big alpha, small beta yield a balanced tree (good for in/out test)
  - Big beta, small alpha yield a smaller tree (good for visibility order)

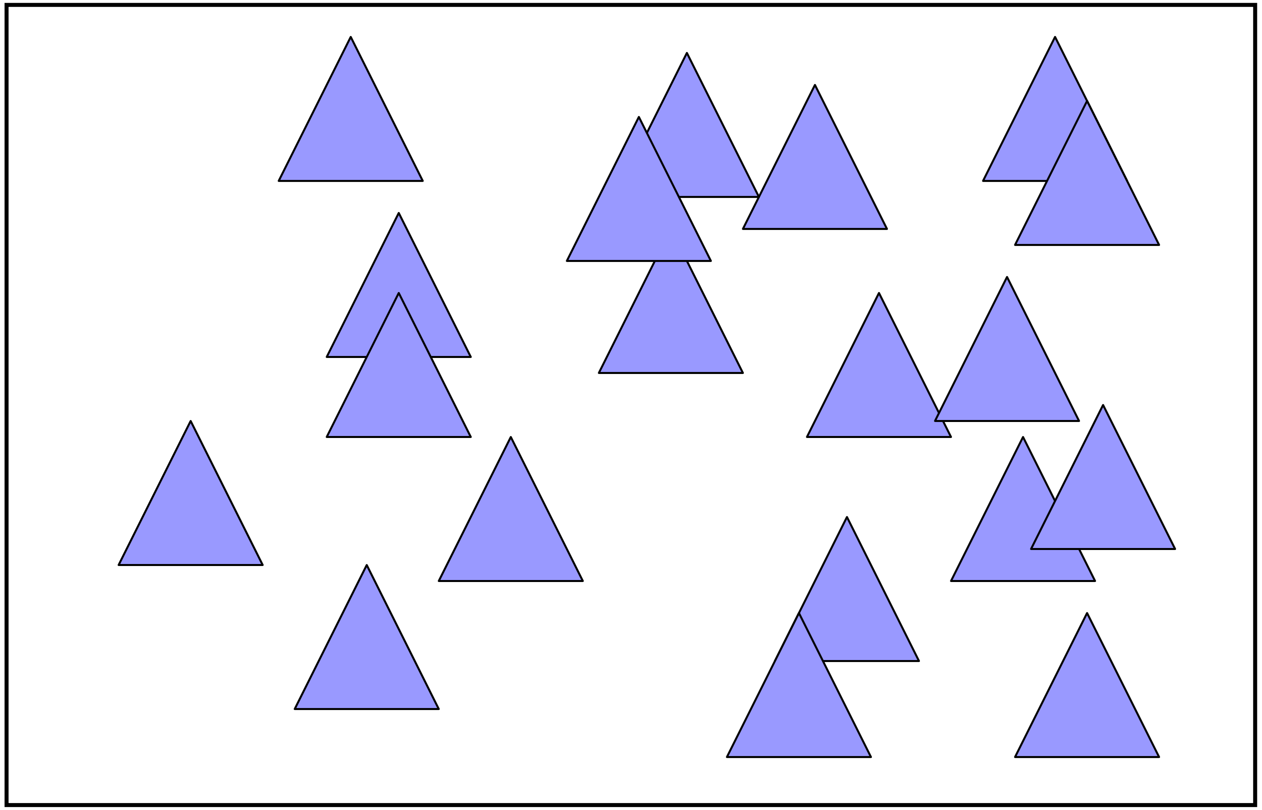
Binary Space Partition-Tree (BSP)

- **Memory occupation:** $O(n)$
  - For each node:
    - (d+1) floatig point numbers (in d dimensions)
    - 2 pointers to child node
- Cost of descending the three:
  - d products, d summations (dot product d+1 dim.)
  - 1 random memory access (follow the pointer)
- Less general data structures can be faster/ less memory consuming

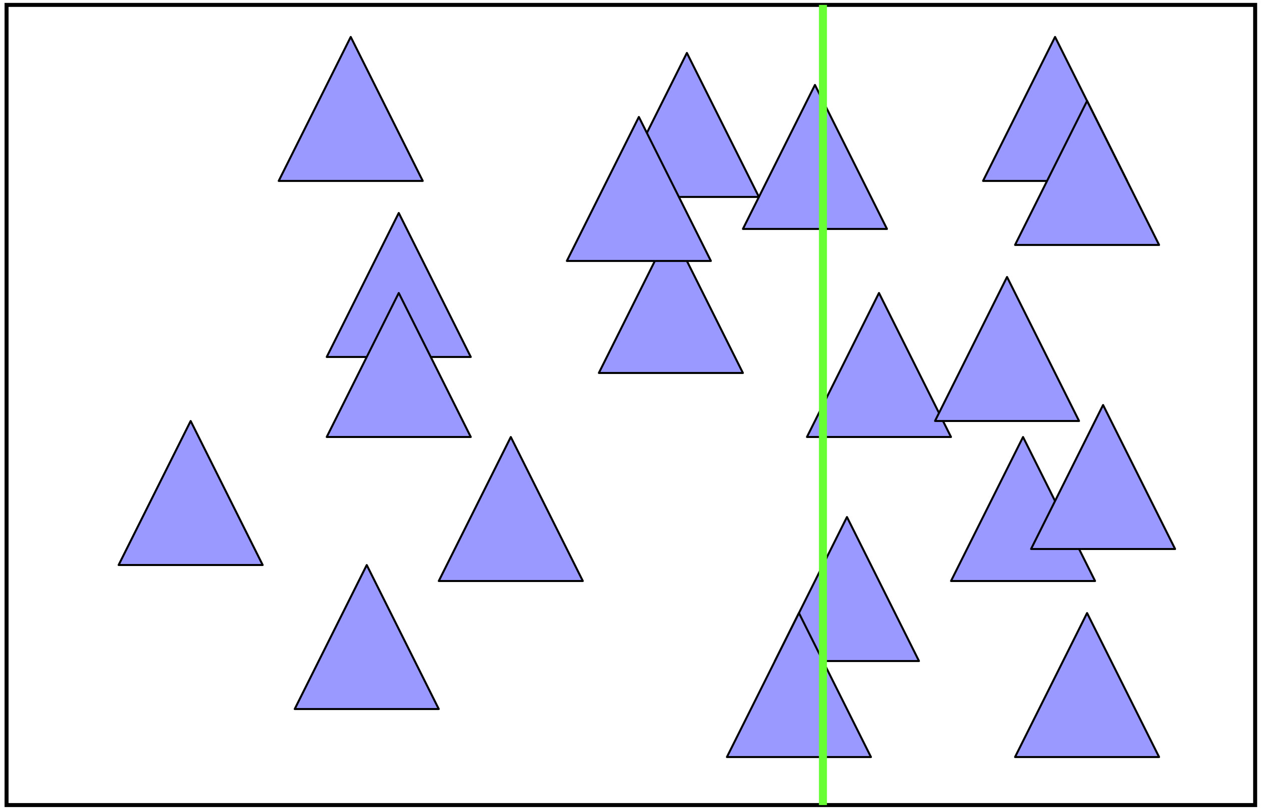
kd-tree

- Kd-tree : k dimensions tree
- È una specializzazione dei BSP in cui i piani di partizione sono ortogonali a uno degli assi principali
- Scelte:
  - L’asse su cui piazzare il piano
  - Il punto sull’asse in cui piazzare il piano
- Vantaggi sui BSP:
  - determinare in quale semispazio risiede un punto costa un confronto
  - La memorizzazione del piano richiede un floating point + qualche bit

