

Spatial Indexing

GMP 25/26

Paolo Cignoni

Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale
delle Ricerche



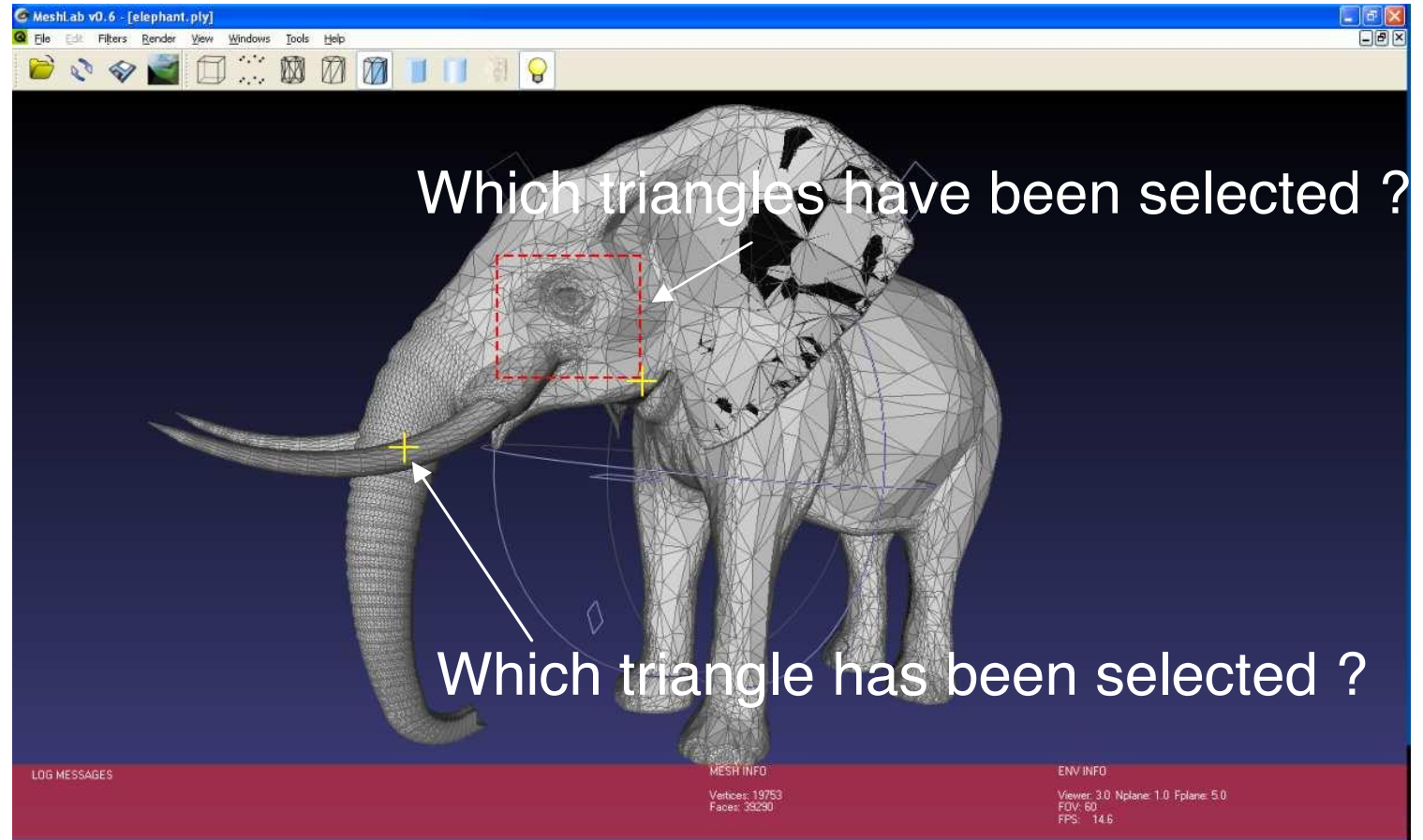
Problem statement

- Let m be a mesh:
 - Which is the mesh element closest to a given point p ?
 - Which are the elements inside a given region on the screen?
 - 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

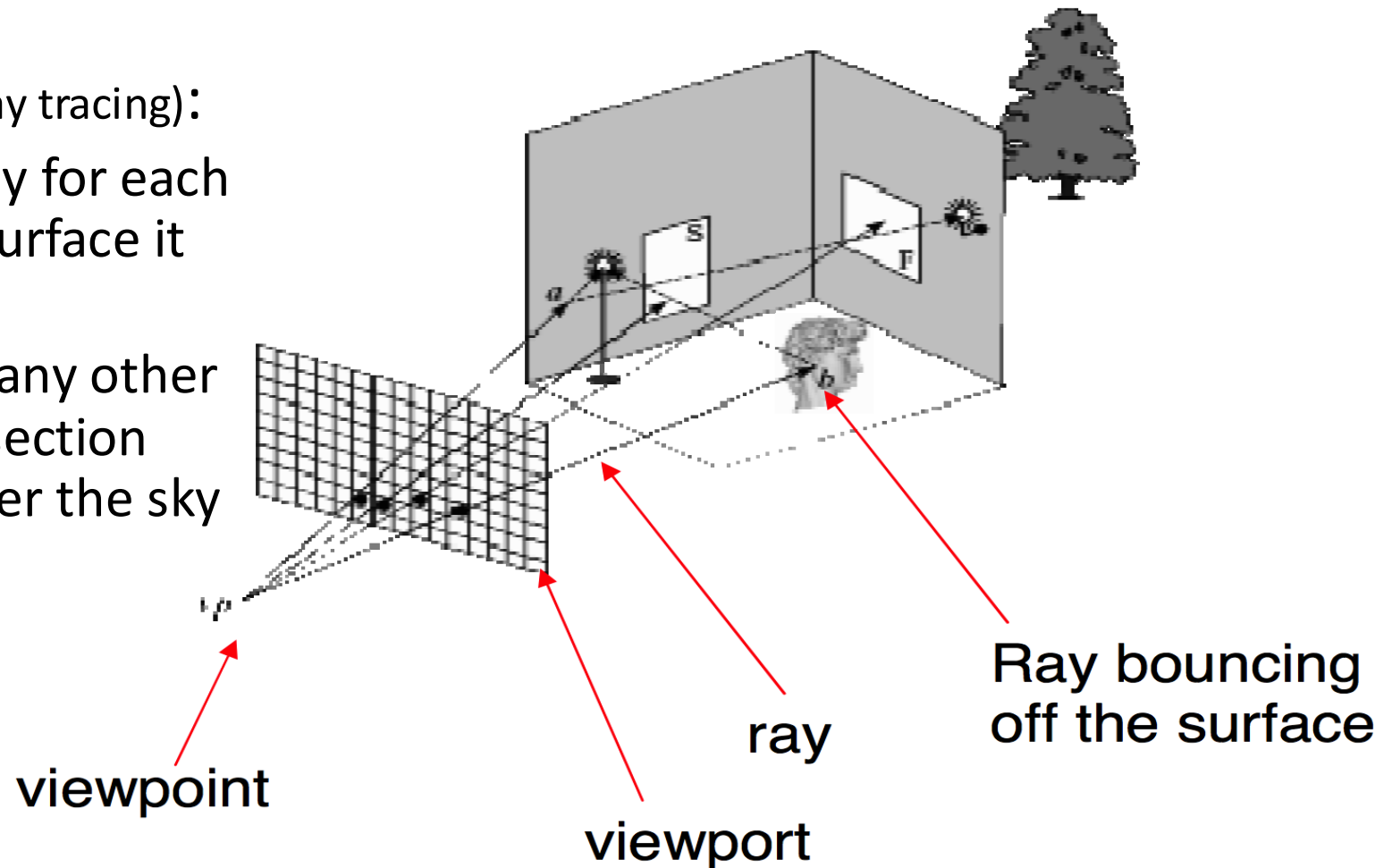
Problem statement

- Picking on a point
- Selecting a region



Problem statement: Rendering

- Path tracing (aka unbiased ray tracing):
 - From the eye, shoot a ray for each pixel, and find the first surface it encounters.
 - From this point shoot many other rays and find their intersection. Recur until you find either the sky or an emissive surface



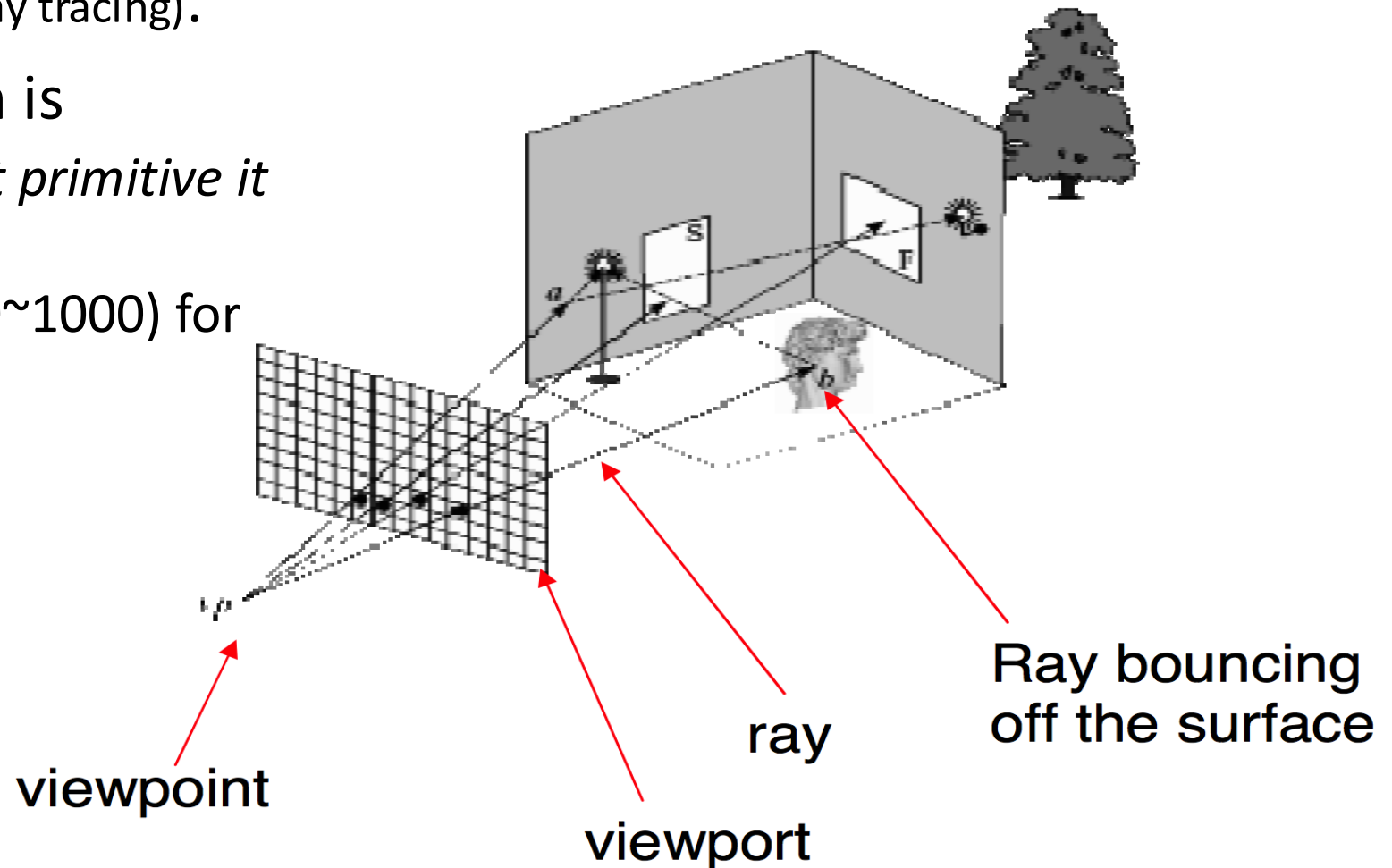
Problem statement: Rendering

- Path tracing (aka unbiased ray tracing):

- The **core** of the problem is

Given a ray find the first primitive it encounters.

