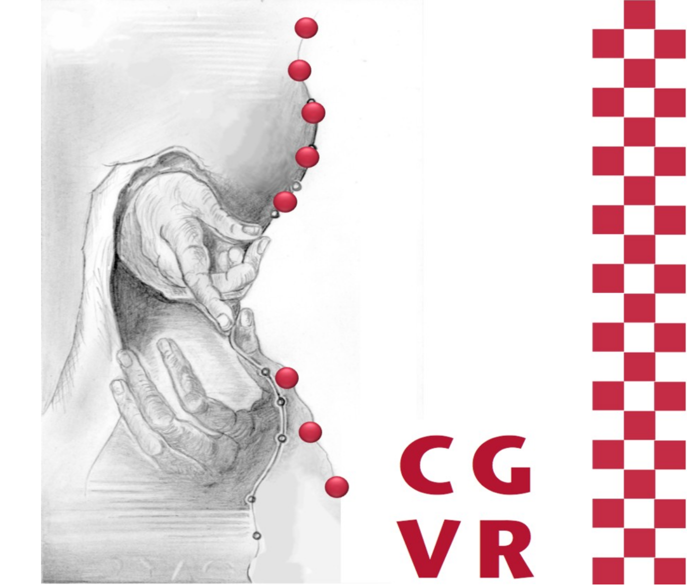



# Advanced Computer Graphics

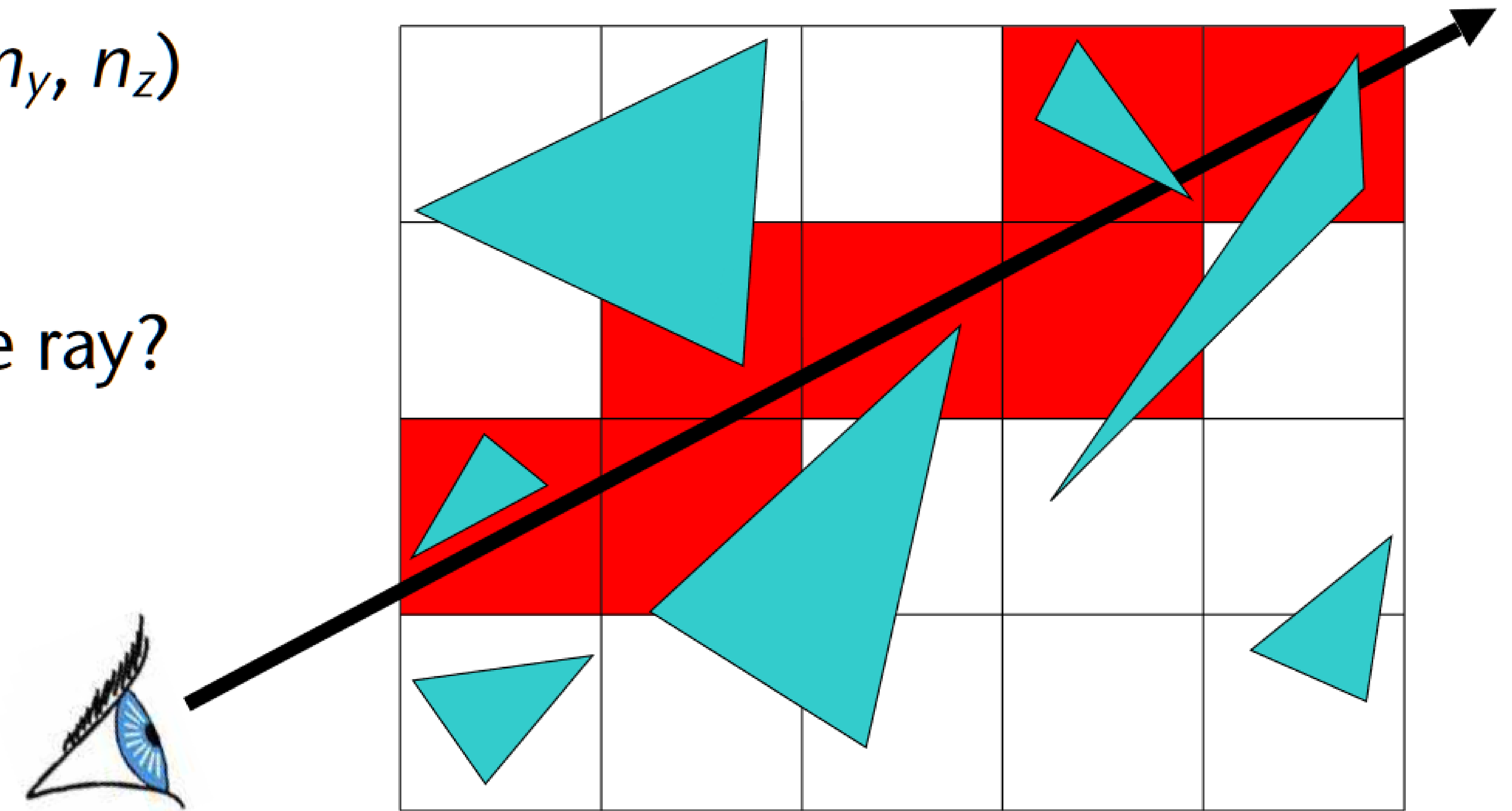
Tutorium 3

# Assignment 3

- Acceleration Data Structures
  - Uniform Grid
  - Octree
  - AABB-Tree
  - Sphere-Tree
  - kD-Tree

