Spatial Acceleration Structures

Why spatial acceleration structures?

For naive ray tracing you have to check for each ray all triangles in the scene

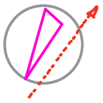
→ this a lot to compute

→ sort primitives spacially in a hierarchy

Bounding Volumes

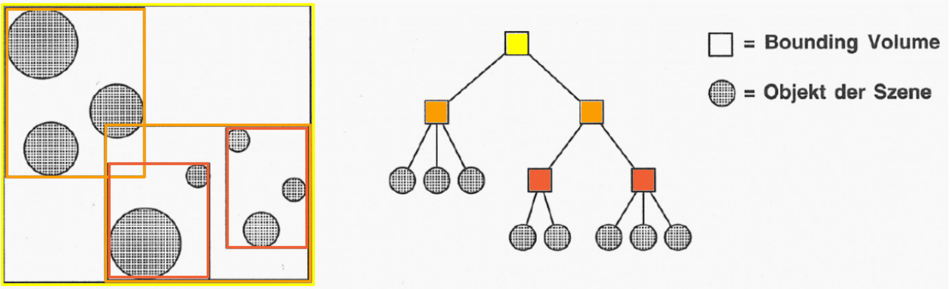
pack primitives into simple volumes → only check primitive intersection, if bounding volume is hit

- sphere: + easy intersection calculation
  - inefficient, because too large



- aabb: + easy intersection calculation
  - sometimes too large
- oriented bounding box (non-axis-alighned): + better fit
  - complex intersection computation

Bounding Volume Hierarchy



organize bounding volumes hierarchically

+ very good adaptivity

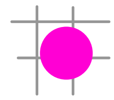
+ efficient traversal O(log n)

- how to arrange BVs?

  ↳ avoid overlapping of bb on same level

Grid

partition scene with equal sized voxels



-> one object can be represented in multiple cells

+ trivial insertion of objects

+ easy construction

- high memory costs (a^4)

- expansive traversal

  ↳ a lot of empty voxels to traverse

  ↳ which voxel to traverse next? (Bresenham Algorithm)

- stop traversal after detecting intersection

  ↳ otherwise overlapping objects will be intersected twice

  ↳ alt. solution: mailboxing (store index / id of intersected primitive

- scene dependent grid resolution ("Teapot in a stadium")

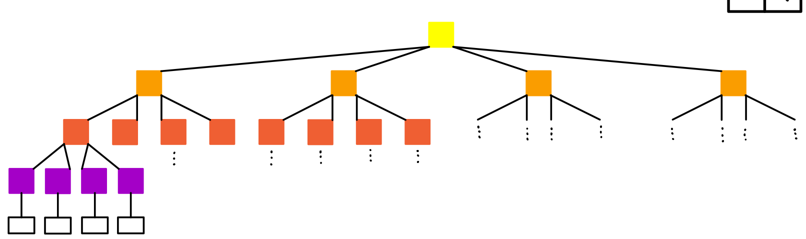
  ↳ solution: hierarchical grid

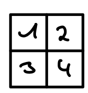
Quadtree

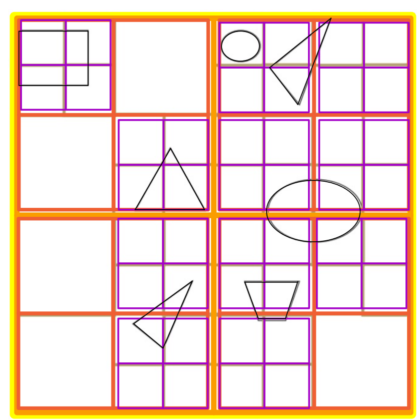
hierarchical subdivision into 4 cells each level

