Computer Graphics

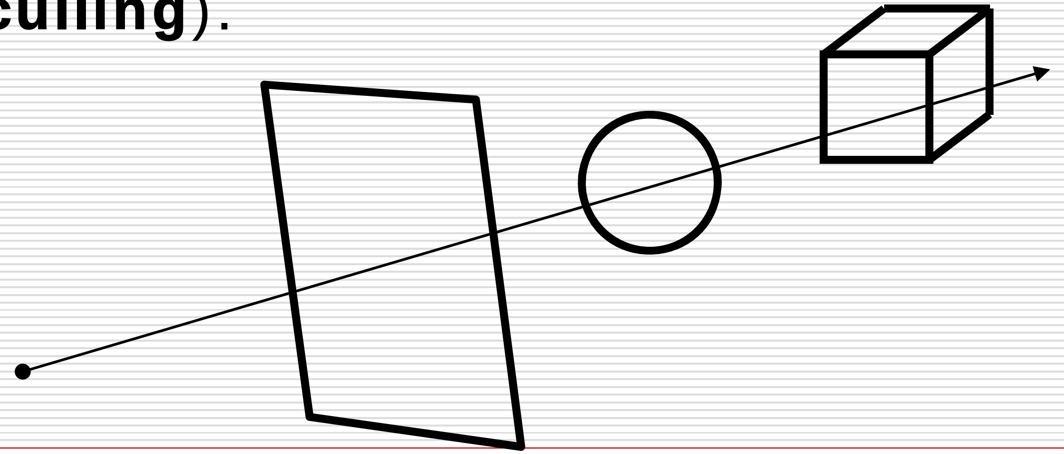
Bing-Yu Chen
National Taiwan University
The University of Tokyo

Hidden-Surface Removal

- Back-Face Culling
- The Depth-Sort Algorithm
- Binary Space-Partitioning Trees
- The z-Buffer Algorithm
- Visible-Surface Ray Tracing (Ray Casting)
- Space Subdivision Approaches

Hidden-Surface Removal =Visible-Surface Determination

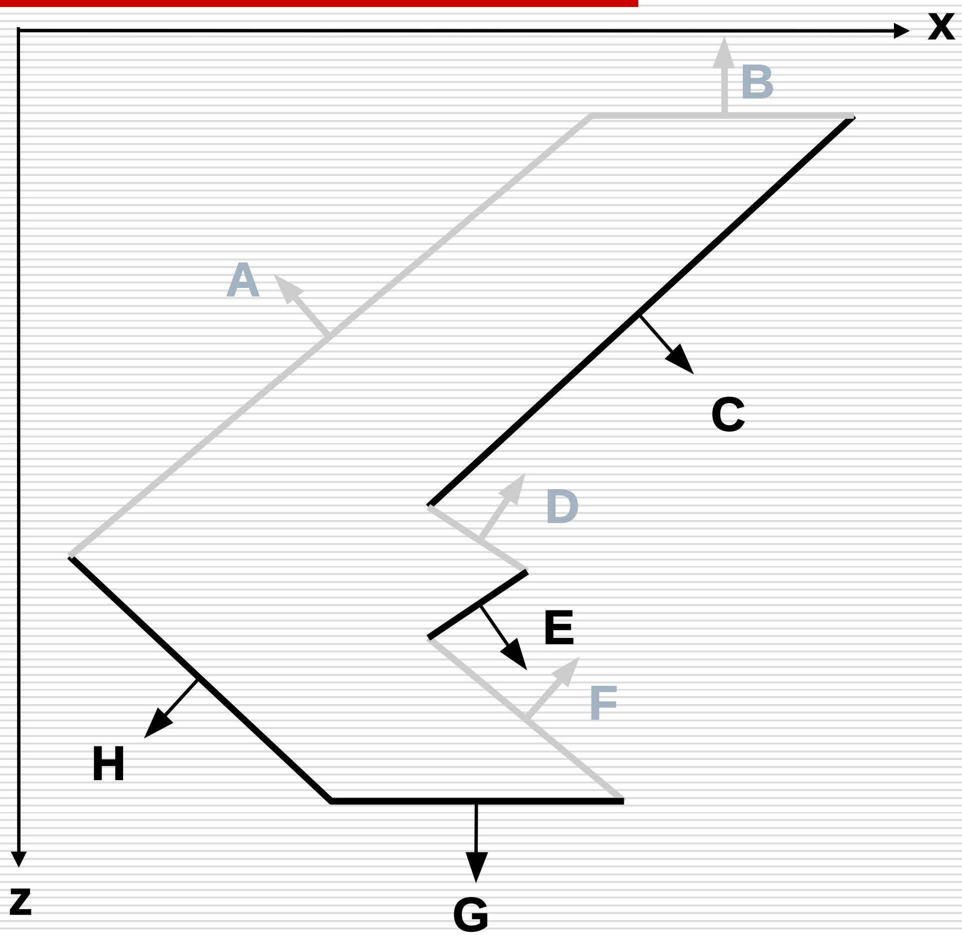
- Determining what to render at each pixel.
- A point is visible if there exists a direct line-of-sight to it, unobstructed by another other objects (**visible surface determination**).
- Moreover, some objects may be invisible because there are behind the camera, outside of the field-of-view, too far away (**clipping**) or back faced (**back-face culling**).



Hidden Surfaces: why care?

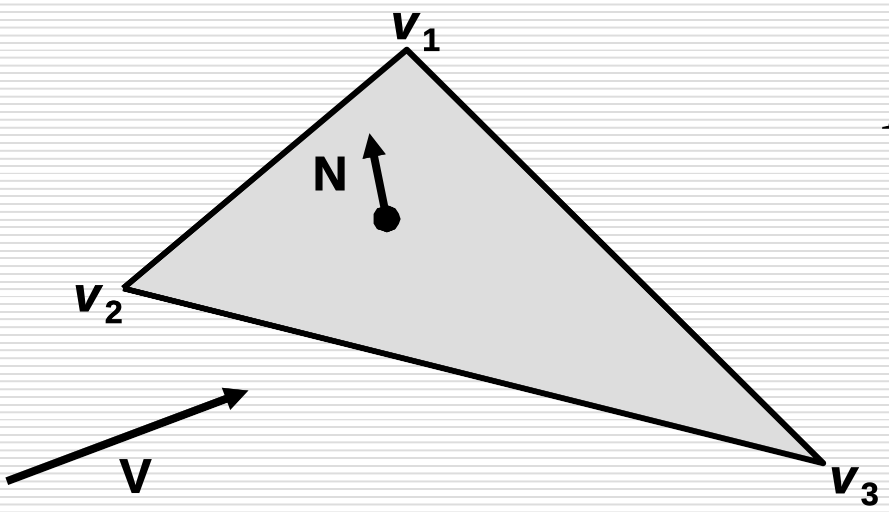
- **Occlusion**: Closer (opaque) objects along same viewing ray obscure more distant ones.
- Reasons for removal
  - Efficiency: As with clipping, avoid wasting work on invisible objects.
  - Correctness: The image will look wrong if we don’t model occlusion properly.

Back-Face Culling = Front Facing



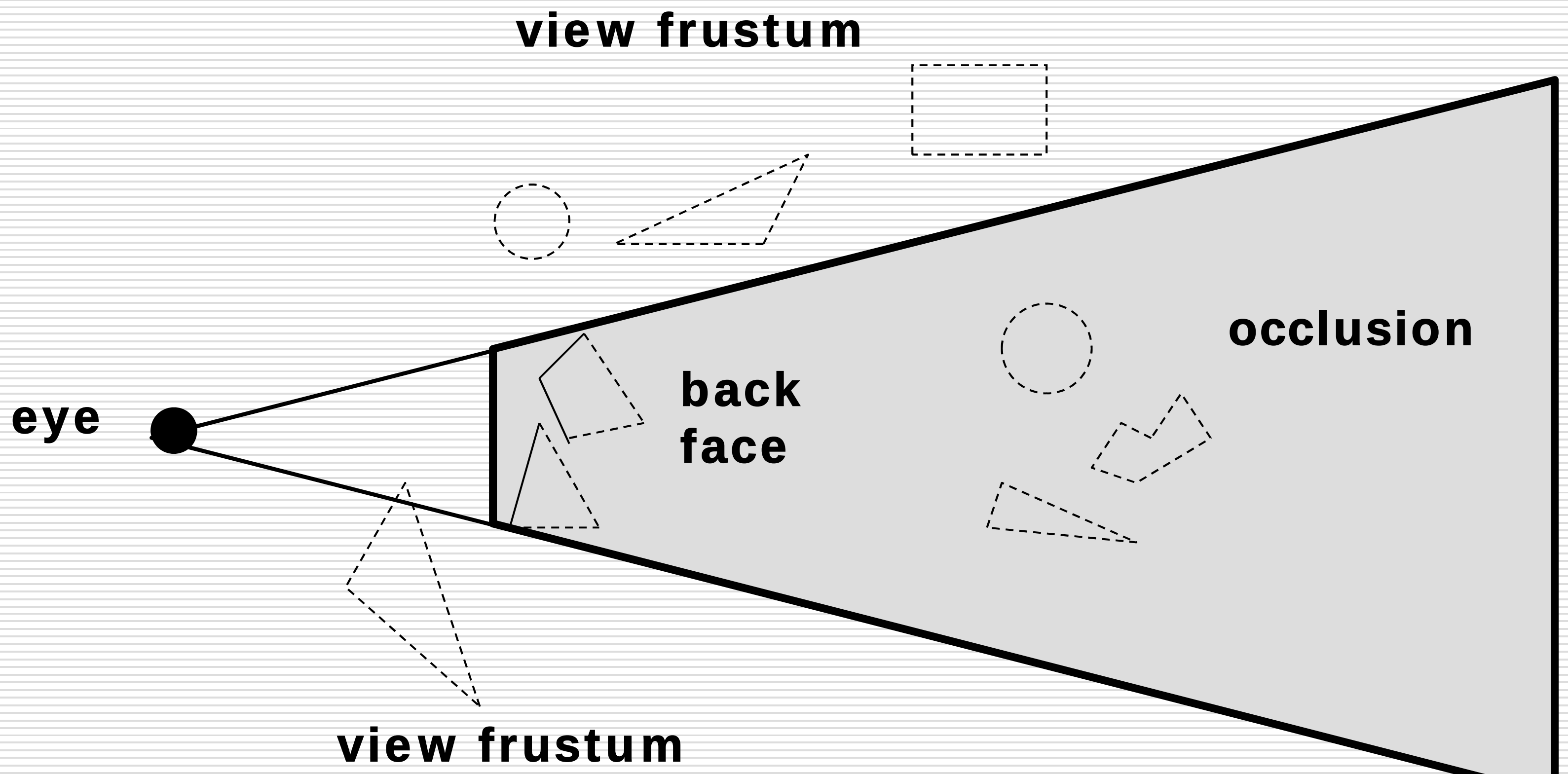
Back-Face Culling = Front Facing

- use cross-product to get the normal of the face (not the actual normal)
- use inner-product to check the facing



$$N = (v_2 - v_1) \times (v_3 - v_1)$$

Clipping (View Frustum Culling)

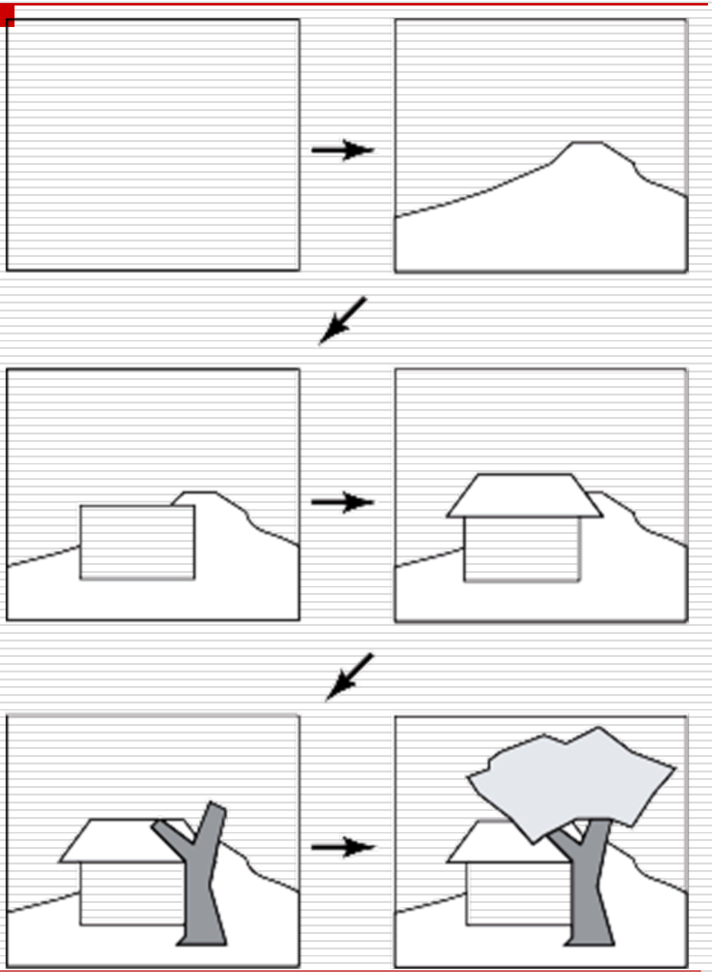


List-Priority Algorithms

- The Painter's Algorithm
- The Depth-Sort Algorithm
- Binary Space-Partitioning Trees

The Painter's Algorithm

- Draw primitives from back to front need for depth comparisons.



The Painter's Algorithm

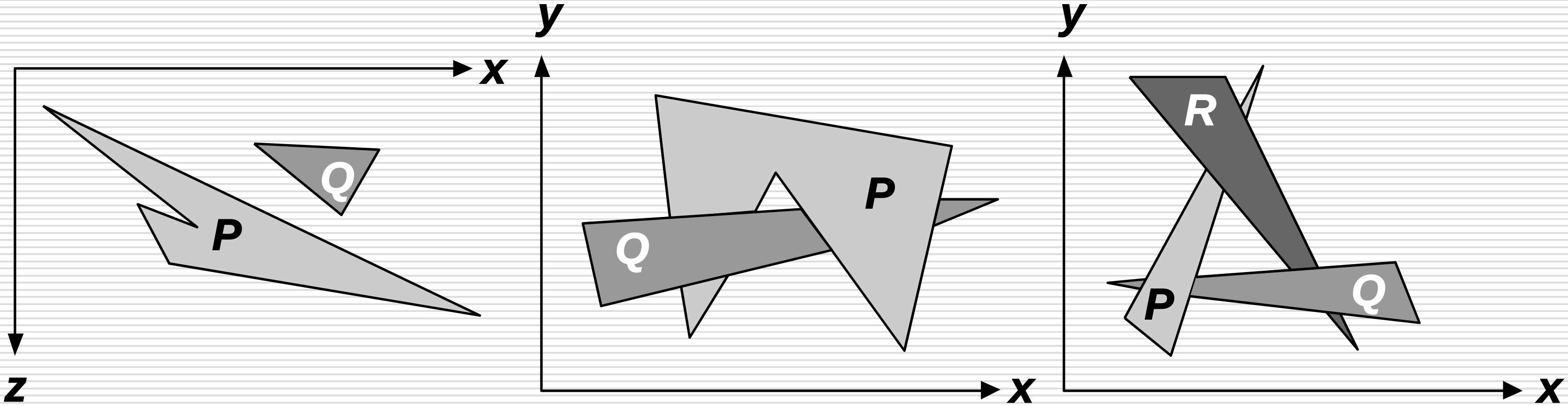
- for the planes with constant *z*
- not for real 3D, just for 2½D

- **sort** all polygons according to the smallest (farthest) *z* coordinate of each
- **scan convert** each polygon in ascending order of smallest *z* coordinate (i.e., back to front)

The Depth-Sort Algorithm

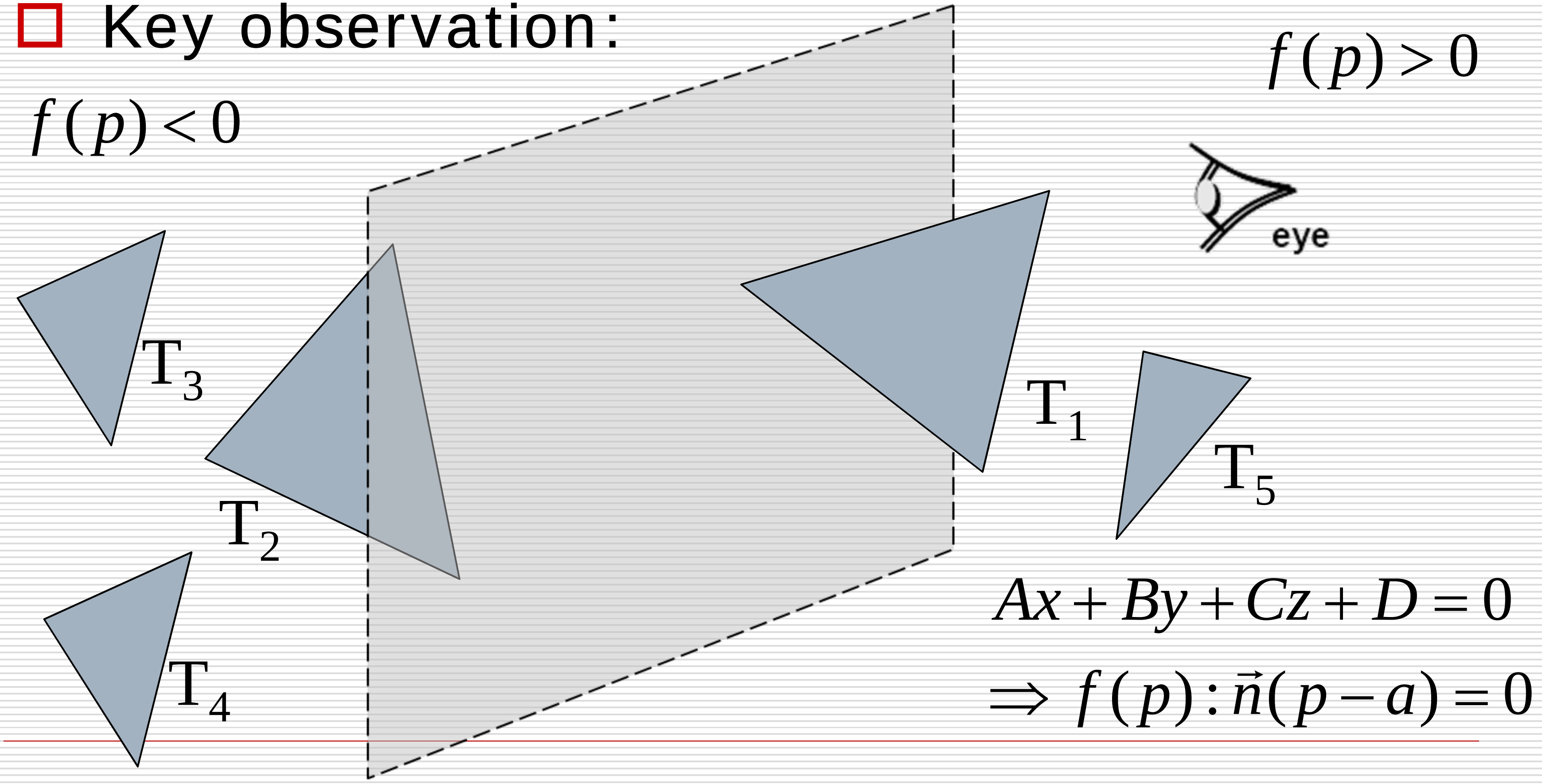
- **sort** all polygons according to the smallest (farthest) *z* coordinate of each
- resolve any ambiguities that sorting may cause when the polygons' *z* extents **overlap**, **splitting** polygons if necessary
- **scan convert** each polygon in ascending order of smallest *z* coordinate (i.e., back to front)