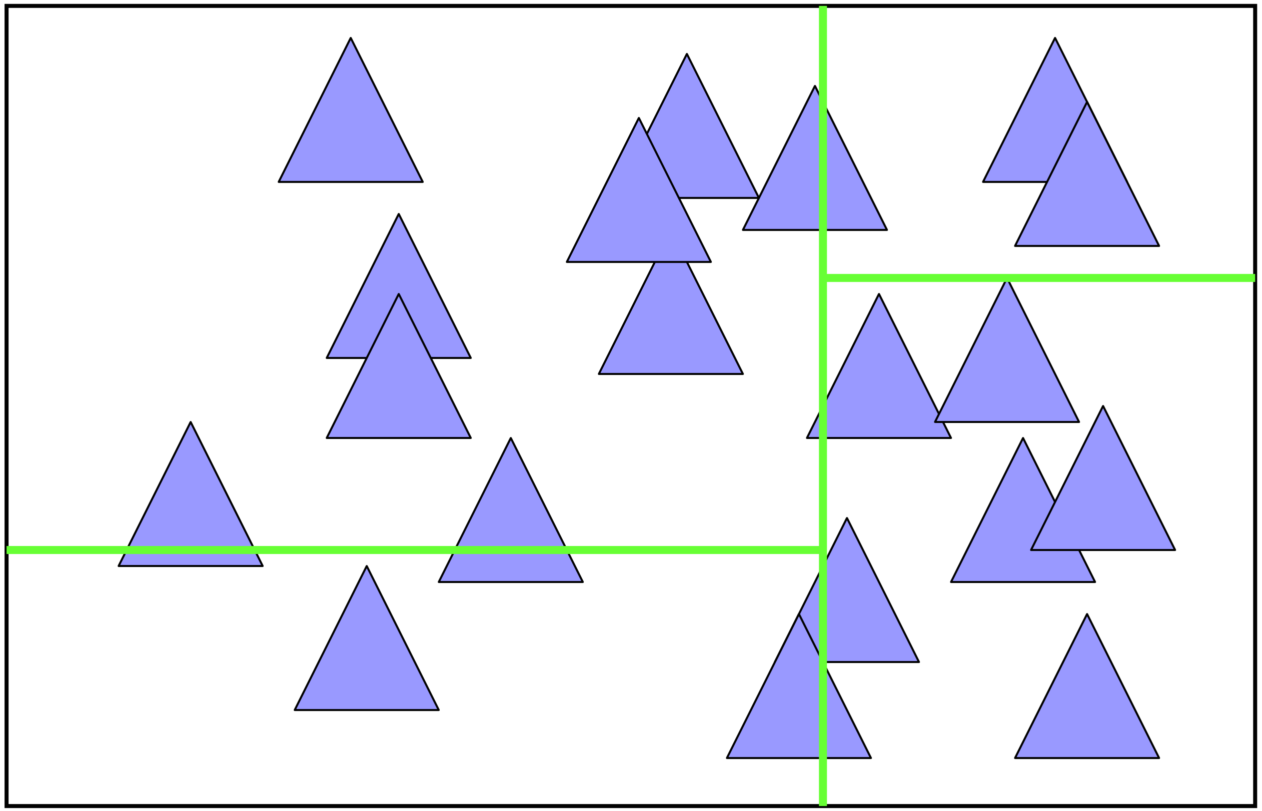
kD-Trees: esempio



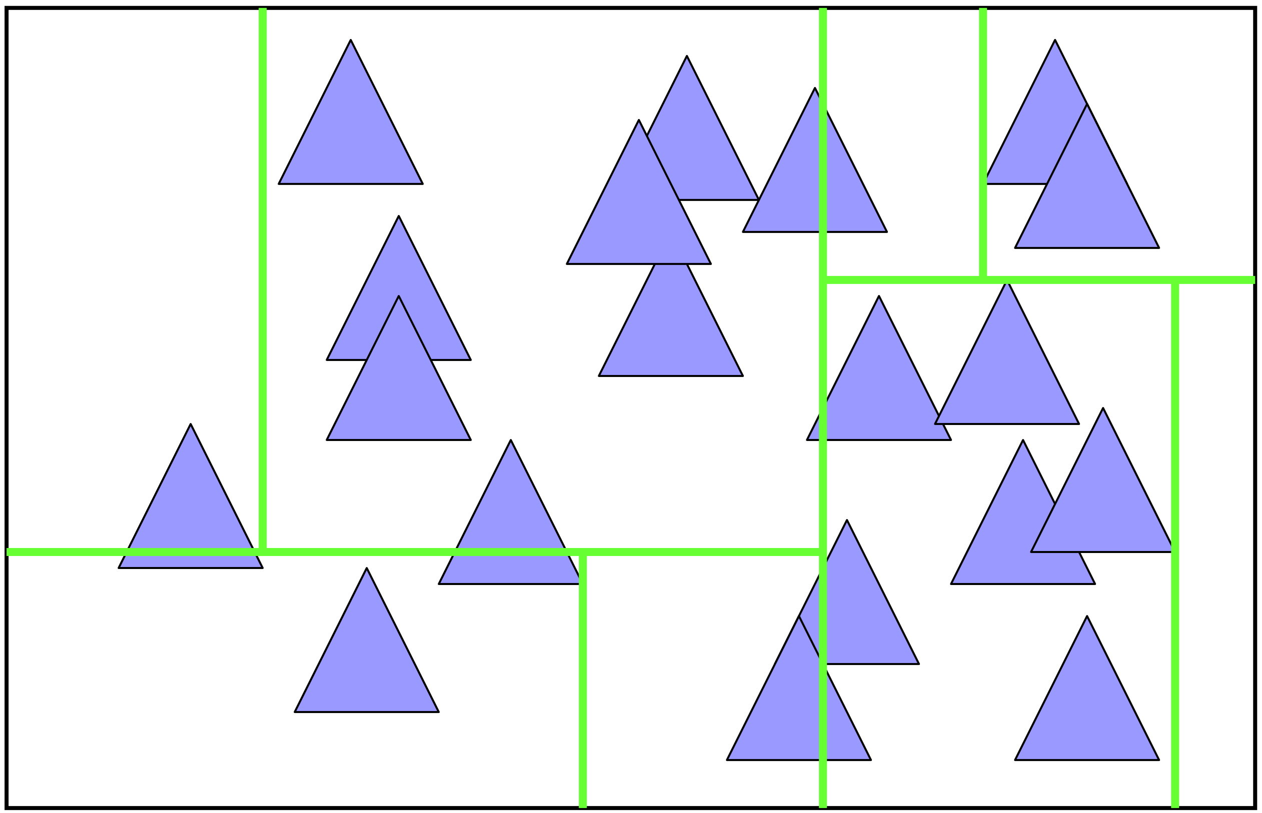
kD-Trees : esempio



kD-Trees : esempio



kD-Trees : esempio



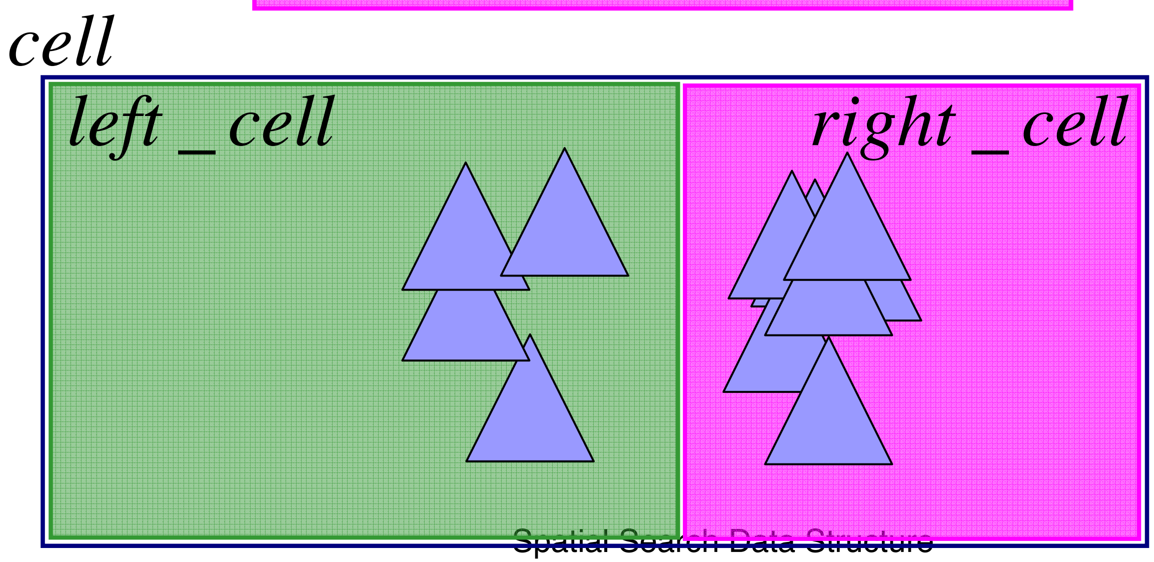
Costruire un kD-tree

- Dati:
  - axis-aligned bounding box (“cell”)
  - lista di primitive geometriche (triangoli)
- Operazioni base
  - Prendi un piano ortogonale a un asse e dividi la cella in due parti **(in che punto?)**
  - Distribuire le primitive nei due insiemi risultanti
  - Ricorsione
  - Criterio di terminazione **(che criterio?)**
- Esempio: se viene usato per il ray-tracing, si vuole ottimizzare per il costo dell’intersezione raggio primitiva

Costruire un kD-tree efficiente per RayCast

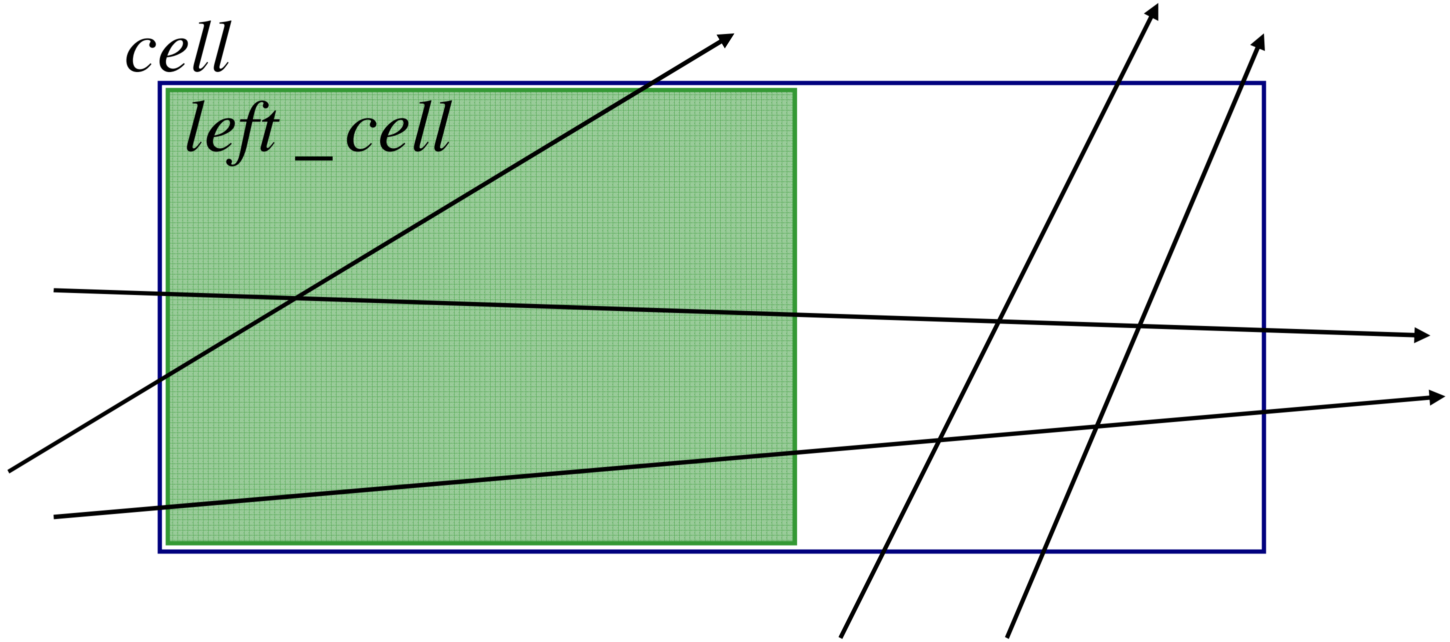
- In che punto dividere la cella?
  - Nel punto che minimizza il **costo**
- Quanto è il costo? Riprendiamo la formula per I BSP

$$Cost(cell) = 1 + \mathrm{Prob}(left\_cell \mid cell)\, Cost(Left) + \mathrm{Prob}(right\_cell \mid cell)\, Cost(Right)$$



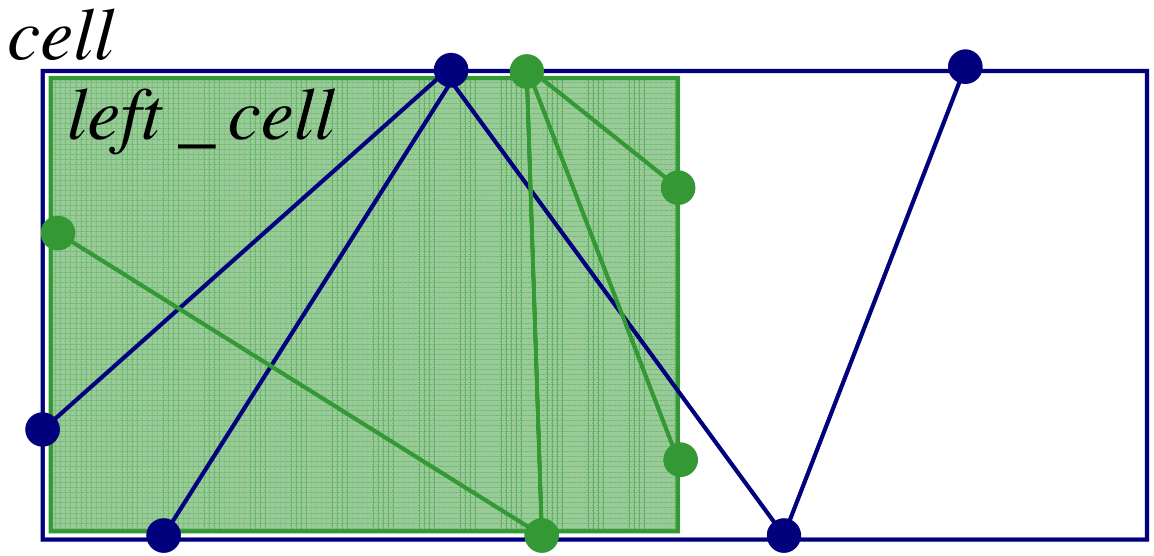
$$\mathrm{Prob}(left\_cell \mid cell)\,Cost(Left)$$

- Sapendo che il raggio interseca la cella *cell*, qual'è la probabilità che intersechi la cella *left_cell* ??