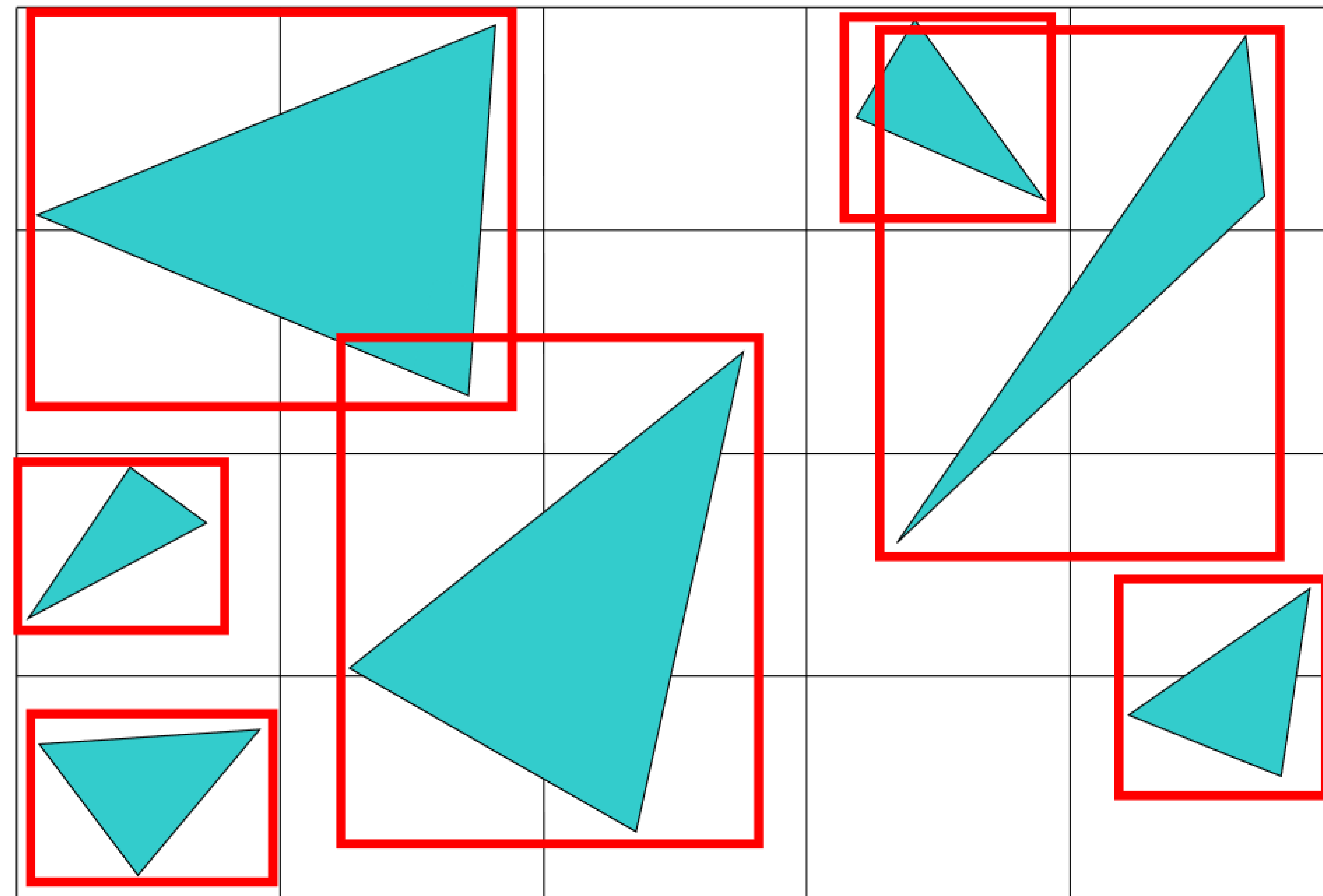
# Uniform Grid

- Approach: partition scene into 3D grid; insert objects in cells; visit all cells along the ray; intersect ray with objects stored in cell
- Construction of the grid:
  - Calculate BBox of the scene
  - Choose a (suitable) grid resolution ($n_x$, $n_y$, $n_z$)
- For each cell intersected by the ray:
  - Is any of the objects in the cell hit by the ray?
  - Yes: return closest hit
  - No: proceed to next cell



# Uniform Grid – Generation

- For each cell store all objects intersecting that cell in a list with that cell ⟶ "insert objects in cells"
  - Each cell has a list that contains pointers to objects
- How to insert objects: use bbox of objects
  - Exact intersection tests are not worth the effort
- Note: most objects are inserted in many cells cells (not just one)



# Uniform Grid – Generation

- Too many cells → slow traversal, heavy memory usage, bad cache utilization
- Too few cells → too many objects/triangles per cell
- Good rule of thumb: choose the size of the cells such that the edge length is about the average size of the polygons/objs (e.g., measured by their bbox)
- If you don't know it (or it's too time-consuming to compute), then choose $n_x, n_y, n_z = \sqrt[3]{N}$ , $N$ = # objects
  - More precisely: resolution = $\lambda\sqrt[3]{N}$ , where $\lambda$ depends on time for intersection & time for step in grid (tune at the end)
  - Consequence: #cells = space complexity $\in O(N)$ [good]
- Another good rule of thumb: try to make the cells cuboid-like

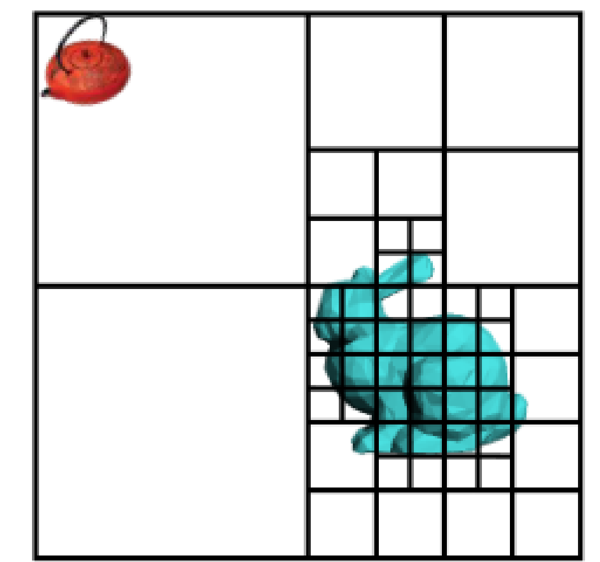
# Uniform Grid – Traversal

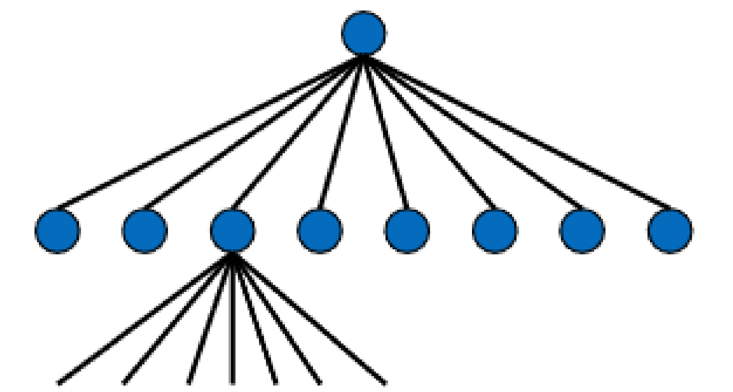
- http://www.cse.yorku.ca/~amana/research/grid.pdf

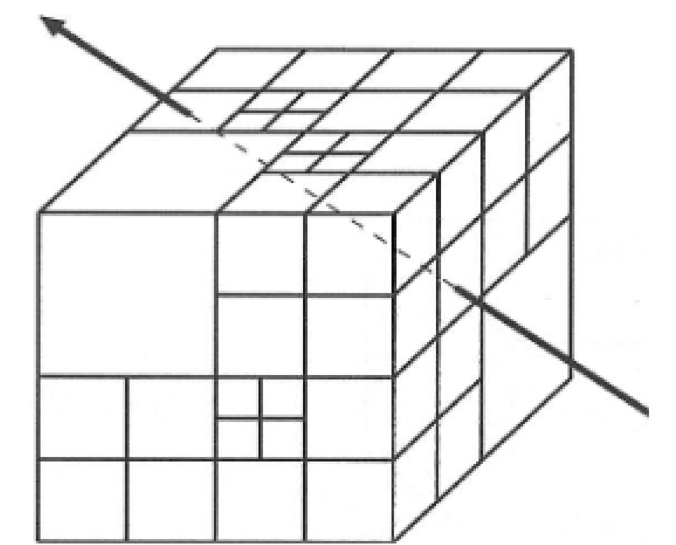
```
list= NIL;
do  {
    if(tMaxX < tMaxY) {
        if(tMaxX < tMaxZ) {
            X= X + stepX;
            if(X == justOutX)
                return(NIL); /* outside grid */
            tMaxX= tMaxX + tDeltaX;
        } else  {
            Z= Z + stepZ;
            if(Z == justOutZ)
                return(NIL);
            tMaxZ= tMaxZ + tDeltaZ;
        }
    } else  {
        if(tMaxY < tMaxZ) {
            Y= Y + stepY;
            if(Y == justOutY)
                return(NIL);
            tMaxY= tMaxY + tDeltaY;
        } else  {
            Z= Z + stepZ;
            if(Z == justOutZ)
                return(NIL);
            tMaxZ= tMaxZ + tDeltaZ;
        }
    }
    list= ObjectList[X][Y][Z];
} while(list == NIL);
return(list);
```