Overlap Cases

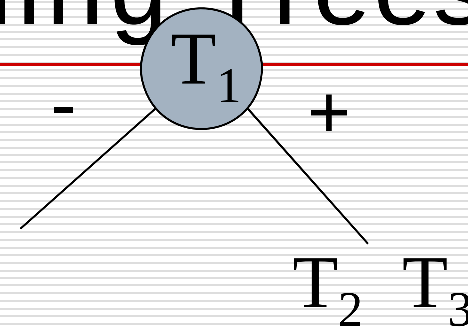


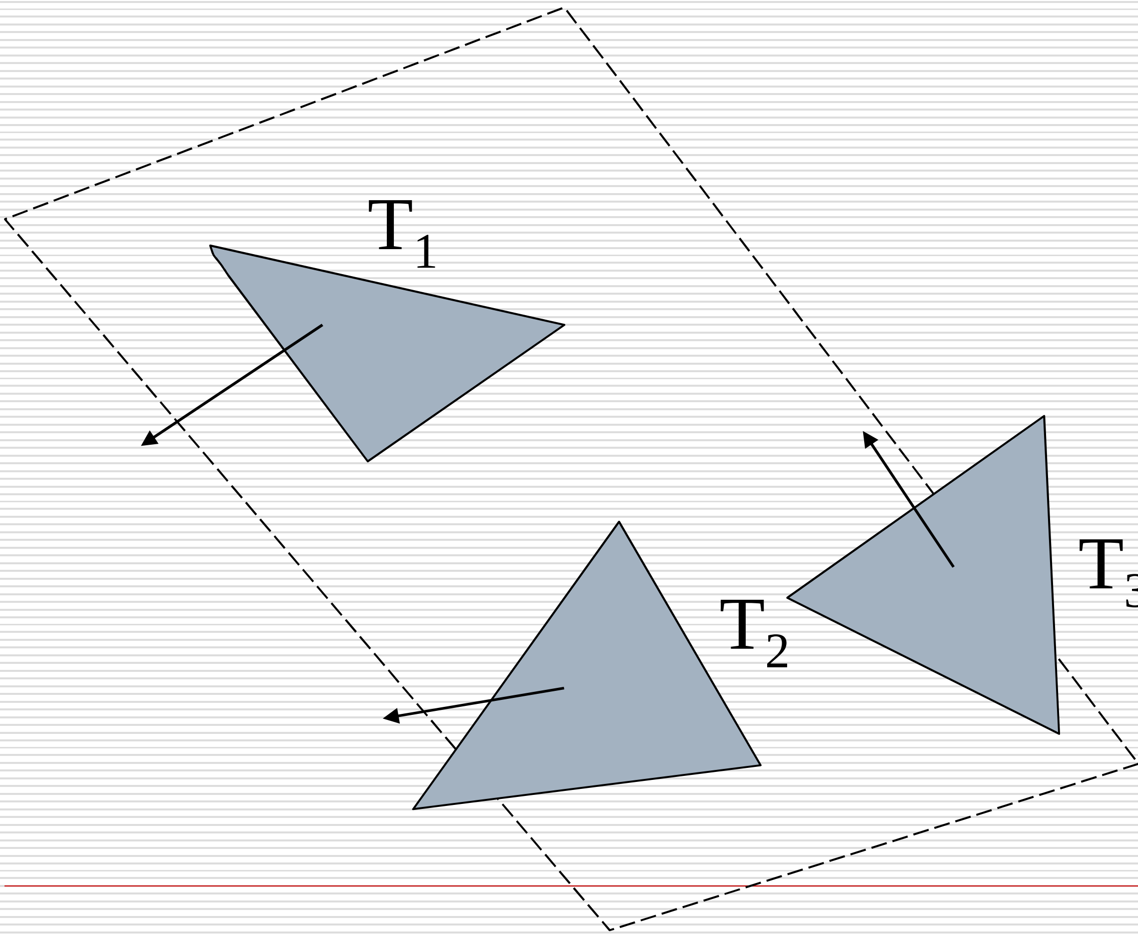
Binary **S**pace-**P**artitioning Trees

- An improved painter's algorithm
- Key observation:

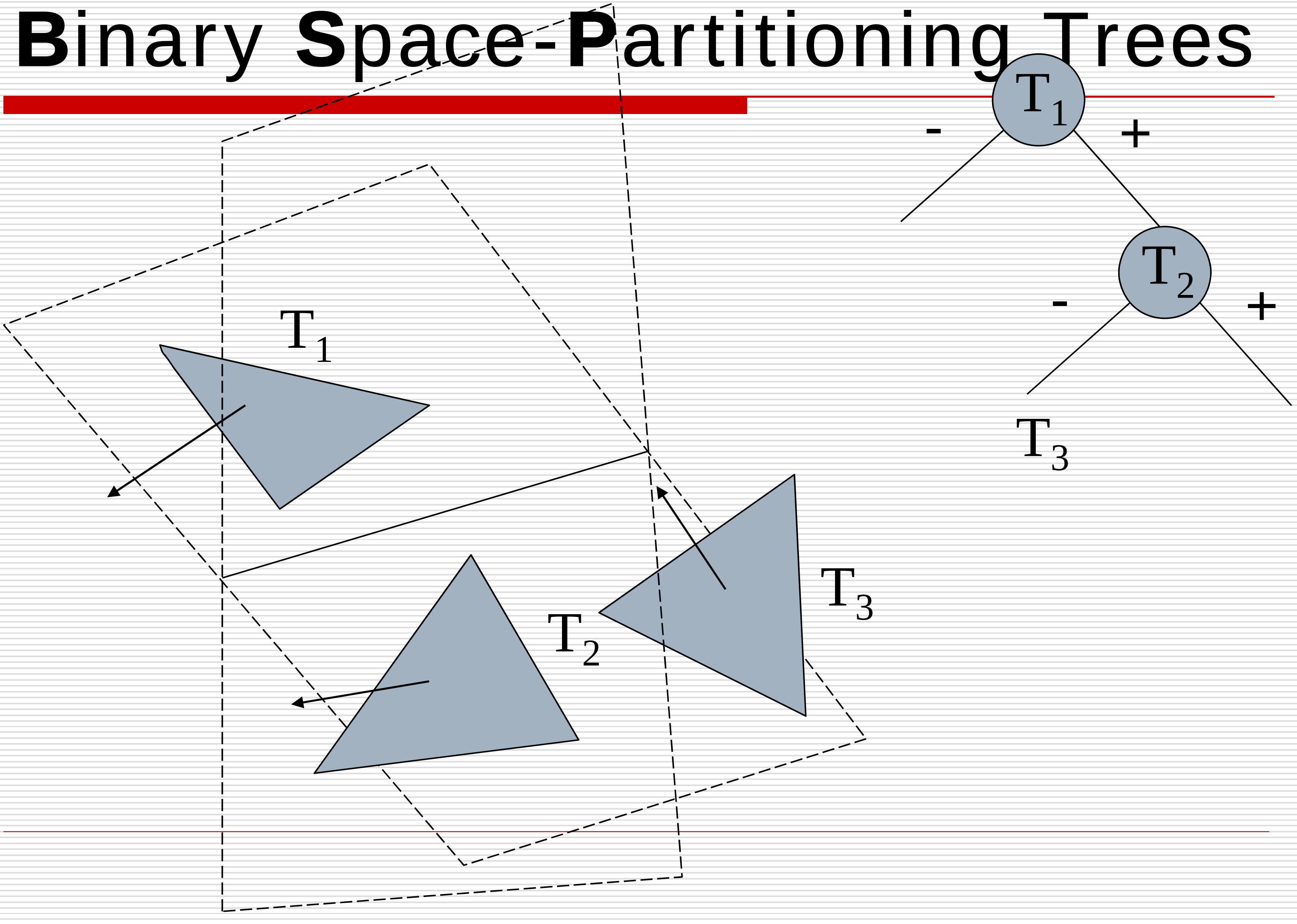


Binary **S**pace-**P**artitioning Trees

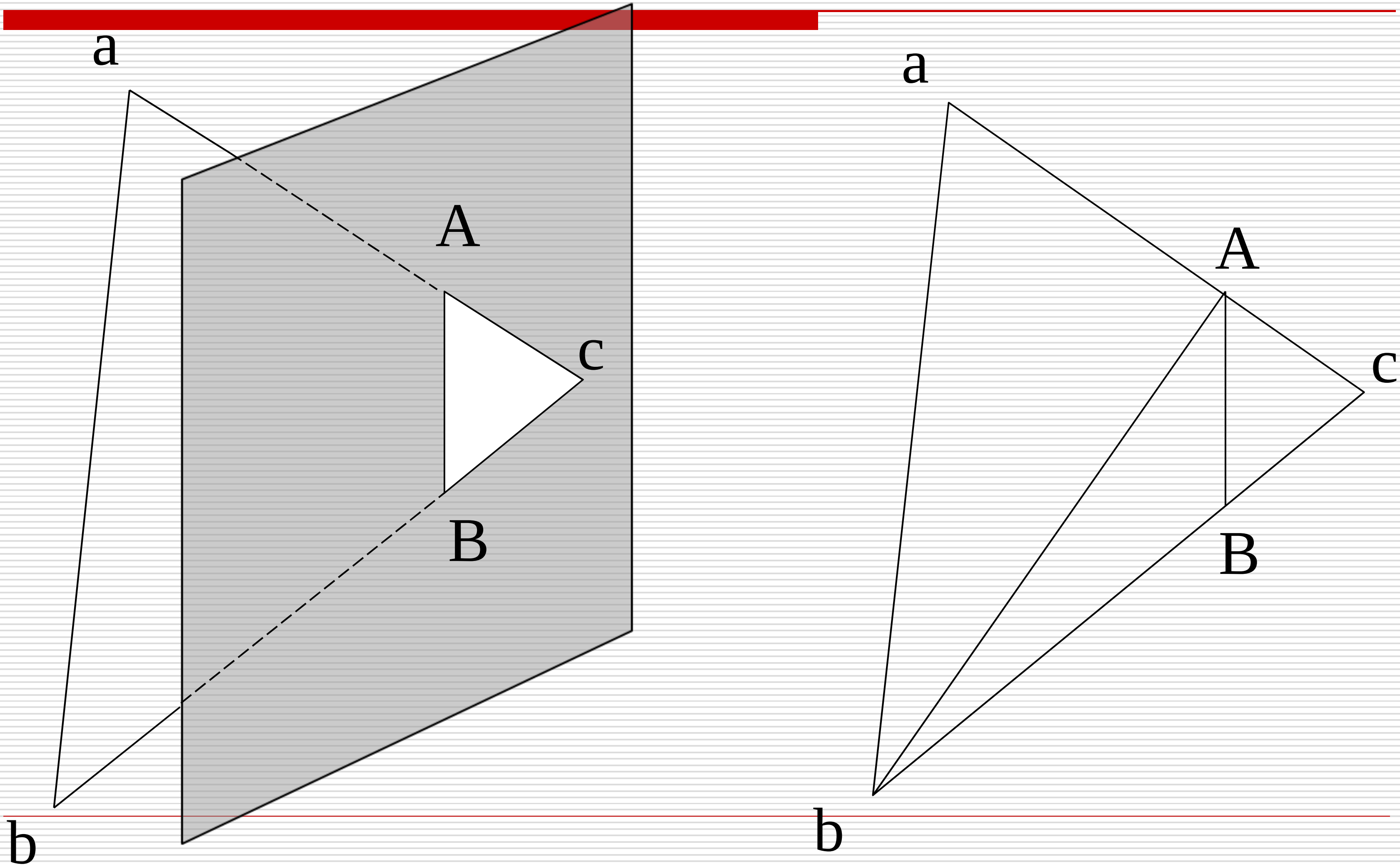




Binary **S**pace-**P**artitioning Trees



Splitting triangles



BSP Tree Construction

```
BSPtree makeBSP(L: list of polygons) {
    if (L is empty) {
        return the empty tree;
    }
    Choose a polygon P from L to serve as root;
    Split all polygons in L according to P
    return new TreeNode (
        P,
        makeBSP(polygons on negative side of P),
        makeBSP(polygons on positive side of P))
}
```

- Splitting polygons is expensive! It helps to choose P wisely at each step.
  - Example: choose five candidates, keep the one that splits the fewest polygons.

BSP Tree Display

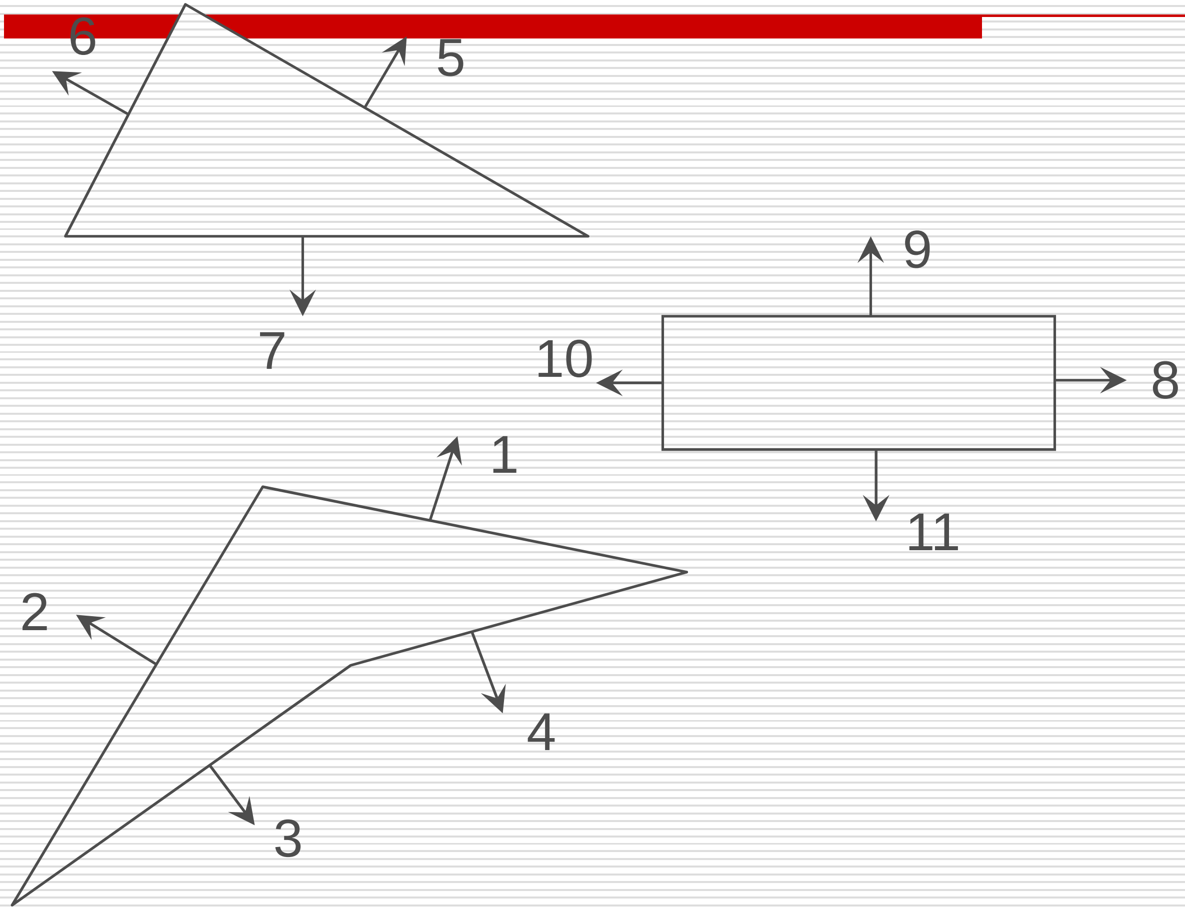
```
void showBSP(v: Viewer, T: BSPtree) {
    if (T is empty) return;
    P = root of T;
    if (viewer is in front of P) {
        showBSP(back subtree of T);
        draw P;
        showBSP(front subtree of T);
    } else {
        showBSP(front subtree of T);
        draw P;
        showBSP(back subtree of T);
    }
}
```

2D BSP demo

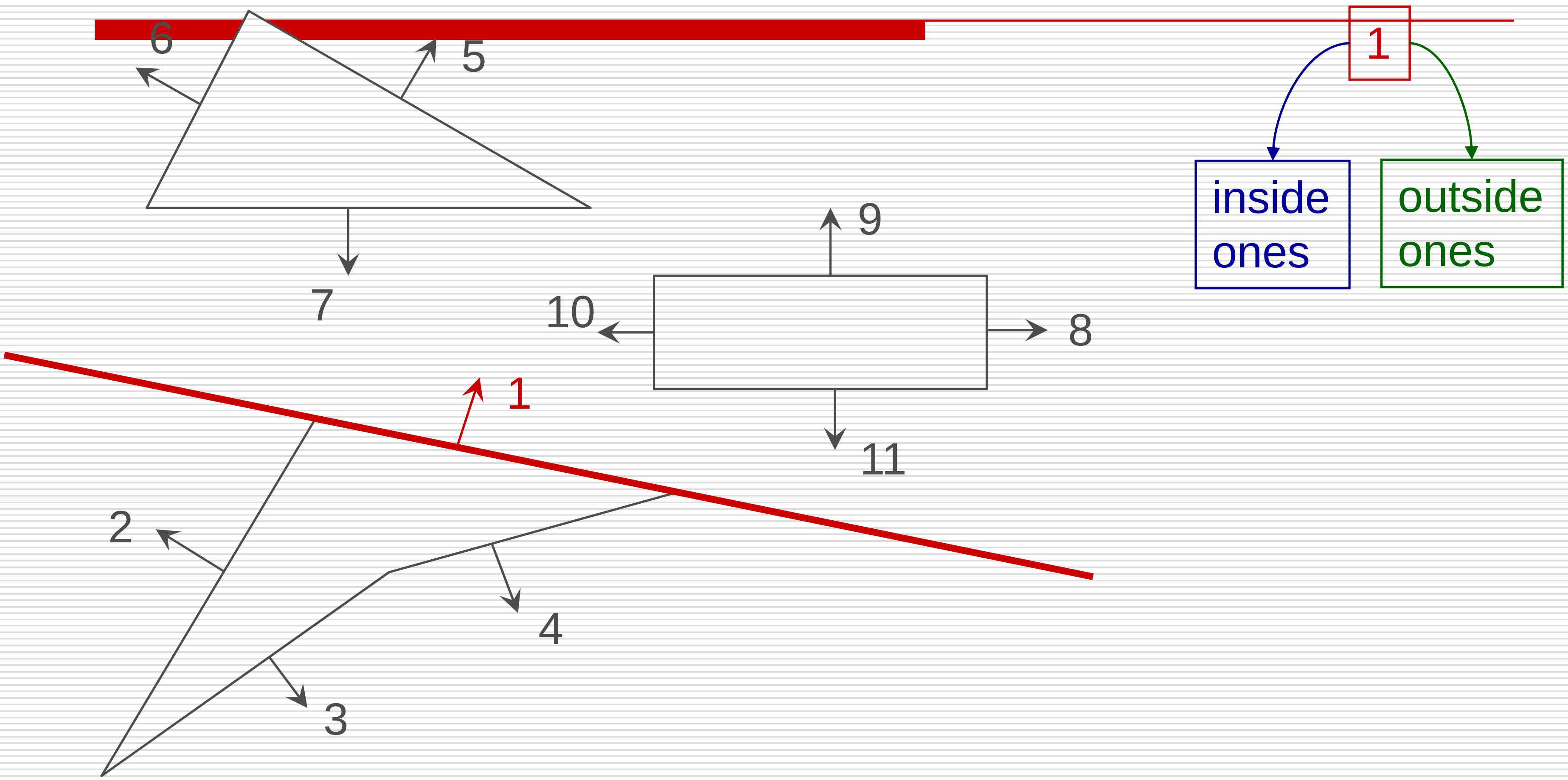
Binary Space-Partitioning Trees

- Same BSP tree can be used for any eye position, constructed only once if the scene if static.
- It does not matter whether the tree is balanced. However, splitting triangles is expensive and try to avoid it by picking up different partition planes.