subdivide until cell is emty or has less then n primitives

-> tree traversal structure







Octree

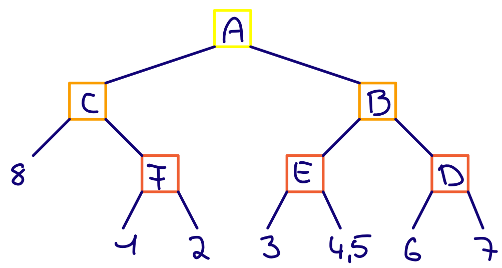
3D subdivision into 8 Voxels

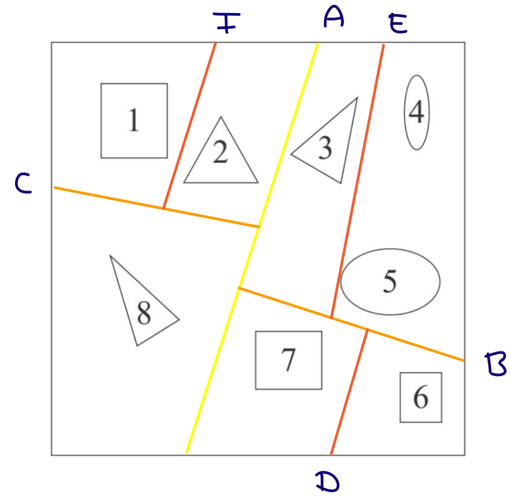
- complex traversal
- slow to refine complex regions

+ simple construction in $O(n)$
+ simple insertion of objects
- complex traversal $O(k \cdot \log n)$
- high memory consumption

Binary Space Partition (BSP)

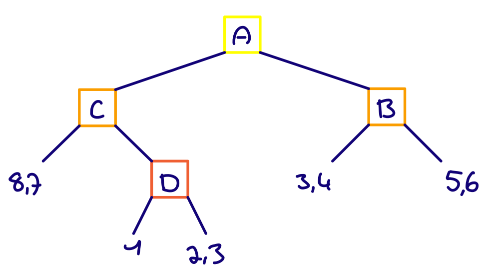
- recursively split space into halves
- „arbitrary" direction of planes

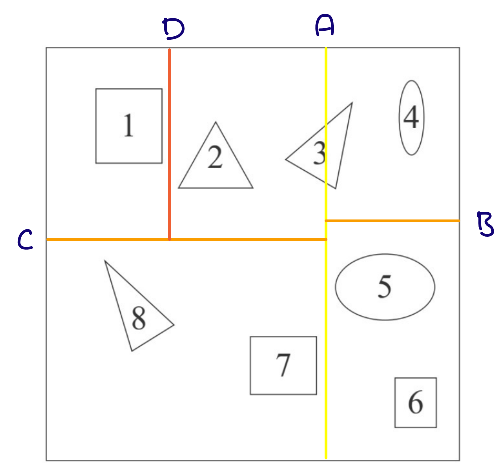




kd-Tree

- same principle as BSP, but axis alinged
- planes are defined by
  - axis flag
  - split point
  - child pointer





BSP / kd-Tree Traversal

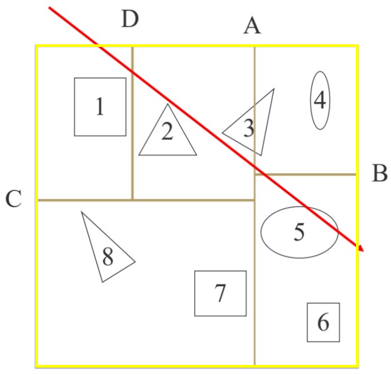
- „Front-to-Back" traversal (start with root node)
- child nodes are traversed in order along the ray
- traversal stops with first intersection
- implementation with stack

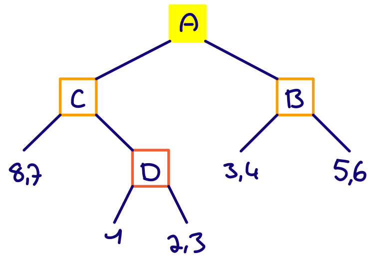
function traverse(ray, root):

```
node = root
intersection == false
while intersection == false:
    if node is leaf:
        intersection = checkTriangleIntersection(node)
        if intersection == false:
            node = pullFromStack
            if node is empty:
                return "no intersection"
            else: continue
    else:
        if node.far_child is hit by ray:
            pushOnStack(node.far_child)
        node = node.close_child
while end
return intersectionPoint(node)
```

traversal example

1.