$\text{Prob}(left\_cell \mid cell)$

$$\text{Prob}[cell \mid left\_cell] = \frac{\#\, raggi\ che\ intersecano\ left\_cell}{\#\, raggi\ che\ intersecano\ \ cell}$$

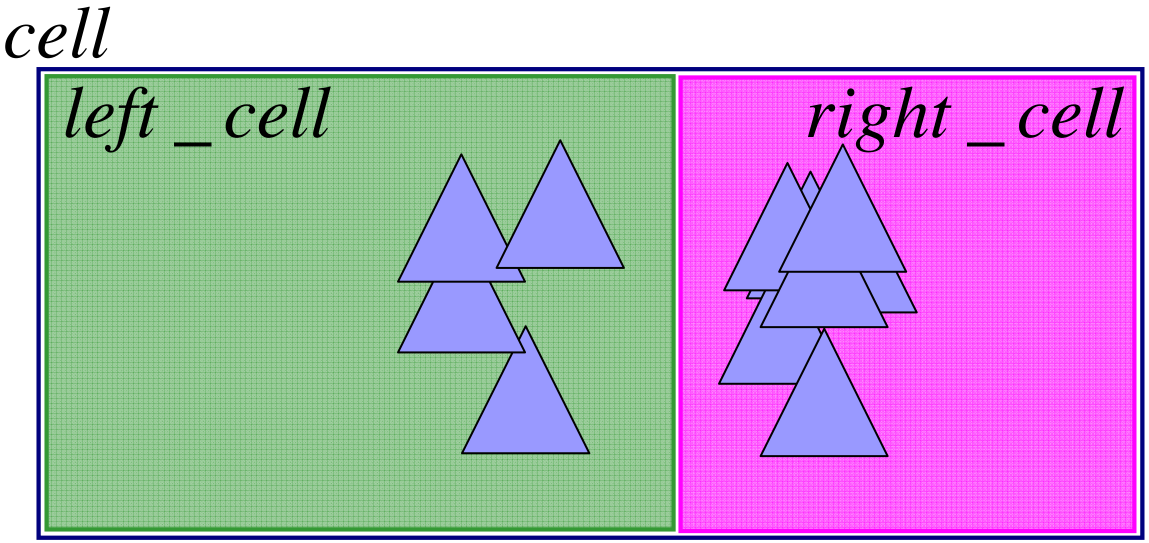


Ogni raggio che interseca una cella corrisponde a una coppia di punti sulla sua superficie. Contiamo le coppie di punti sulla superficie delle celle

$$\text{Prob}[cell \mid left\_cell] = \frac{\displaystyle\int_{\sigma(left\_cell)} \left(\int_{\sigma(left\_cell)} da \right) da}{\displaystyle\int_{\sigma(cell)} \left(\int_{\sigma(cell)} da \right) da} = \frac{Area(left\_cell)^2}{Area(cell)^2} = \frac{Area(left\_cell)}{Area(cell)}$$

$$cost(left\_cell)$$

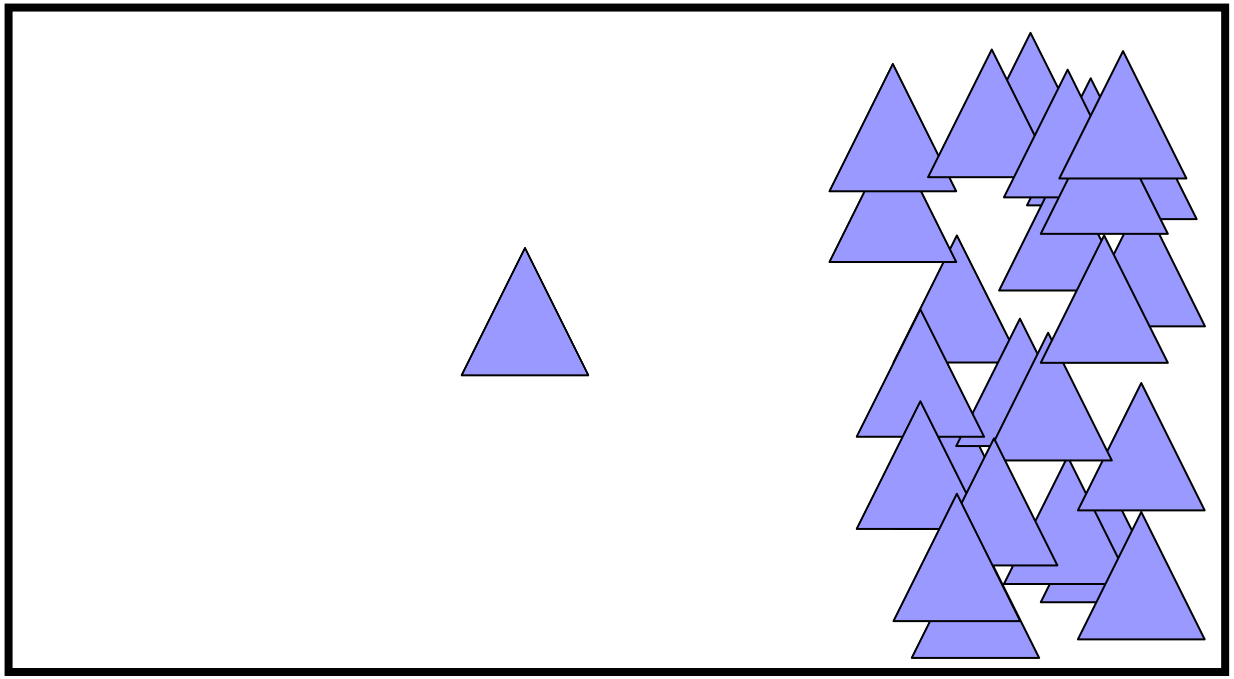
- Sapendo che il raggio interseca la cella *left_cell*, qual'è il costo di testare l'intersezione con i triangoli?
- Si approssima con il numero di triangoli che toccano la cella



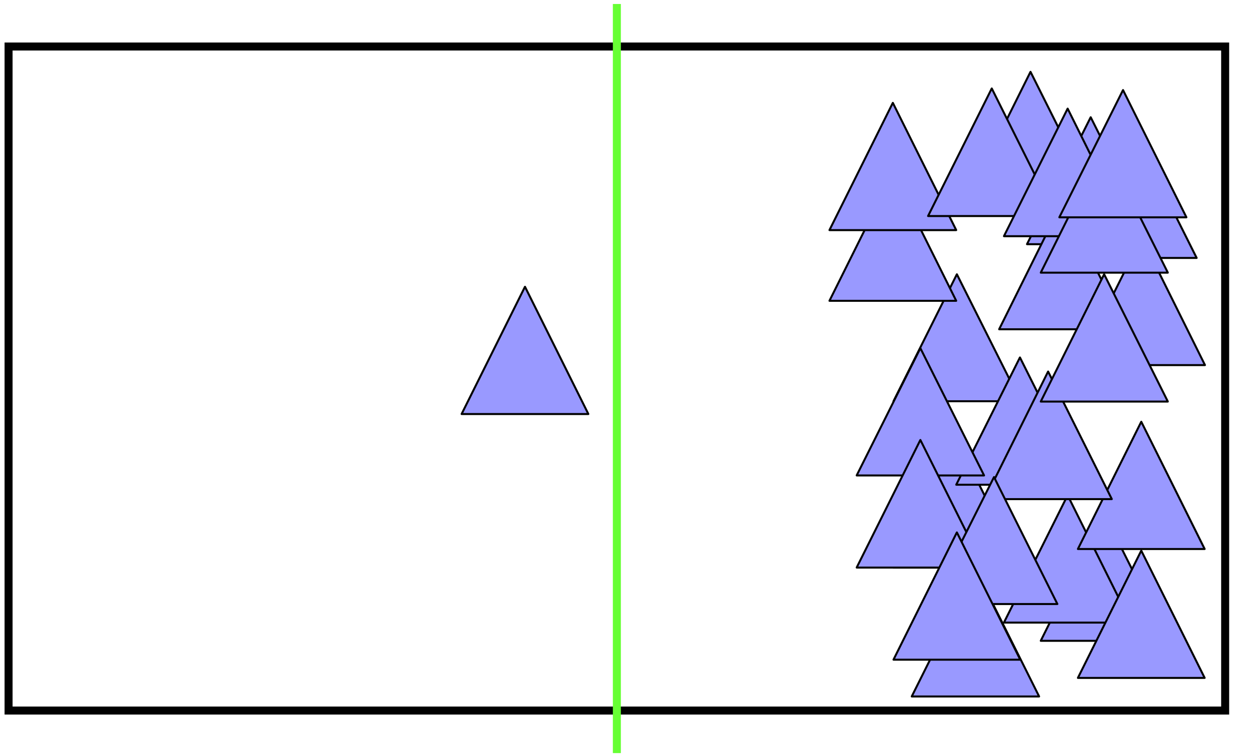
$$Cost(left\_cell) = 4$$

Esempio

- Come si suddivide la cella qui sotto?