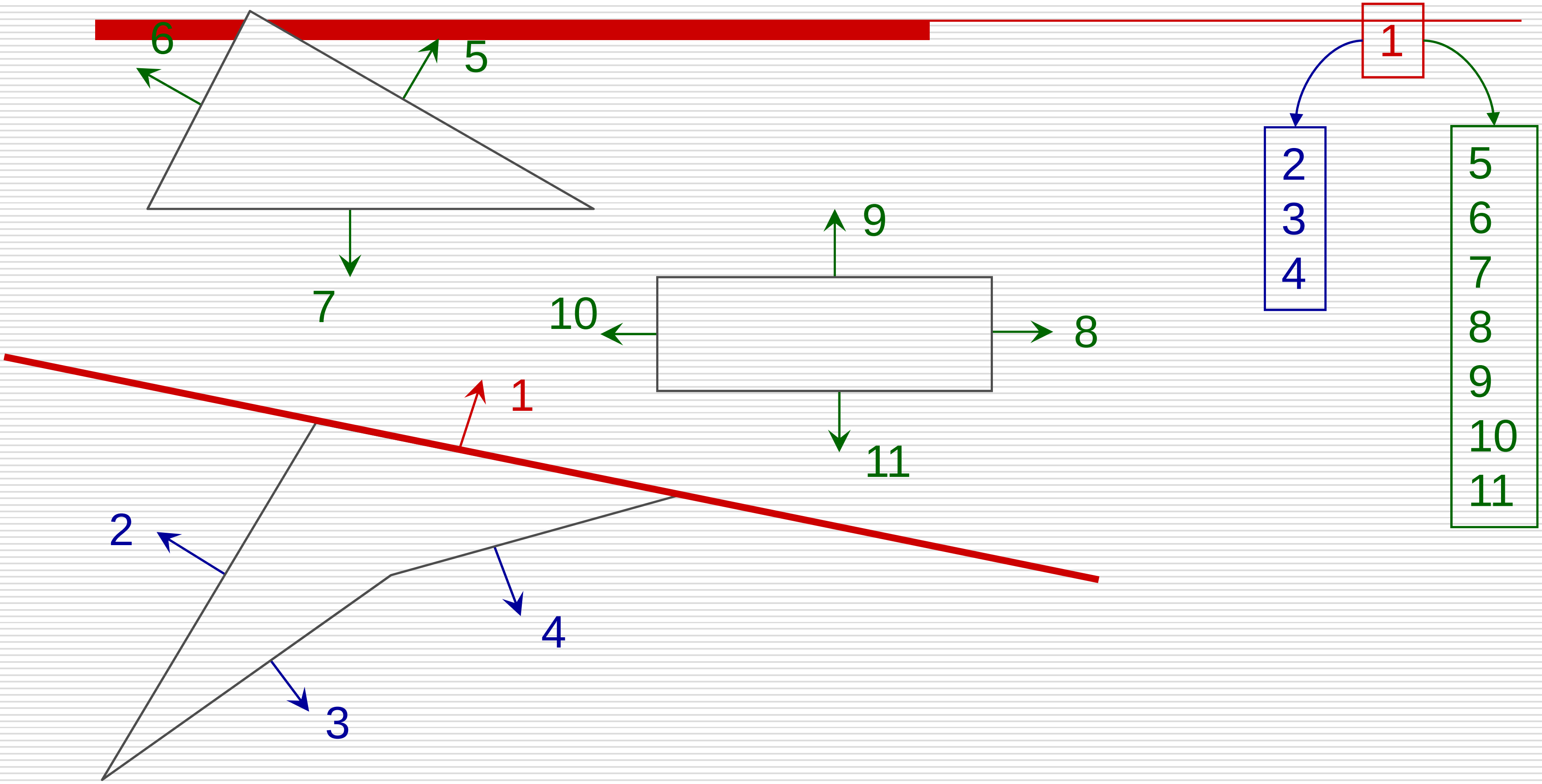
BSP Tree



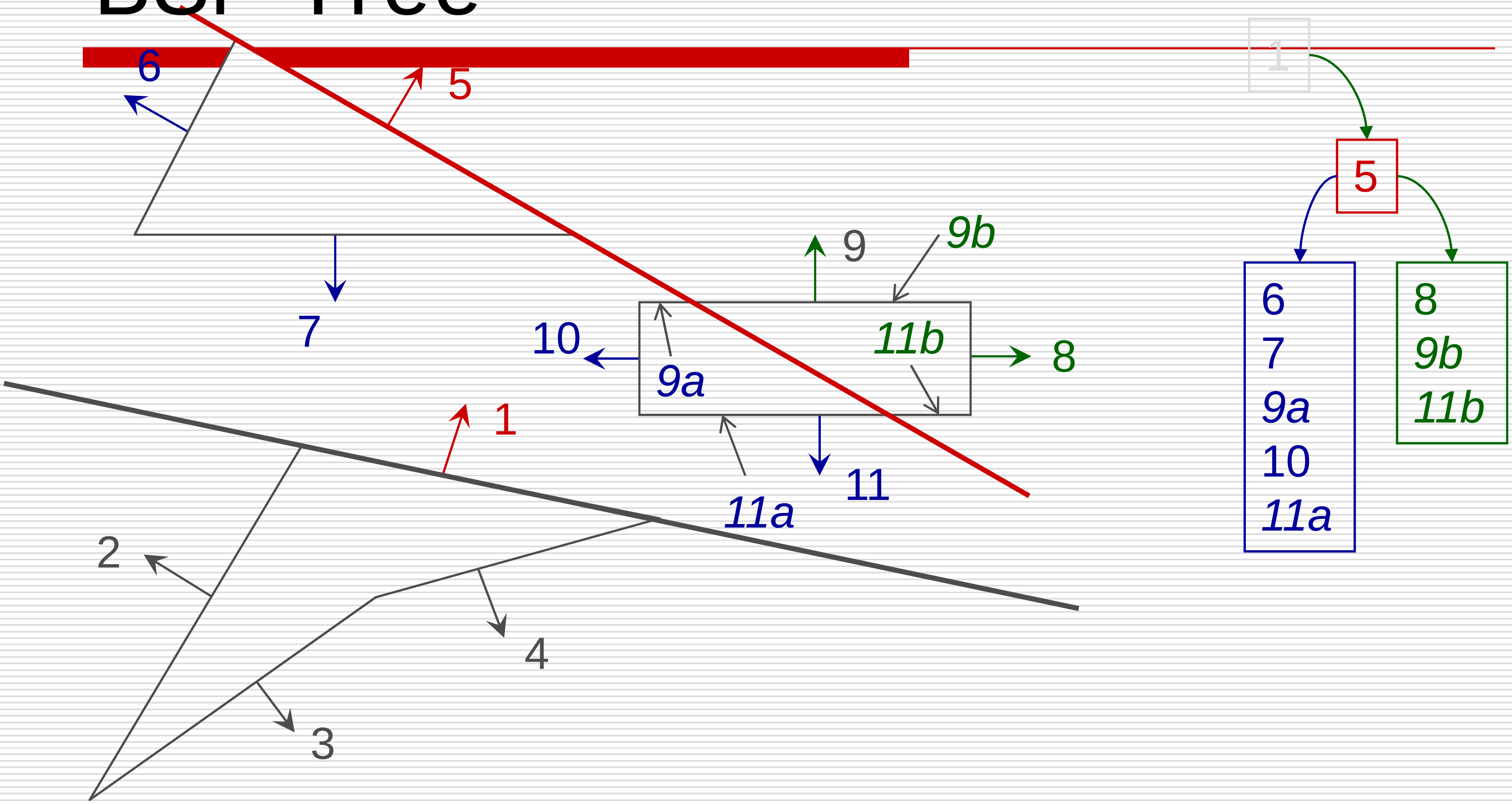
BSP Tree



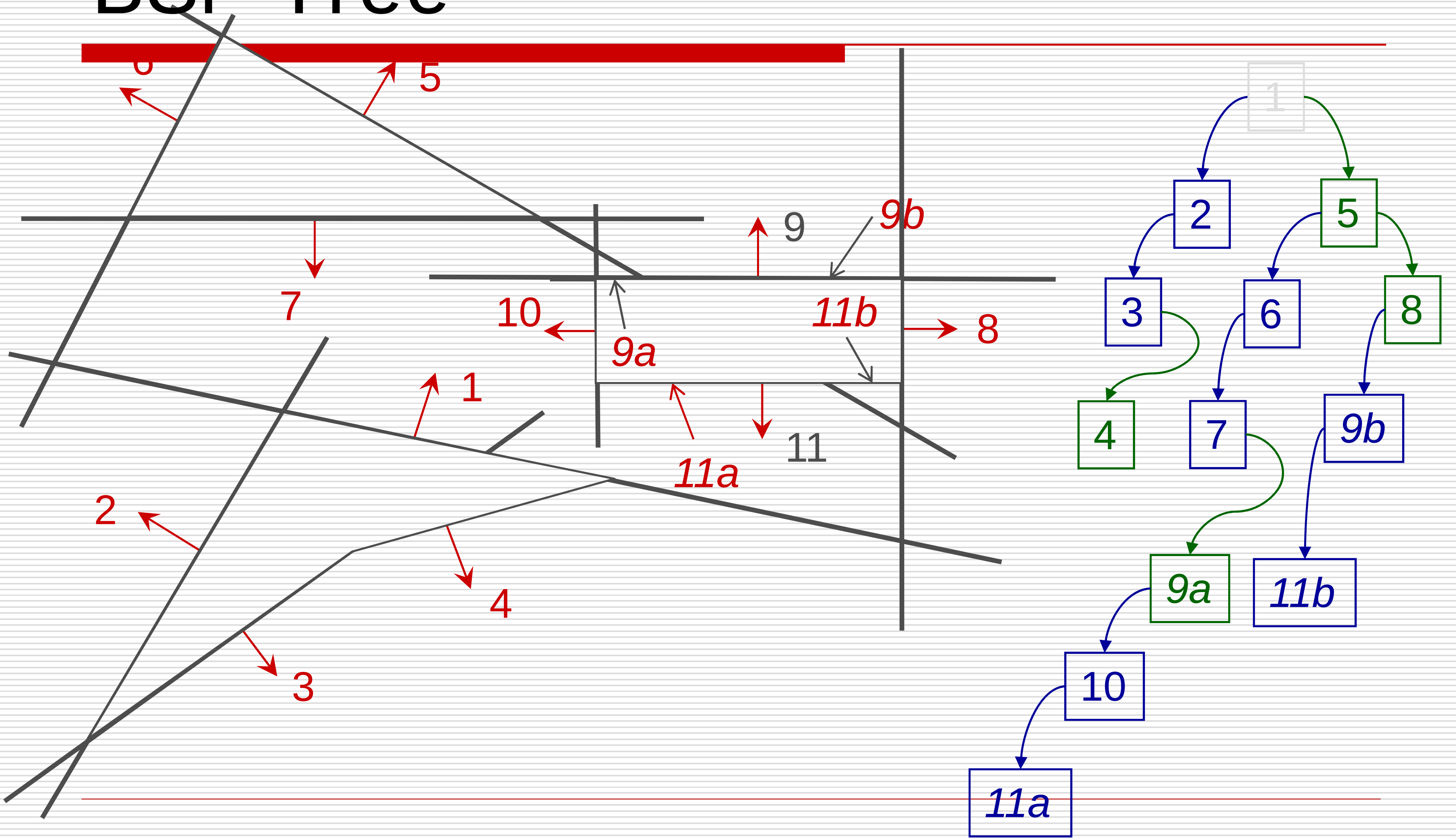
BSP Tree



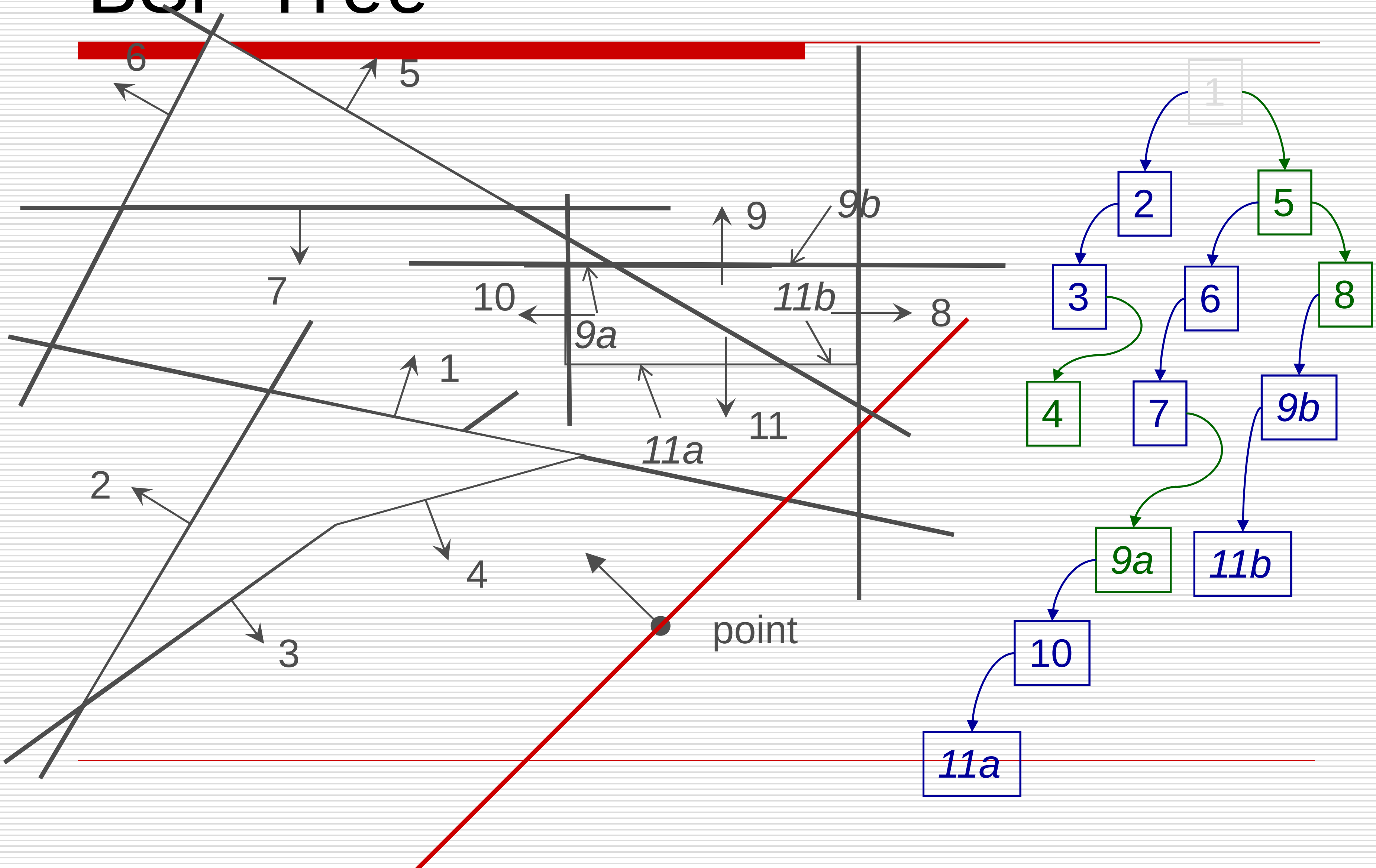
BSP Tree



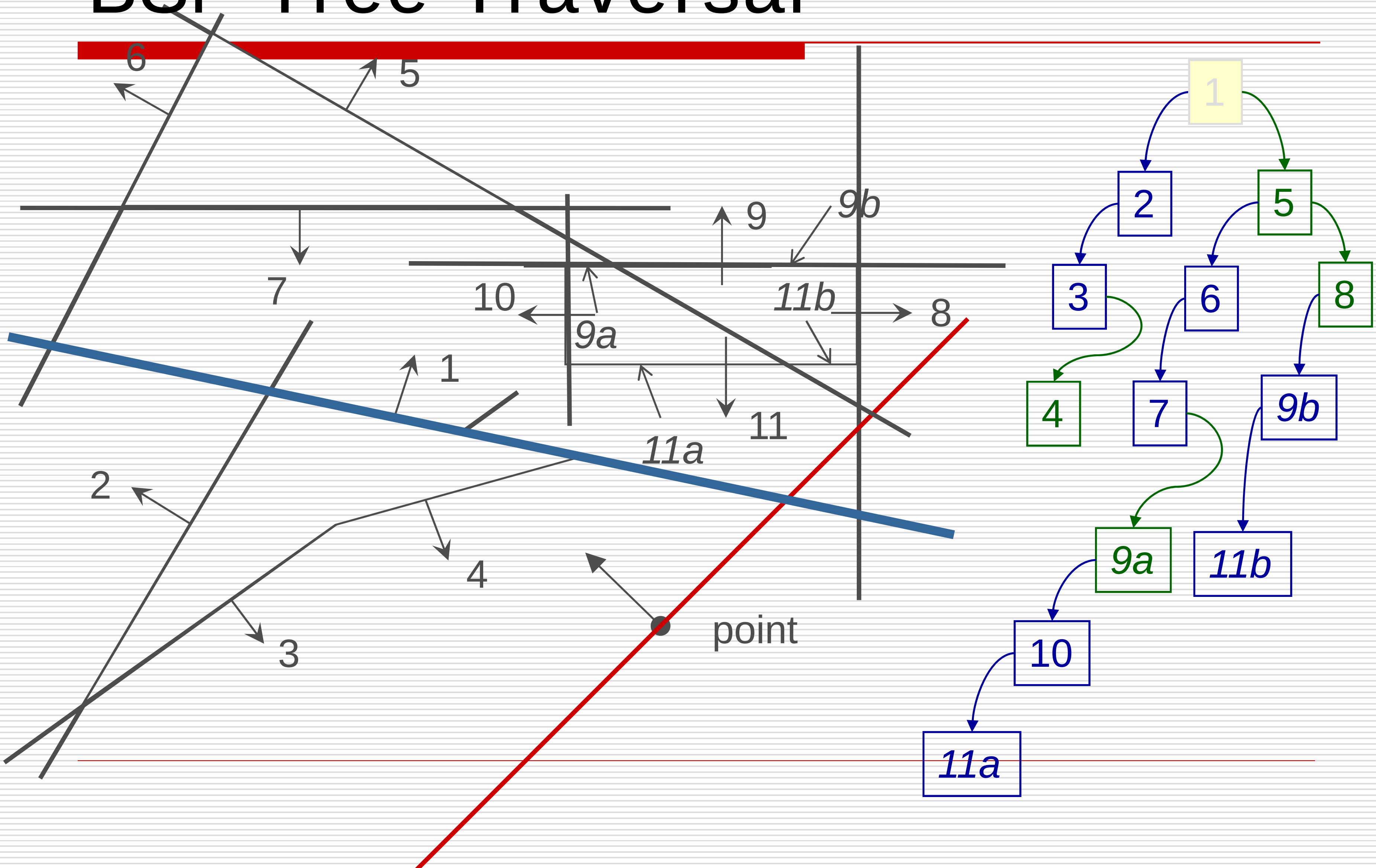
BSP Tree



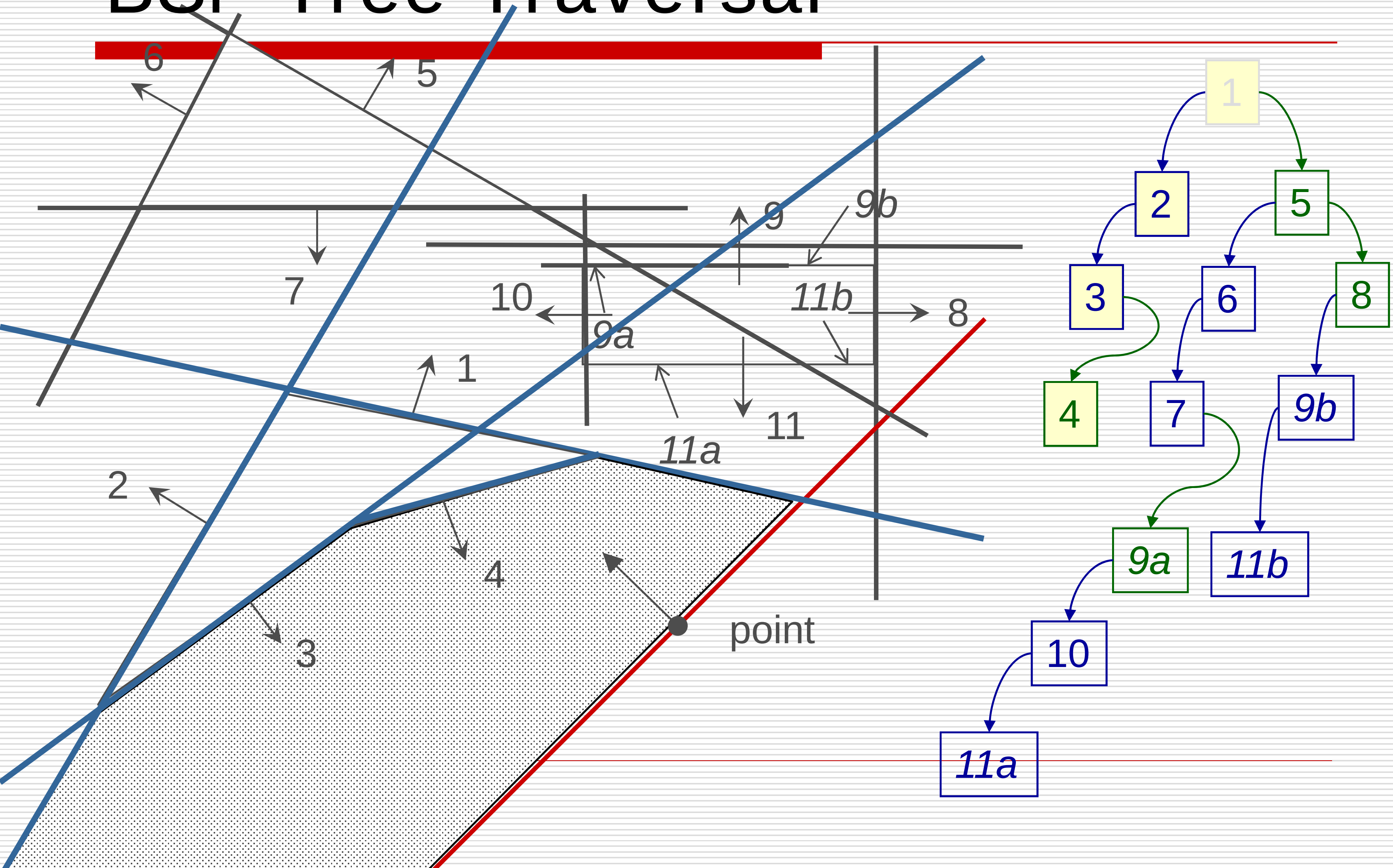
BSP Tree



BSP Tree Traversal



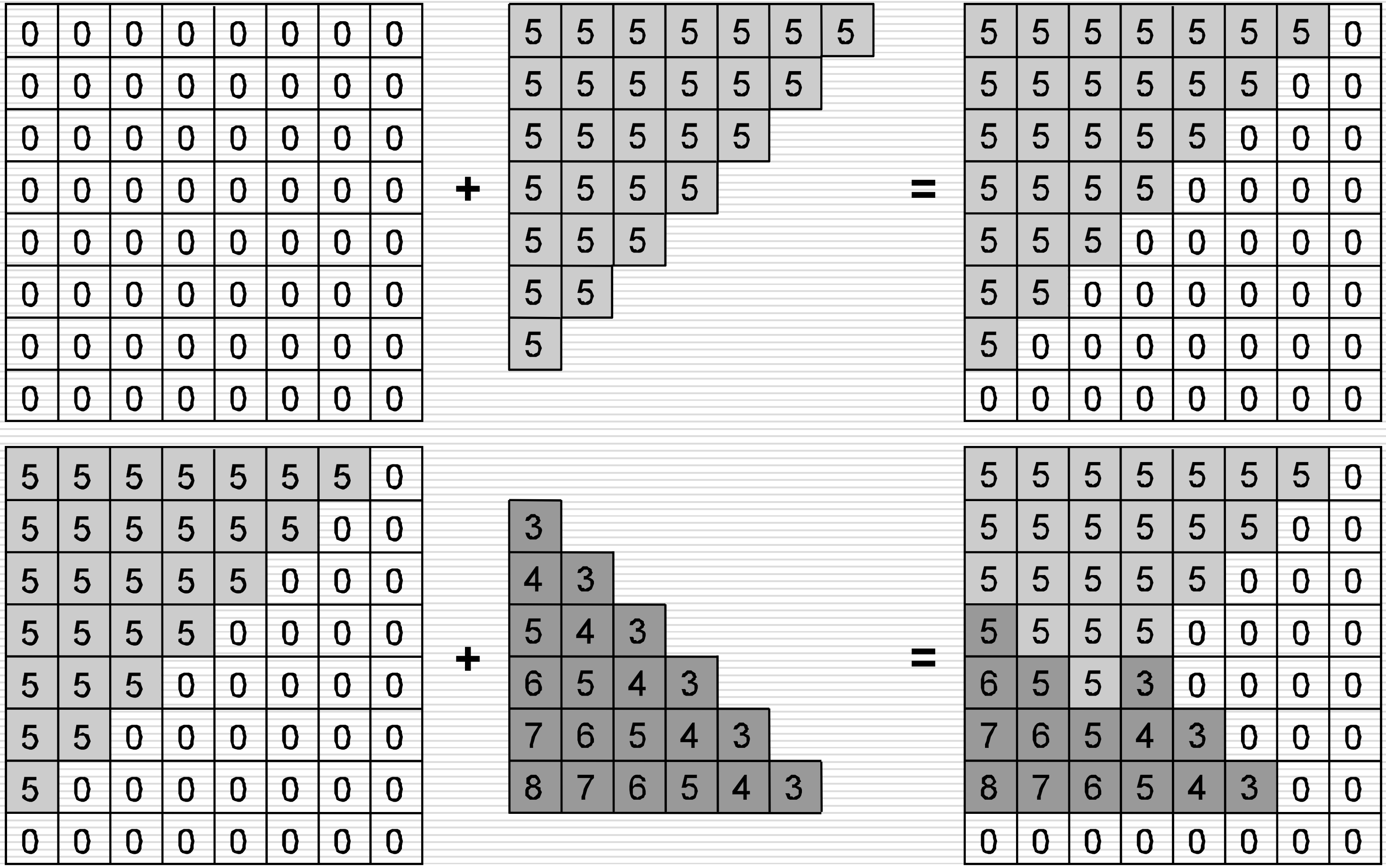
BSP Tree Traversal



The z-Buffer Algorithm

- Resolve depths at the pixel level
- Idea: add Z to frame buffer, when a pixel is drawn, check whether it is closer than what's already in the frame buffer