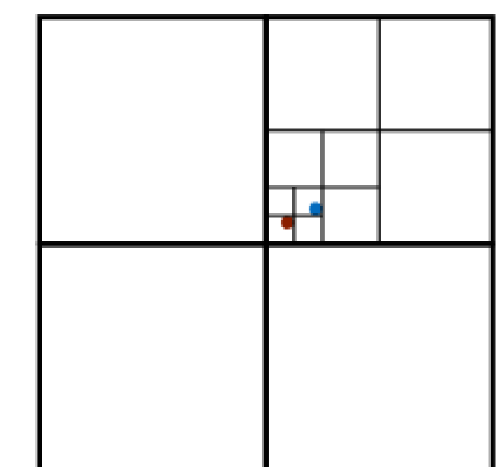
# Octree

- Construction:
  - Start with the bbox of the whole scene
  - Subdivide a cell into 8 equal sub-cells
  - Stopping criterion: the number of objects, and maximal depth
- Advantage: we can make big strides through large empty spaces
- Disadvantages:
  - Relatively complex ray traversal algorithm
  - Sometimes, a lot of levels are needed to discriminate objects







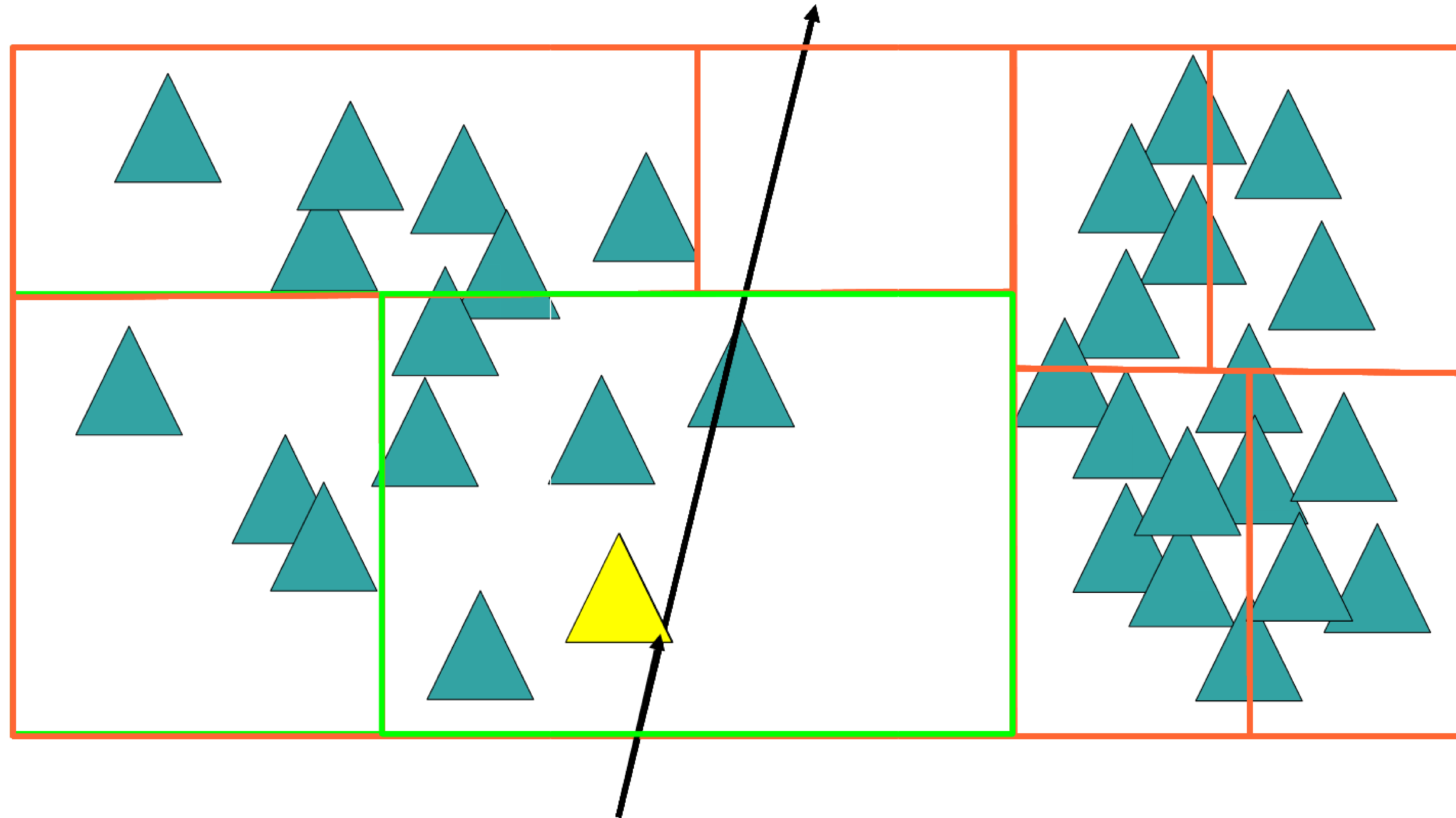


# Octree – Traversal

- https://bertolami.com/files/octrees.pdf

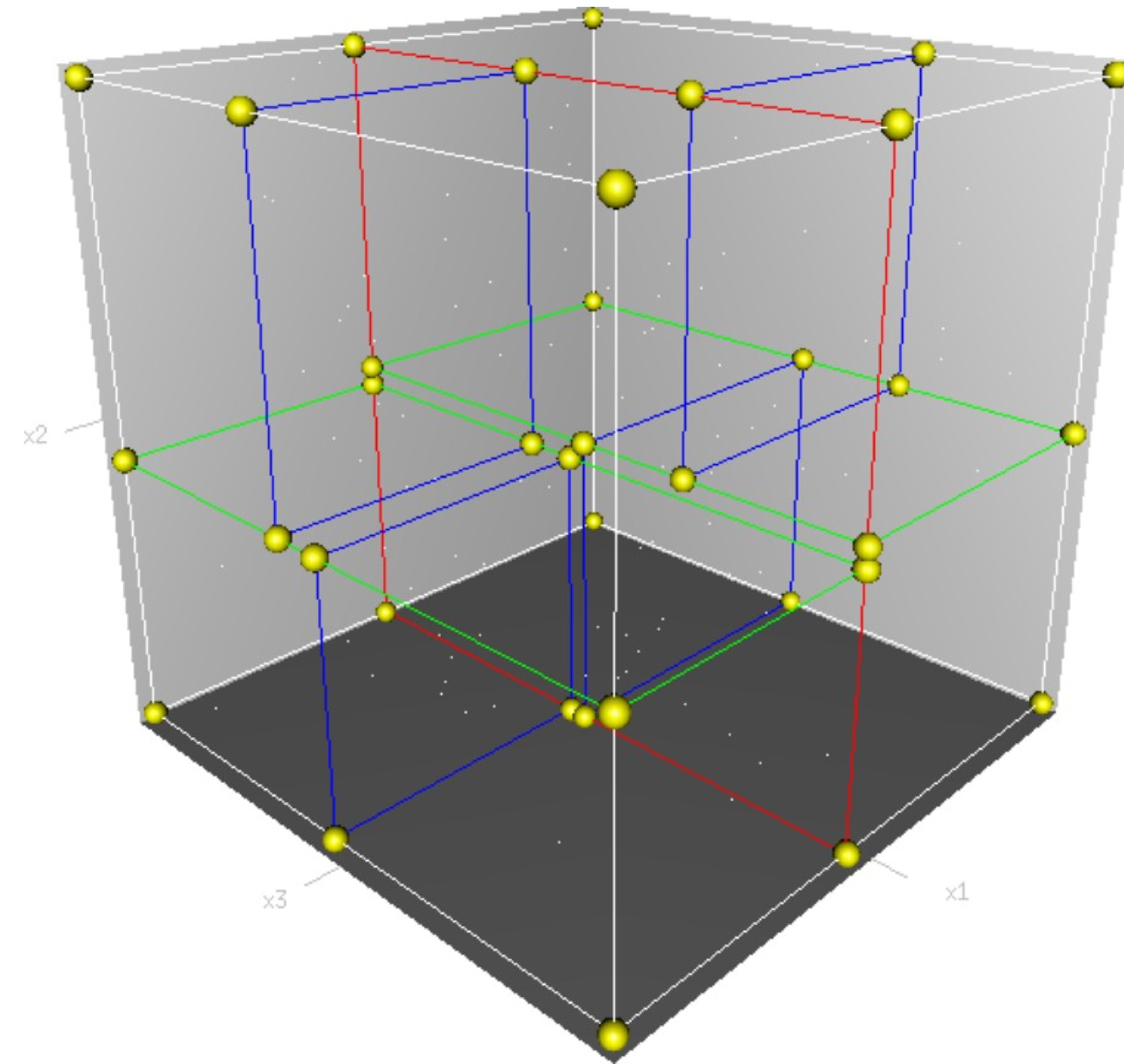
```
void Octree::Trace(ray, collision) {
  root_node.Trace(ray, collision)
}

void OctreeNode::Trace(ray, collision) {
  if ray does not intersect node bounds:
    return

  if a collision has already been detected and it’s closer
      than the ray entry point into this node:
    return

  if node is a leaf:
    for each polygon in node:
      if ray intersects polygon:
        if intersection is closer to ray origin than collision:
          update collision with current intersection
  else:
    for each child_node:
       child_node.Trace(ray, collision)
}
```