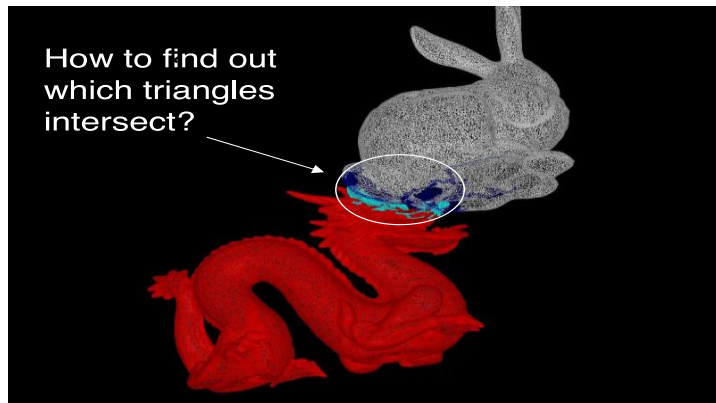
- You shoot many rays ($10 \sim 1000$) for each hit surface
- Primitives can easily be $O(10^5) \sim O(10^9)$



Problem statement: Dynamics/Simulation

- Simulating rigid body dynamics requires mainly two tasks:

- Computing the position according to current forces
- Computing what are the new forces according the current positions
 - Reaction forces after collision



Problem statement

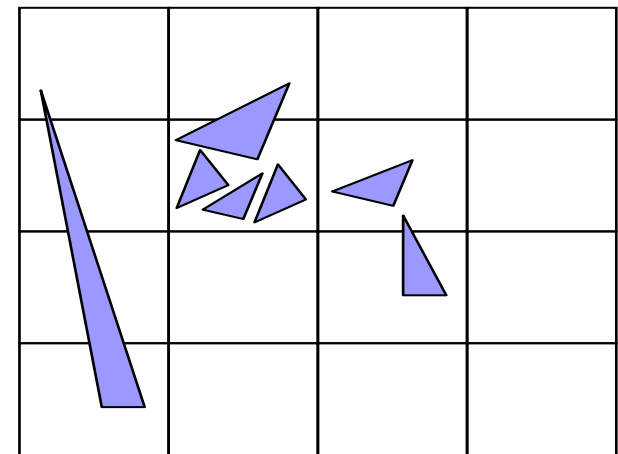
- Without any spatial search data structure, the solutions to these problems require $O(n)$ time, where n is the numbers of primitives ($O(n^2)$ for the collision detection)
- Spatial data structure can make it (average) almost **constant** or expected logarithmic.
- Strong complexity lower bound (worst case log) are possible only for restricted (often not-practical) settings.
 - Hard to be proved, reasonable heuristics are the the standard

Indexing Structures

- Two Class of structures
- **Non-Hierarchical** / Flat space subdivision
 - It would seem trivial, but there are reasons for them
- **Hierarchical**
 - Divide et impera / adaptive subdivision

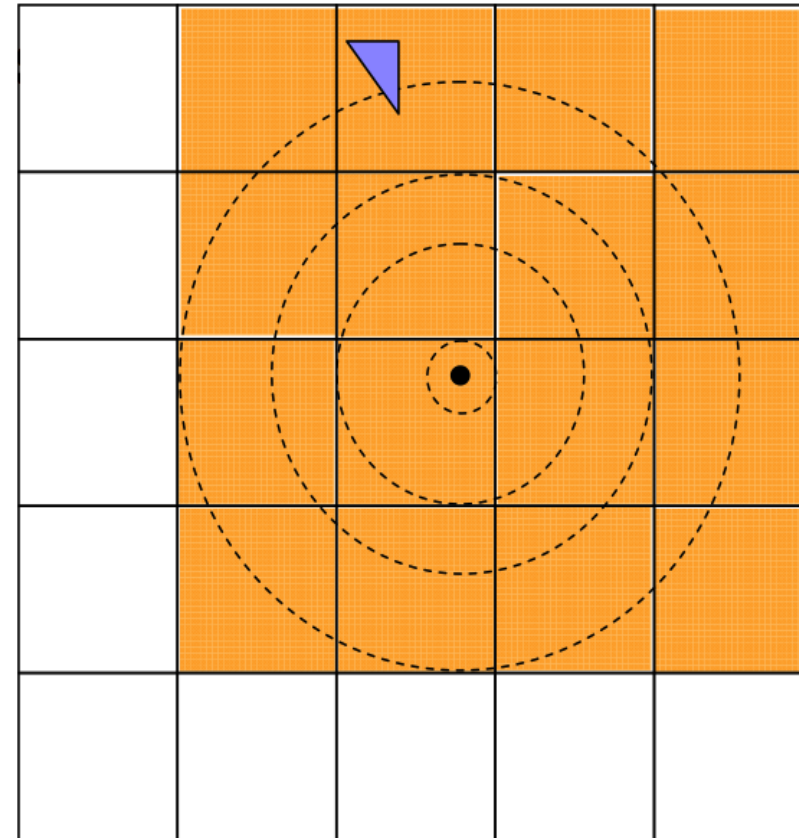
Uniform Grid

- **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)
 - All the cells spanned by each primitive
- **Regular grids access by position is trivial:**
 - If you want to know if something is at (x,y,z) just use integer division...



Uniform Grid

- **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: $O(1)$



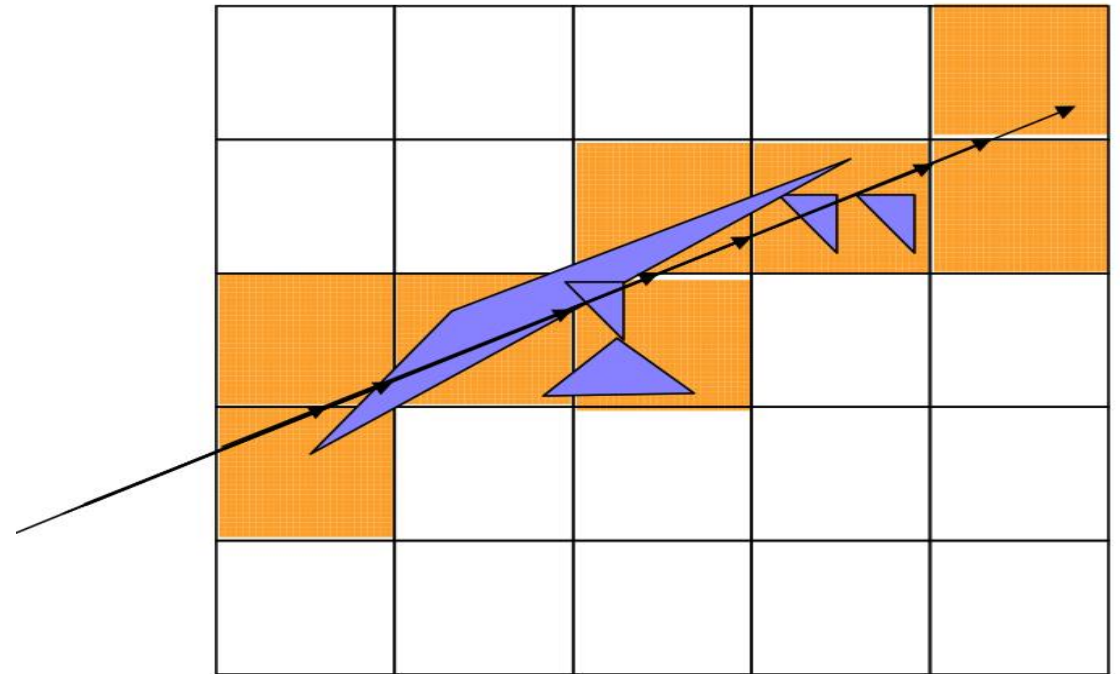
Uniform Grid

- **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)

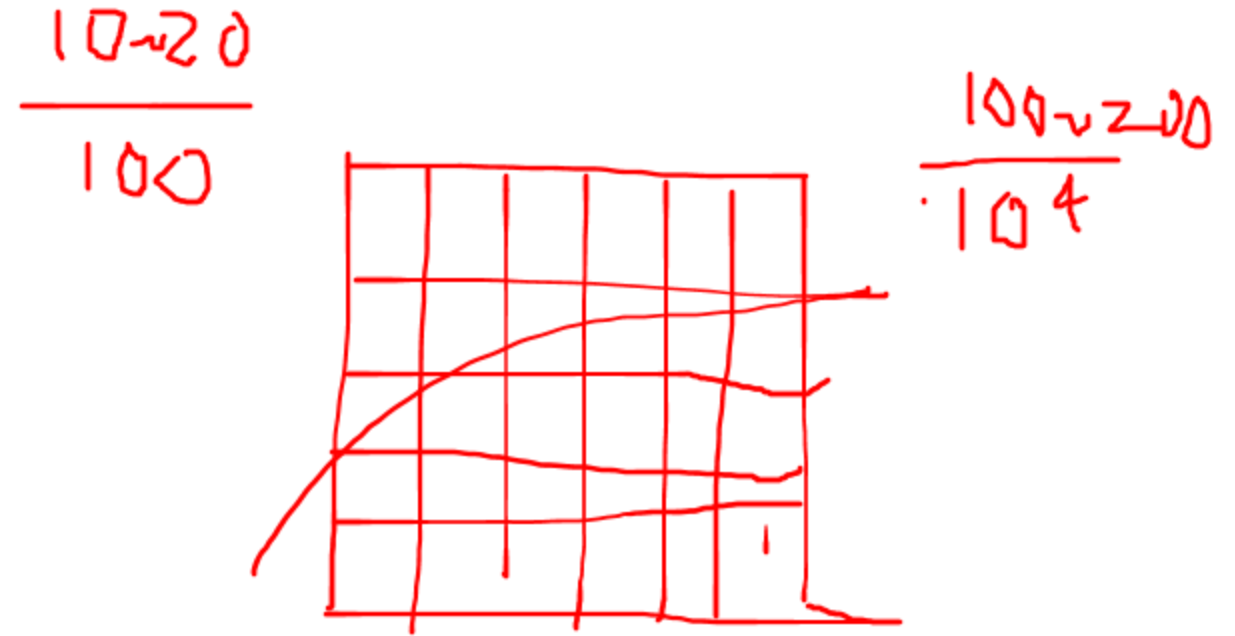
- **Cost:**

- Worst: $O(\#cells + n)$
- Aver: $O(\sqrt[d]{\#cells} + \sqrt[d]{n})$



Uniform Grid

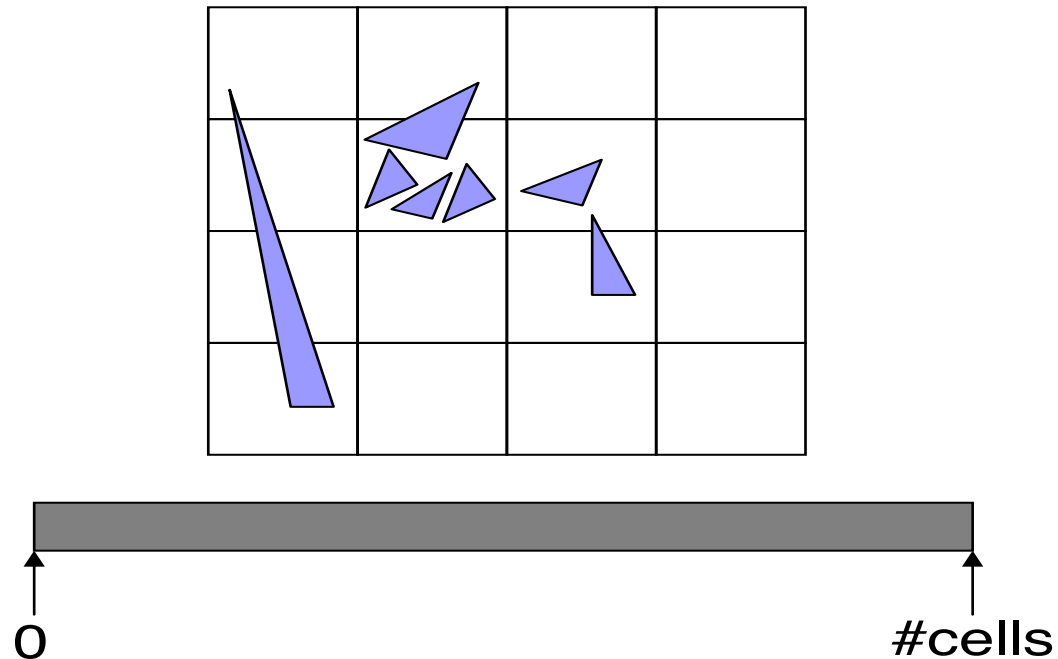
- **Memory occupation:** $O(\# \text{ cells} + n)$
- **Pros:**
 - Easy to implement
 - Fast query
- **Cons:**
 - Memory consuming
 - Performance **very** sensitive to distribution of the primitives.