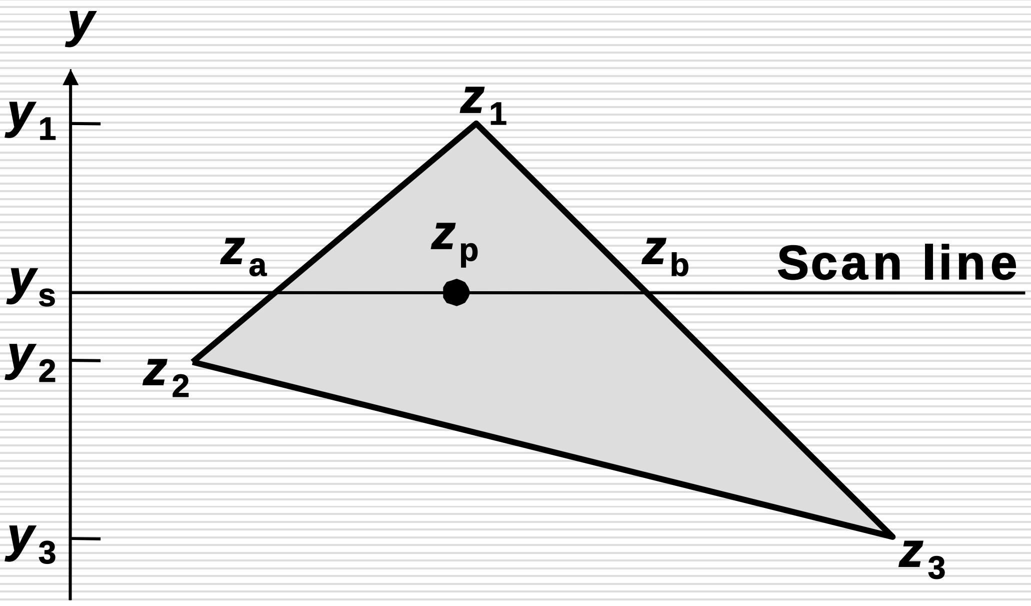
The z-Buffer Algorithm



The z-Buffer Algorithm

```
void zBuffer() {
    int pz;
    for (each polygon) {
        for (each pixel in polygon's projection) {
            pz=polygon's z-value at (x,y);
            if (pz>=ReadZ(x,y)) {
                WriteZ(x,y,pz);
                WritePixel(x,y,color);
            }
        }
    }
}
```

The z-Buffer Algorithm



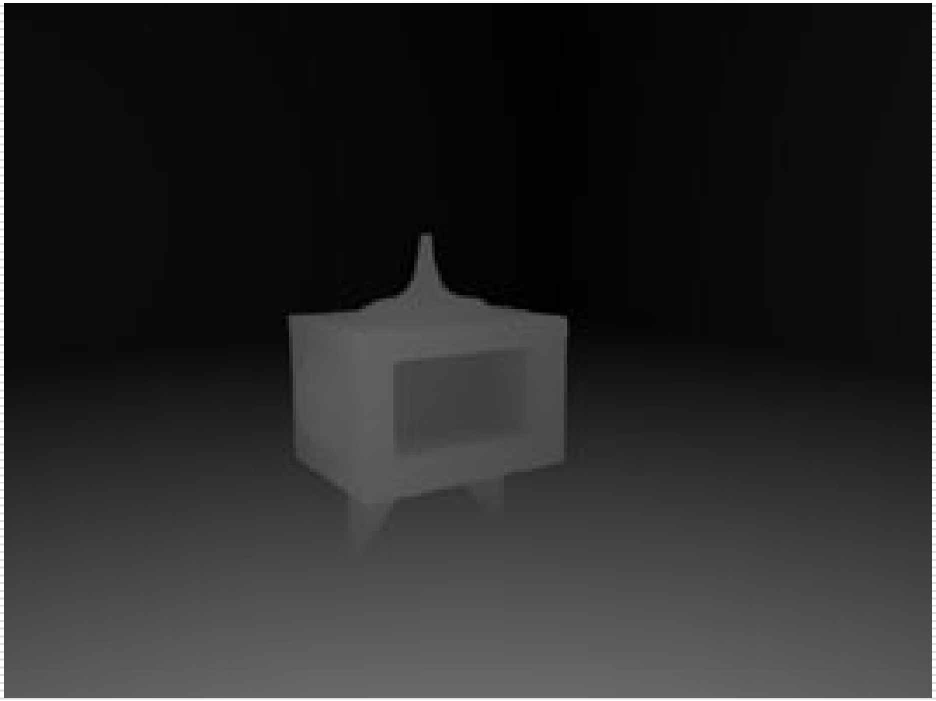
$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

z-Buffer: Example

color buffer



depth buffer

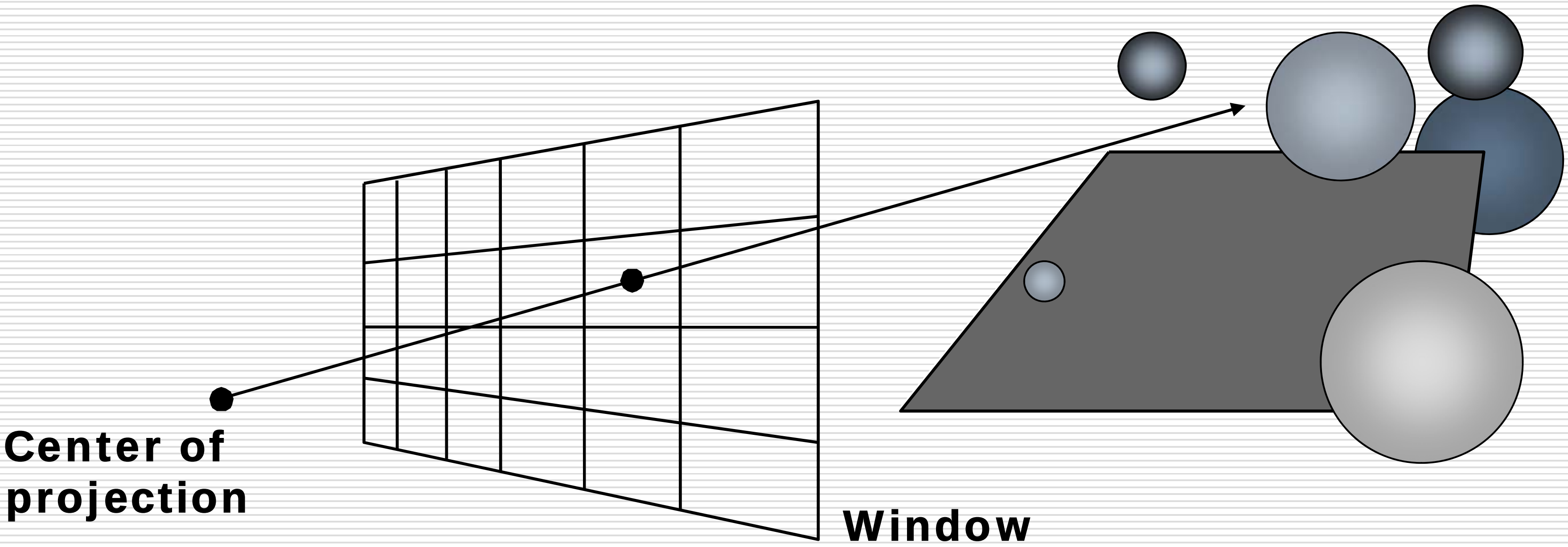
The z-Buffer Algorithm

- Benefits
  - Easy to implement
  - Works for any geometric primitive
  - Parallel operation in hardware
    - independent of order of polygon drawn
- Limitations
  - Memory required for depth buffer
  - Quantization and aliasing artifacts
  - Overfill
  - Transparency does not work well

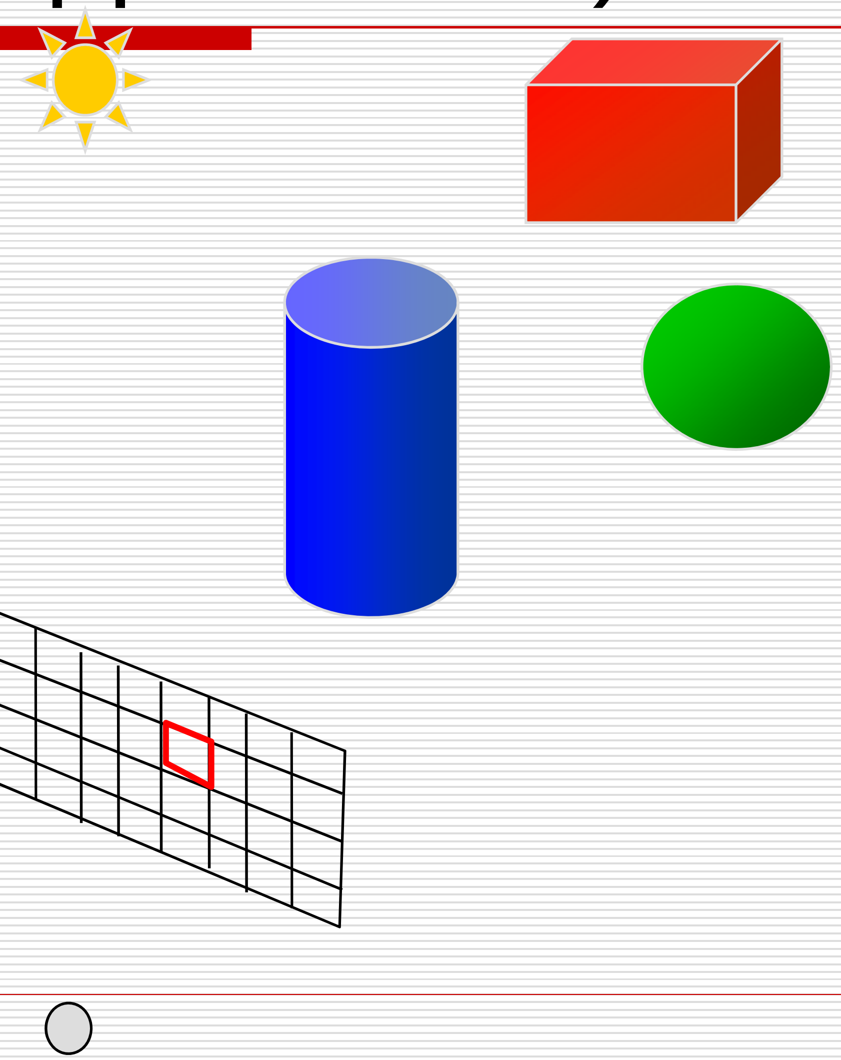
Ray Tracing = Ray Casting

```
select center of projection and window on viewplane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        for (each object in scene) {
            if (object is intersected and is closest considered thus far)
                record intersection and object name;
        }
        set pixel's color to that at closest object intersection;
    }
}
```

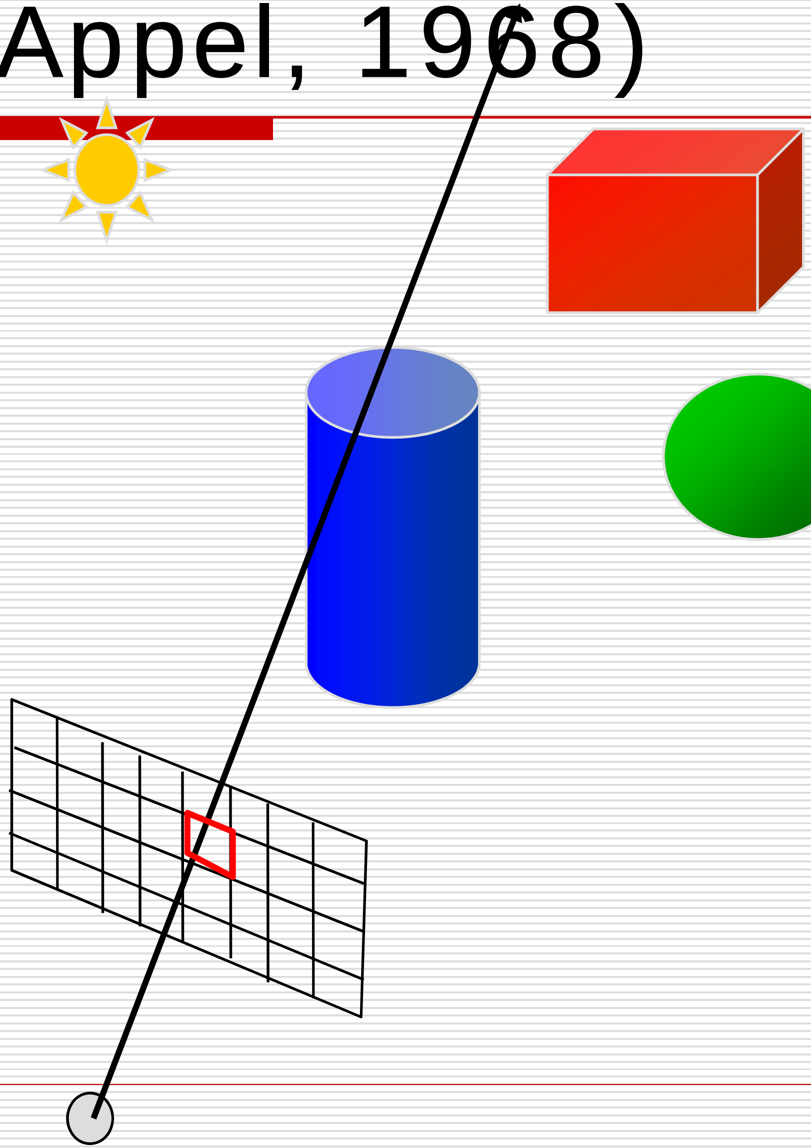
Ray Casting



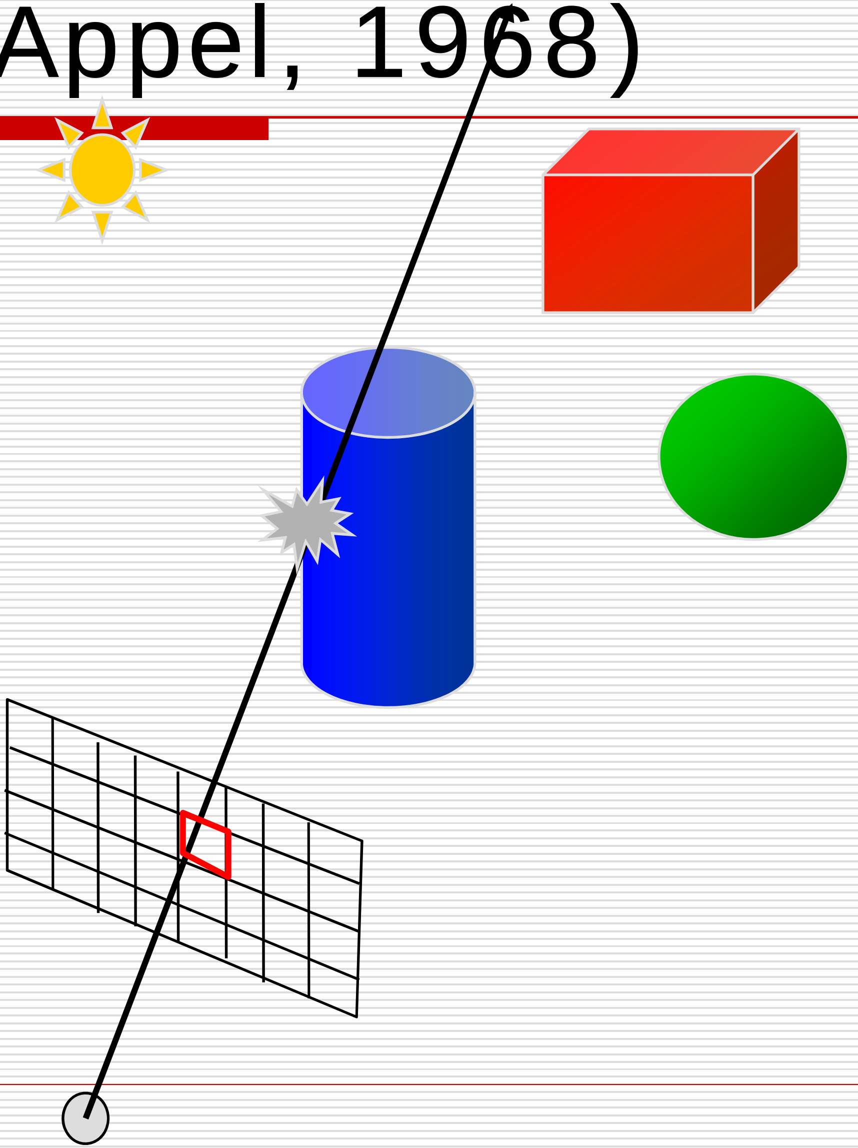
Ray Casting (Appel, 1968)



Ray Casting (Appel, 1968)

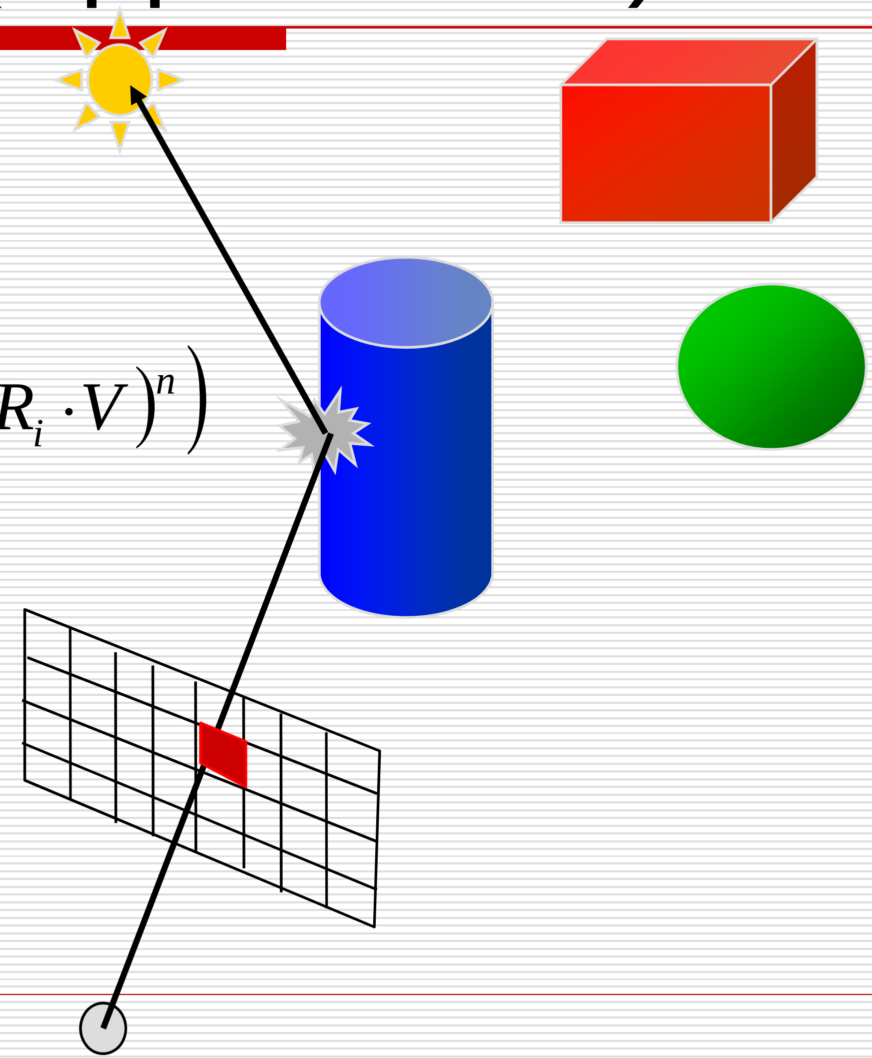


Ray Casting (Appel, 1968)