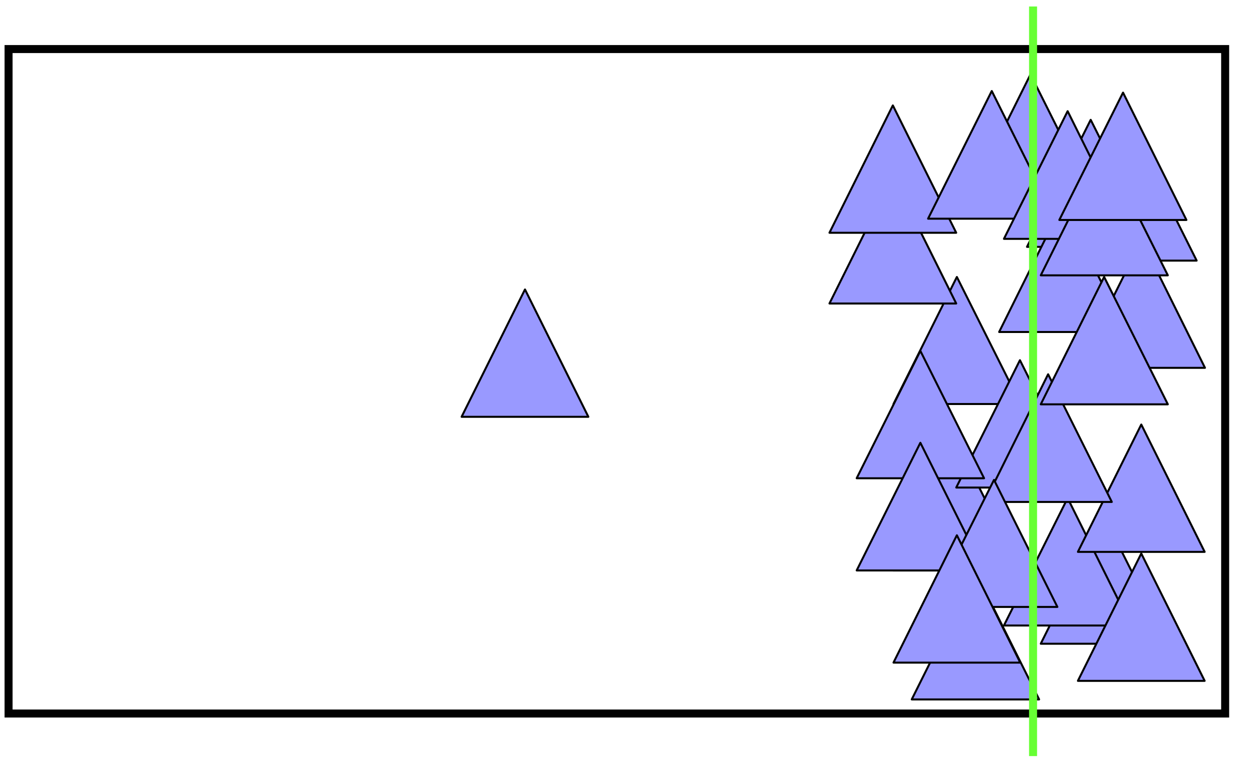


A metà



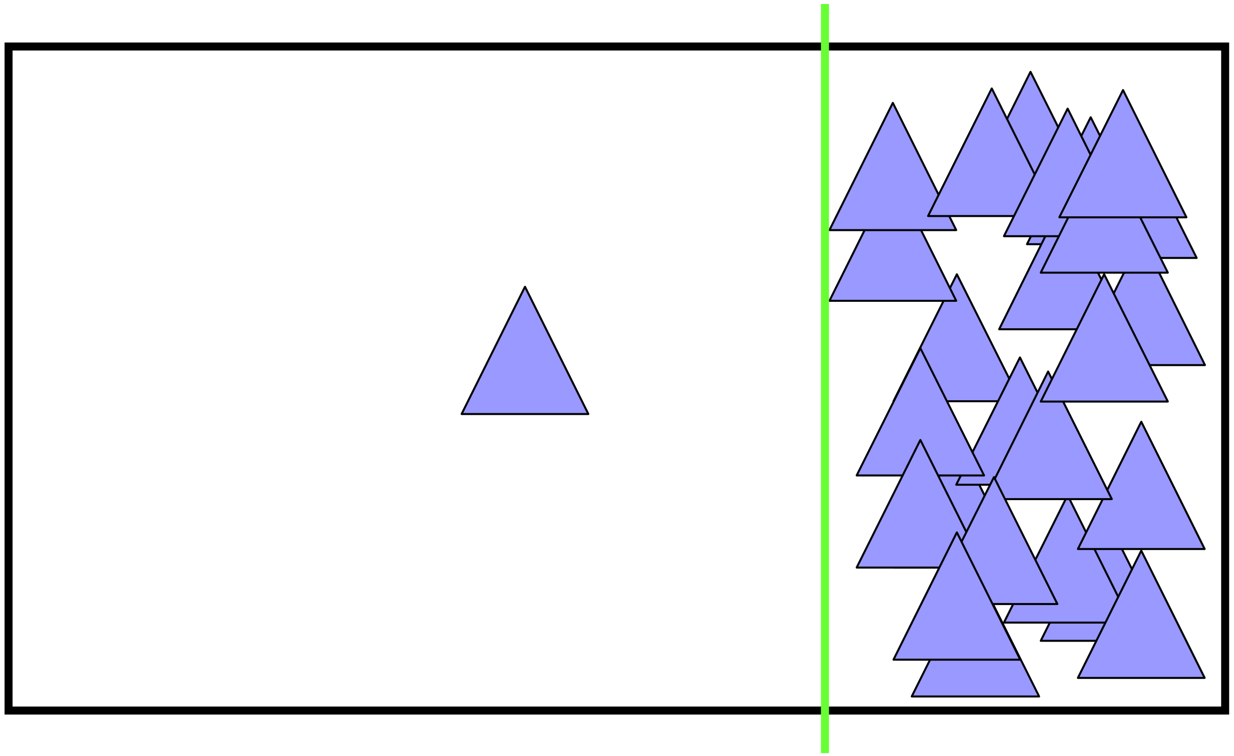
- Non tiene conto delle probabilità
- Non tiene conto dei costi

Nel punto mediano



- Rende uguali i costi di *left_cell* e *right_cell*
- Non tiene conto delle probabilità

Ottimizzando il costo



- Separa bene spazio vuoto
- Distribuisce bene la complessità

Range Query with kd-tree

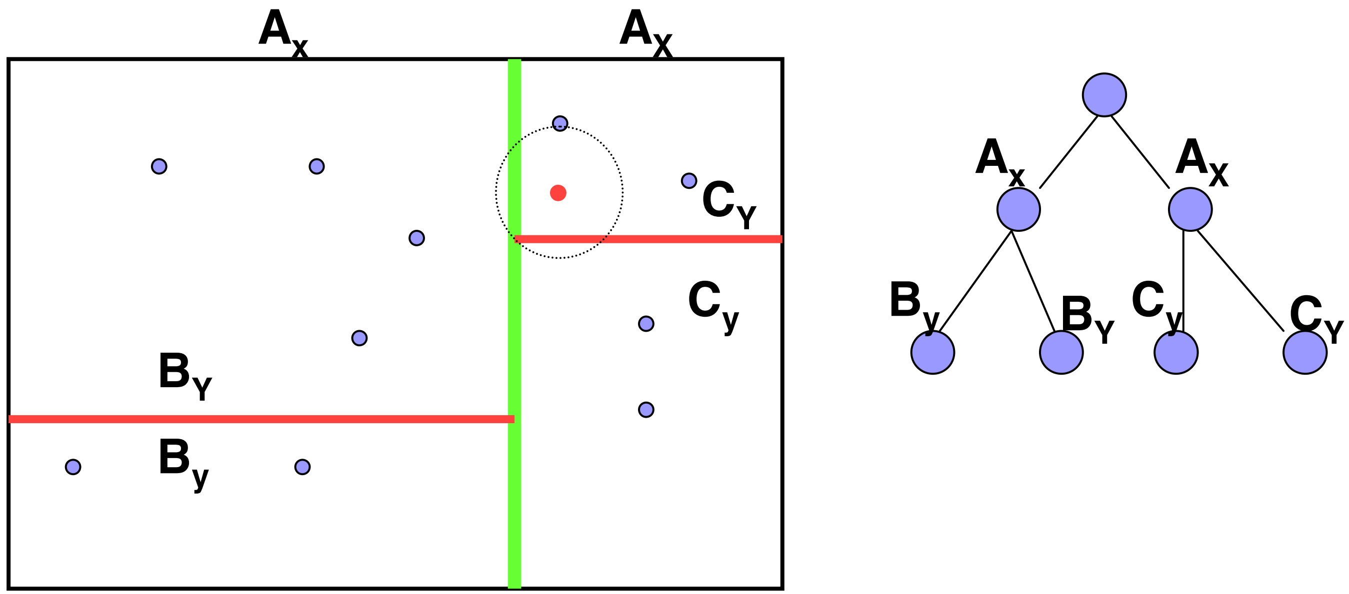
- **Query**: return the primitives inside a given box
- Algorithm:
  - Compute intersection between the node and the box
  - If the node is entirely inside the box add all the primitives contained in the node to the result
  - If the node is entirely outside the box return
  - If the nodes is **partially** inside the box recur to the children
- **Cost:** if the leaf nodes contain one primitive and the tree is balanced:

$$O(n^{1-\frac{1}{d}} + k)$$

  n number of primitives, *d* dimension
- $O(n^{2d})$ possible results

Nearest Neighbor with kd-tree

- **Query**: return the nearest primitive to a given point *c*
- **Algorithm**:
  - Find the nearest neighbor in the leaf containing c
  - If the sphere intersect the region boundary, check the primitives contained in intersected cells

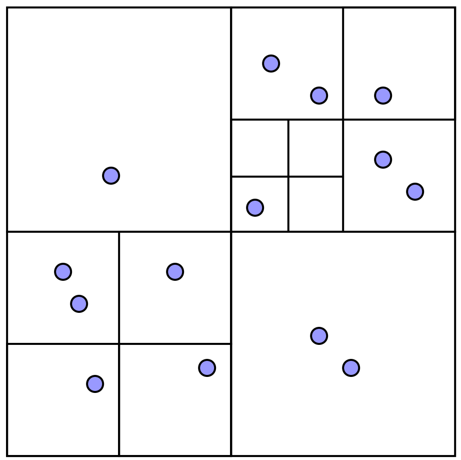


Spatial Search Data Structure

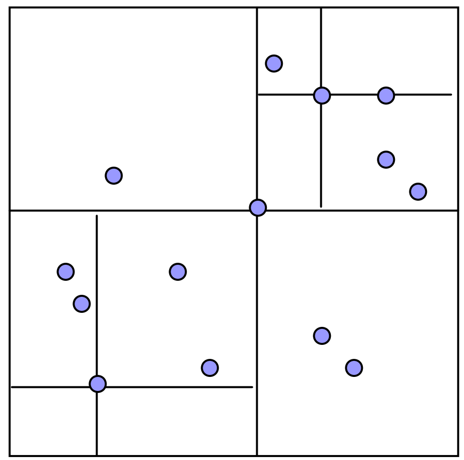
Quad-Tree (2d)

- The plane is recursively subdivided in 4 subregions by couple of orthogonal planes

Region Quad-tree

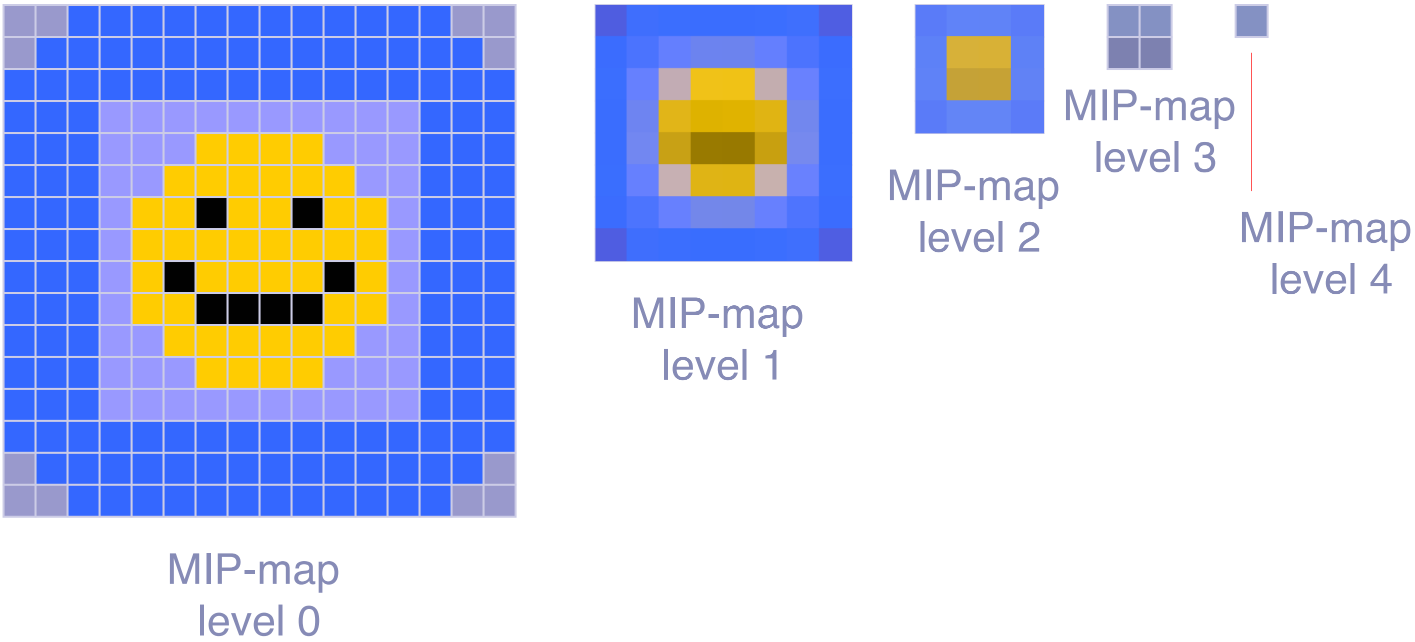


Point Quad-tree



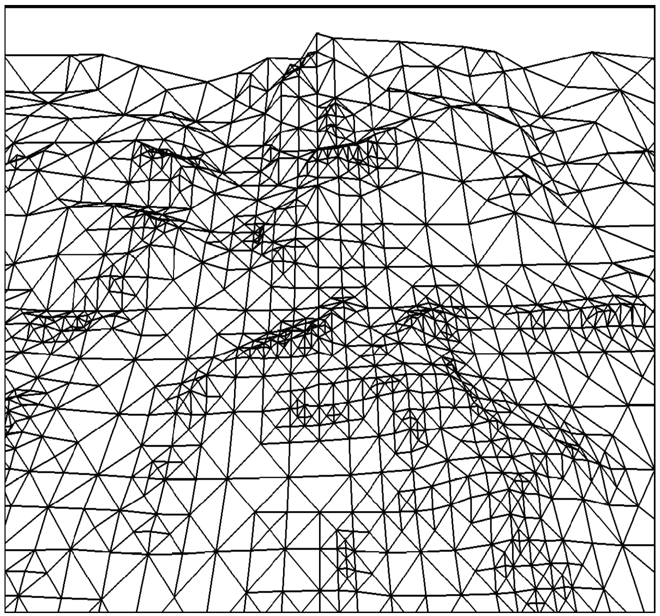
Quad-Tree (2d): examples

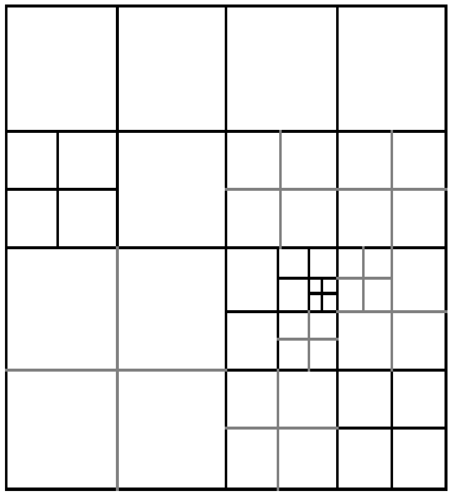
- Widely used:
  - Keeping level of detail of images