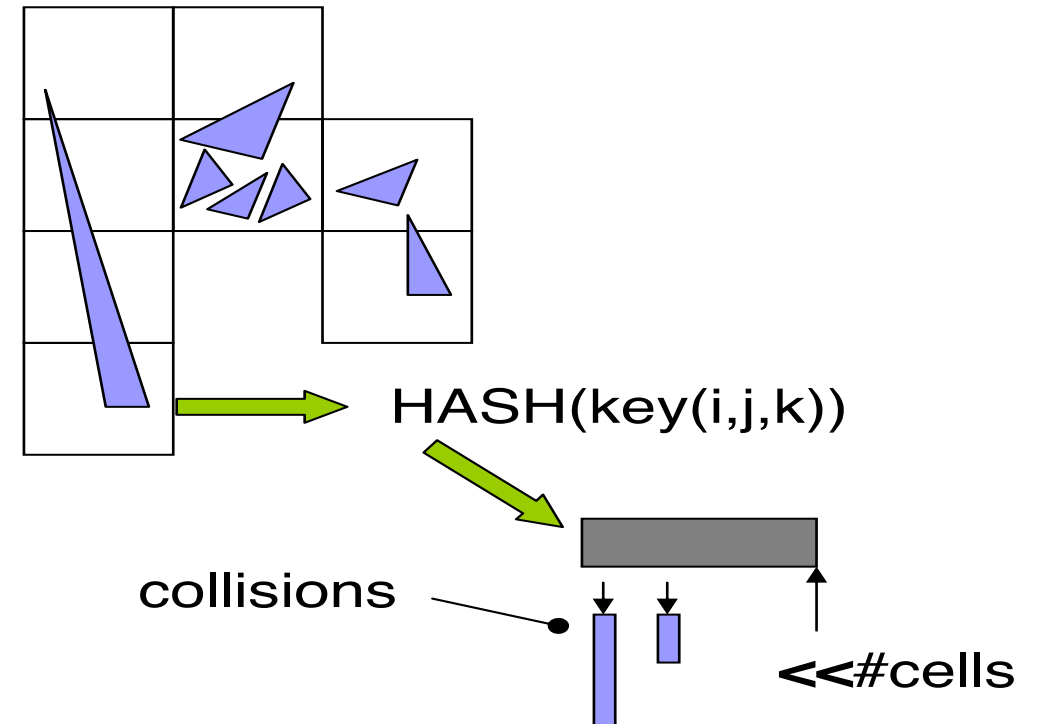
Spatial Hashing

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

Uniform grid



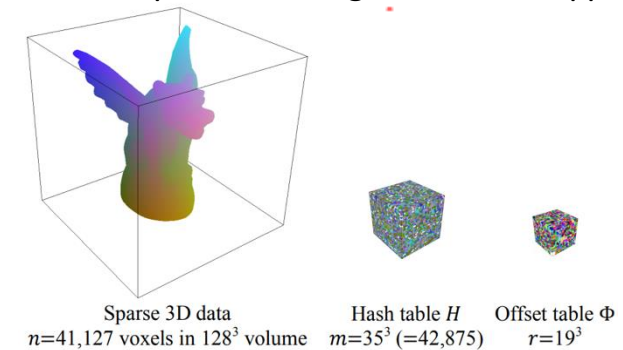
Spatial hashing



Spatial Hashing

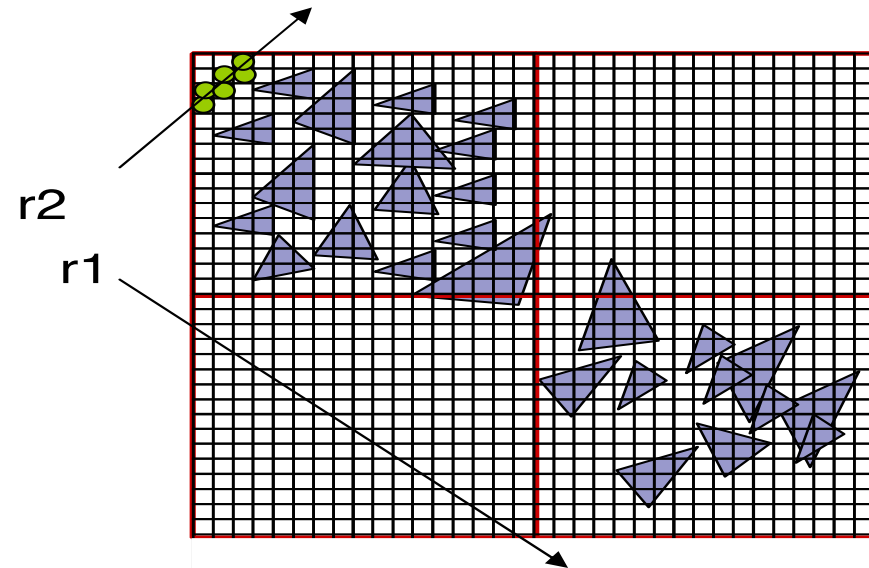
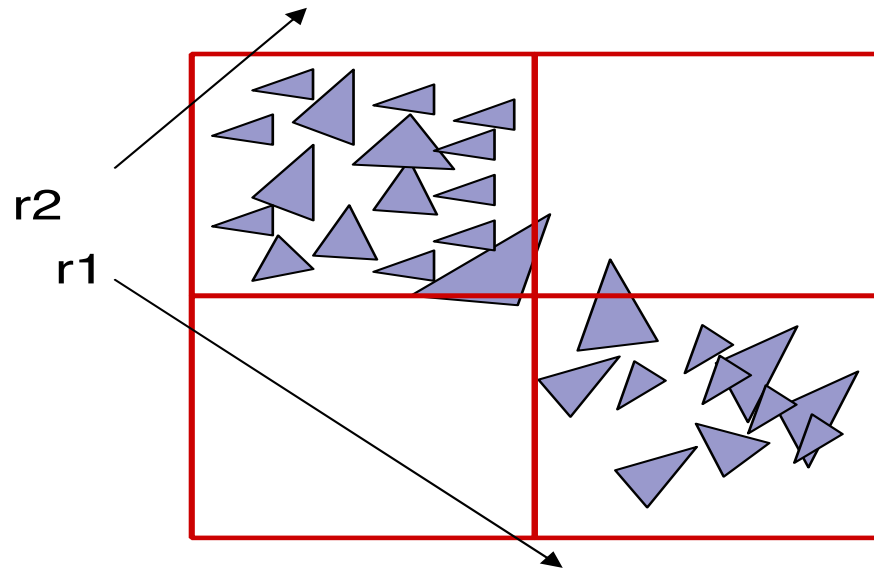
- **Cost:** same as UG, except that in worst case the access to a cell is $O(\#cells)$ because of collisions
- **Memory occupation:**
 - **Worst.:** *all volumetric cells are used*
 - **Aver. :** *only a few surface intersecting cells are allocated*
- **Pros**
 - Fast query if **good hashing** is done
 - Easy to implement
 - Less memory consuming
- **Cons:**
 - Performance **very** sensitive to distribution of the primitives.

Perfect spatial hashing [Lefebvre,Hoppe 06]



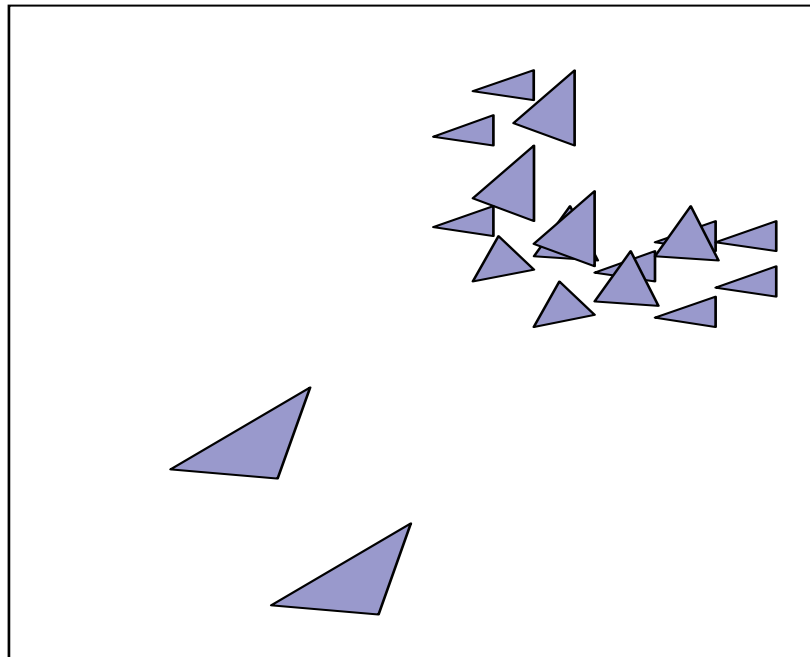
UG Approach: Cell Size

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



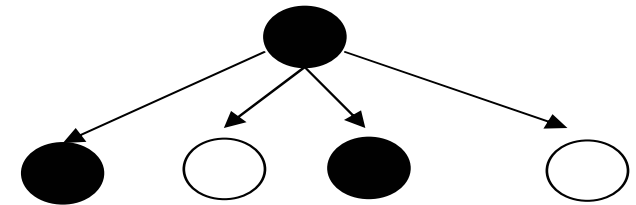
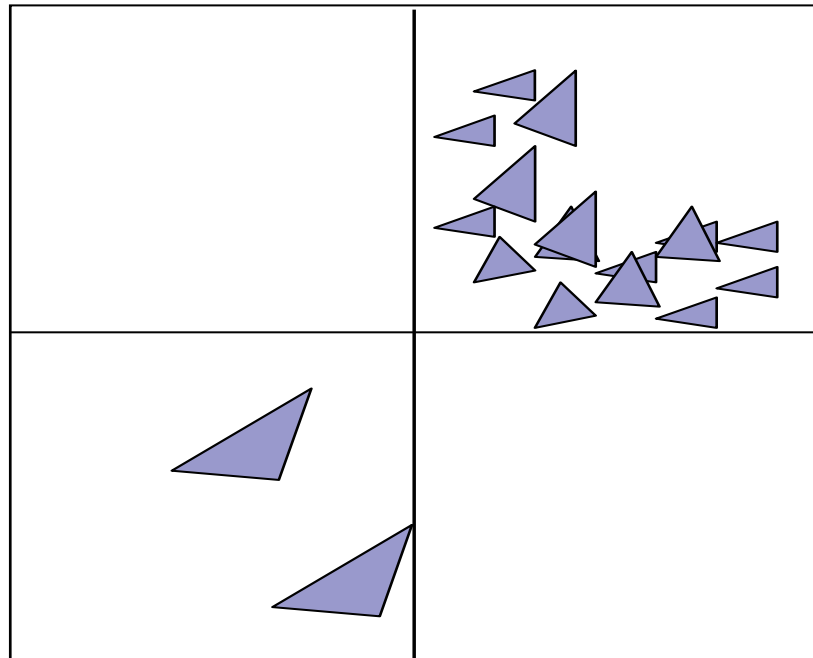
Hierarchical Indexing

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



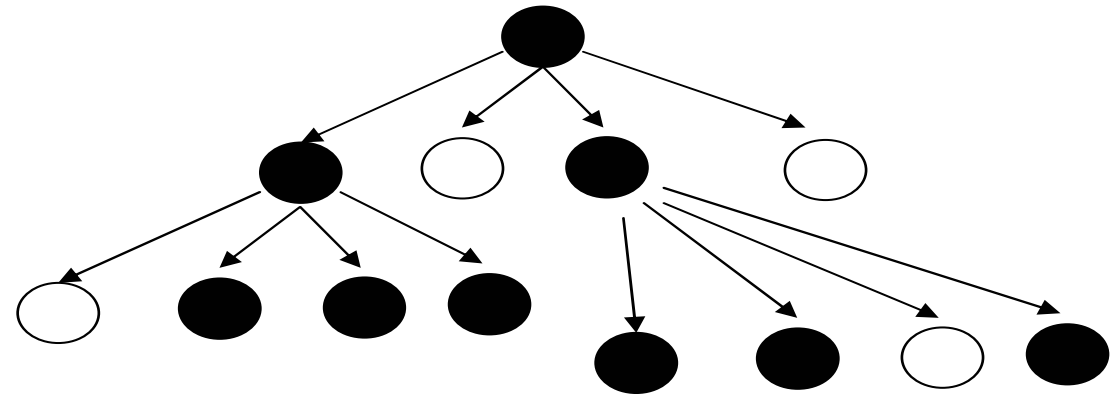
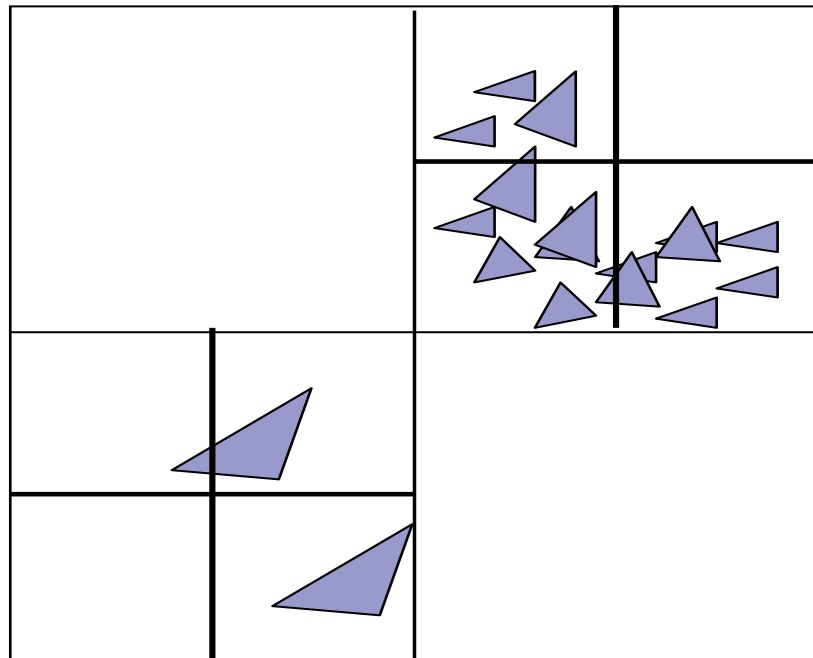
Hierarchical Indexing