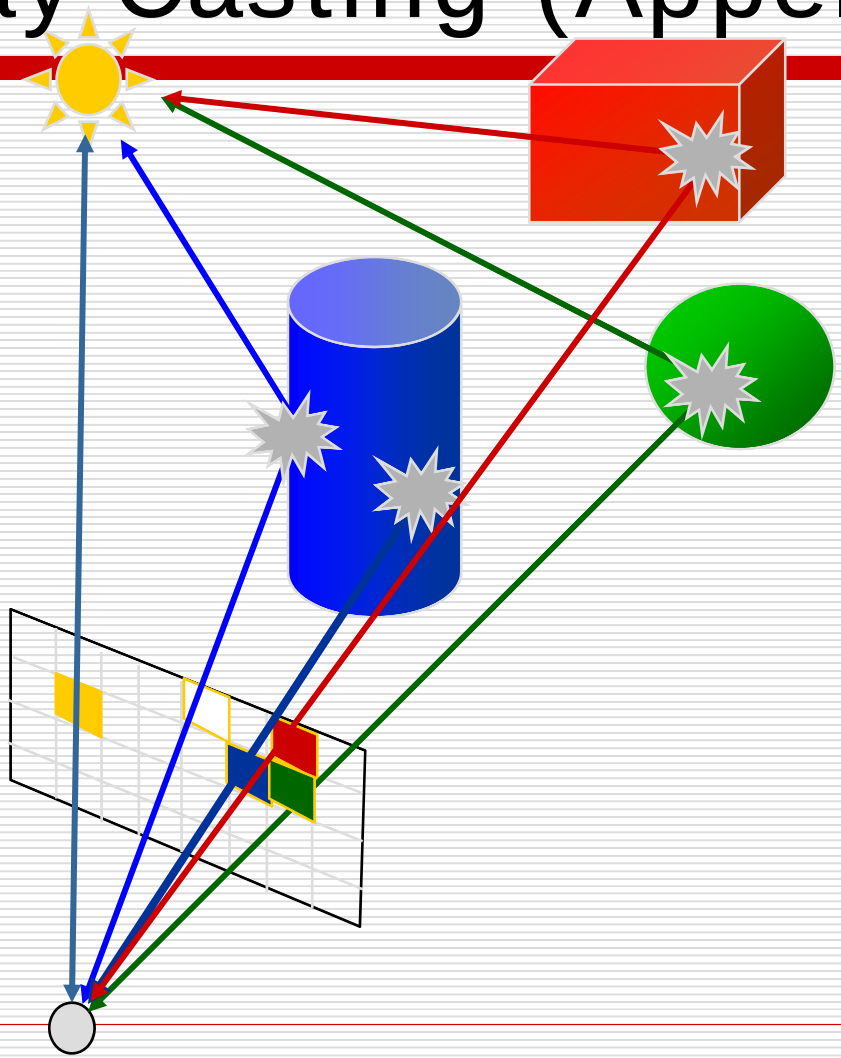


Ray Casting (Appel, 1968)

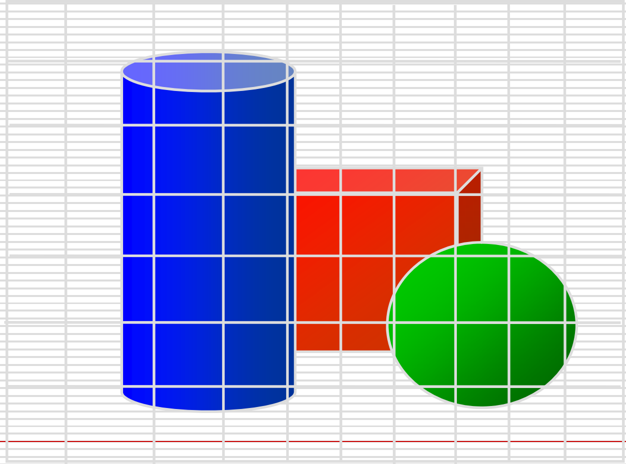
$$k_a I_a + \sum_{i=1}^{nls} I_i \left(k_d \left(L_i \cdot N \right) + k_s \left(R_i \cdot V \right)^n \right)$$



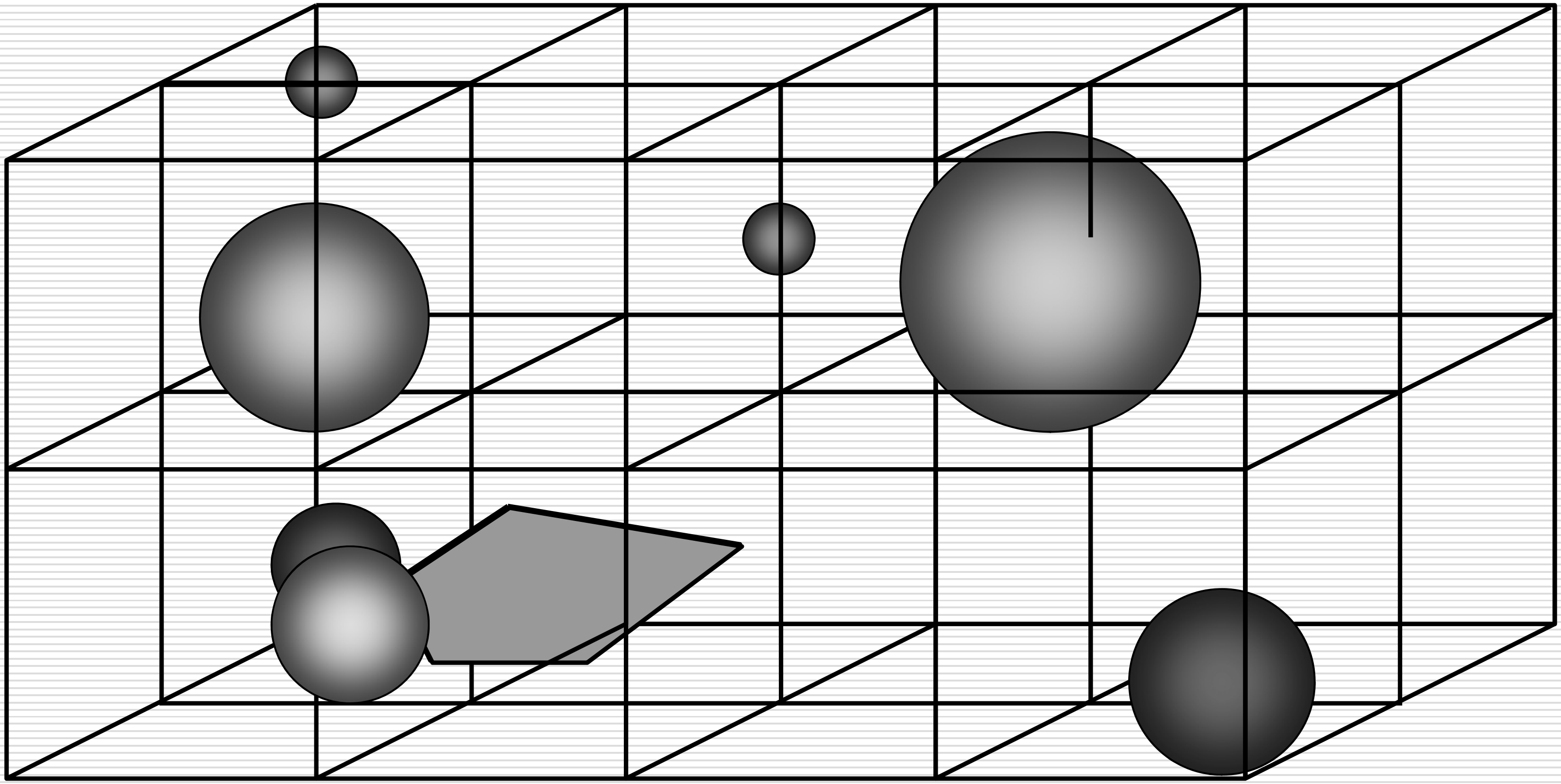
Ray Casting (Appel, 1968)



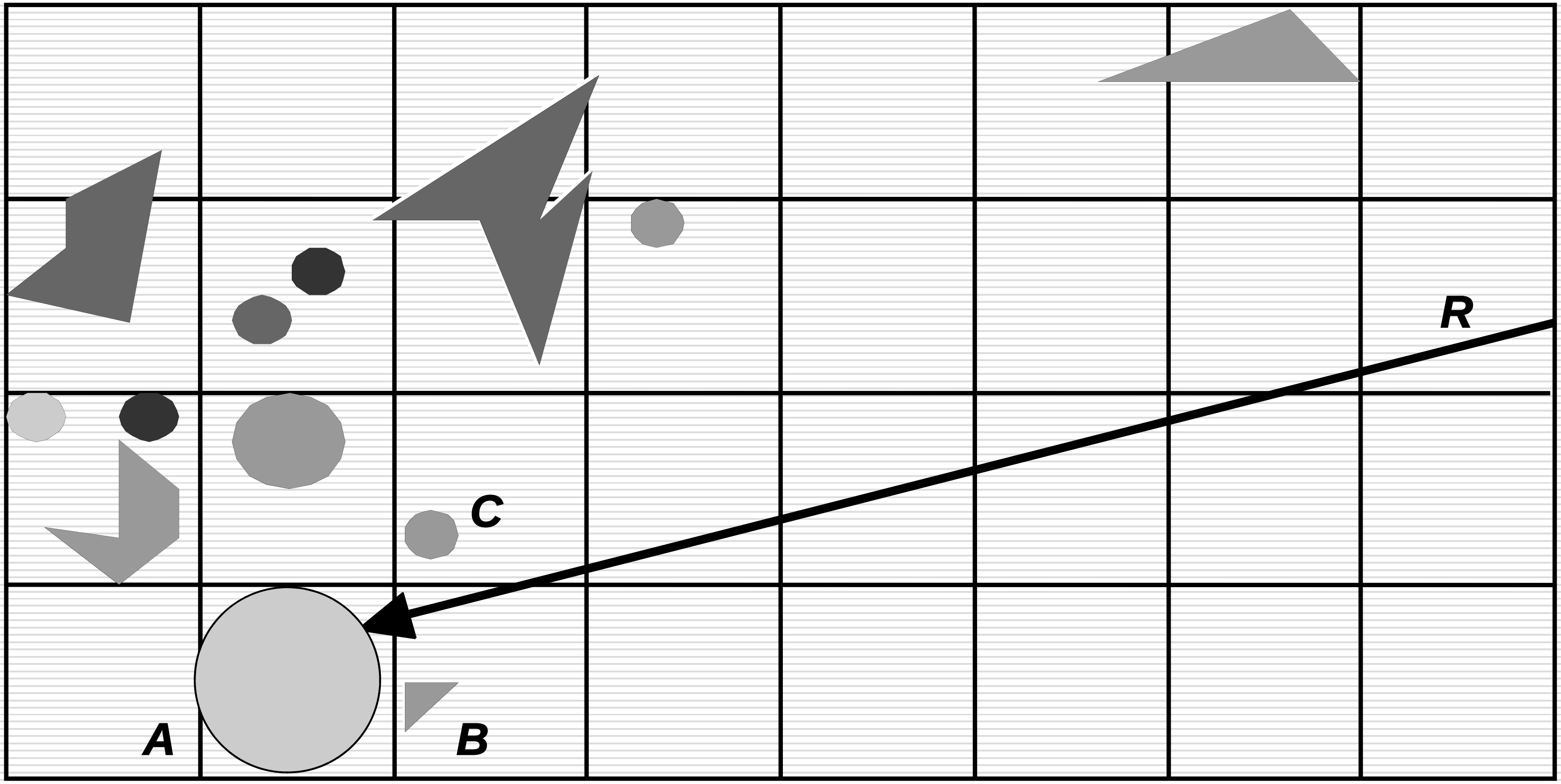
direct illumination



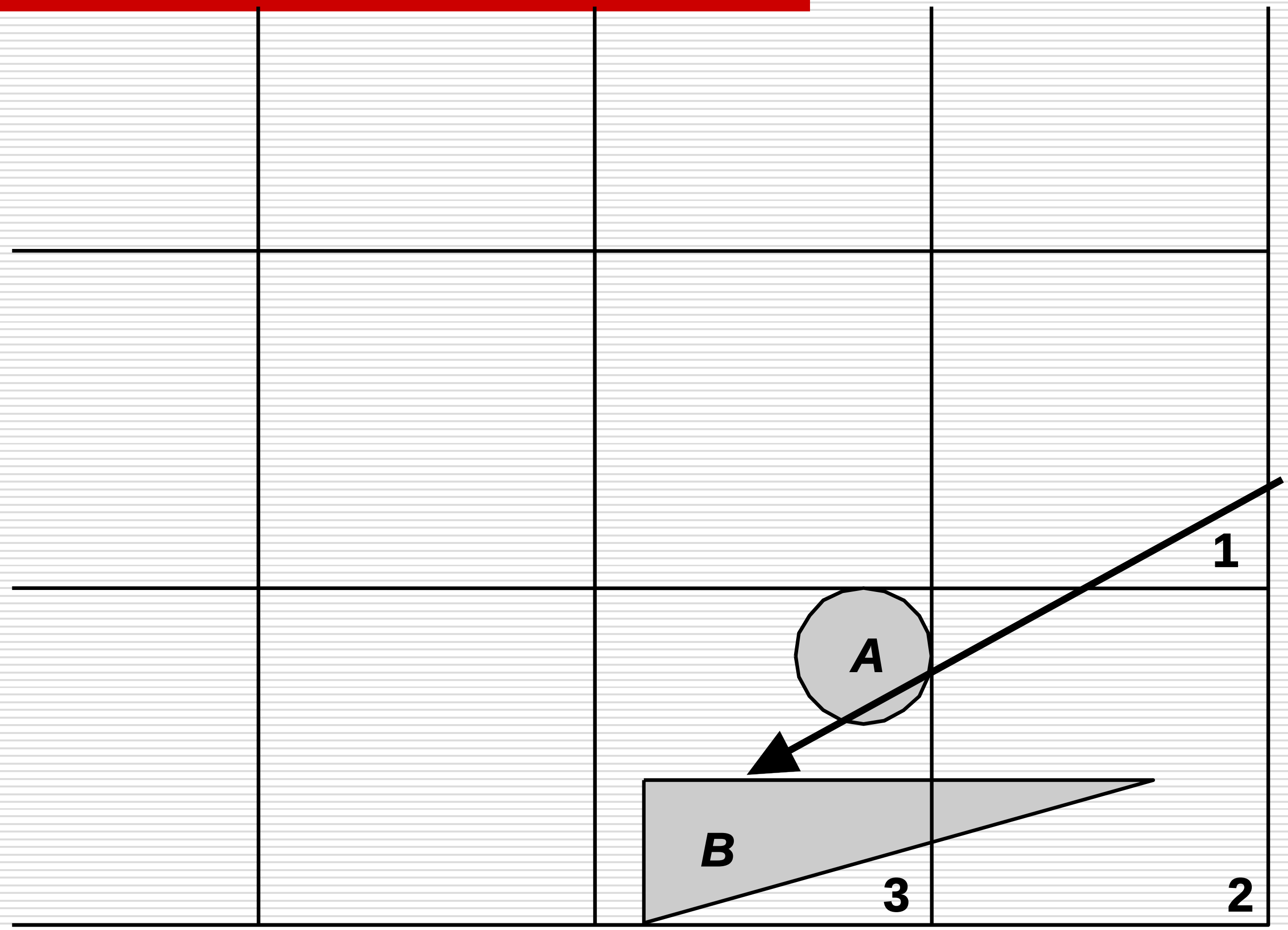
Spatial Partitioning



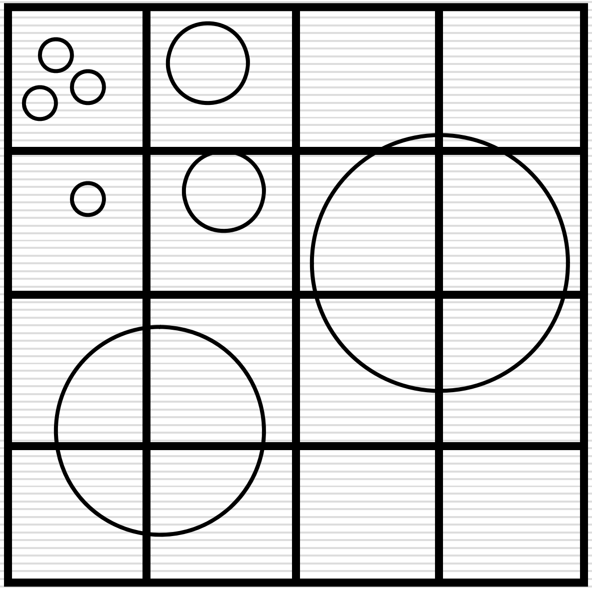
Spatial Partitioning



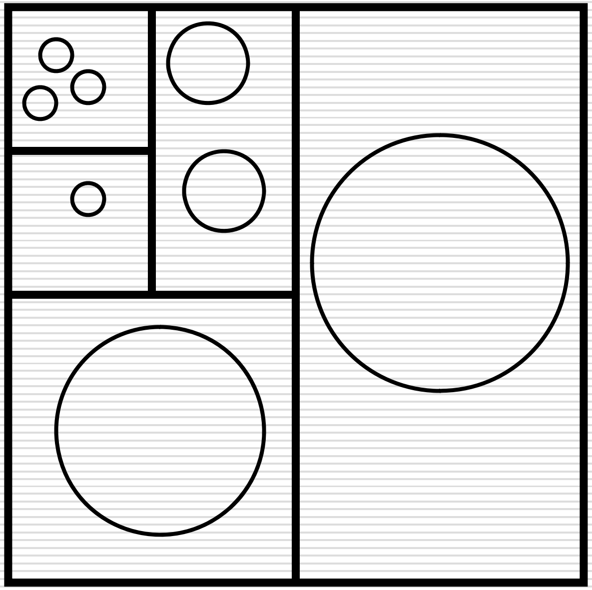
Spatial Partitioning



Space Subdivision Approaches

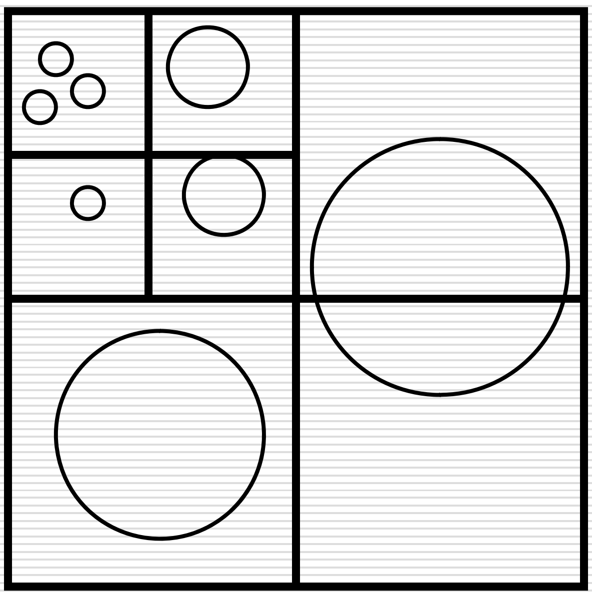


Uniform grid

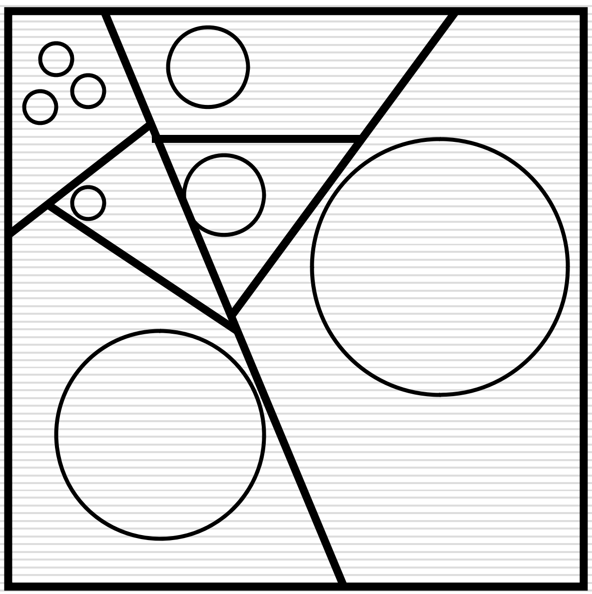


K-d tree

Space Subdivision Approaches

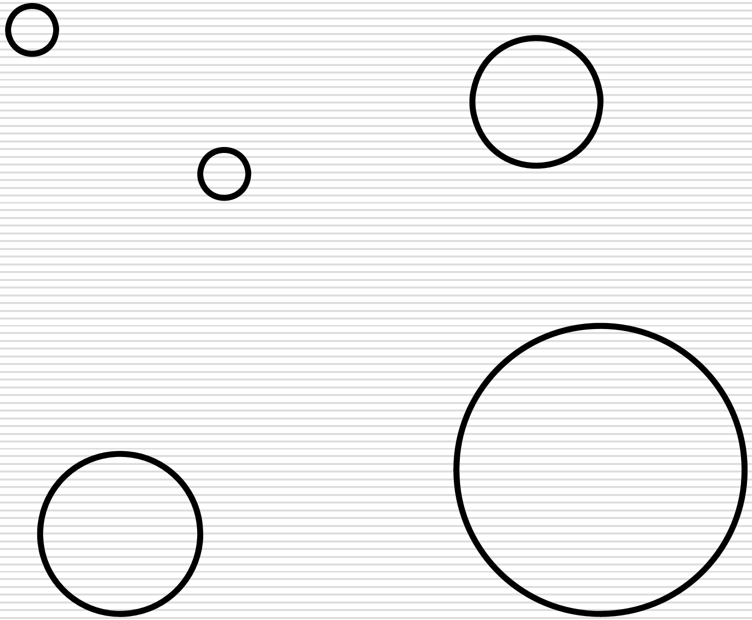


Quadtree (2D)
Octree (3D)