Quad-Tree (2d): examples

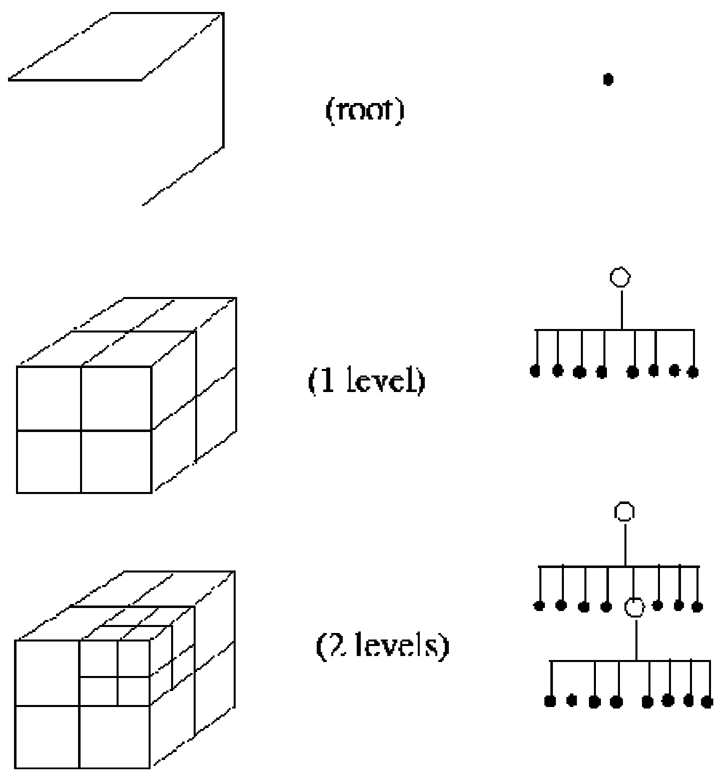
- Widely used:
  - Terrain rendering: each cross in the quatree is associated with a height value





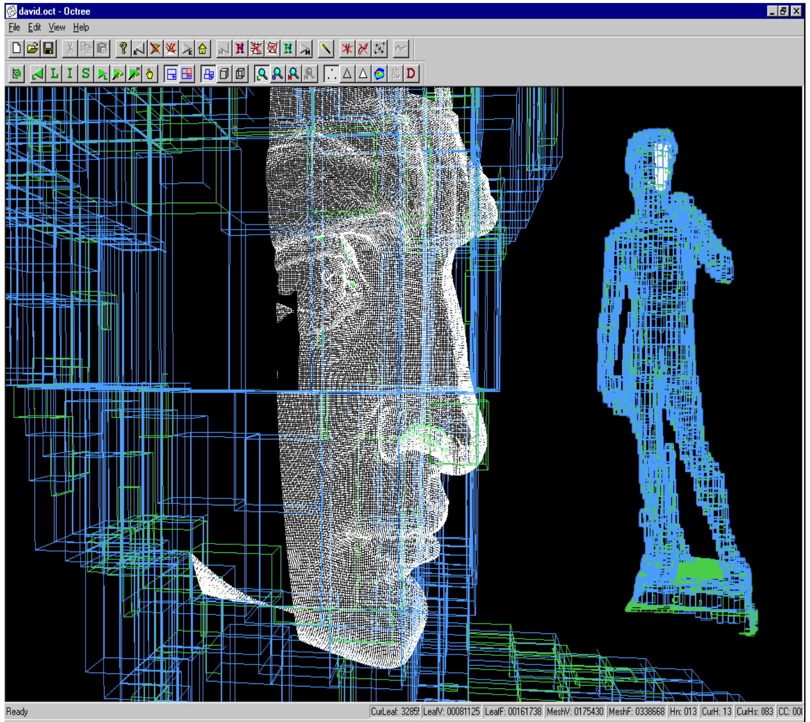
Oct-Tree (3d)

- The same as quad-tree but in 3 dimensions



Oct-Tree (3d) : Examples

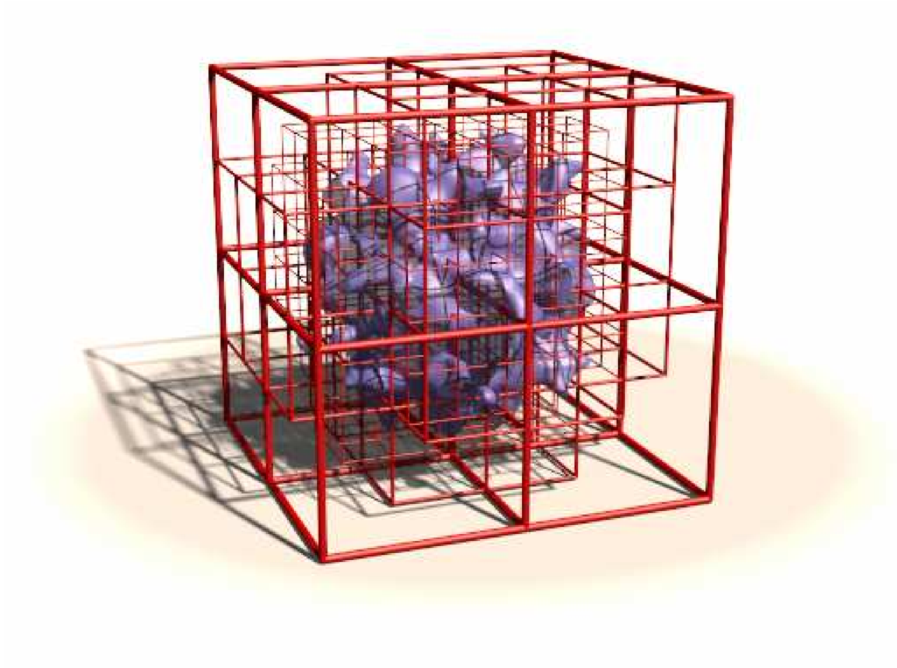
- Processing of Huge Meshes (ex: simplification)
- Problem: mesh do not fit in main memory
- Arrange the triangles in a oct-tree



Spatial Search Data Structure

Oct-Tree (3d) : Examples

- Extraction of isosurfaces on large dataset
  - Build an octree on the 3D dataset
  - Each node store min and max value of the scalar field
  - When computing the isosurface for alpha, nodes whose interval doesn't contain alpha are discarded



Advantages of quad/oct tree

- Position and size of the cells are implicit
  - They can be explored without pointers (convenient if the hierarchies are complete) by using a linear array where:

quadtree

$$Children(i) = 4i+1,\ldots,4*(i+1)$$
$$Parent(i) = \lfloor i/4 \rfloor$$

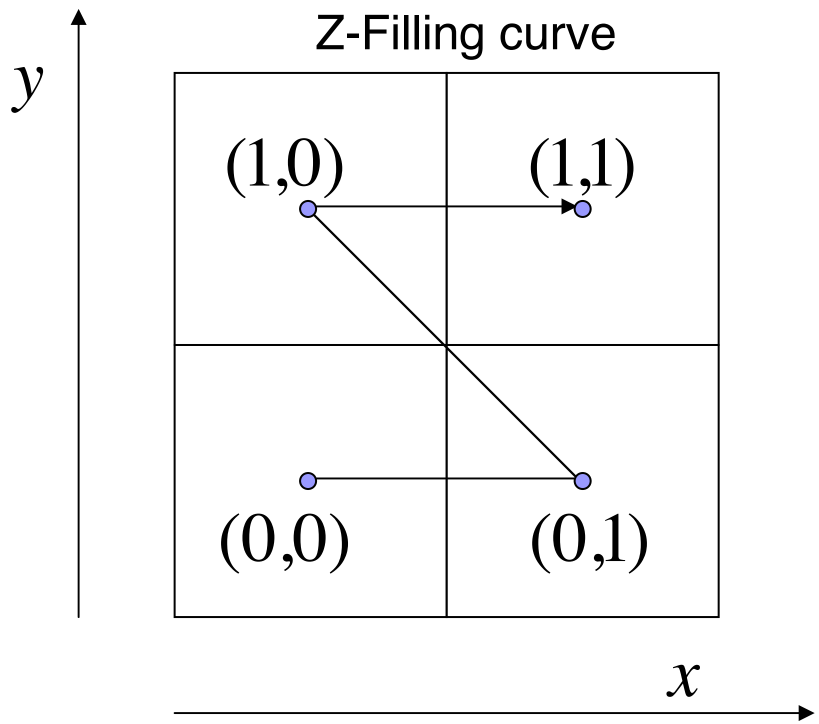
octree

$$Children(i) = 8i+1,\ldots,8*(i+1)$$
$$Parent(i) = \lfloor i/8 \rfloor$$

Z-Filling Curves

- Position and size of the cells are implicit
  - They can be indexed to preserve locality, i.e.

$$\textit{Spatially close} \rightarrow \textit{close in memory}$$



Easy conversion between position in space and order in the curve

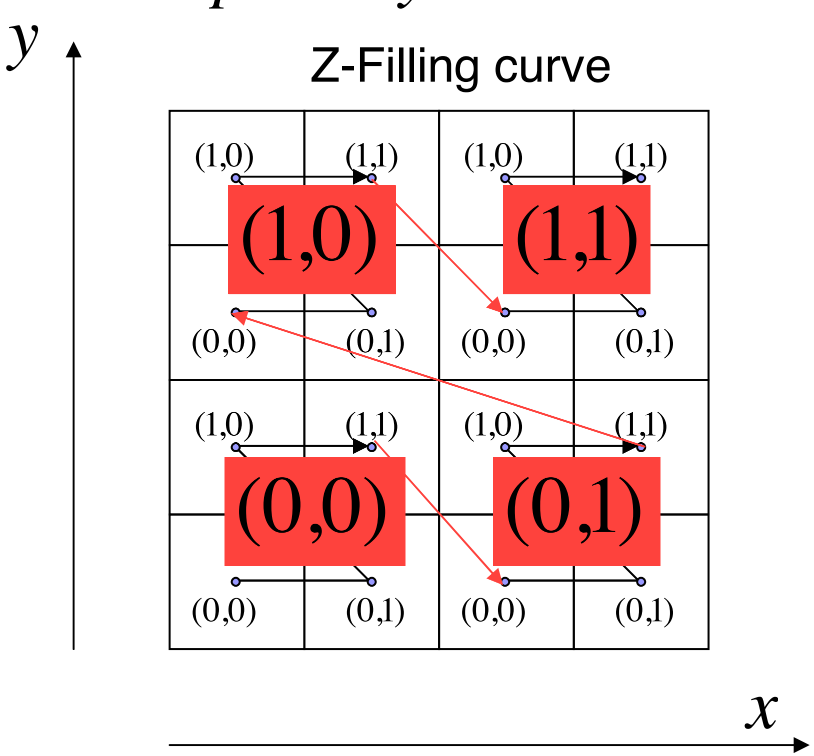
Just use the 0..1 coordinates as bits

00 01 10 11

Z-Filling Curves

- Position and size of the cells are implicit
  - They can be indexed to preserve locality, i.e.