# kD-Tree



[Slide courtesy Martin Eisemann]

# kD-Tree



# kD-Tree – Generation

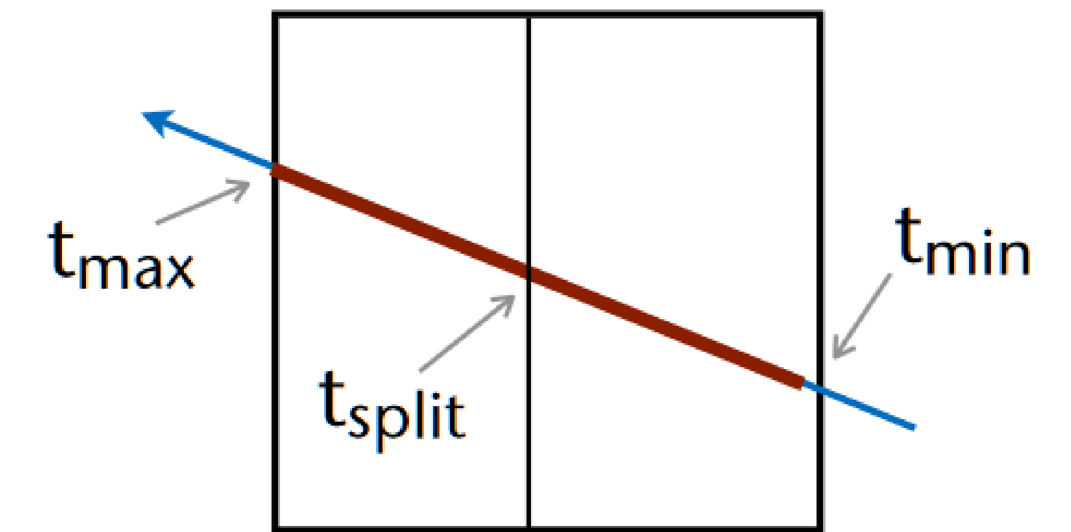
- **Given:**
  - An axis-aligned bbox enclosing part of the scene (cell / node of the kd-tree)
    - At the root, the box encloses the whole universe
  - List of the geometry primitives contained in this cell
- **The procedure (top down):**
  1. Choose an axis-aligned plane, with which to split the cell
  2. Distribute the geometry among the two children
     - Some polygons need to be assigned to both children
  3. Do a recursion, until the stopping criterion is met
- Remark: each cell (whether leaf or inner node) defines a box, without the box ever being explicitly stored anywhere
  - (Theoretically, such boxes could be half-open boxes, if we start at the root with the complete space)