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



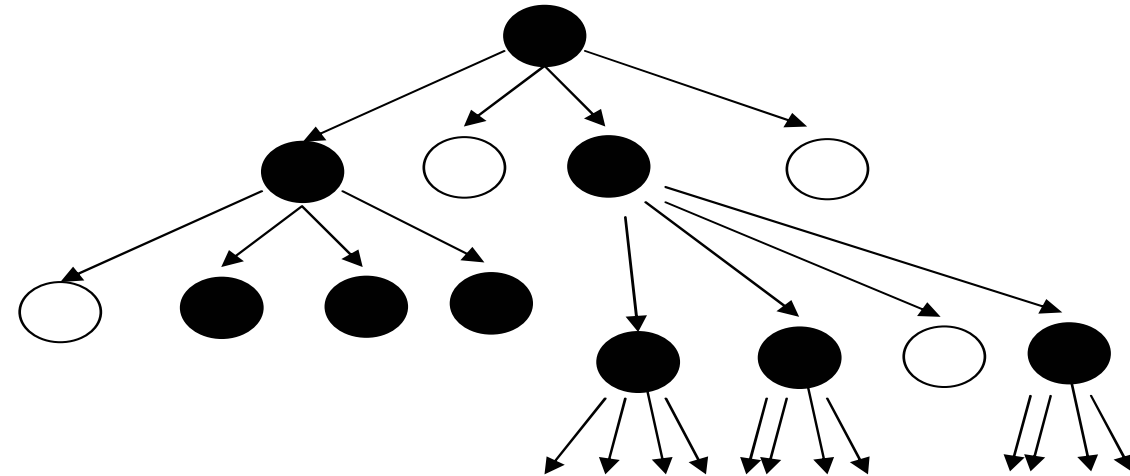
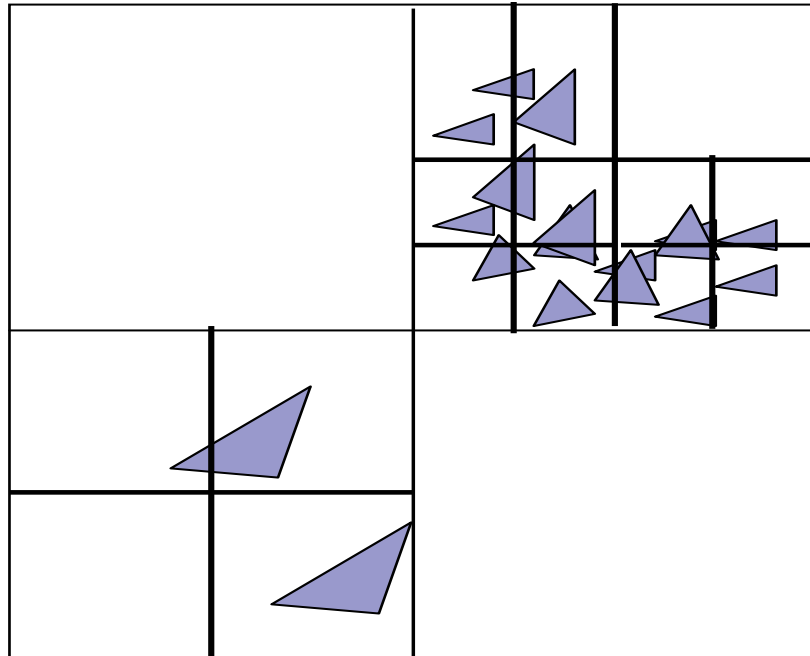
Hierarchical Indexing

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



Hierarchical Indexing

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



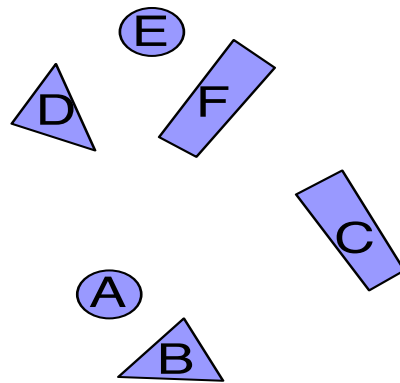
Basic Facts

- The queries correspond to a visit of the tree
 - The complexity is sublinear (logarithmic) in the number of nodes
 - 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)

- **Description:**

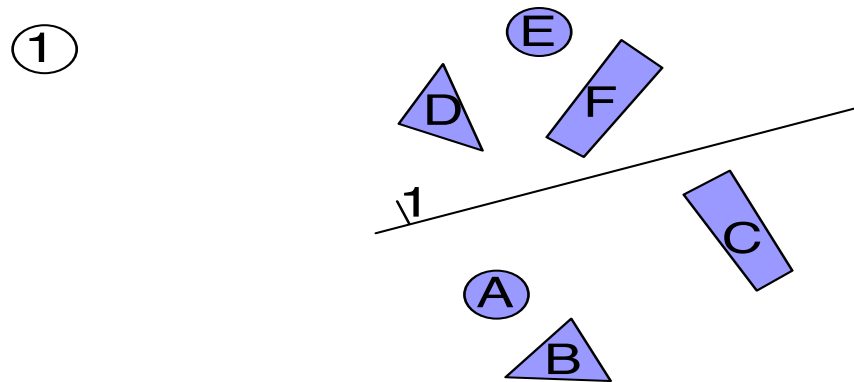
- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**



Binary Space Partition-Tree (BSP)

- **Description:**

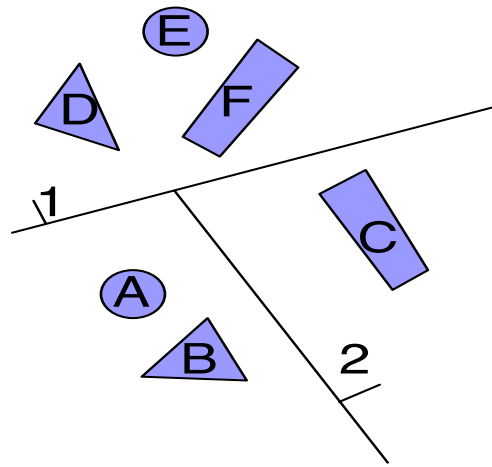
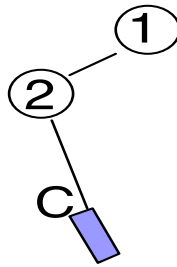
- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**



Binary Space Partition-Tree (BSP)

- **Description:**

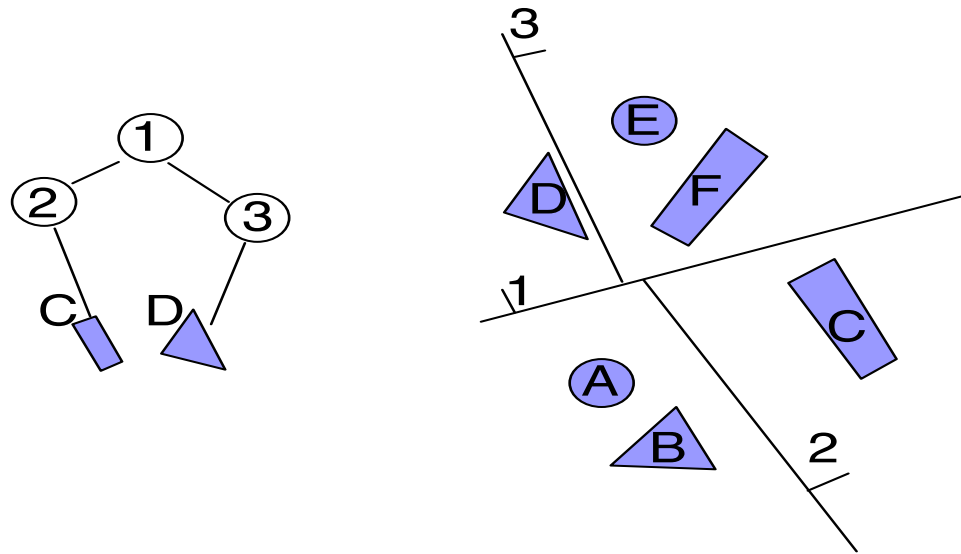
- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**



Binary Space Partition-Tree (BSP)

- **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**

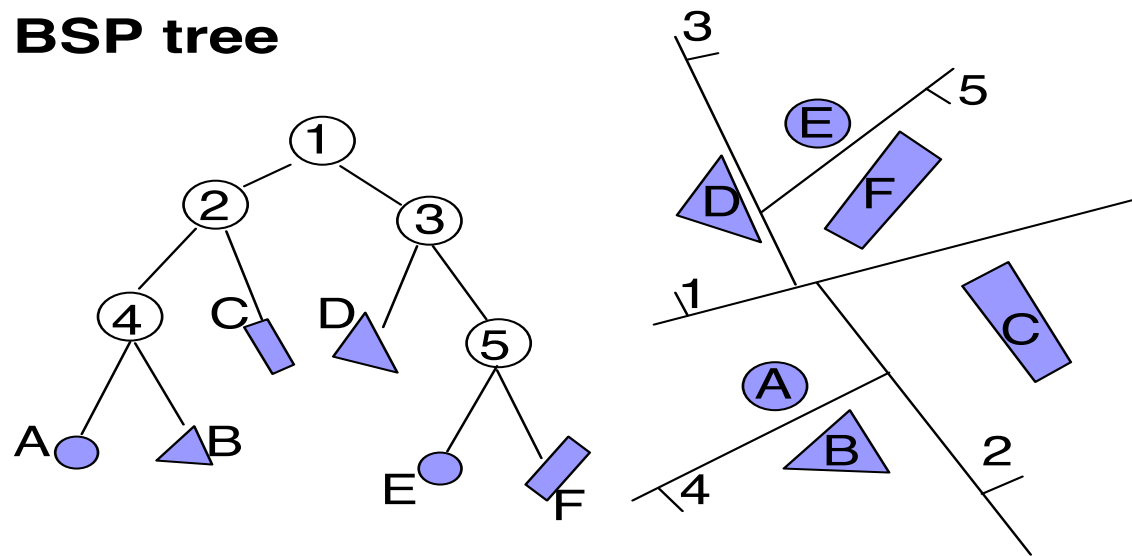


Binary Space Partition-Tree (BSP)

- **Description:**

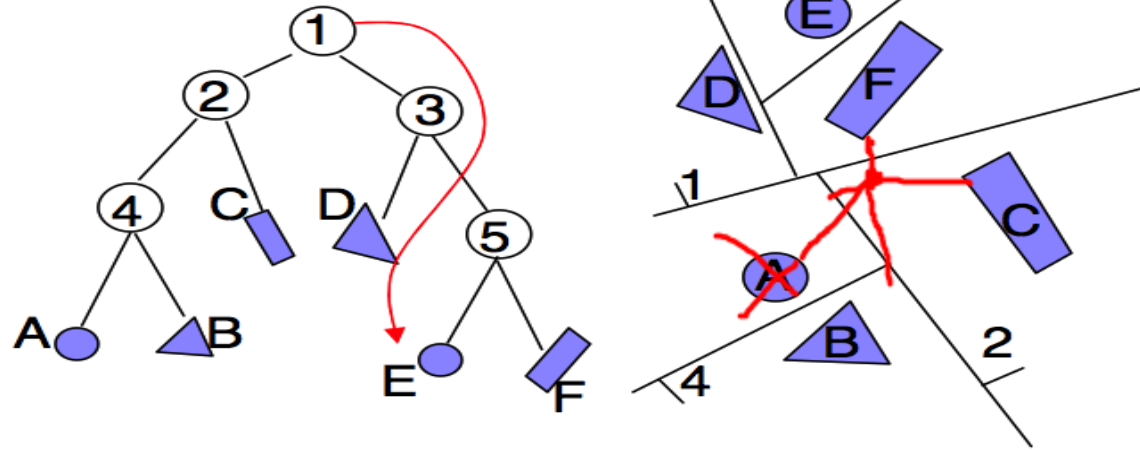
- It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
- therefore a node always corresponds to a **convex region**