$$Spatially\ close \rightarrow close\ in\ memory$$



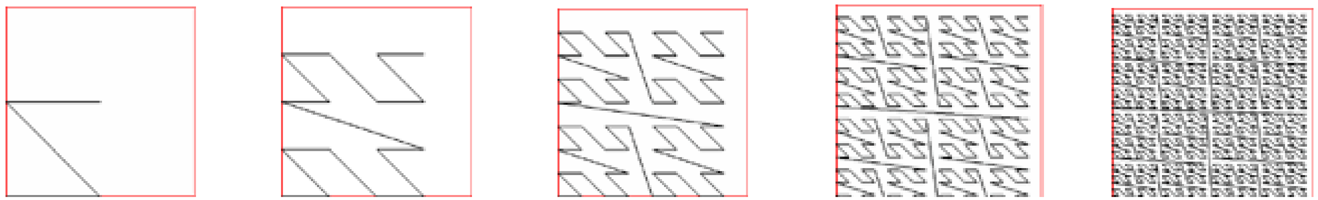
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

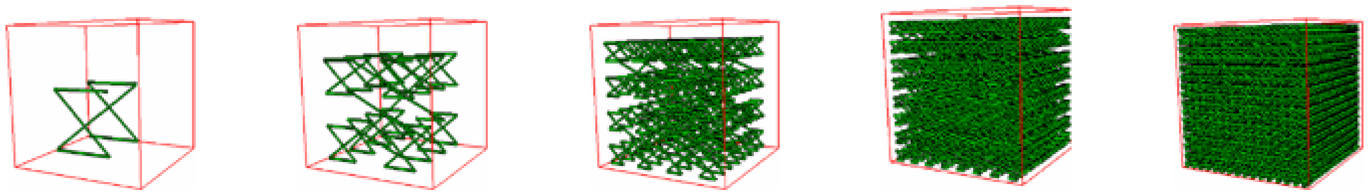
Z-Filling Curves

- Conversion from spatial coordinates to index.
  - Write the coord values in binary
  - Interleave the bits

$$
\begin{array}{ccccccccccc}
x & = & & b_0^x & & b_1^x & & b_2^x & \ldots & & b_n^x \\
y & = & b_0^y & & b_1^y & & b_2^y & & \ldots & b_n^y & \\
id & = & b_0^y & b_0^x & b_1^y & b_1^x & b_2^y & b_2^x & \ldots & b_n^y & b_n^x
\end{array}
$$

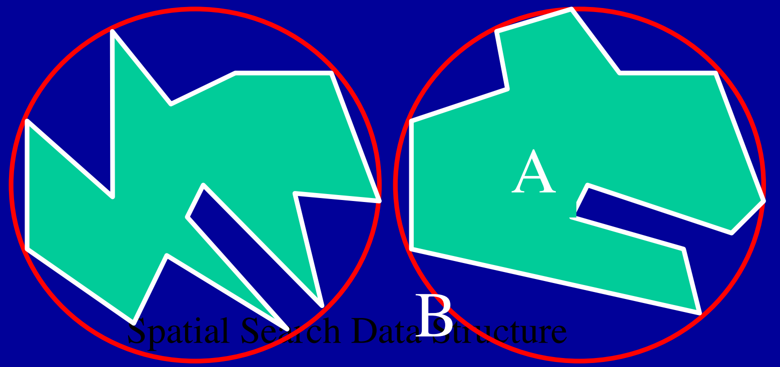
Hierarchical Z-Filling Curves





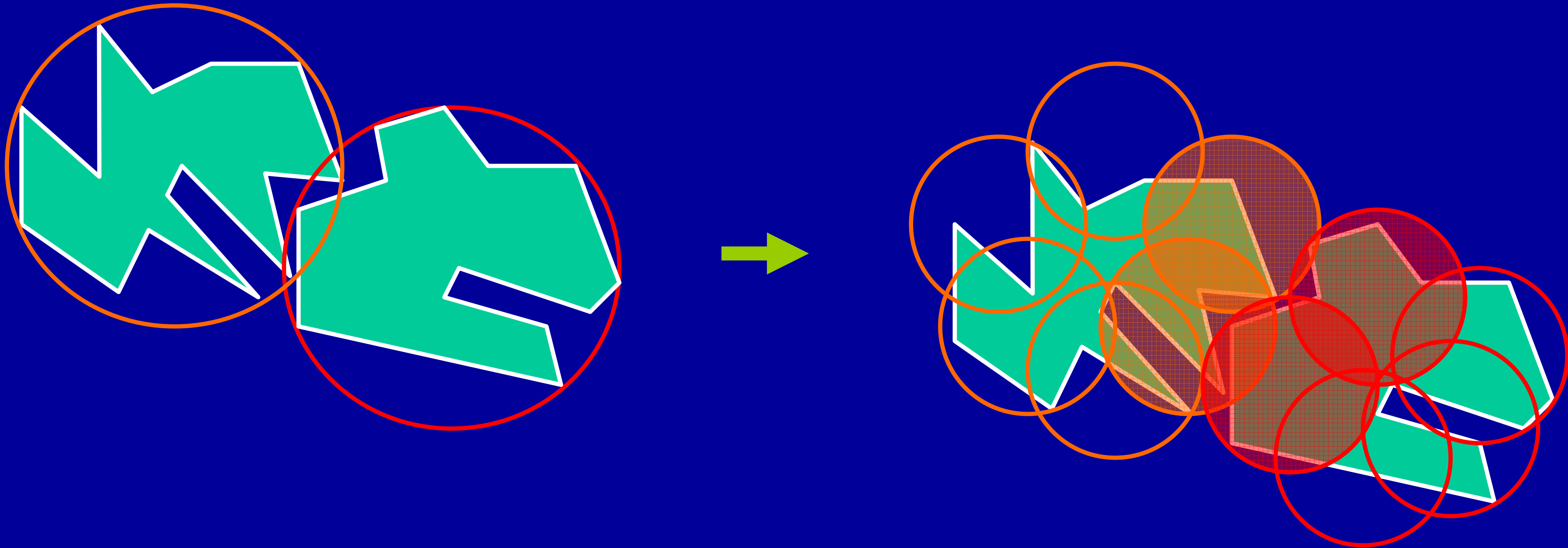
Bounding Volumes Hierarchies

- If a volume B includes a volume A, it is called *bounding volume* for A
- No object can intersect A without intersecting B
- If two bounding volumes do not overlap, the same hold for the volumes included



The Principle

- What if they do overlap?
- Refine.



Questions!

- What kind of Bounding Volumes?
- What kind of hierarchy?
- How to build the hierarchy?
- How to update (if needed) the hierarchy?
- How to transverse the hierarchy?

All the literature on CD for non-convex objects is about answering these questions.

Cost

$$T_c = N_v*C_v + N_n*C_n + N_s*C_s$$

v : visited nodes

n : couple of bounding volumes tested for overlap

s : couple of polygons tested for overlap

N: number of

C: Cost

BHV - Desirable Properties (2)

- The hierarchy should be able to be constructed in an automatic predictable manner
- The hierarchical representation should be able to approximate the original model to a high degree or accuracy
  - allow quick localisation of areas of contact
  - reduce the appearance of object repulsion

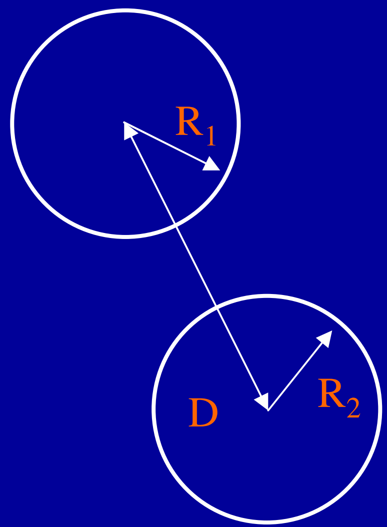
BHV - Desirable Properties

- The hierarchy approximates the bounding volume of the object, each level representing a tighter fit than its parent
- For any node in the hierarchy, its children should collectively cover the area of the object contained within the parent node
- The nodes of the hierarchy should fit the original model as tightly as possible

Sphere-Tree

[O'Rourke and Badler 1979 , Hubbard 1995a & 1996, Palmer and Grimsdale 1995, Dingliana and O'Sullivan 2000]

- Nodes of BVH are spheres.
- Low update cost C_u
  - translate sphere center
- Cheap overlap test C_v

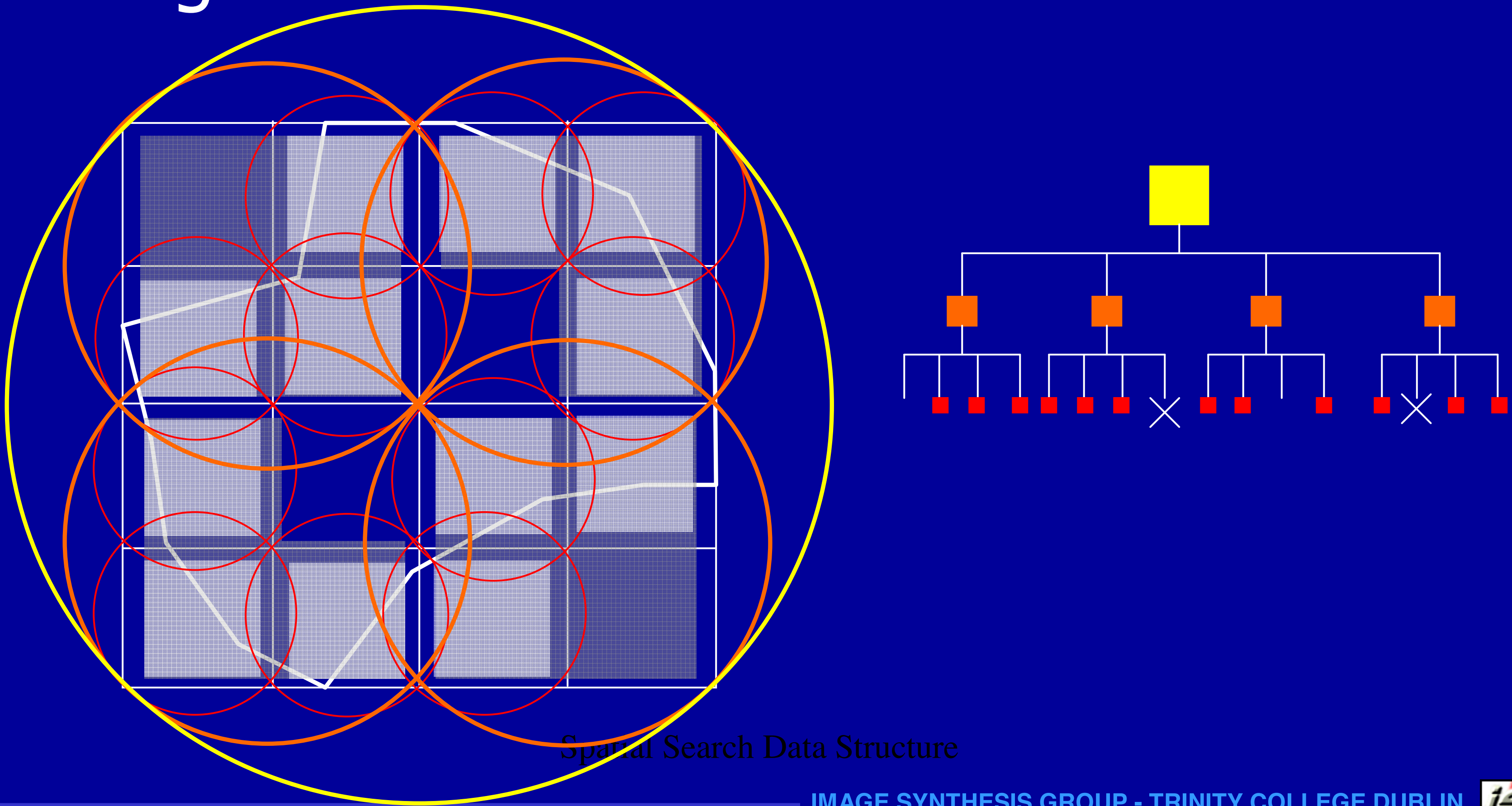


$$D^2 < (R_1 + R_2)^2$$

- Slow convergence to object geometry
  - Relatively high N_u & N_v

Sphere-Tree Construction Dingliana and O'Sullivan 2000

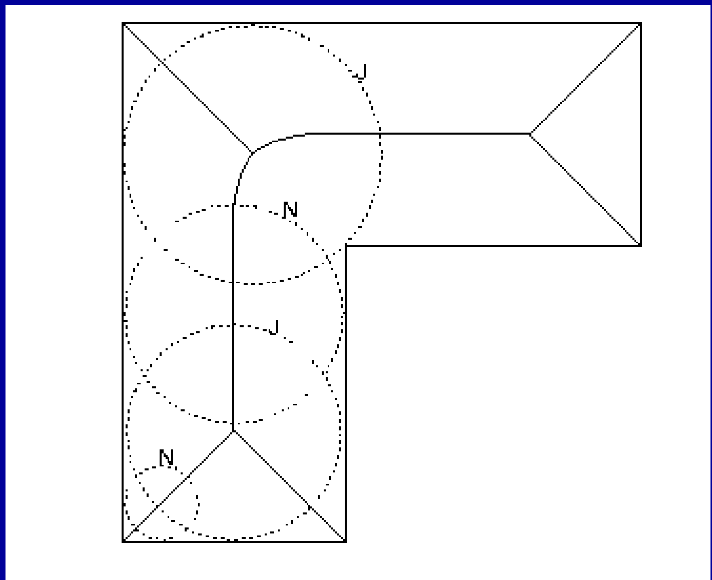
- Spheres placed around the boxes of a regular oct-tree