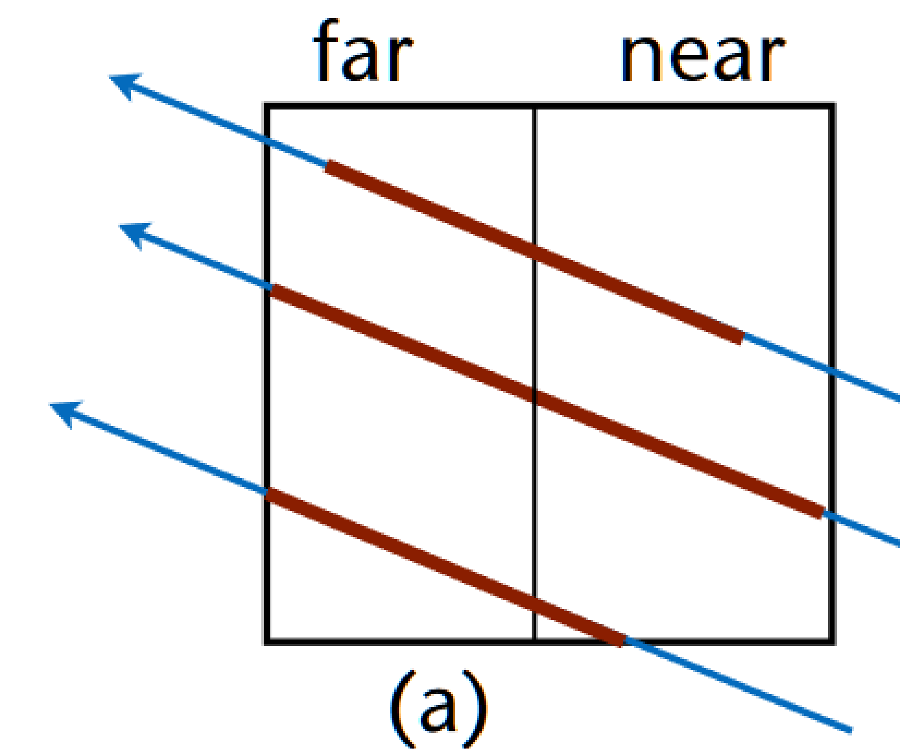
# kD-Tree – Generation

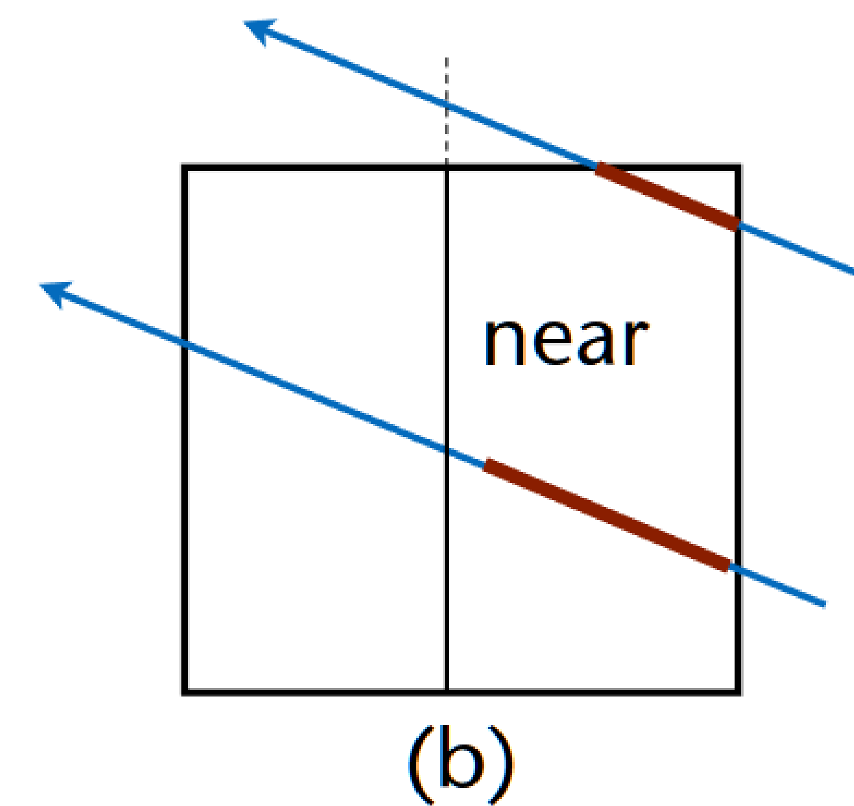
- **Naïve selection of the splitting plane:**
  - Splitting axis:
    - Round Robin (x, y, z, x, y, z, ...)
    - Best: split along the longest axis of the node's region (not bbox of its contents!)
  - Split position (along the splitting axis):
    - Middle of the cell
    - Median of the geometry
- **In case the intended application is known: use a cost function!**
  - Choose a splitting plane such that the *expected* costs of a ray test are minimal
  - Try all 3 axes: search for the minimum along each axis
  - Choose axis / split position with the smallest minimum

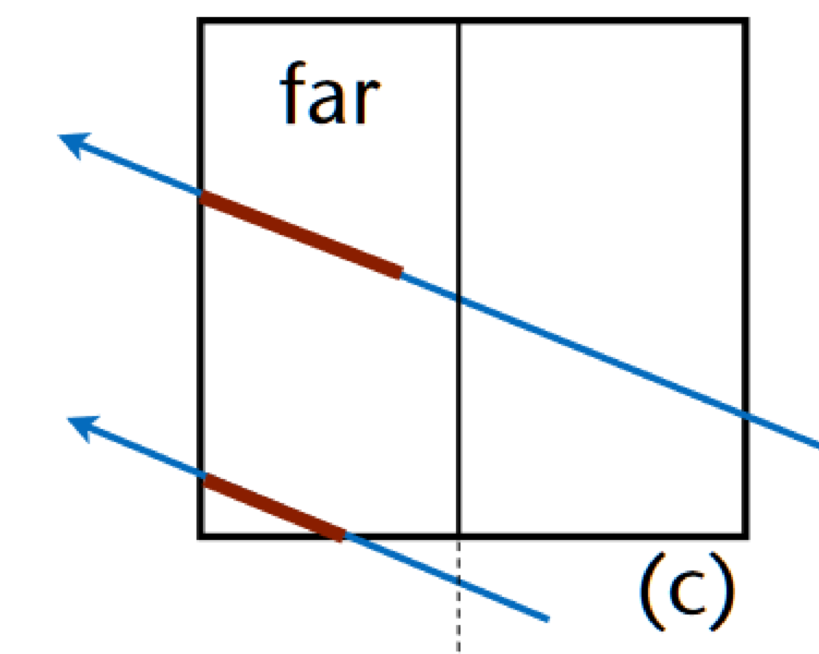
# kD-Tree – Traversal

- Intersect ray with root-box → $t_{min}$, $t_{max}$
- Recursion:
  - Update [$t_{min}$, $t_{max}$] throughout tree traversal
  - Intersect ray with splitting plane → $t_{split}$
  - We need to consider the following three cases:
    a) First traverse the "near", then the "far" subtree
    b) Only traverse the "near" subtree
    c) Only traverse the "far" subtree