BSP tree



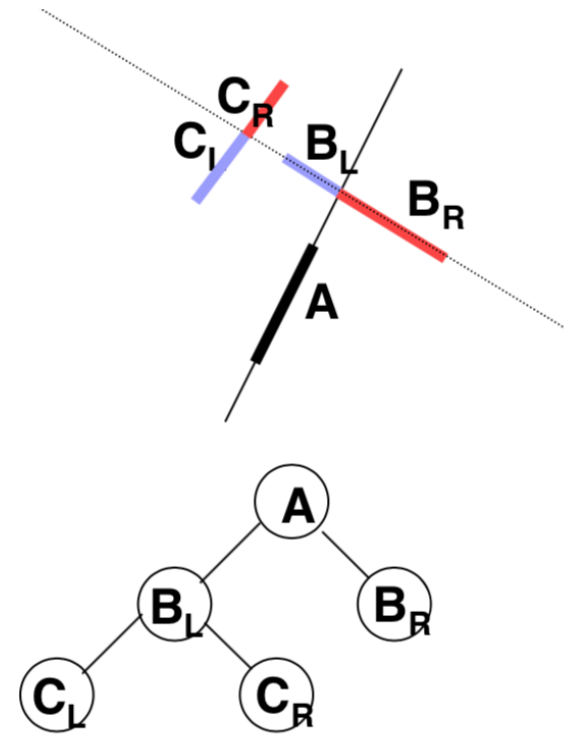
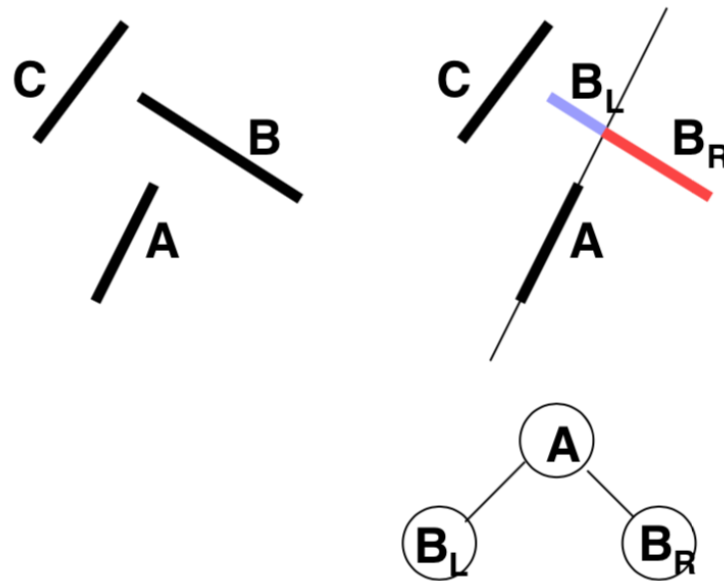
Binary Space Partition-Tree (BSP)

- **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: $O(\log n)$



Binary Space Partition-Tree (BSP)

- What could go wrong?
 - What happen to split primitives? Can I bound them?
- Where to place the plane?
- A common strategy is:
 - Primitives are planar faces:
 - Use one of the primitive as splitting plane and decompose the rest



BSP-Tree Cost

- 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
$$C(T) = 1 + P(T_L) C(T_L) + P(T_R) C(T_R)$$
 - Where $P(T_L)$ is probability that T_L is visited given that T has been visited.

BSP Tree Cost

- How to choose the splitting primitive?
- Try to guess the cost:

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

- We choose the primitive that minimize

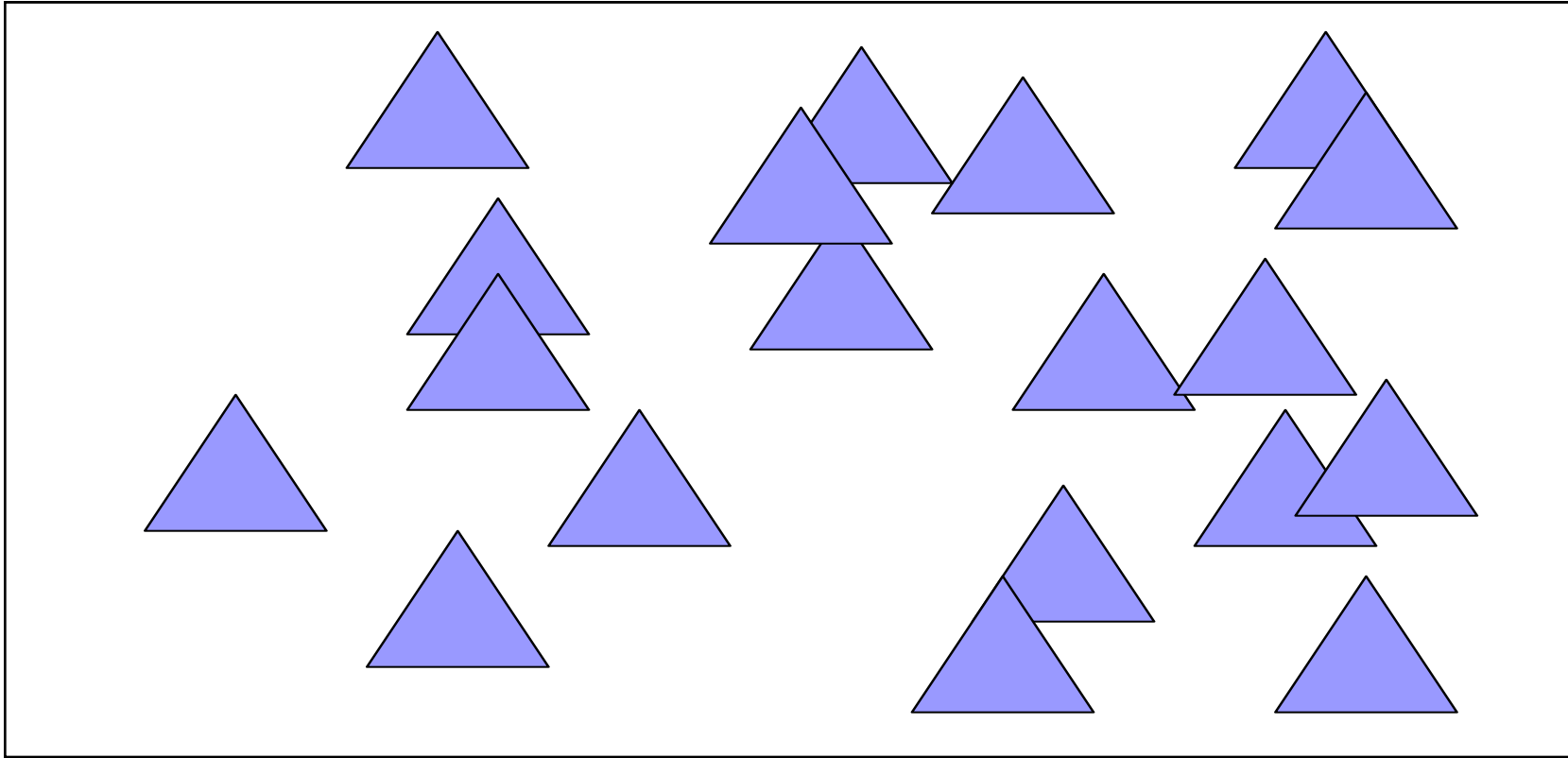
$$1 + |S(T_L)|\alpha + |S(T_R)|\alpha + \beta s$$

- S_L number of primitives in the left subtree
- s number of primitives split by the chosen primitive
- Big α , small β yield a balanced tree
- Big β , small α yield a smaller tree

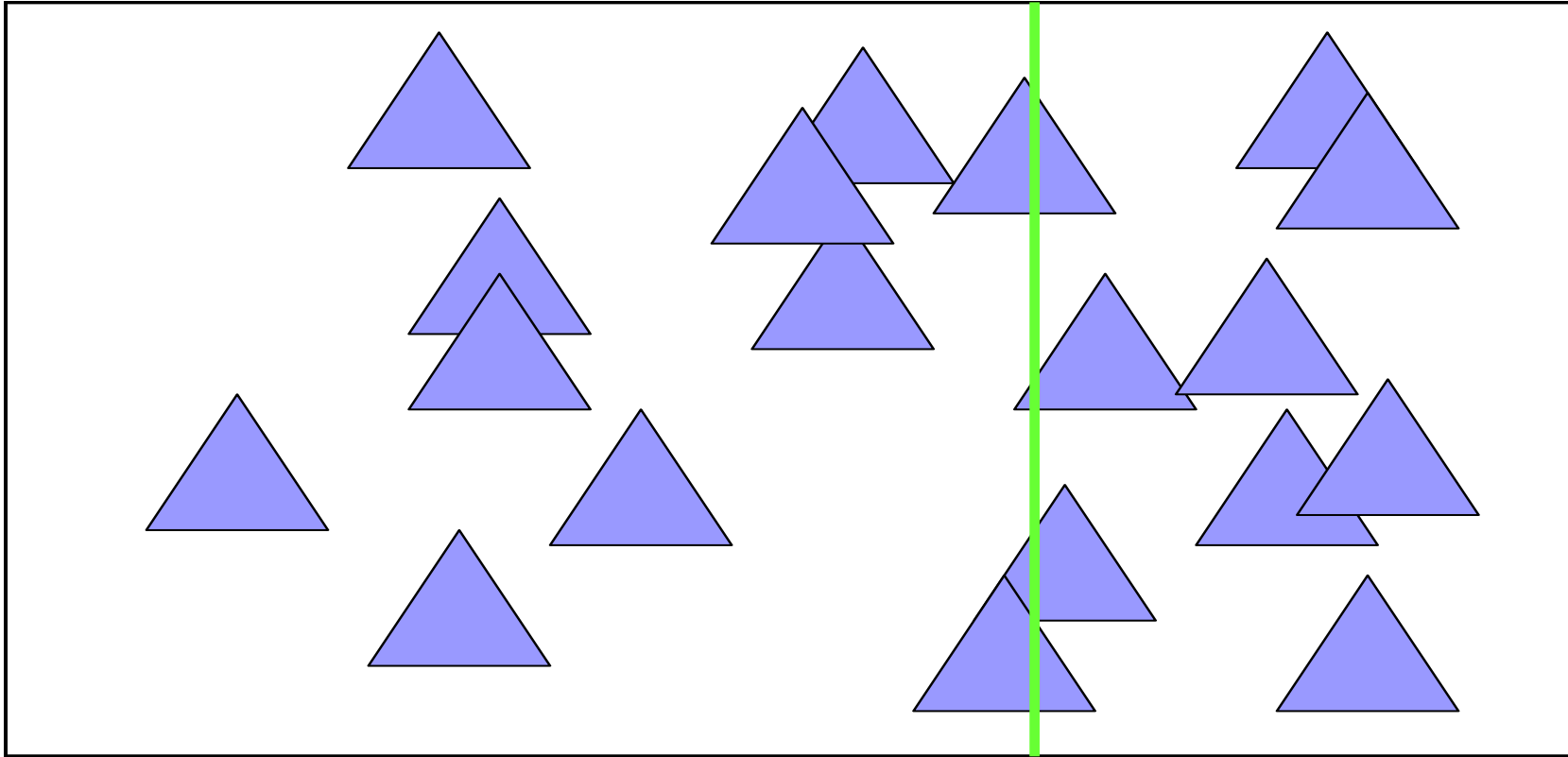
KD-Tree

- Kd-tree : k dimensions tree
- It's a special kind of BSP tree with axis-aligned bisector planes
- It depends on:
 - Chosen Axis
 - Point on axis where to define the plane
- Advantages wrt BSP:
 - Test are really fast (to explore the tree)
 - Lower memory consumption