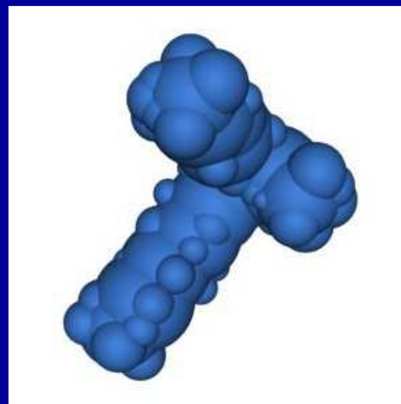


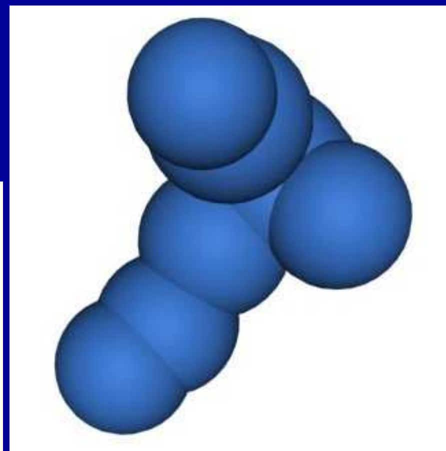
Spatial Search Data Structure

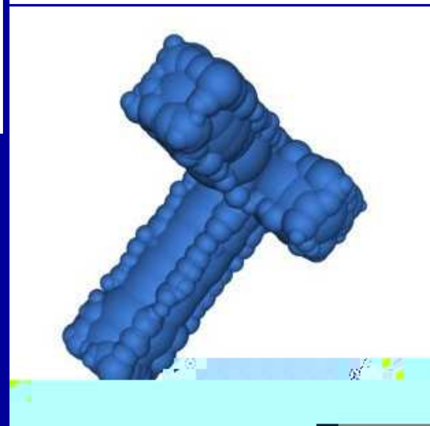
Sphere-Tree Construction Hubbard 1995a & 1996,

- Spheres placed along the Medial-Axis (transform)









Axis-Aligned Bounding Box

[van den Bergen 1997]

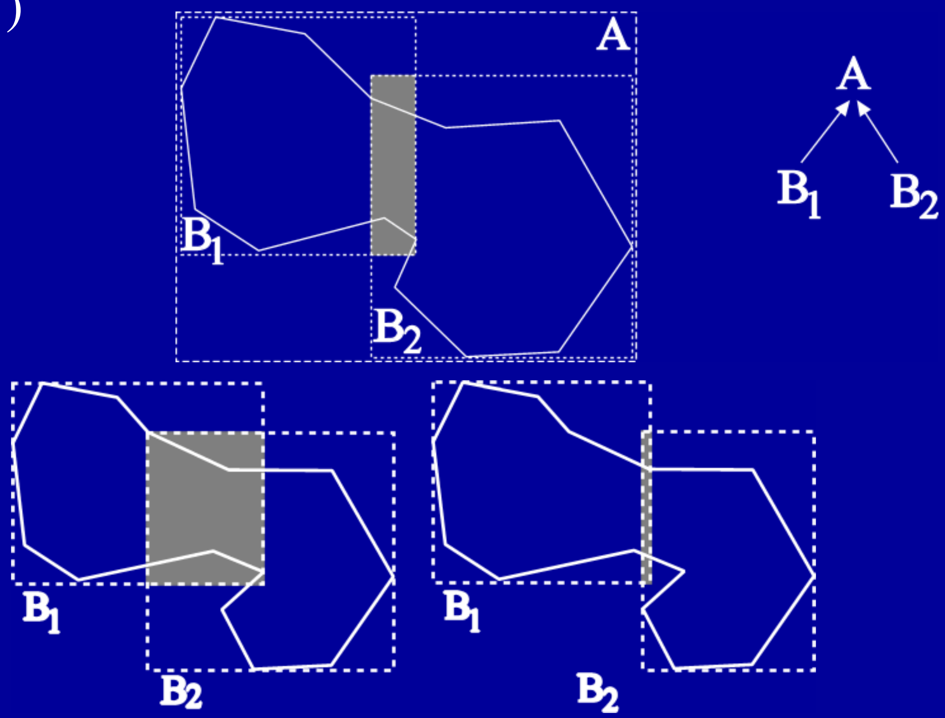
- The bounding volumes are axis aligned boxes (in the *object* coordinate system)
- The hierarchy is a binary tree (built top down)
- Split of the boxes along the longest edge at the median (equal number of polygons in both children)

Axis-Aligned Bounding Box

- The hierarchy of boxes can be quickly updated :
- let $Sm(R)$ be the smallest AABB of a region R and r_1, r_2 two regions.

$$Sm(Sm(r_1) \cup Sm(r_2)) = Sm(r_1 \cup r_2)$$

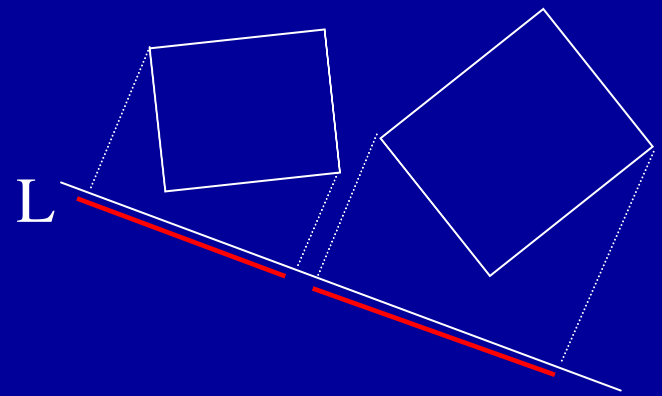
- The hierarchy is updated in *O(n)* time
- Note: this is not the same as rebuilding the hierarchy



refitting *rebuilding*

AABB - Overlap

If two **convex polyhedra** do not overlap, then there exists a direction L such that their projections on L do not overlap. L is called Separating Axis

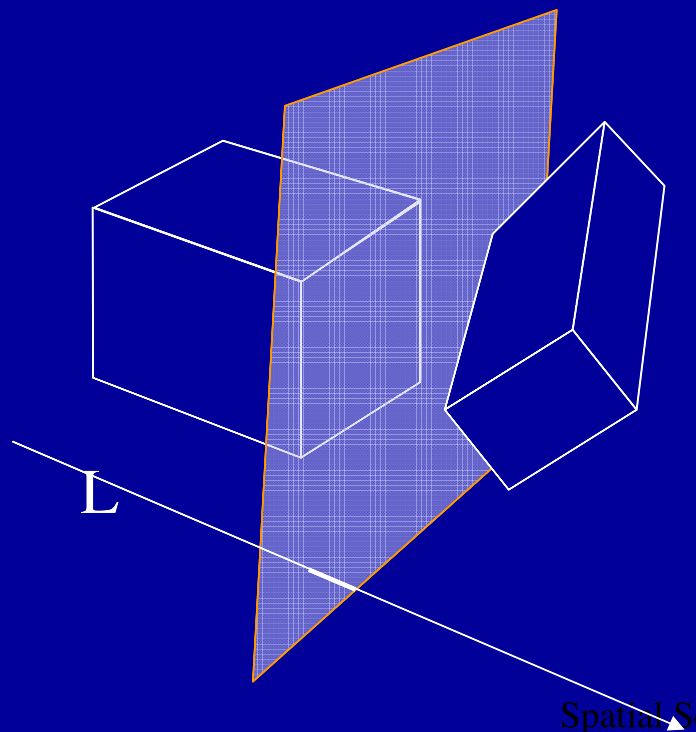


Separating Axis Theorem: L can only be one of the following:

- Normal to a face of one of the polyedra
- Normal to 2 edges, one for each polyedron

AABB - Overlap

Ex: There are 15 possible axes for two boxes: 3 faces from each box, and 3x3 edge direction combinations



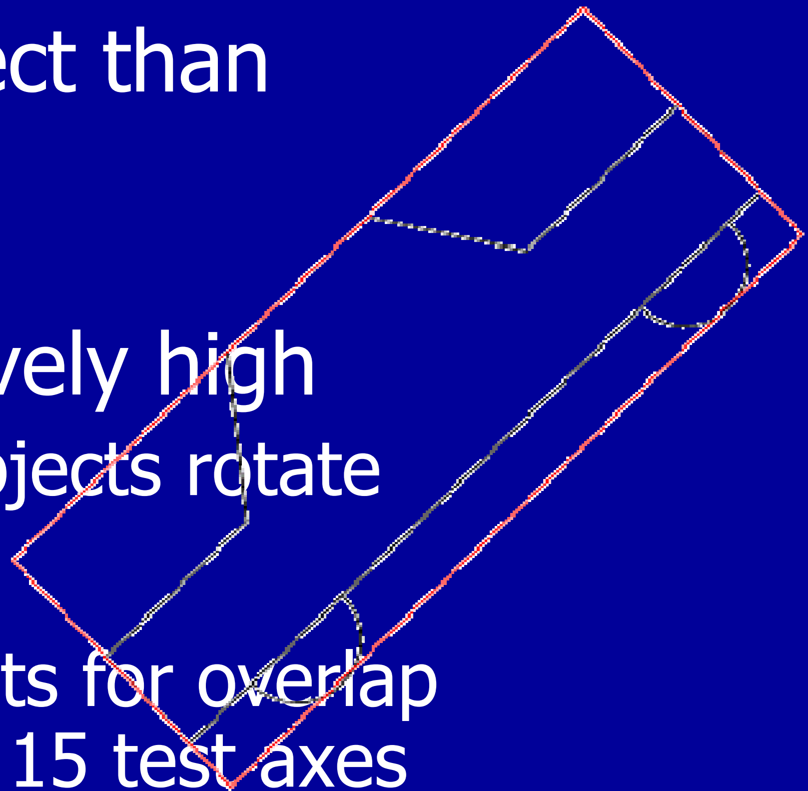
Note: SA is a normal to a face 75% of the times

Trick: Ignore the tests on the edges!