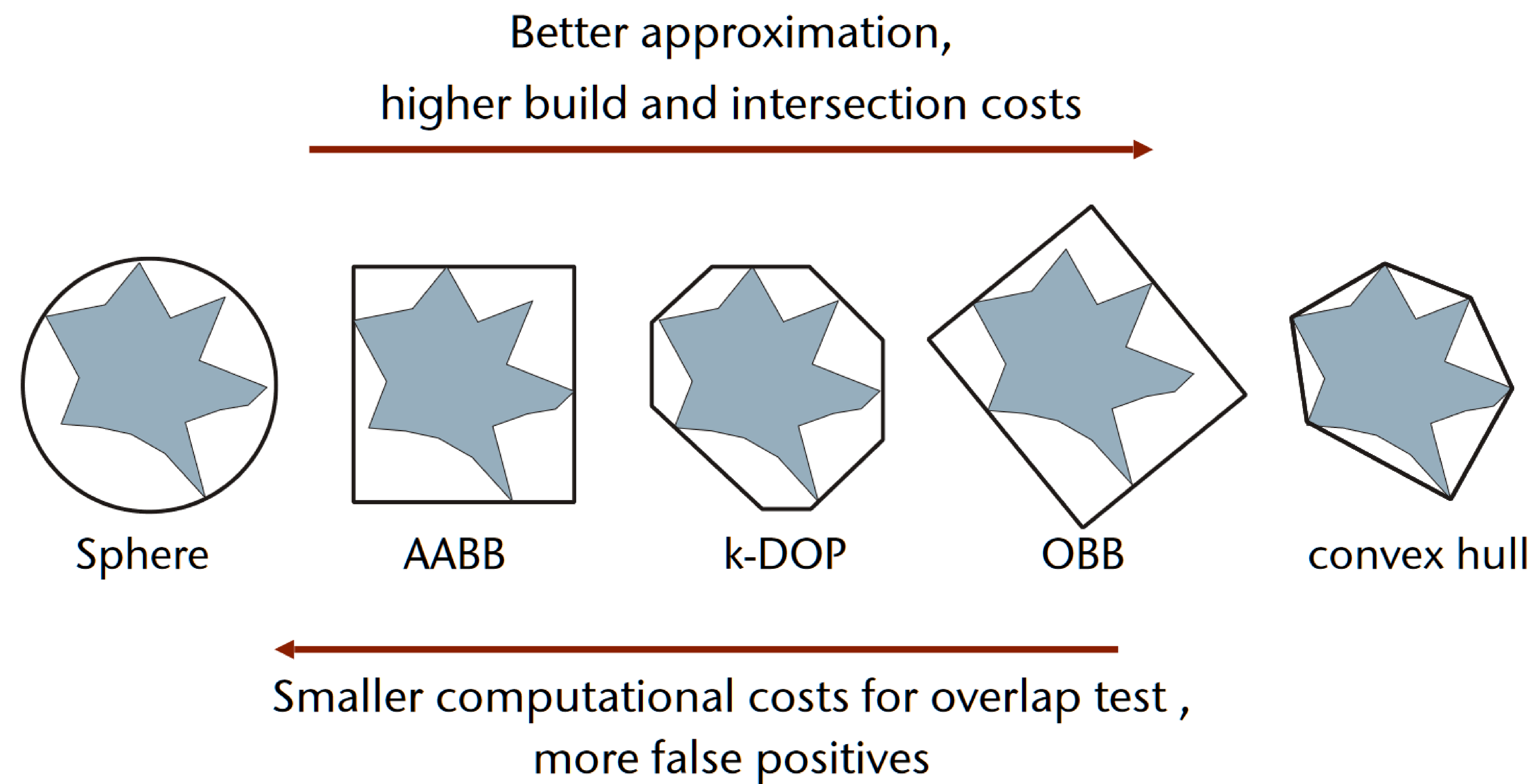



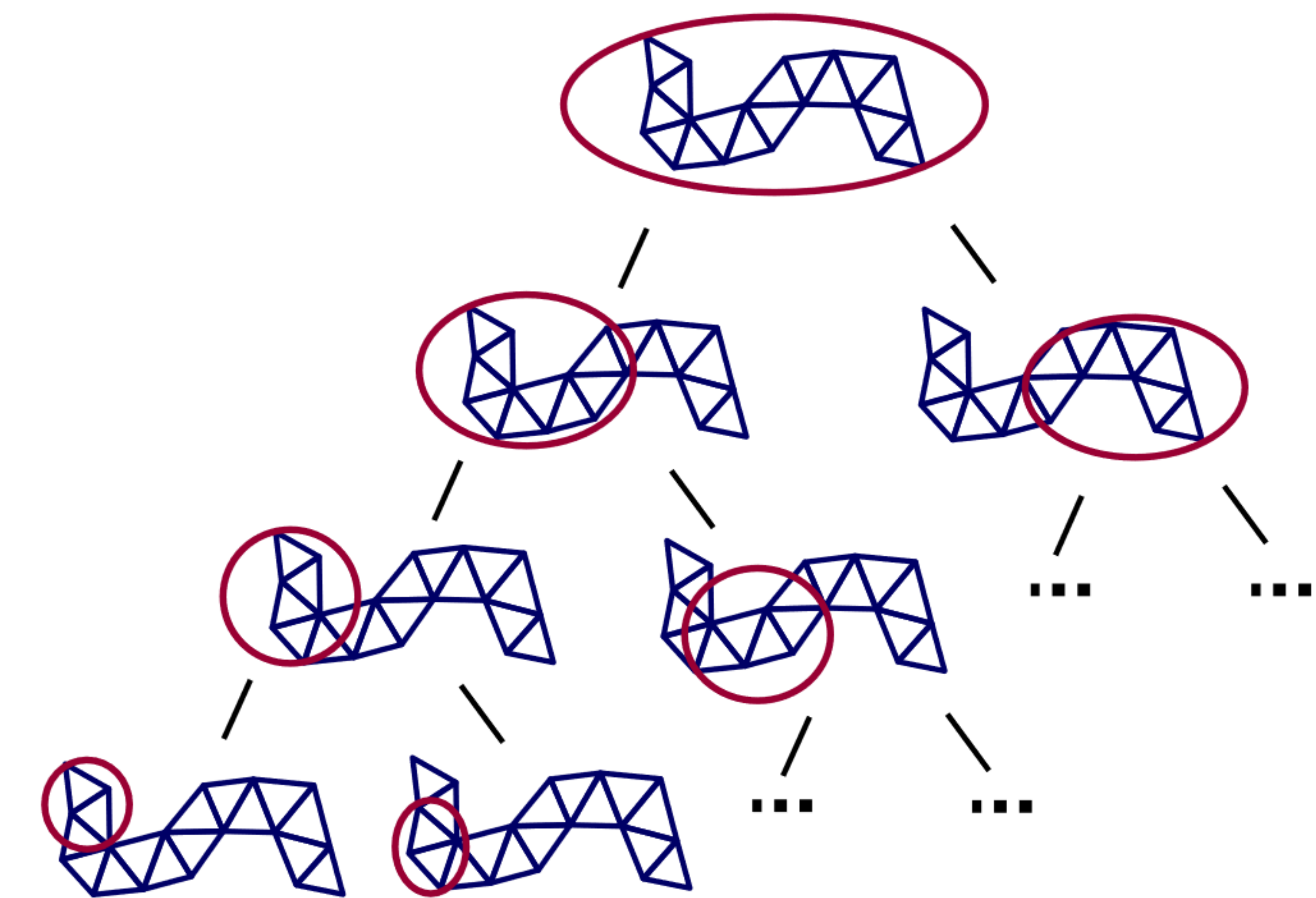



# kD-Tree – Traversal

```
traverse( Ray r, Node n, float t_min, float t_max ):
if n is leaf:
  intersect r with each primitive in object list,
    discard those farther away than t_max
  return object with closest intersection point (if any)
t_split = signed distance along r to splitting plane of n
near = child of n containing origin of r
far  = the "other" child of n
if t_split > t_max:
  return traverse( r, near, t_min, t_max )          // (b)
else if t_split < t_min:
  return traverse( r, far, t_min, t_max )           // (c)
else:                                               // (a)
  t_hit = traverse( r, near, t_min, t_split )
  if t_hit < t_split:
    return t_hit                                    // early exit
  return traverse( r, far, t_split, t_max )
```