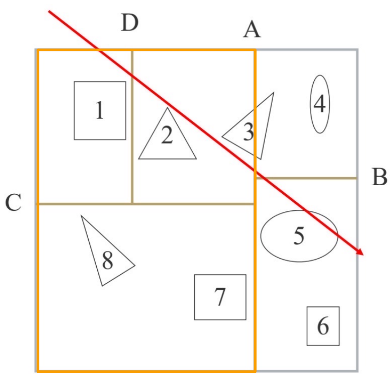


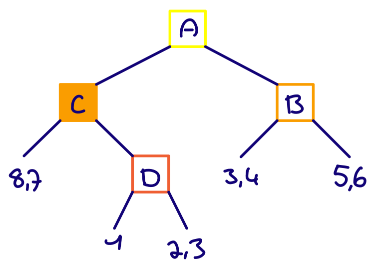
Process	Stack
A	

depending on intersection point with plane A, we know, we hit the outer bounding box

-> check both children, first the closer one

2.

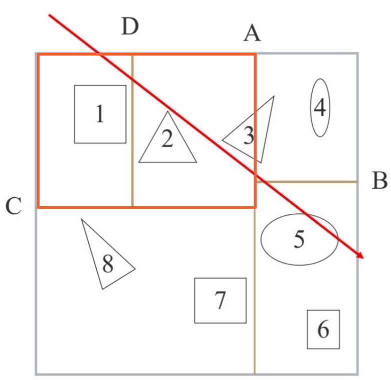


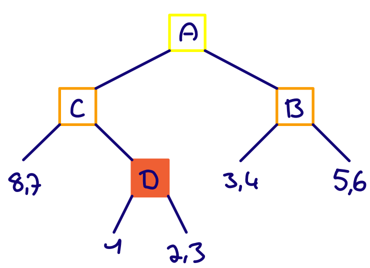


Process	Stack
C	B

because the ray isn't intersecting plane C, we know it only intersects the closer child

3.



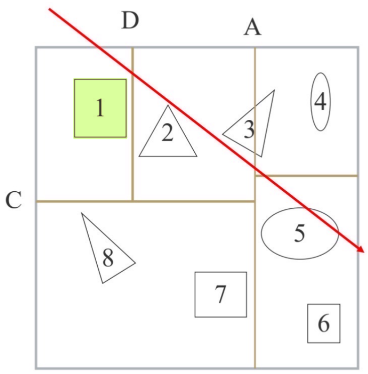


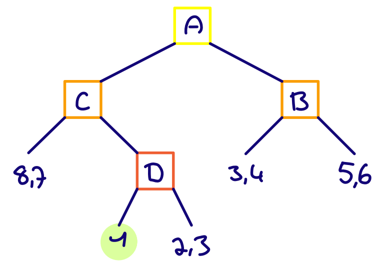
Process	Stack
D	B

both objects in D could be intersected

-> check the closer one first, stack the others

4.



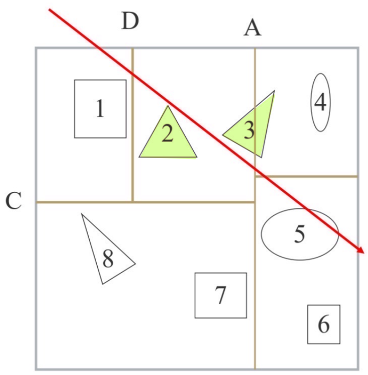


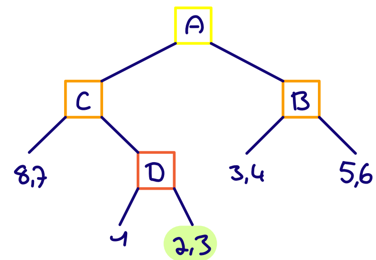
Process	Stack
1	B
	2,3

object intersection check is negative

→ pop next one from stack

5.