Object Oriented Bounding Box

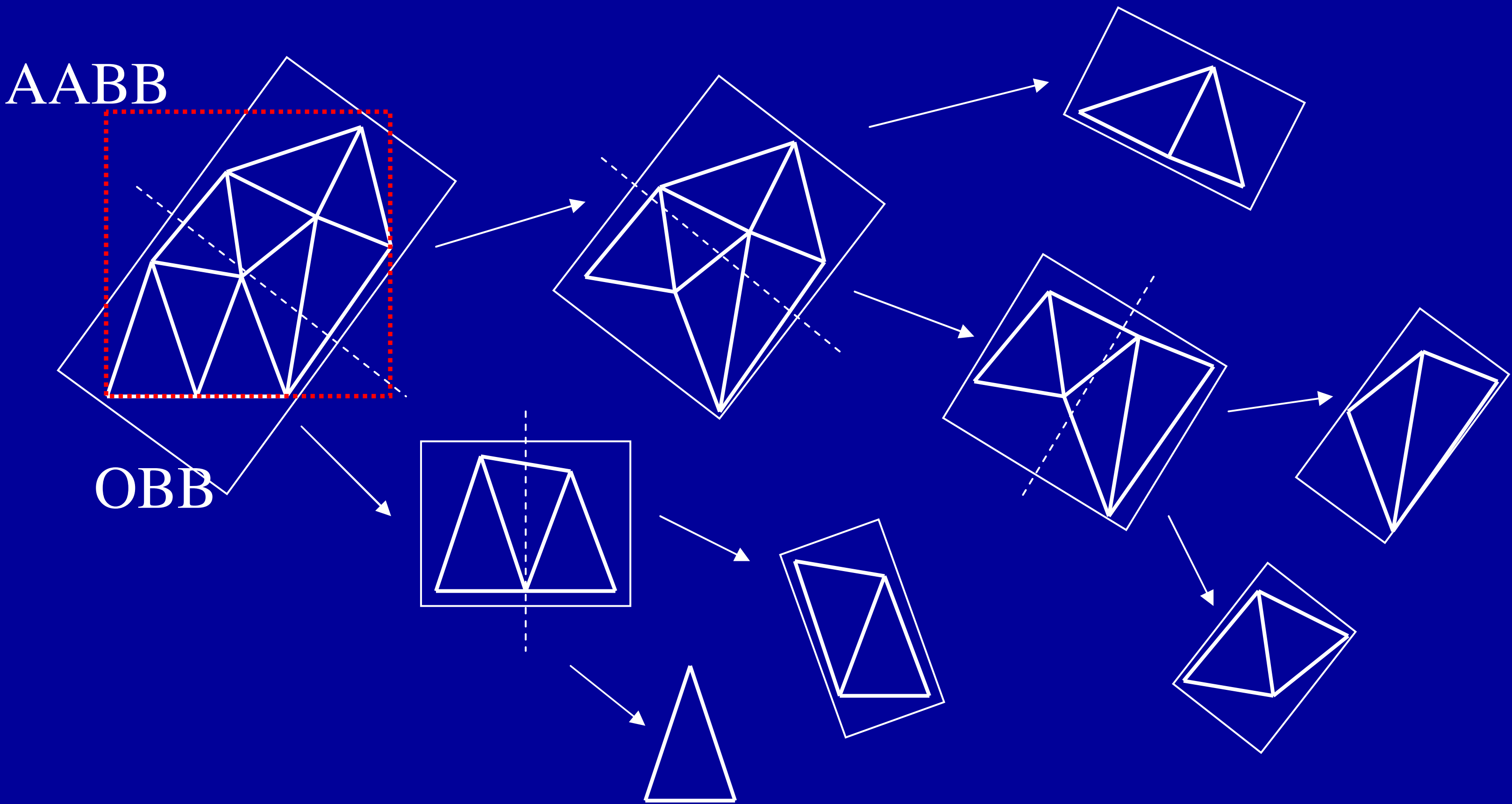
[Gottschalk et al. 1996]

- Better coverage of object than AABB
  - Quadratic convergence
- Update cost C_u is relatively high
  - reorient the boxes as objects rotate
- Overlap cost C_v is high
  - Separating Axis Test tests for overlap of box's projection onto 15 test axes



Oriented Bounding Box

[Gottschalk et al. 1996]



Spatial Search Data Structure

Building an OBB

- The OBB fitting problem requires finding the orientation of the box that best fits the data
- Principal Components Analysis:
  - Point sample the convex hull of the geometry to be bound
  - Find the mean and covariance matrix of the samples
  - The mean will be the center of the box
  - The eigenvectors of the covariance matrix are the principal directions – they are used for the axes of the box
  - The principle directions tend to align along the longest axis, then the next longest that is orthogonal, and then the other orthogonal axis

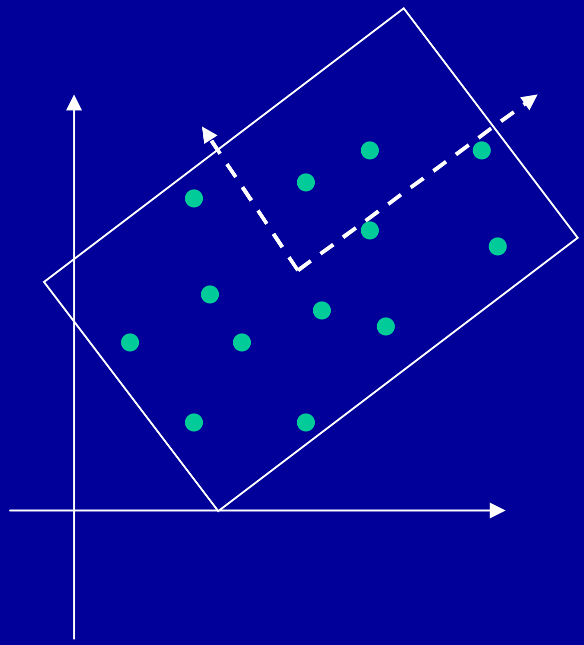
Principal Component Analysis

$$c = \frac{1}{3n}\sum_{h=1}^{n} p^h$$

$$Cov_{ij} = \frac{1}{3n}\sum_{h=1}^{n}(p_i^h - c_i)(p_j^h - c_j)$$

$$Cov \quad is \quad symmetric \Rightarrow eigen\ vectors$$

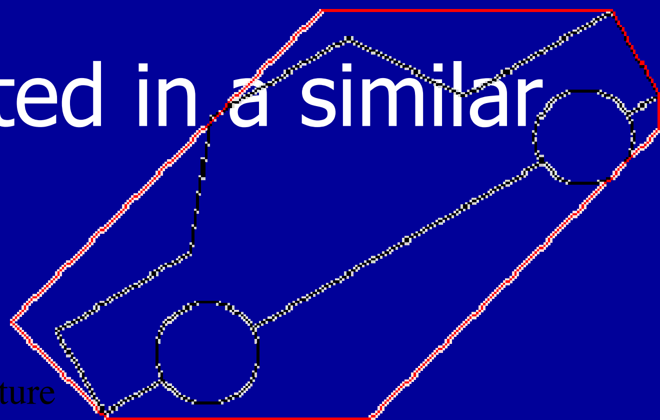
$$form\ an\ \ orthogonal\ basis$$



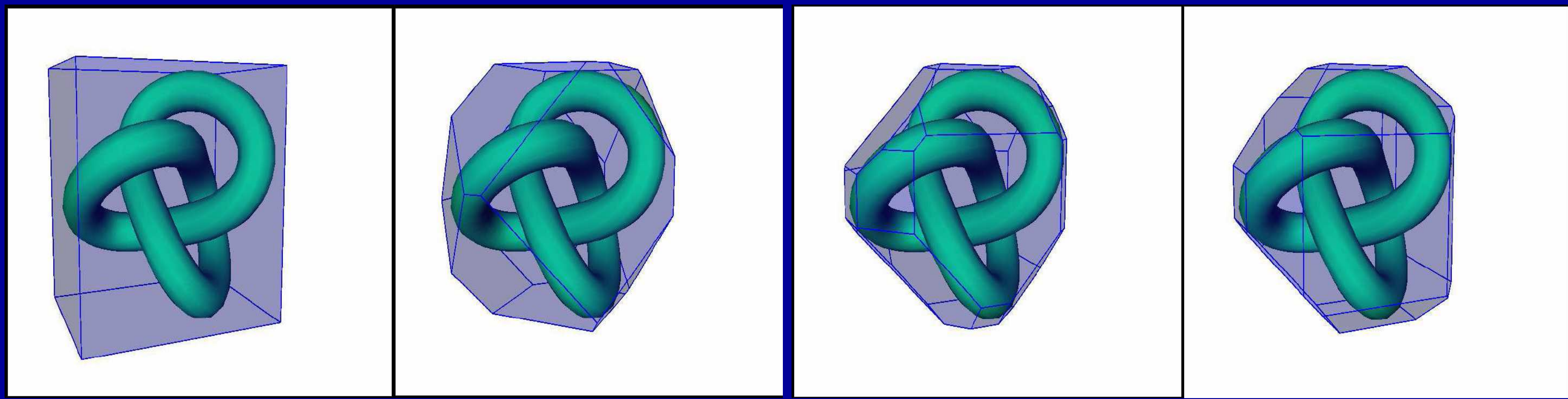
Discrete Oriented Polytope

[Klosowski et al. 1997]

- Convex polytope whose faces are oriented normal to *k* directions: $n \in \{0, \pm 1\}^d$
- Overlap test similar to OBB
  - *k/2* pairs of co-linear vectors
  - k/2 overlap tests
- *k*-DOP needs to be updated in a similar way as the AABB
- AABB is a 6-DOP



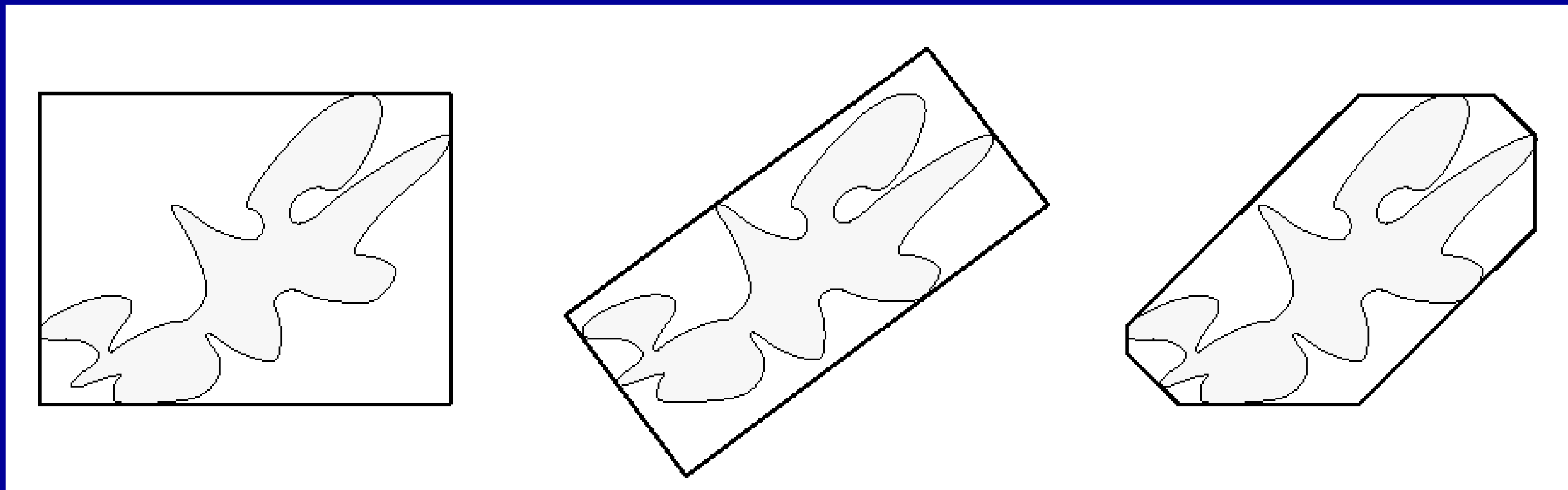
K-Dops examples



6-dop 14-dop 18-dop 26-dop

Discrete Oriented Polytope

[Klosowski et al. 1997]



AABB OBB 6-DOP