# BVH (AABB-Tree, Sphere-Tree)



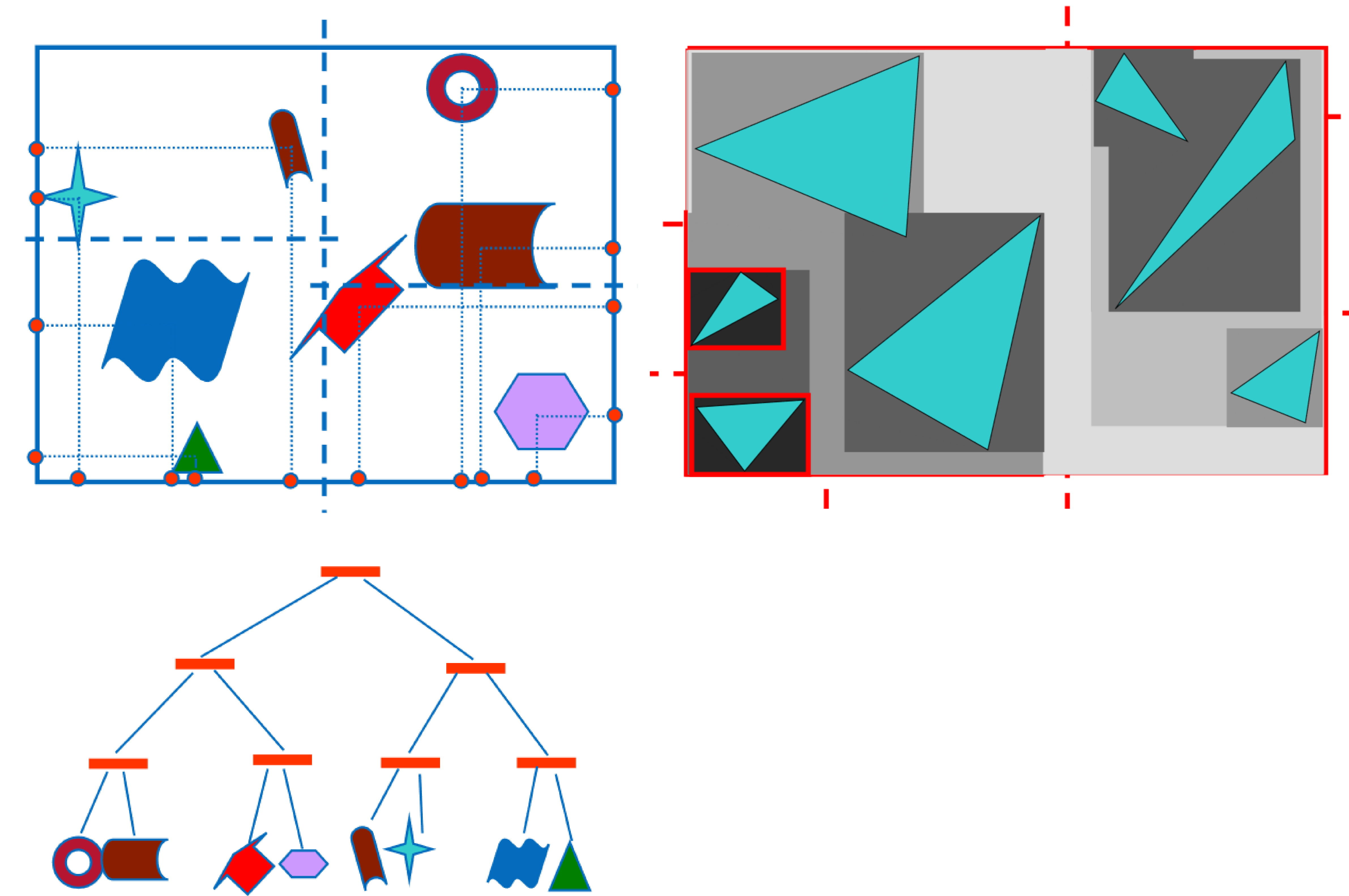
# BVH – Generation

- There are many possible principles:

1. Given by modeling process (e.g., in form of a scene graph)
2. Bottom-up:
   - Recursively combine objects/BV's and enclose in (larger) BV
   - Problem: how to choose the objects/BV's to be combined?
3. Iterative Insert:
   - Start with empty tree, iteratively add polygons, let each polygon "sift" through the tree [Goldsmith/Salmon]
4. **Top-down**:
   - Partition the set of primitives recursively
   - Problem: how to partition the set?

# BVH – Generation – Median Cut

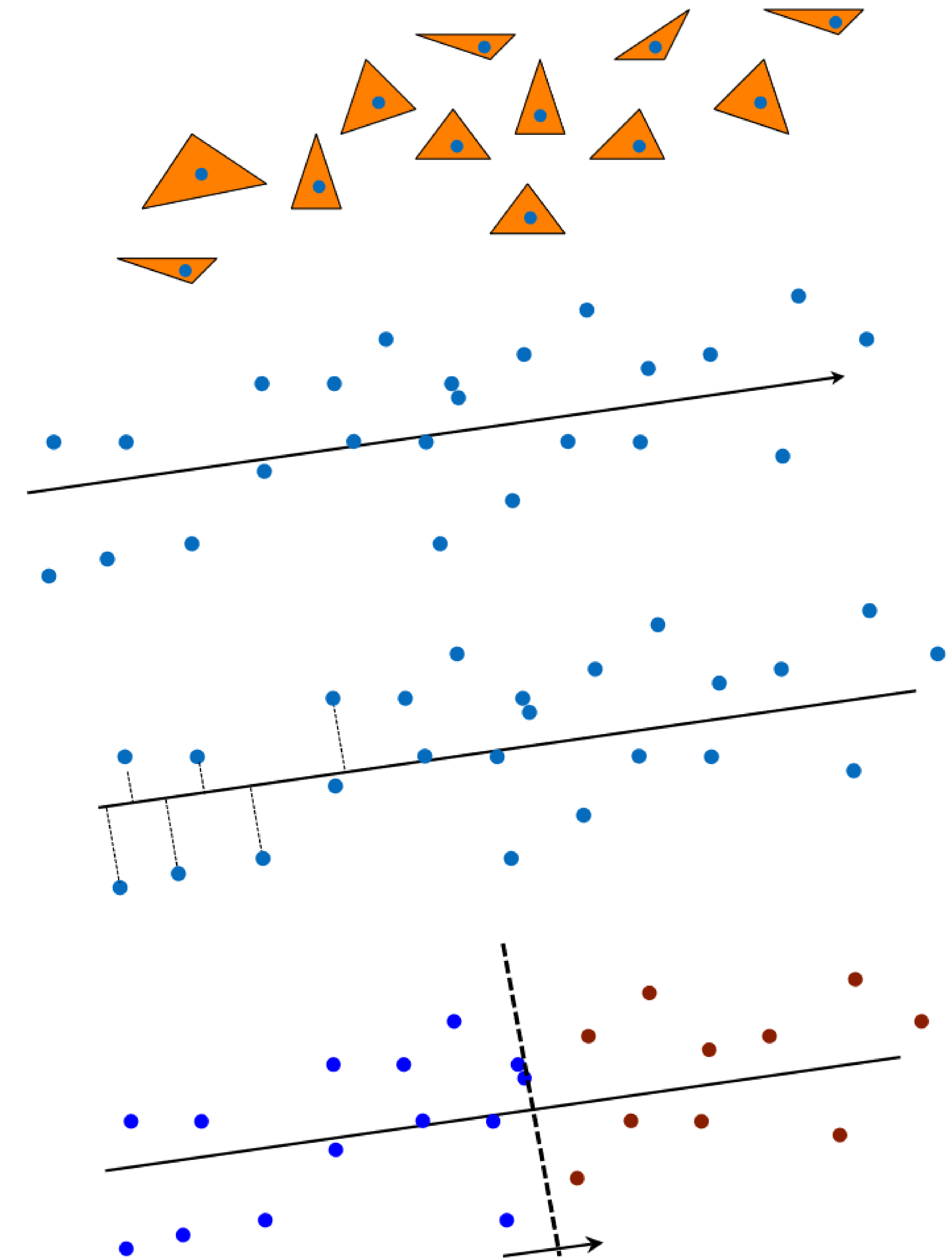
1. Construct elementary BVs around all objects
2. Sort all objects according to their "center" along the x-axis
3. Partition the scene along the *median* on the x-axis; assign half of the objects to the left and the right sub-tree, resp.
   1. Variant: cyclically choose a different axis on each level
   2. Variant: choose the axis with the longest extent
4. Repeat 1-3 recursively
   - Terminate, when a node contains less than *n* objects