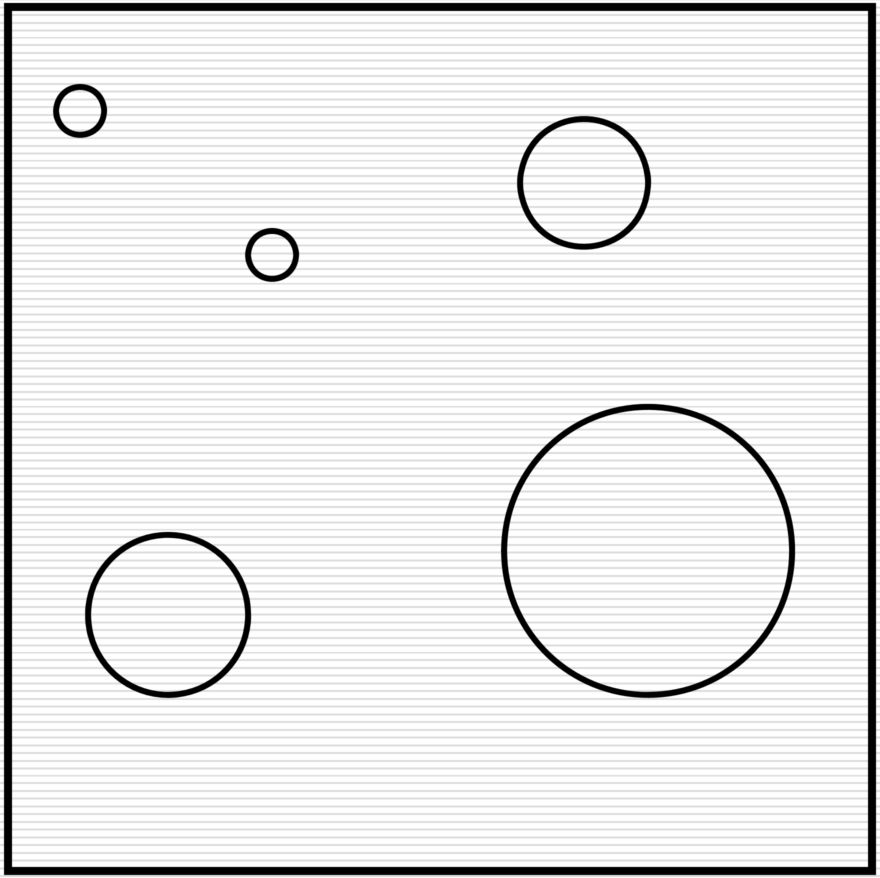


BSP tree

Uniform Grid



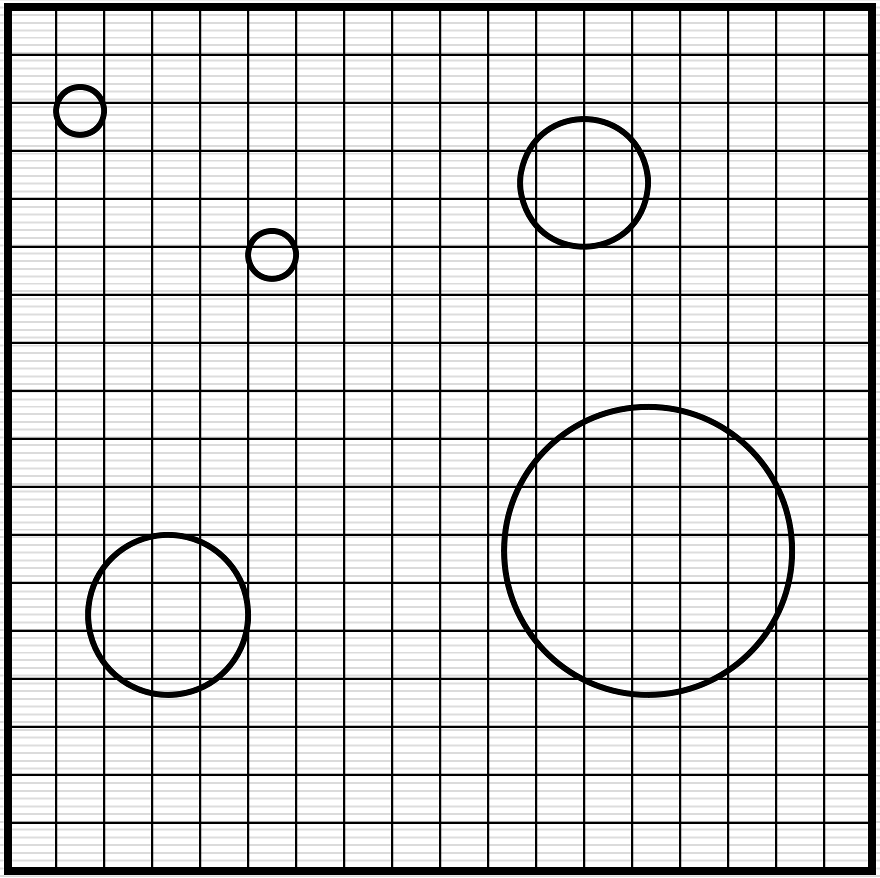
Uniform Grid



Preprocess scene

1. Find bounding box

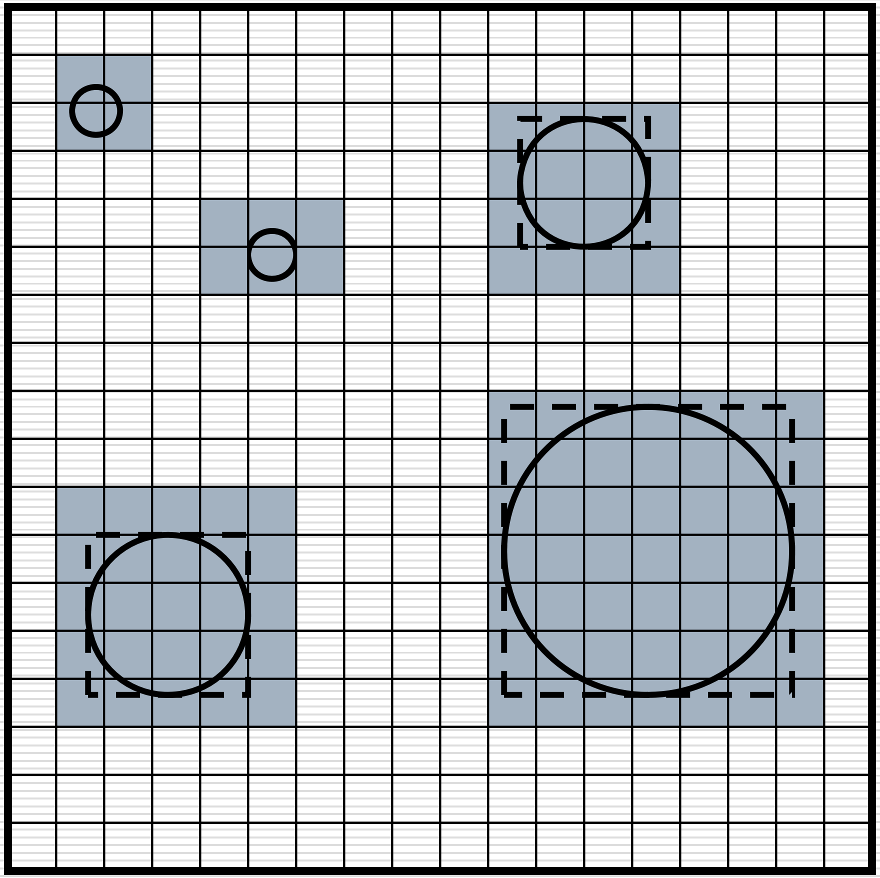
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution

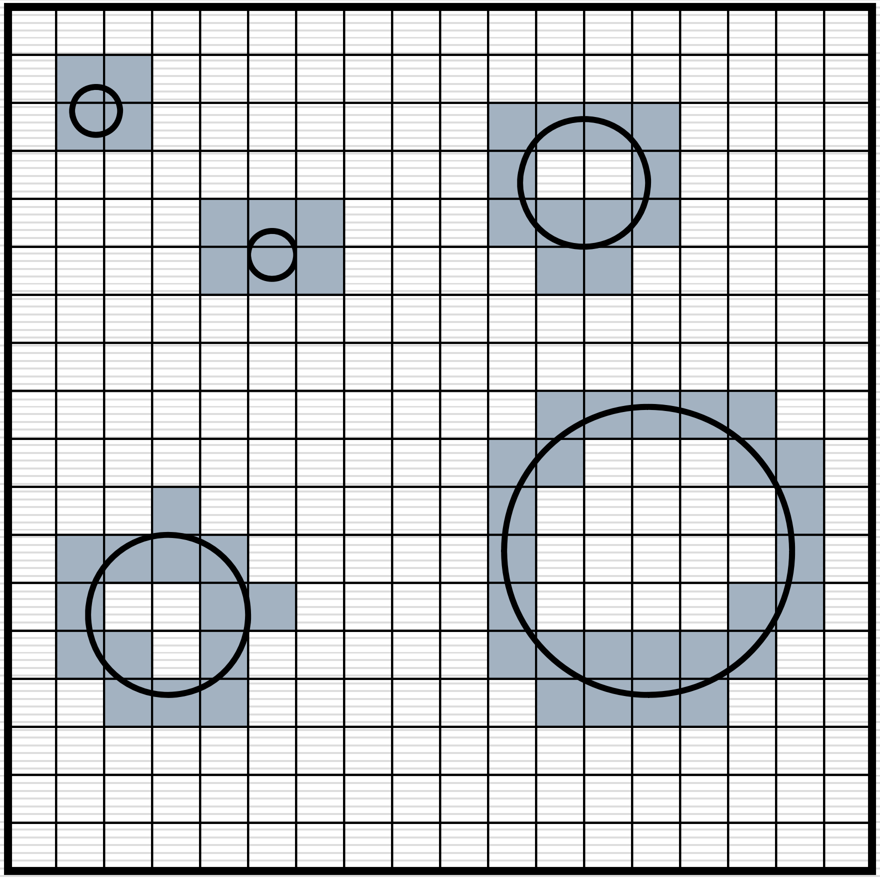
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell

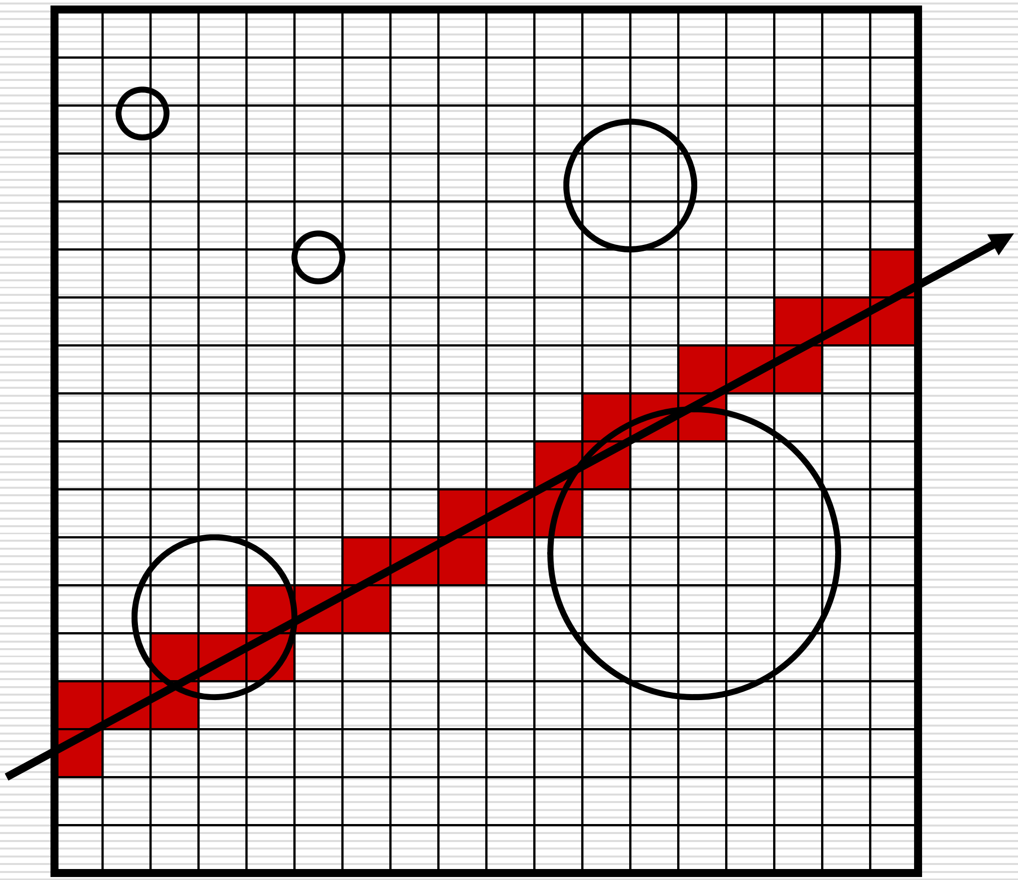
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell
4. Check that object overlaps cell (expensive!)

Uniform Grid Traversal

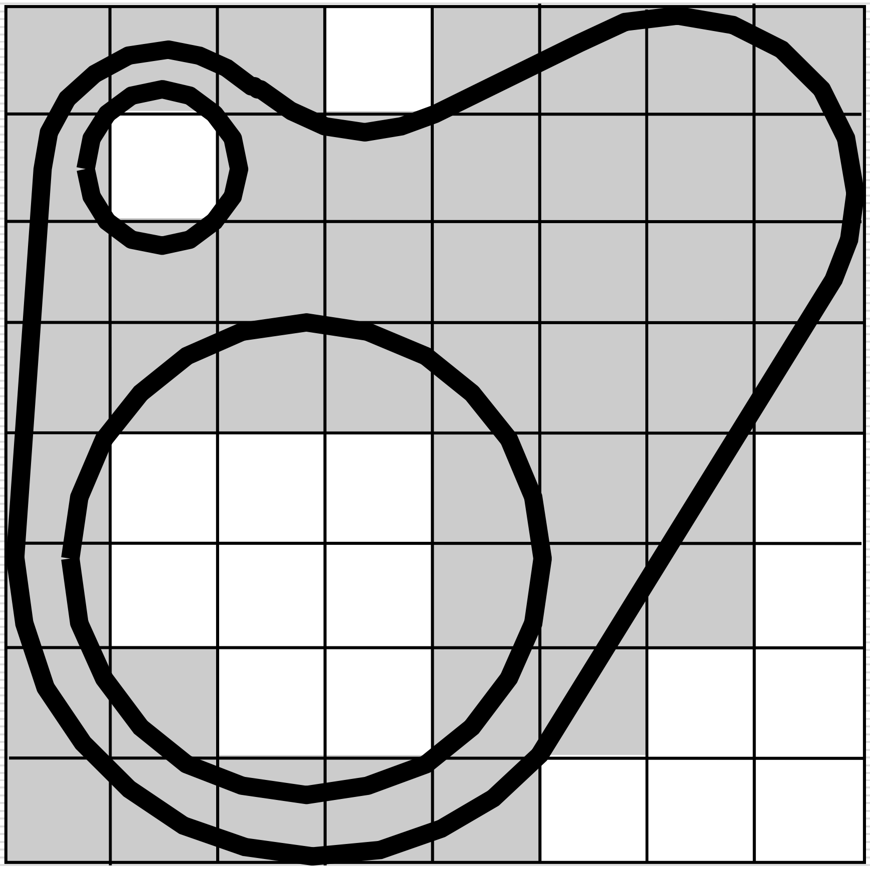


Preprocess scene

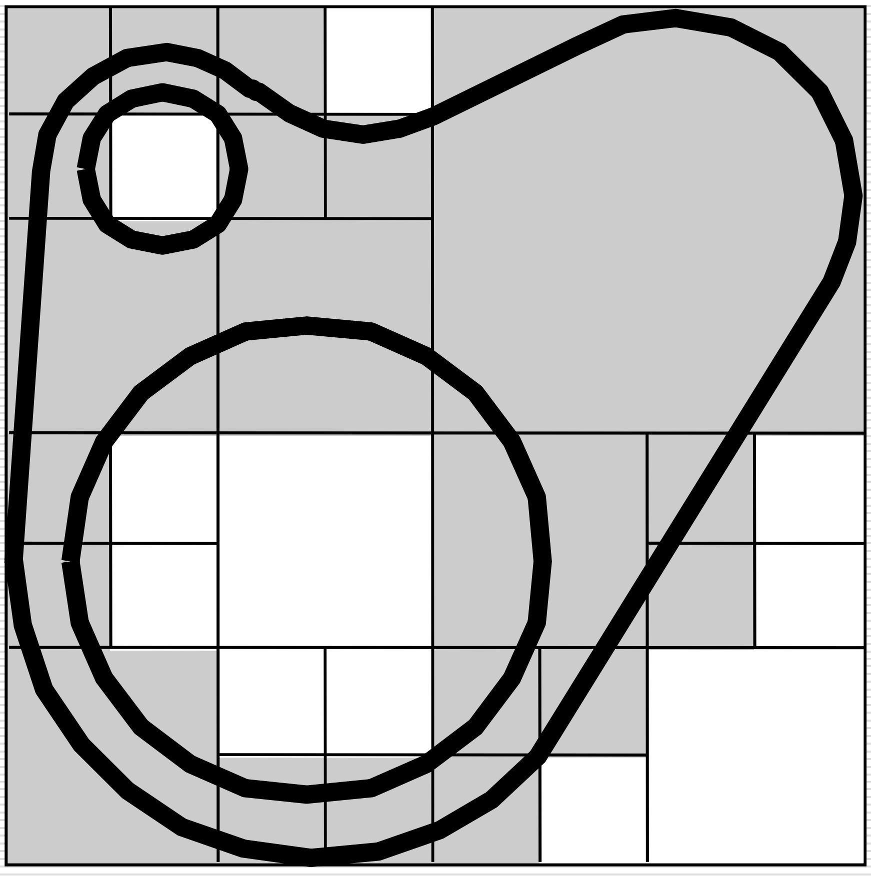
Traverse grid

3D line = 3D-DDA

From Uniform Grid to Quadtree

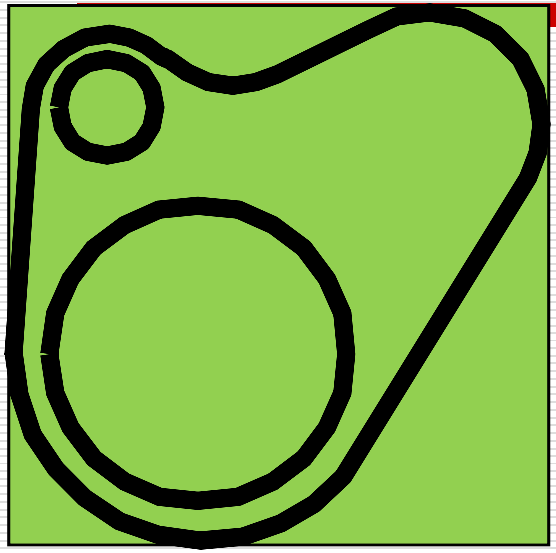


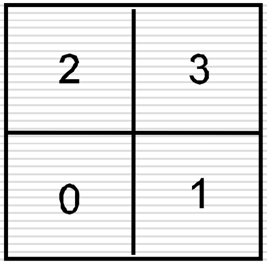
Quadtree (Octrees)