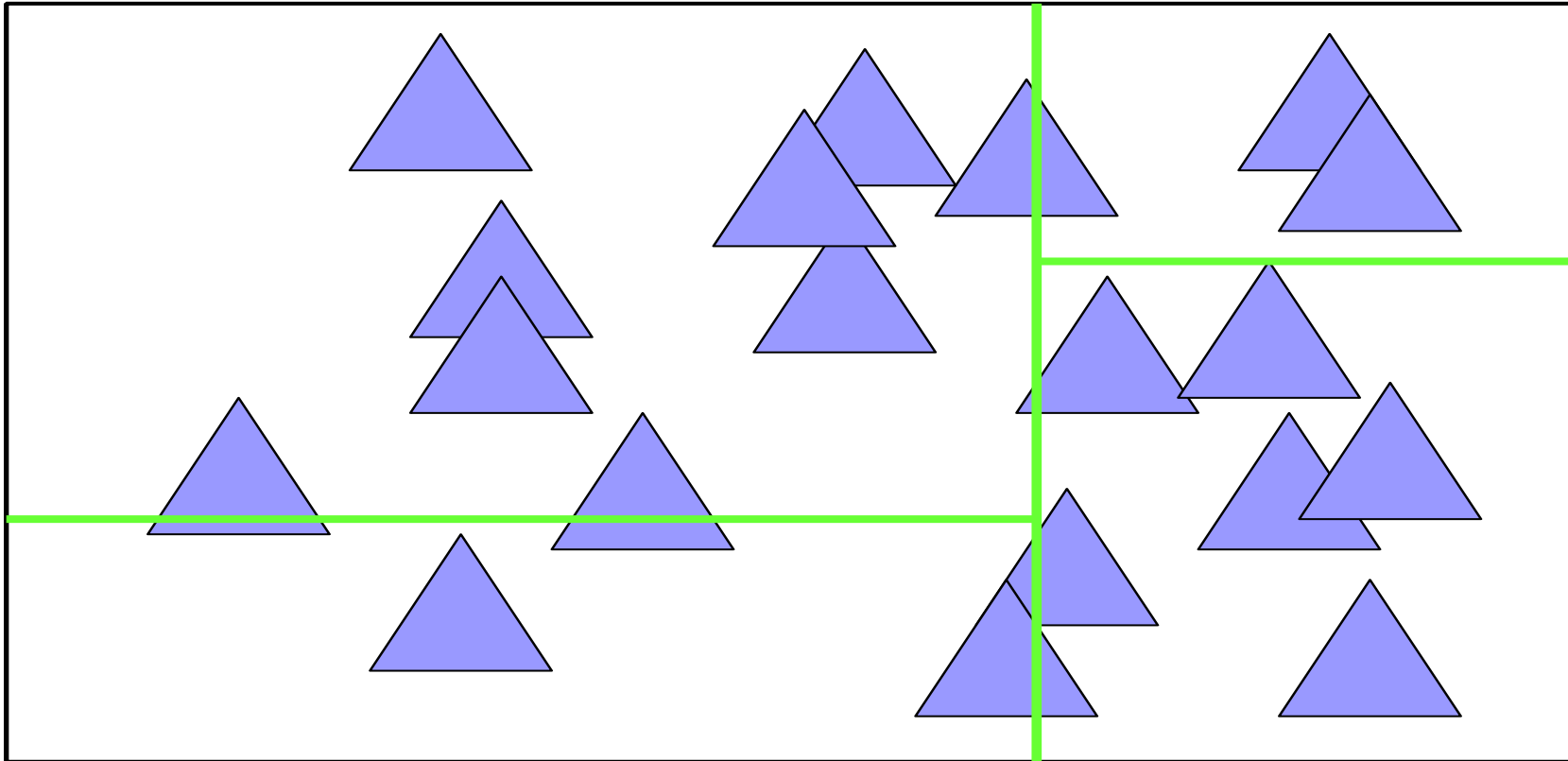
KD-Tree



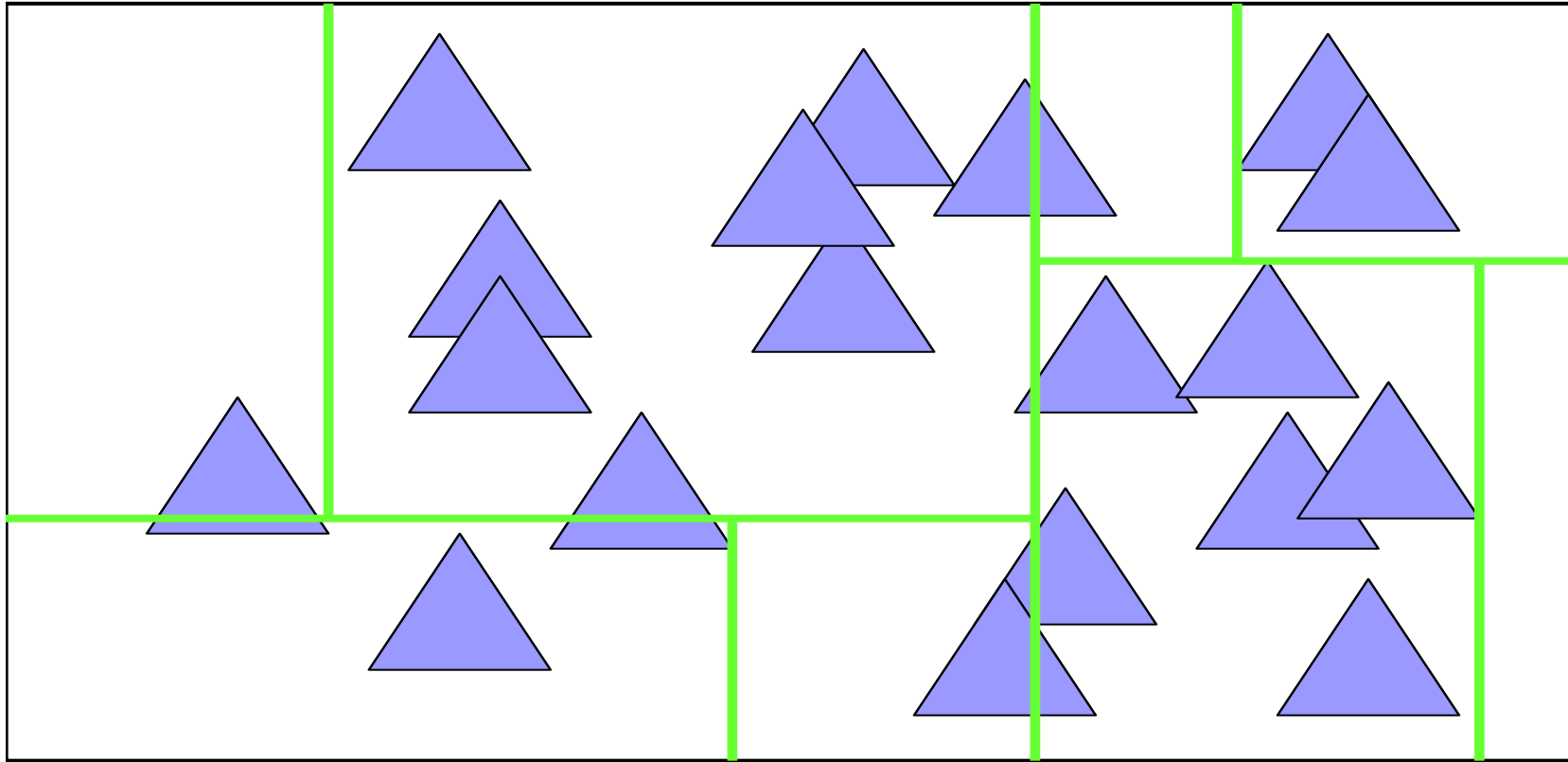
KD-Tree



KD-Tree

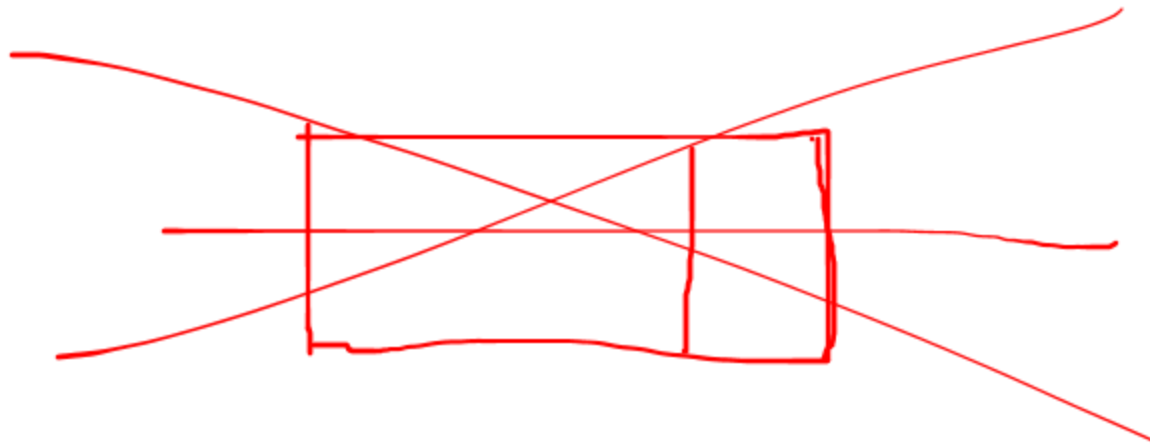


KD-Tree



Kd-tree More on cost

- Example Ray intersection
- $C(T) = 1 + P(T_L)C(T_L) + P(T_R)C(T_R)$
- The cost of a final leaf is roughly the number of primitives
 - (you have to test them)
- $P(T_L)$ is more interesting:
$$P(T_L) = \frac{|\text{rays intersecting } T_L|}{|\text{rays intersecting } T|}$$



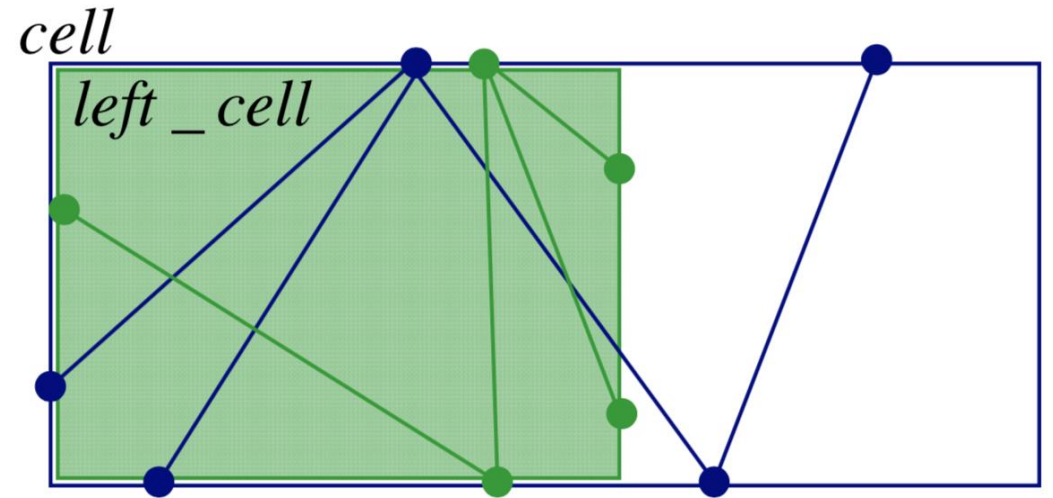
Kd-tree More on cost

$$P(T_L) = \frac{|\text{rays intersecting } T_L|}{|\text{rays intersecting } T|}$$

You can consider rays as pairs of points over the surface of the cell.

Intuitively a ray (p_1, p_2) that hits T hits also T_L IFF either p_1 or p_2 are on T_L

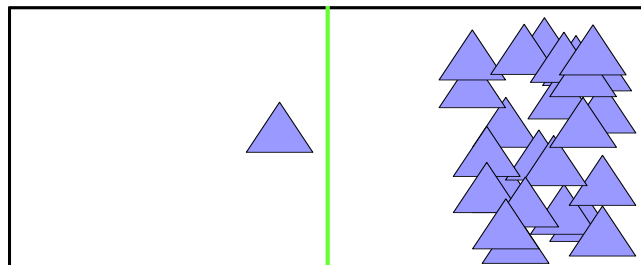
With a few assumptions on ray distrib.