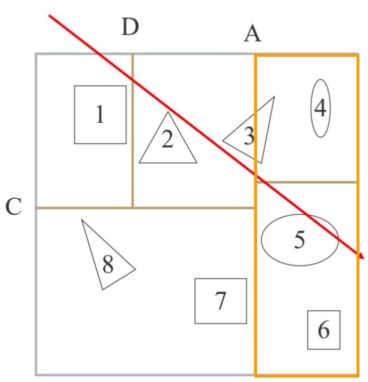


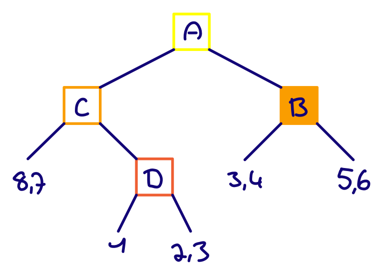
Process	Stack
2,3	B

object intersection check is negative

→ pop root child node B from stack

6.

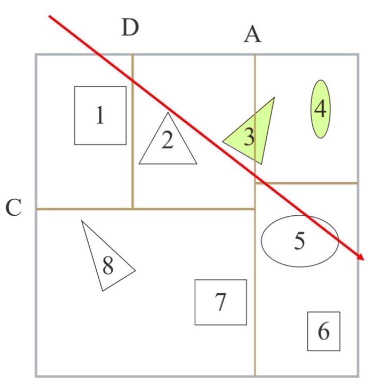


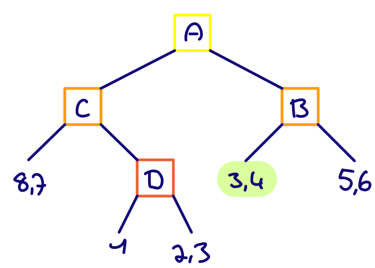


Process	Stack
B	

intersection with plane B means both children are hit by the ray → first check closer objects

7.



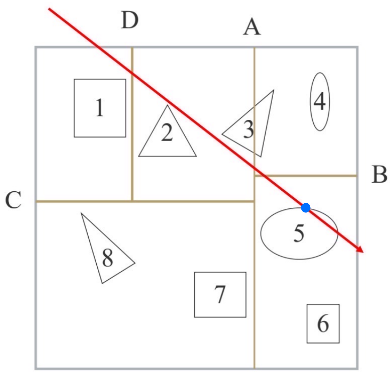


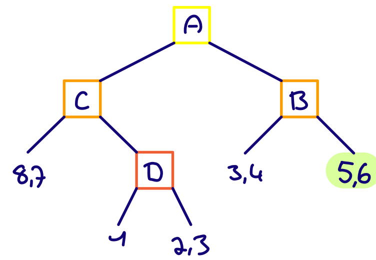
Process	Stack
3,4	5,6

no intersection detected

→ pop other objects from stack

8.





Process	Stack
5,6	

intersection with object 5 detected

→ terminate traversal

Surface Area Heuristic (SAH)

build good kd-Trees by accounting probability and costs

- probability of a ray hits the cell is proportional to its size
- cost of the cell is given by triangle count → more triangles = higher costs

$$C(\text{cell}) = C\_\text{trav} + p(\text{hit}\,L) \cdot C(L) + p(\text{hit}\,R) \cdot C(R)$$

=> produces large chunks of empty space

=> automatically and rapidly isolates complexity

Large Scenes

what if spatial data does not fit into memory?

- Lazy build: build subtree only when it's needed (potential intersection)

  \+ no memory wasted
  \- can be slow, same subtree has to be build over and over again

- lazy build with caching: each needed subtree will be stored when it's built for the first time (if memory is full, some subtrees will be deleted)

  \- lots of tests
  \- inefficient deletion of subtrees

- Multi-Level-Hierarchy: first levels of tree are fixed, lazy build on lower levels

  \+ often visited nodes (upper levels) don't have to be rebuild

→ Ray-Reordering: reminds frequently hit elements and keeps them in memory

Dynamic Scenes

in moving scenes you have to update your spacial structure fast

→ combine kd-Tress with BVH in lower levels to bounding kd-Trees