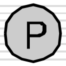
subdivide the space adaptively

Quadtree Data Structure

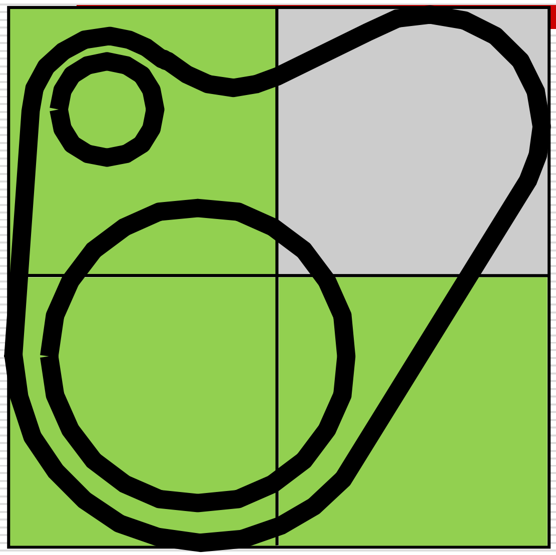


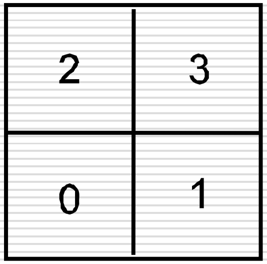


Quadrant Numbering

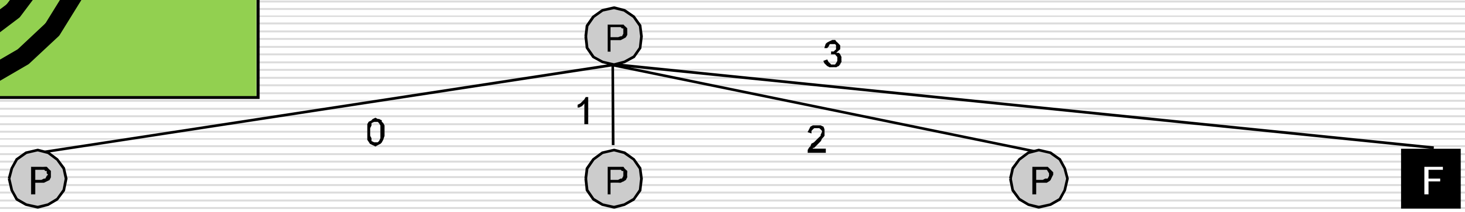


Quadtree Data Structure

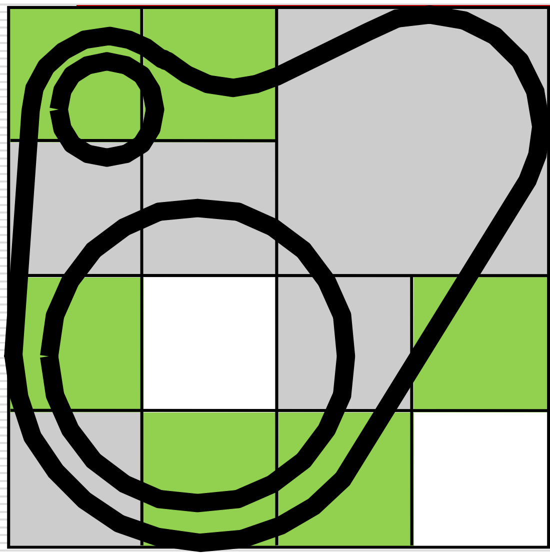


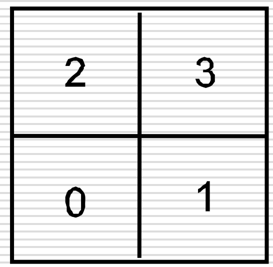


Quadrant Numbering

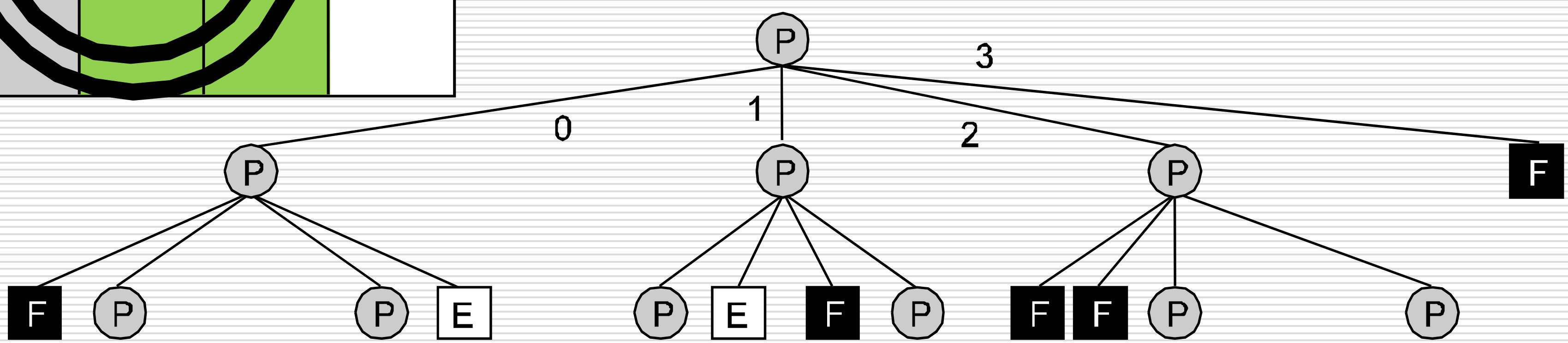


Quadtree Data Structure

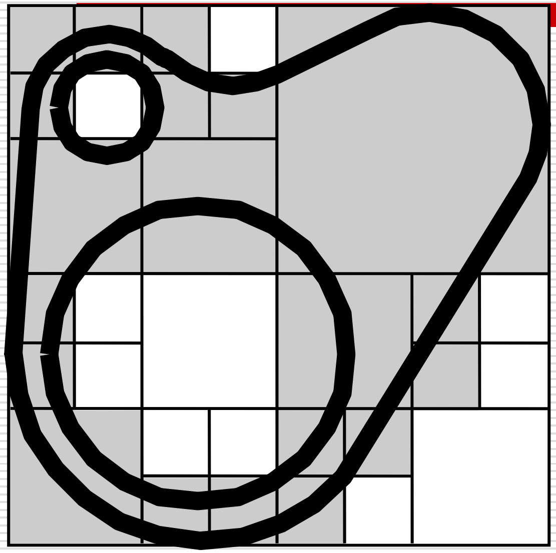


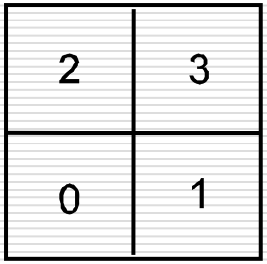


Quadrant Numbering

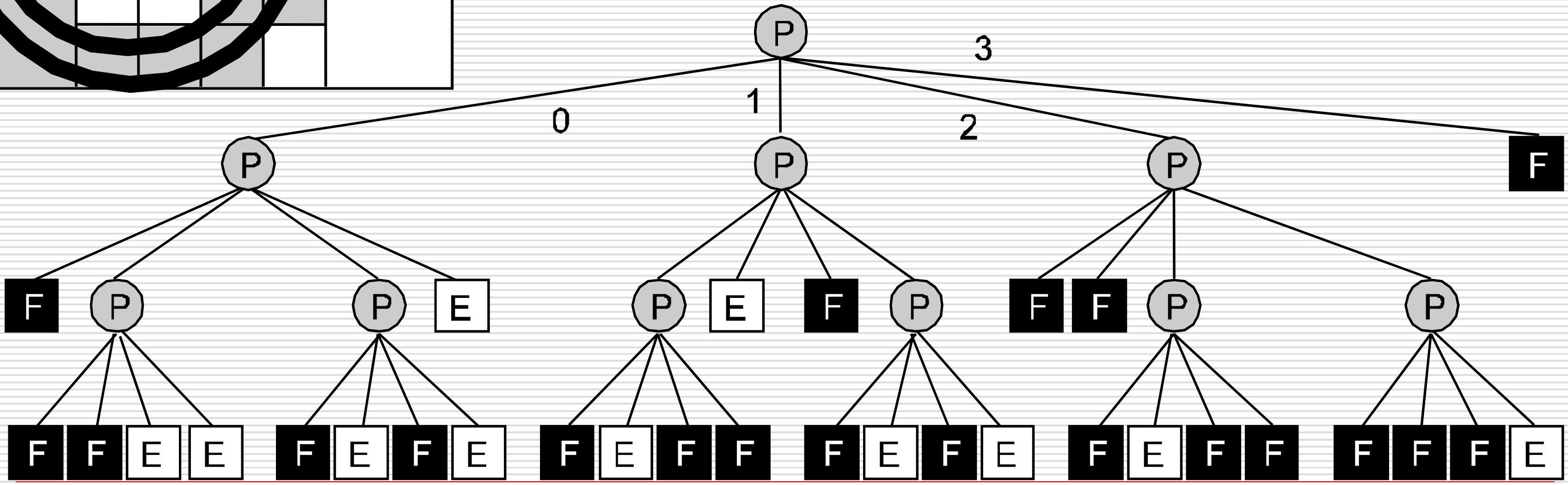


Quadtree Data Structure

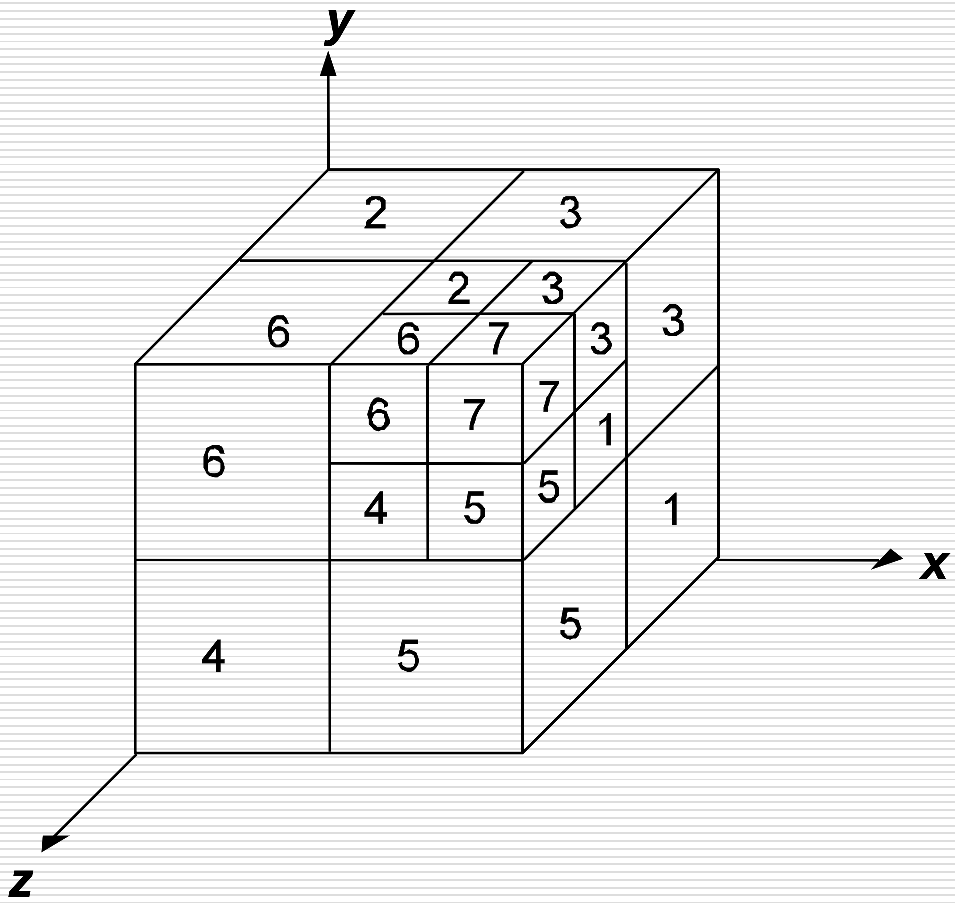