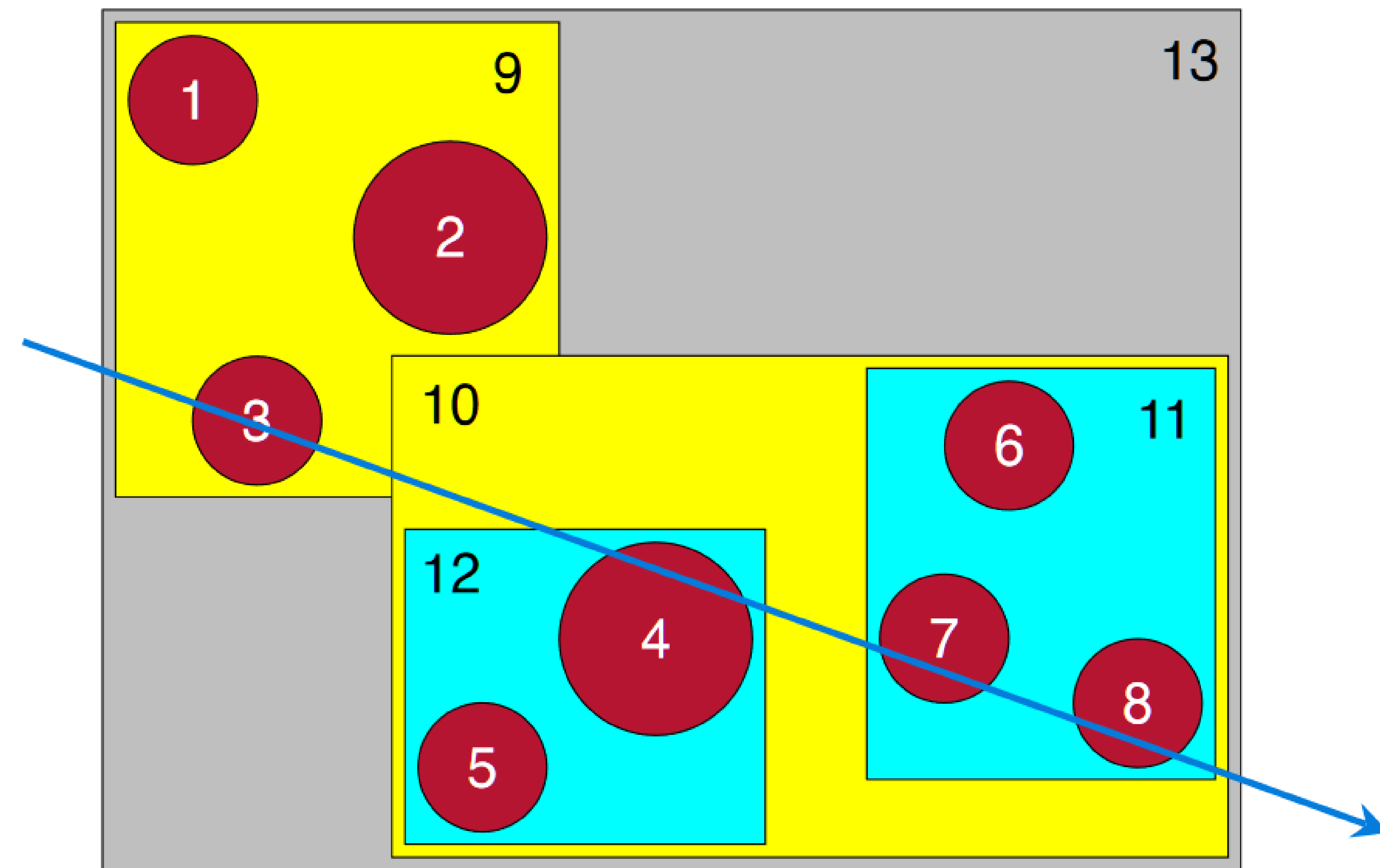
# BVH – Generation – Principle Comp. Analysis

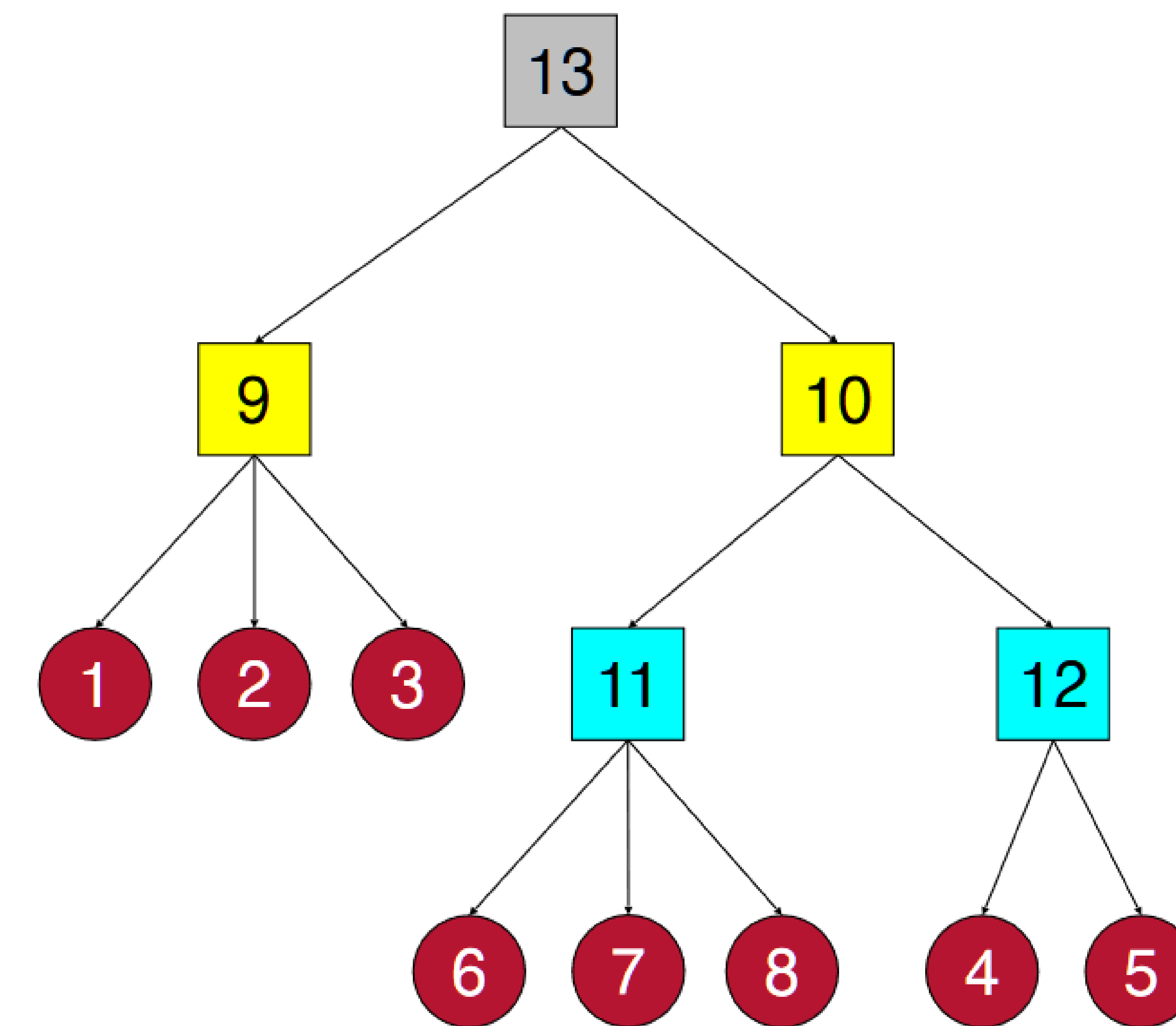


1. Represent all polygons by their midpoints
2. Calculate axis of largest extent (using PCA)
3. Project all midpoints onto that axis and sort
4. Search minimum of $C(B)$ by plane sweep



# BVH – Traversal



# BVH – Traversal

- Problem: the order by which nodes are visited with pure depth-first search (DFS) depends *only* on the topology of the tree
- Better: consider the spatial layout of the BV's, too
- Criterion: distance between origin of ray and intersection with BV (= *lower bound on distance of enclosed primitives*)
- Consequence: should not use simple recursion / stack any more
- Use **priority queue**

# BVH – Traversal

- Maintain a p-queue
  - Contains all BVs (= BVH nodes) that still need to be visited
  - Sorted by their distance from ray origin (along ray)

```
Pqueue q ← init with root
closest_hit = ∞
while q not empty:
  node ← extract front from q          // = nearest BV
  if dist(node) <= closest_hit:        // else: skip this subtree
    if node is leaf:
      intersect ray with all polygons in node
      update closest_hit, if any polygon is closer
    else                               // inner node
      forall children of node:
        if ray intersects child:
          insert child in q with its distance
```

# BVH – Traversal

- **Warning:** the closest ray-BV intersection and the closest ray-primitive intersection can occur in different BV's!