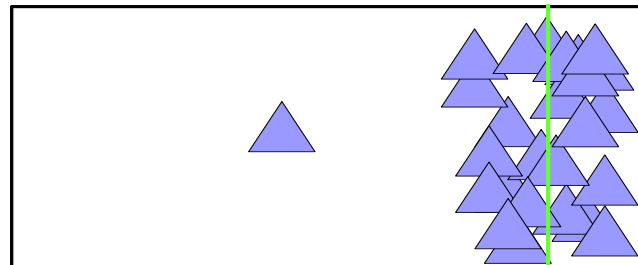
$$P(T_L) = \frac{|\text{surface area } T_L|}{|\text{surface area } T|}$$

KD-Tree:construction

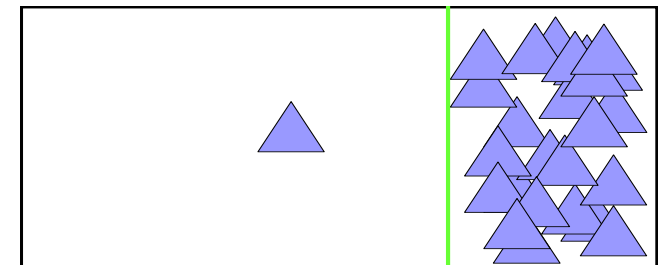
- Input:
 - axis-aligned bounding box (“cell”)
 - List of triangles
- Base Operations
 - Split a cell using an axis aligned plane (**where?**)
 - Distribute triangles among the two sets
 - Recursive call



In the middle



median



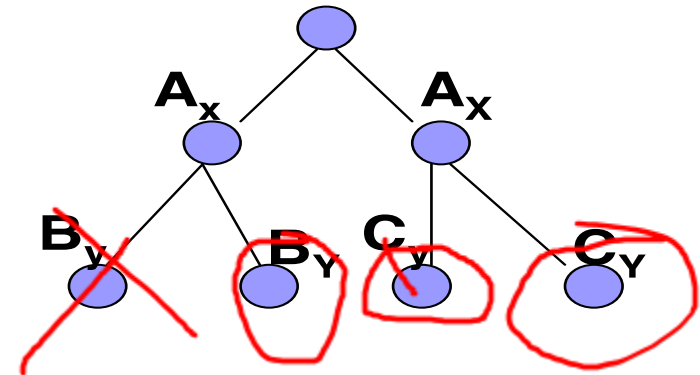
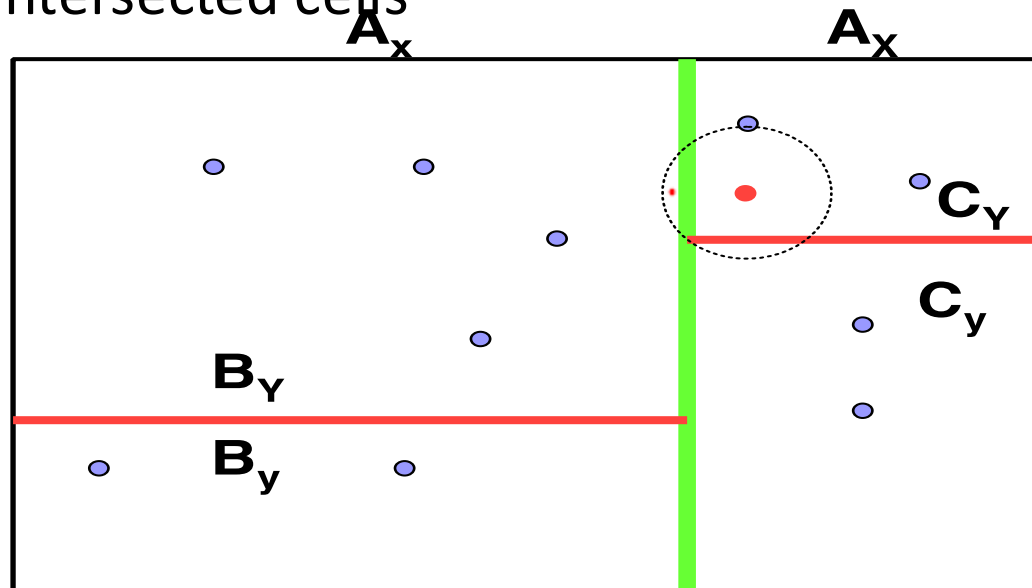
Cost optimized

KD-Tree:range query

- **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 node is **partially** inside the box recur to the children
- **Cost:** if the leaf nodes contain one primitive and the tree is balanced: $O(n^{1-\frac{1}{d}} + k)$ $n = \# \text{primitives}$ $d = \text{dimension}$
- $O(n^{2d})$ possible results

Nearest Neighbor with kd-tree

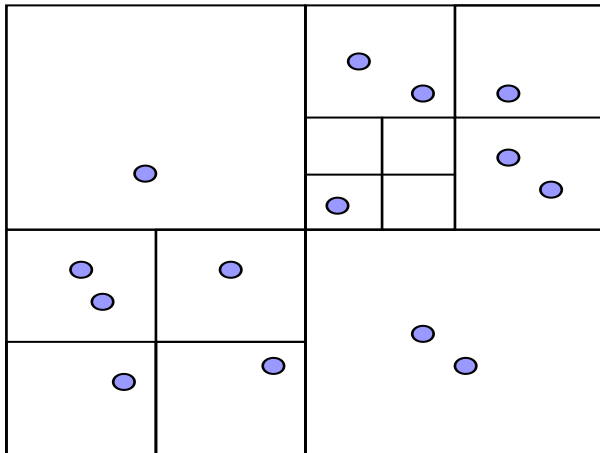
- **Query:** return the nearest primitive to a given point c
- **Algorithm:**
 - Find the nearest neighbor in the leaf containing c
 - If the sphere intersect the region boundary, check the primitives contained in intersected cells



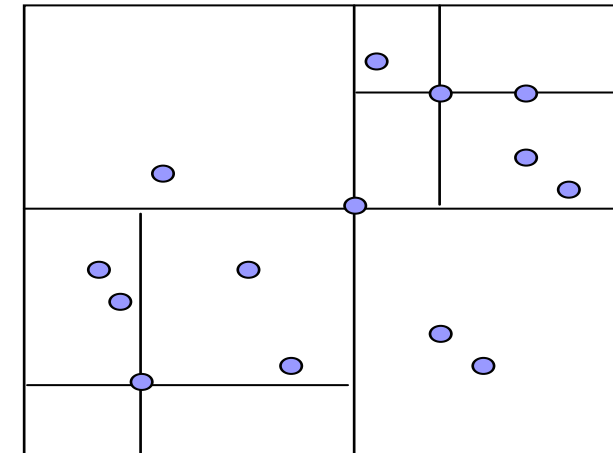
Quad-Tree (2D)

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

Region Quad-tree

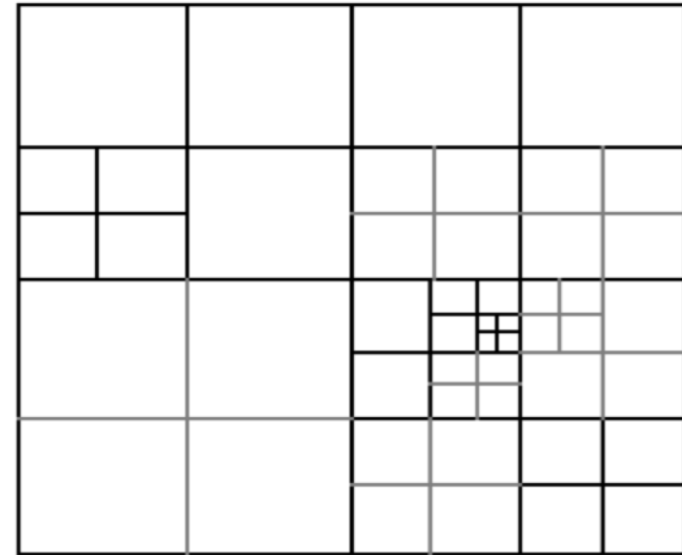
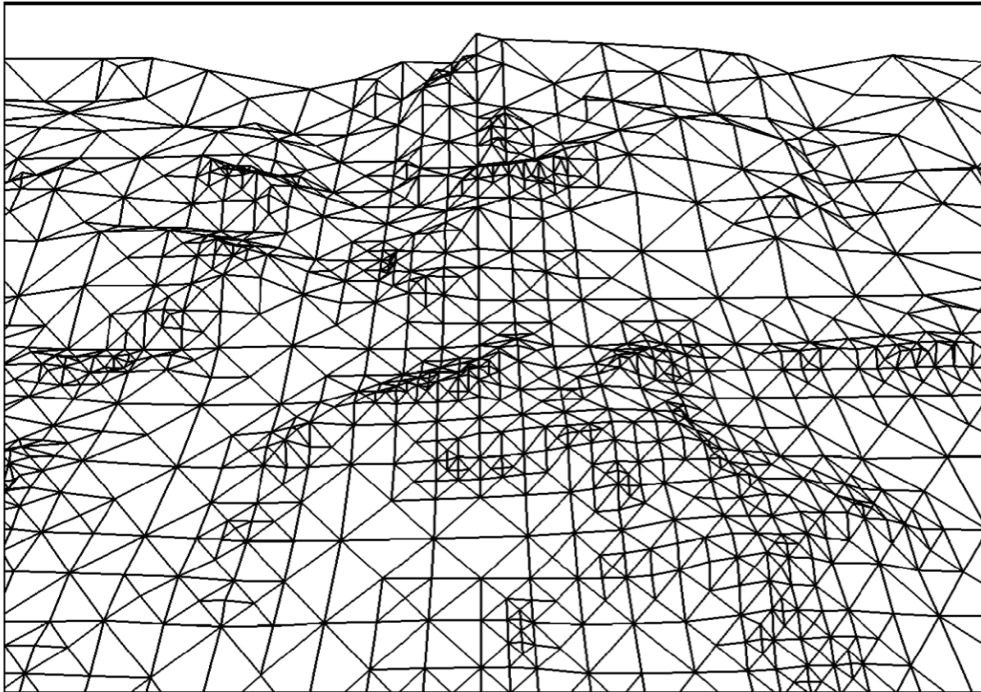


Point Quad-tree



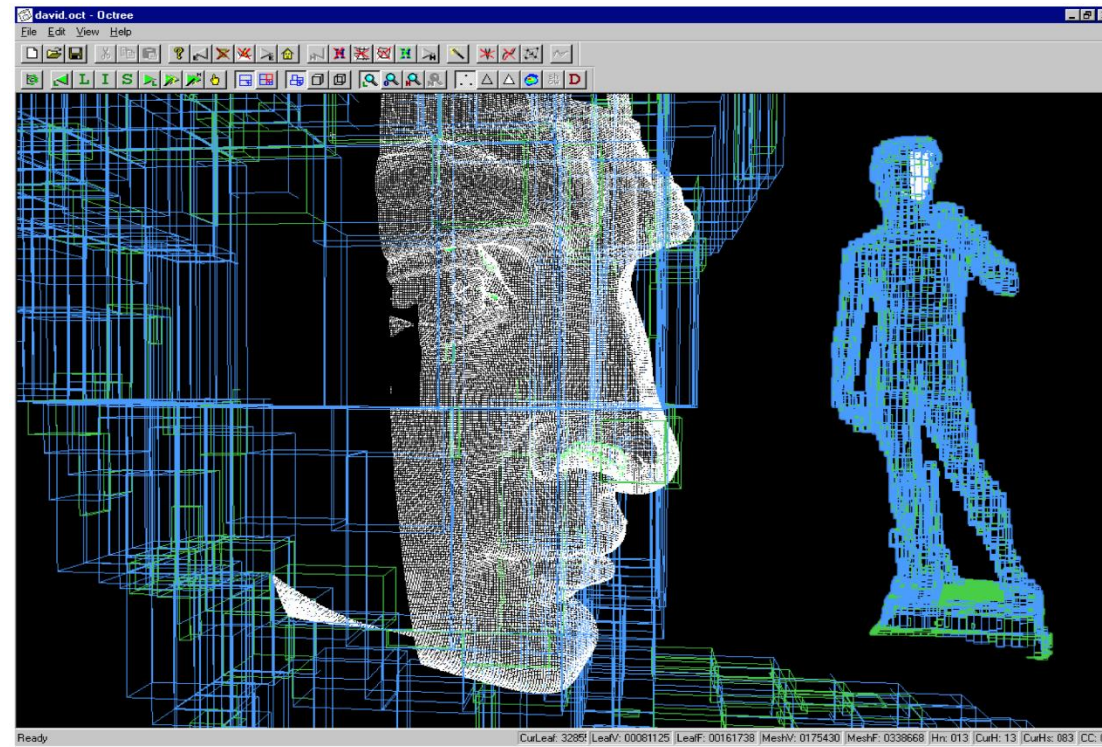
Quad-Tree (2d):example

- 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:



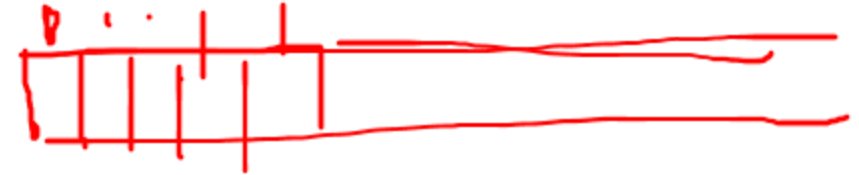
Large meshes: out of core

Oct-Tree (3d)