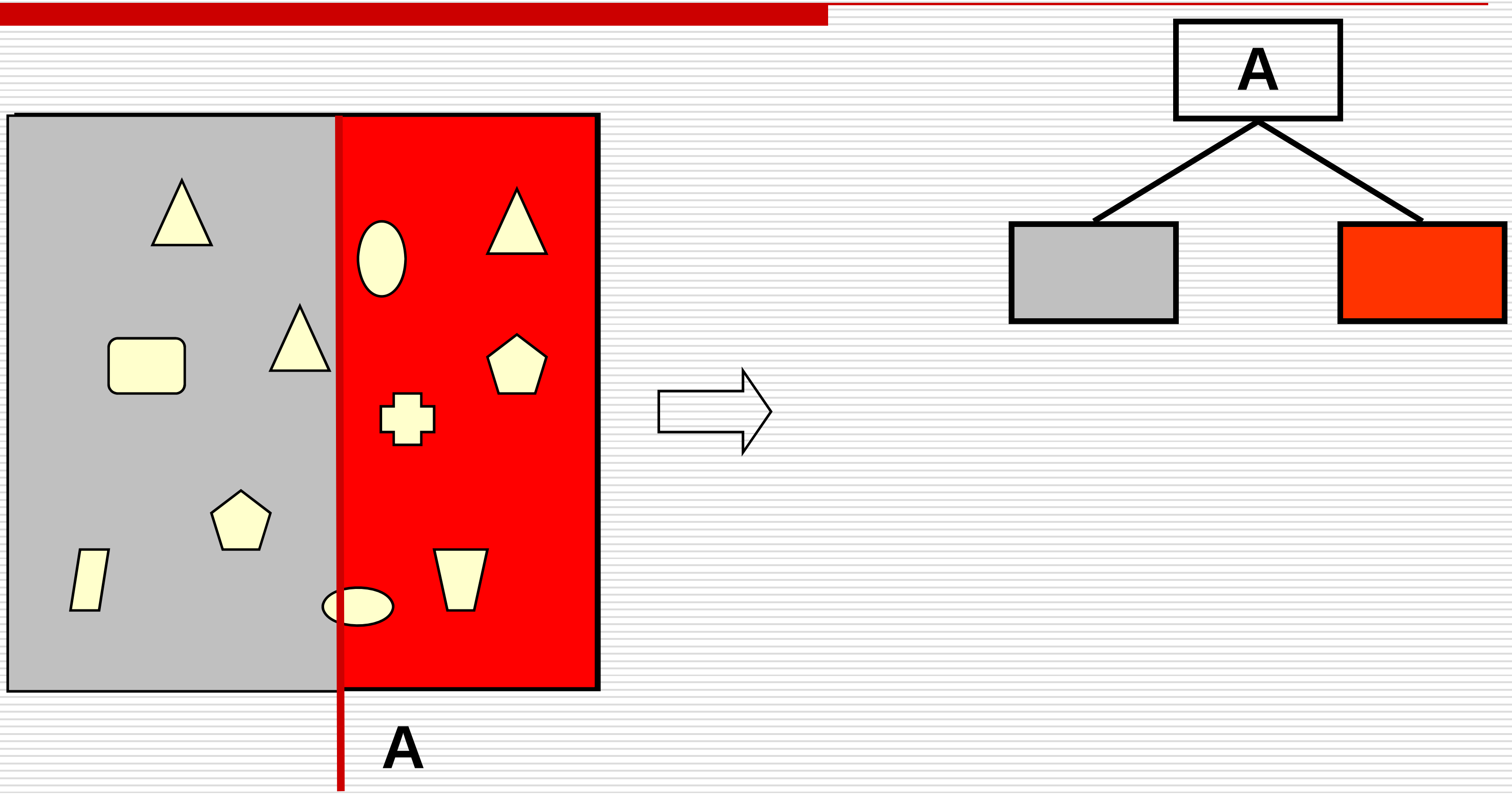
Quadrant Numbering



From Quadtree to Octree

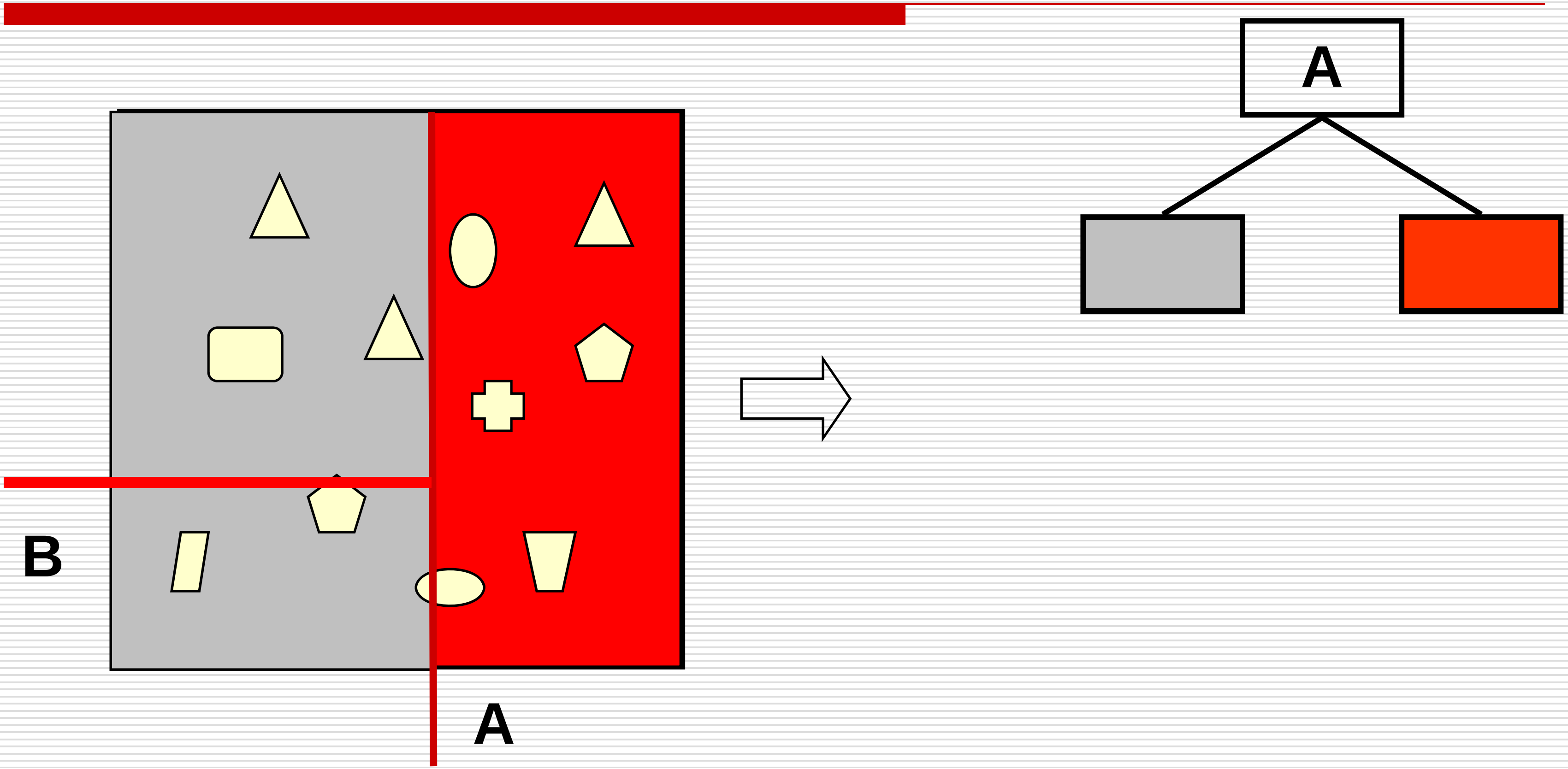


K-d Tree

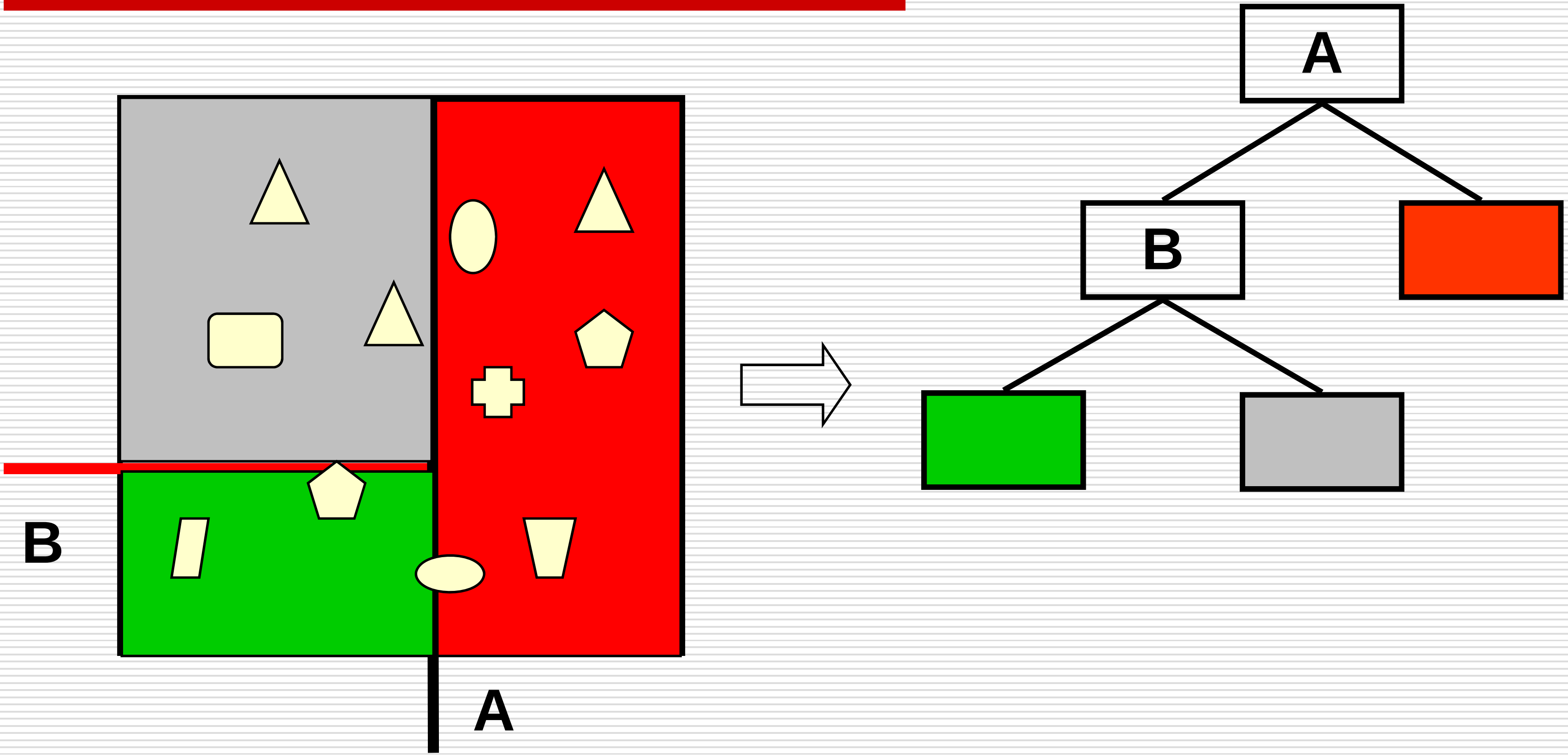


Leaf nodes correspond to unique regions in space

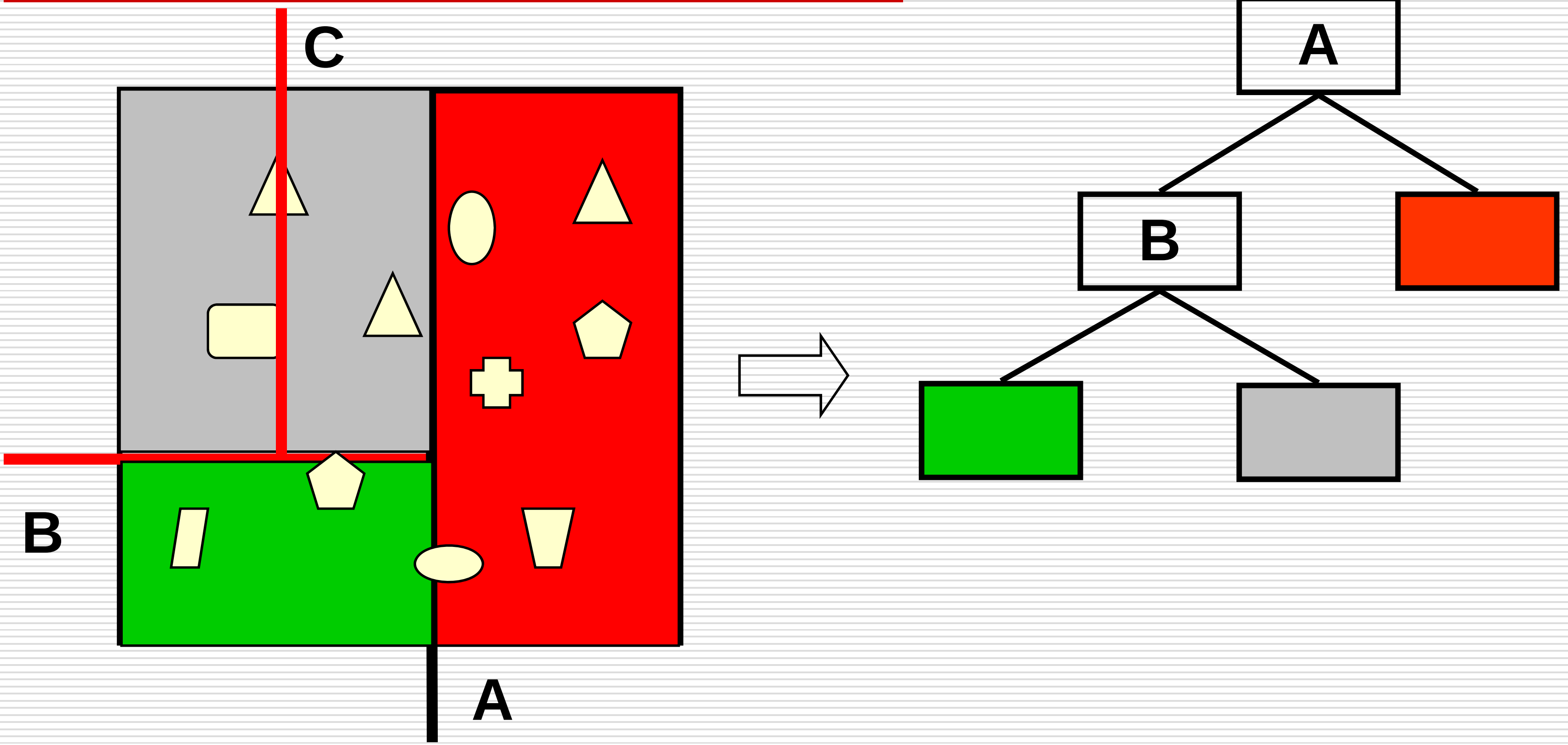
K-d Tree



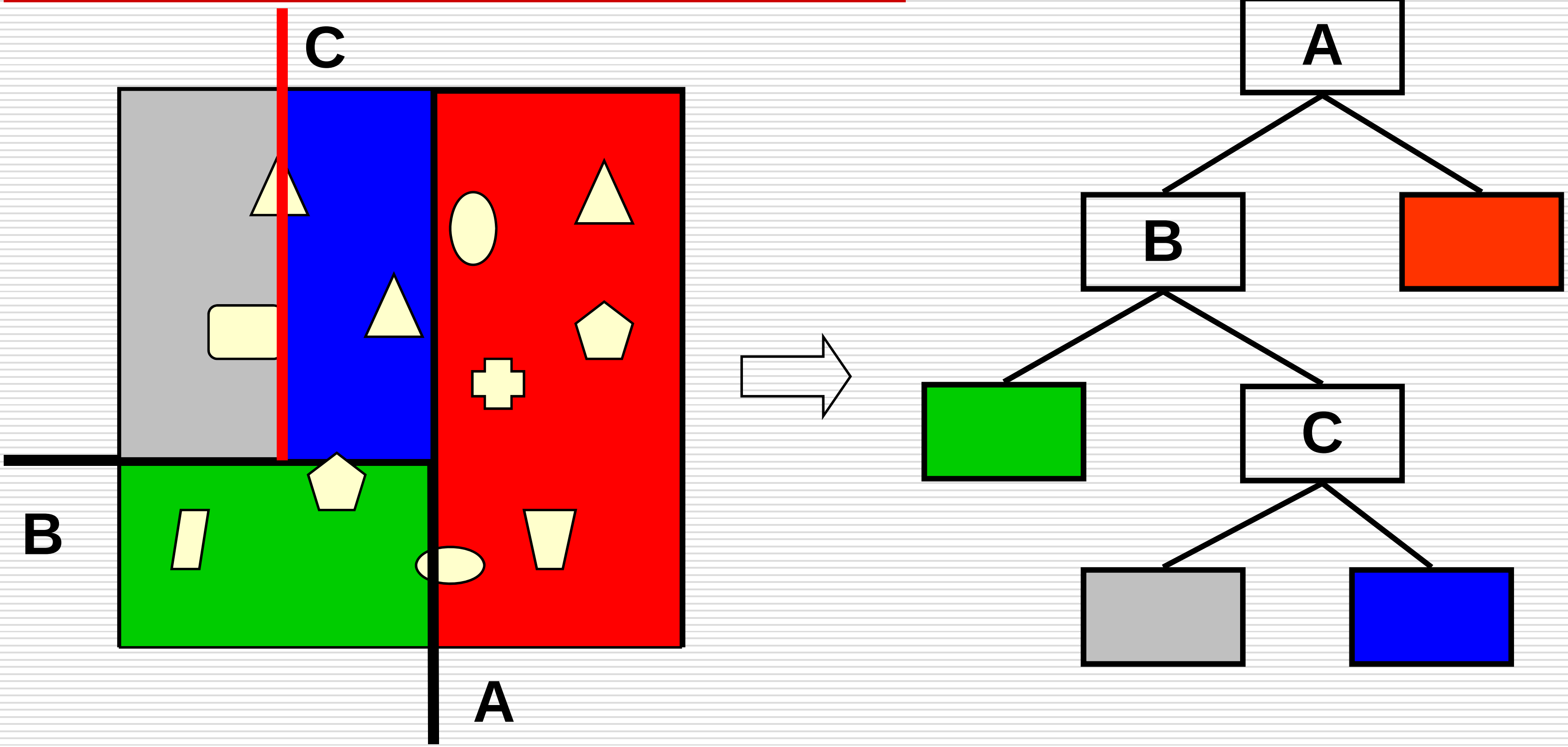
K-d Tree



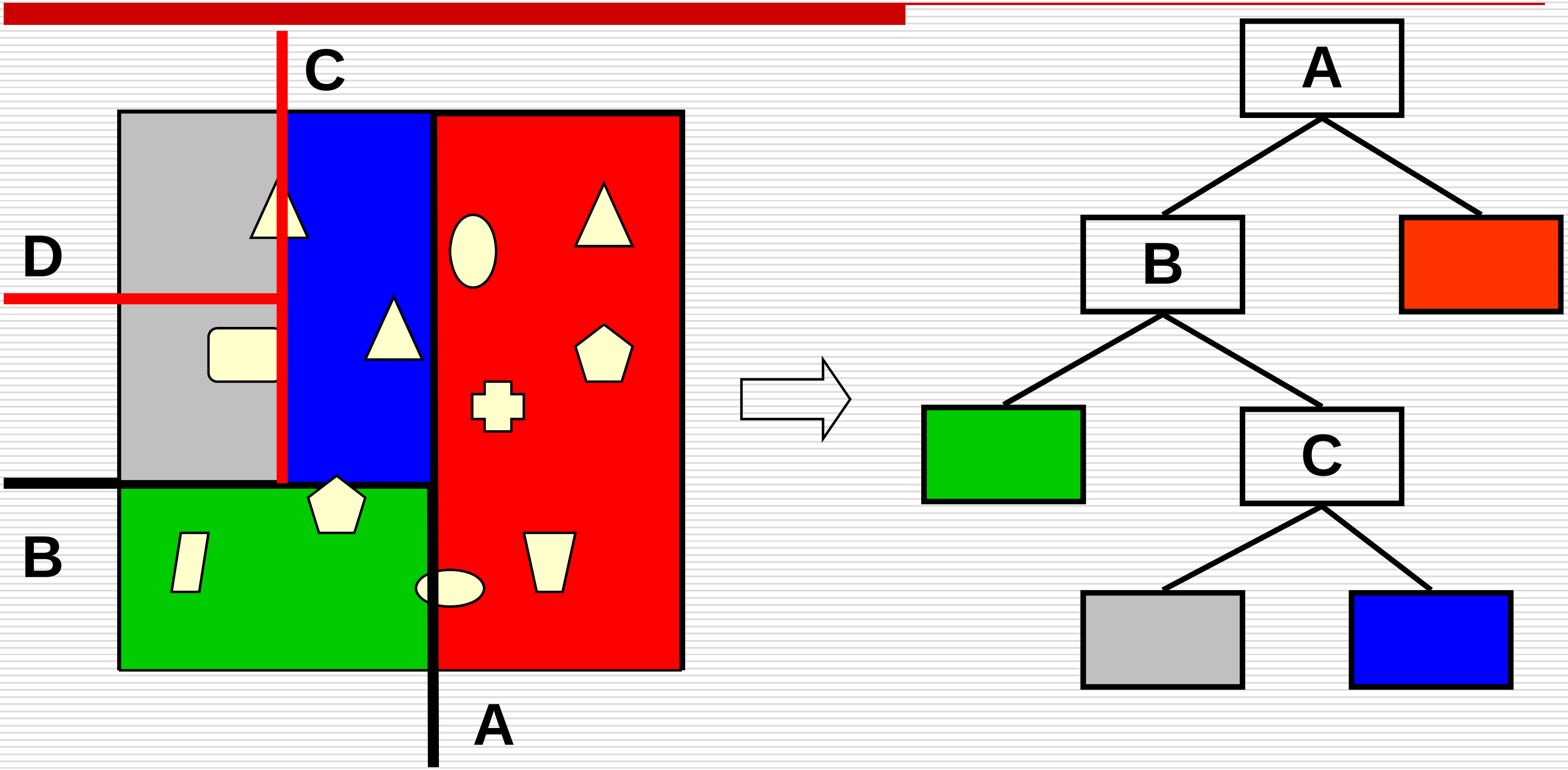
K-d Tree



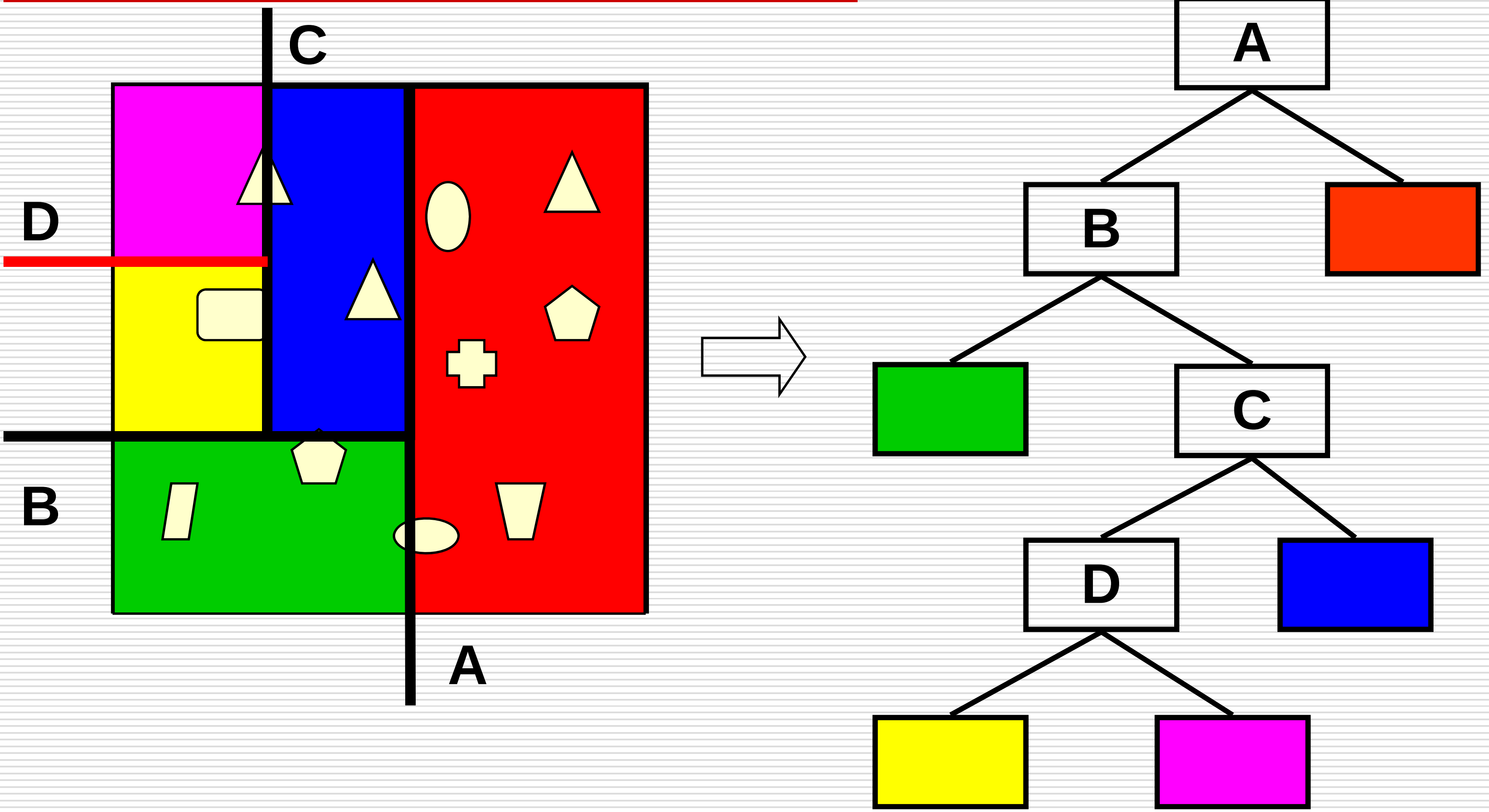
K-d Tree



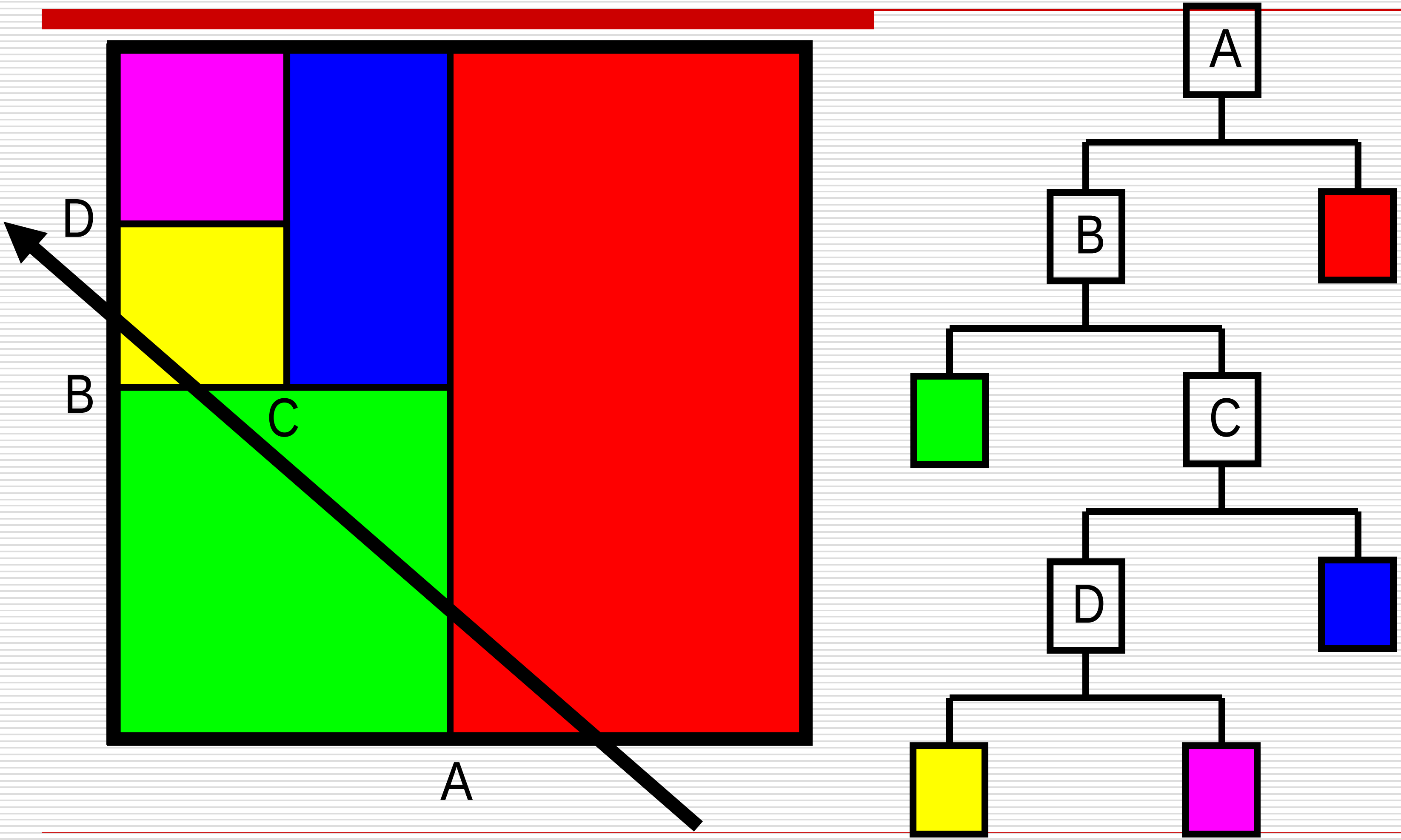
K-d Tree



K-d Tree

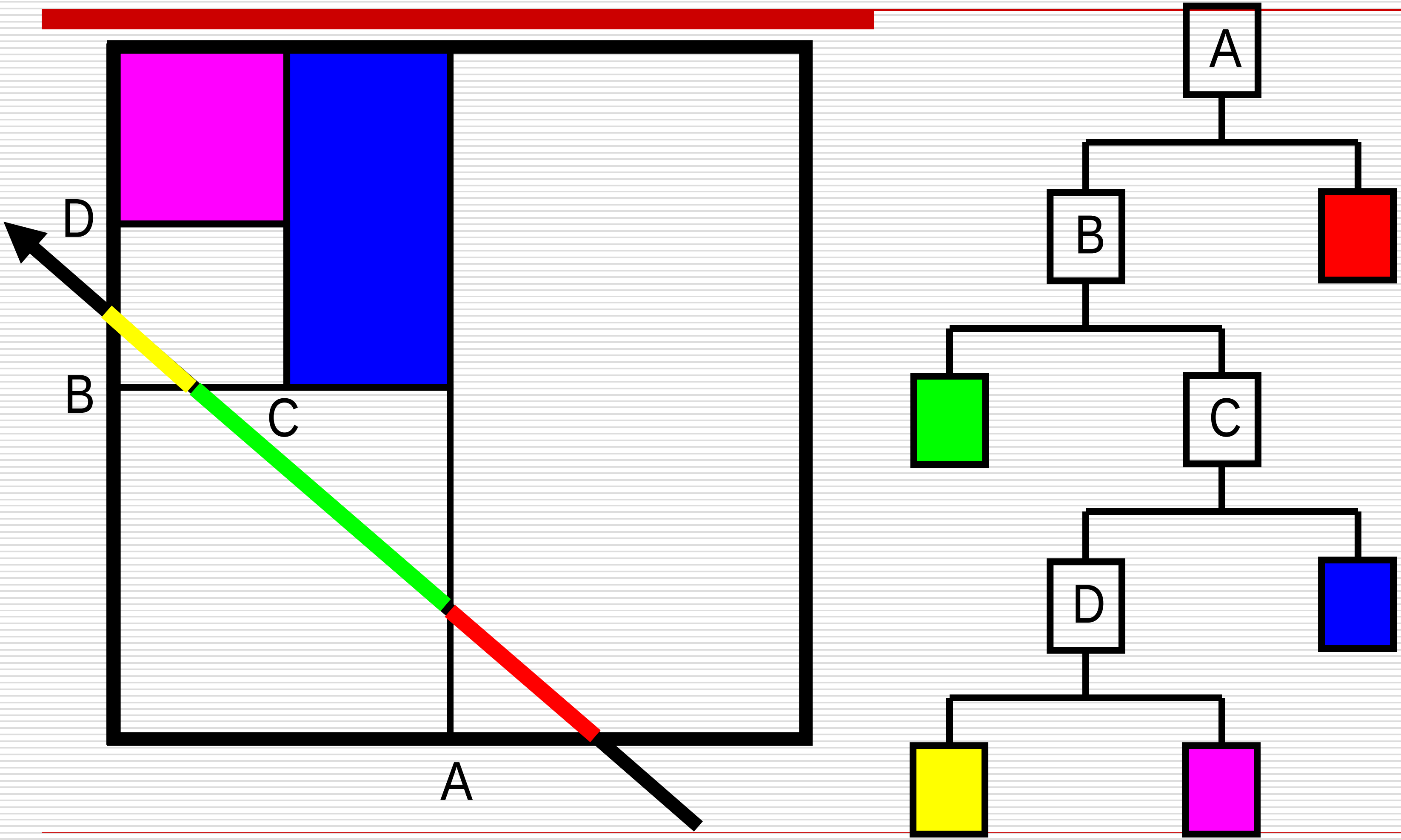


K-d Tree



Leaf nodes correspond to unique regions in space

K-d Tree Traversal



Leaf nodes correspond to unique regions in space