- 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 by using a linear array (convenient only if the hierarchies are complete) where:

quadtree

$$Children(i) = 4i + 1, \dots, 4 * (i + 1)$$

$$Parent(i) = \lfloor i / 4 \rfloor$$

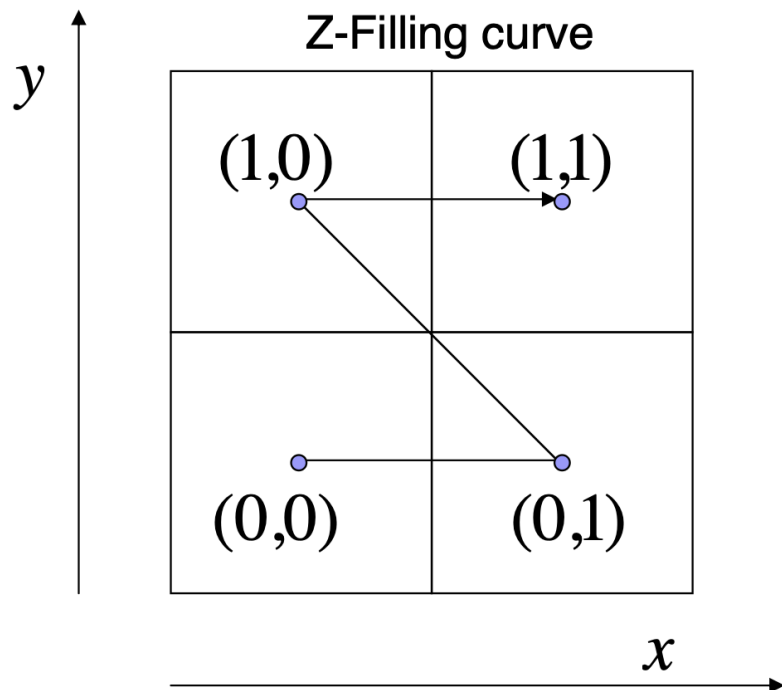
octree

$$Children(i) = 8i + 1, \dots, 8 * (i + 1)$$

$$Parent(i) = \lfloor i / 8 \rfloor$$

Z-Filling Curves

- Position and size of the cells are implicit
- They can be indexed to preserve locality, i.e
Spatially close -> Close in memory



Easy conversion
between position in
space and order in the
curve

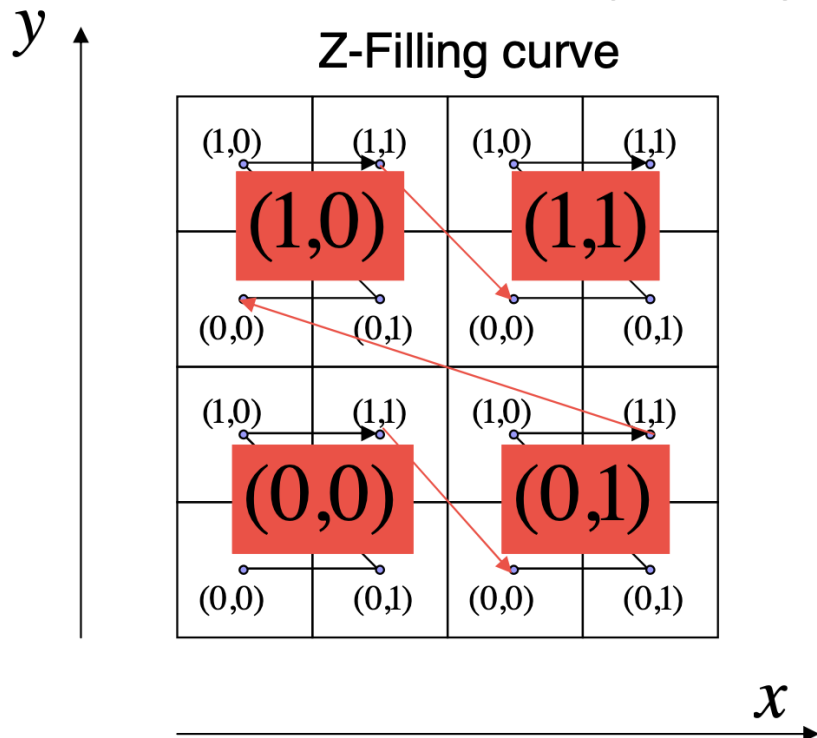
Just use the 0..1
coordinates as bits

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 -> 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

00	00
00	01
00	10
00	11
01	00
01	01
01	10
01	11
10	00
10	01
10	10
10	11
11	00
11	01
11	10
11	11

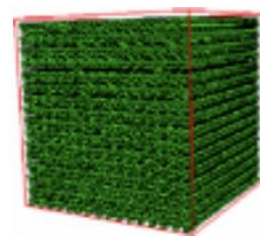
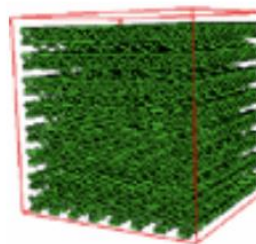
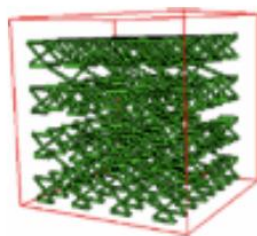
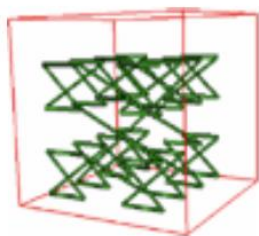
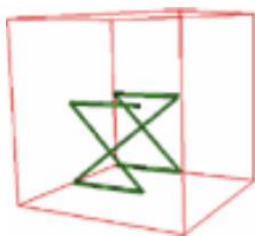
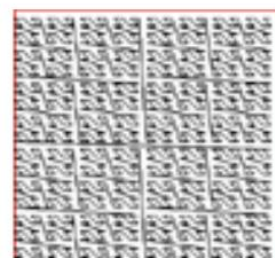
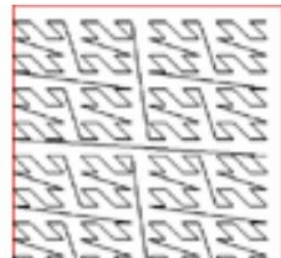
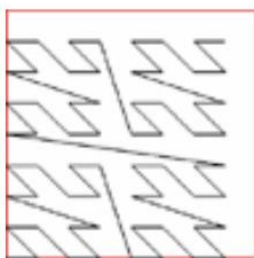
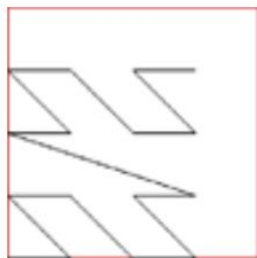
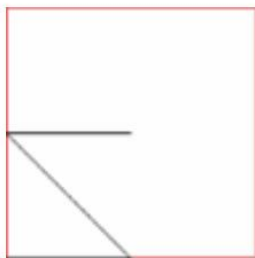
Z-Filling Curves

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

$$\begin{array}{l} x = \quad b_0^x \quad b_1^x \quad b_2^x \quad \dots b_n^x \\ y = b_0^y \quad b_1^y \quad b_2^y \quad \dots b_n^y \end{array}$$

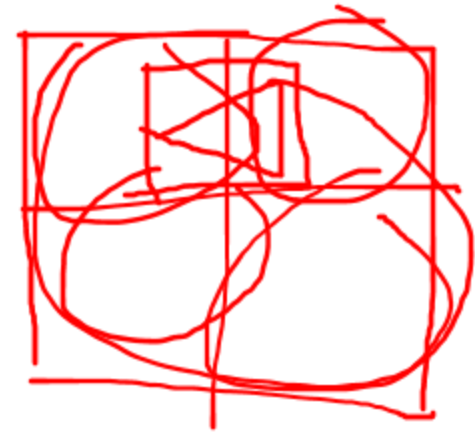
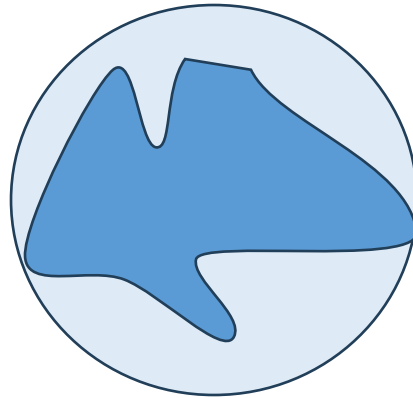
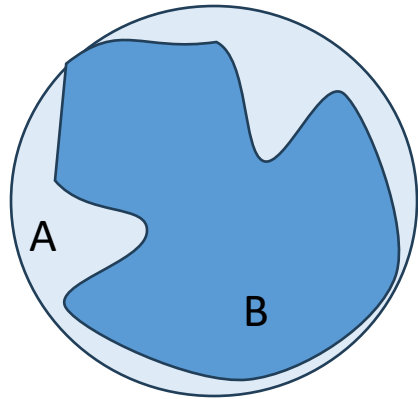
$$id = b_0^y \ b_0^x \ b_1^y \ b_1^x \ b_2^y \ b_2^x \ \dots \ b_n^y \ b_n^x$$

Z-Filling Curves



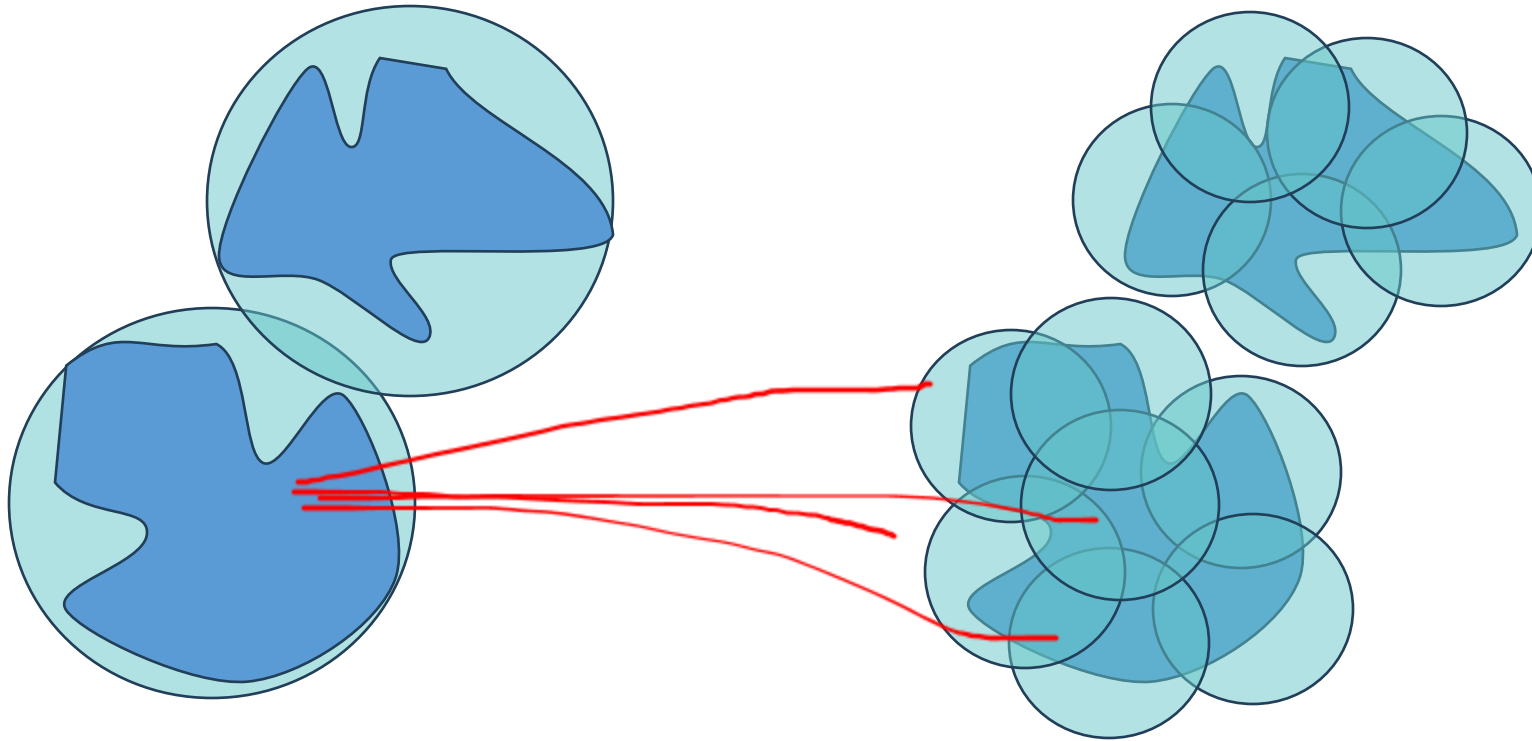
Bounding Volume Hierarchies (BVH)

- E.g. coping with the fact that perfect partition does not happen
 - 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



Bounding Volume Hierarchies (BVH)

- If overlaps, refine



Bounding Volume Hierarchies (BVH)

Questions:

SAMPLE

- 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?

Most of the literature on BVH is about answering these questions

Conclusion

- No perfect data structure
- Depend a lot on your pattern of query
 - Close to surface vs random
 - Static vs dynamic