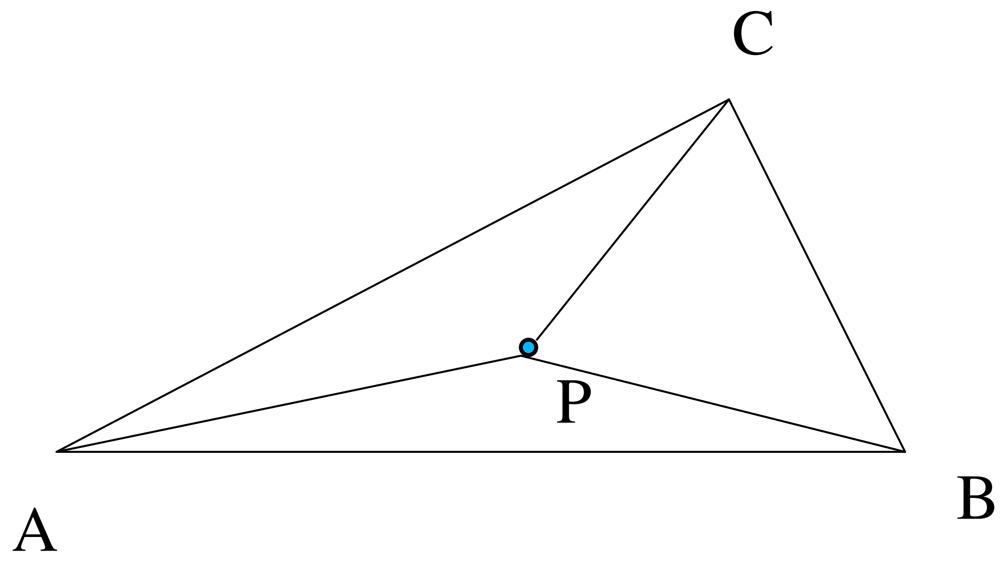
# Computer Graphics
## - Ray-Tracing II -

**Hendrik Lensch**

## Overview

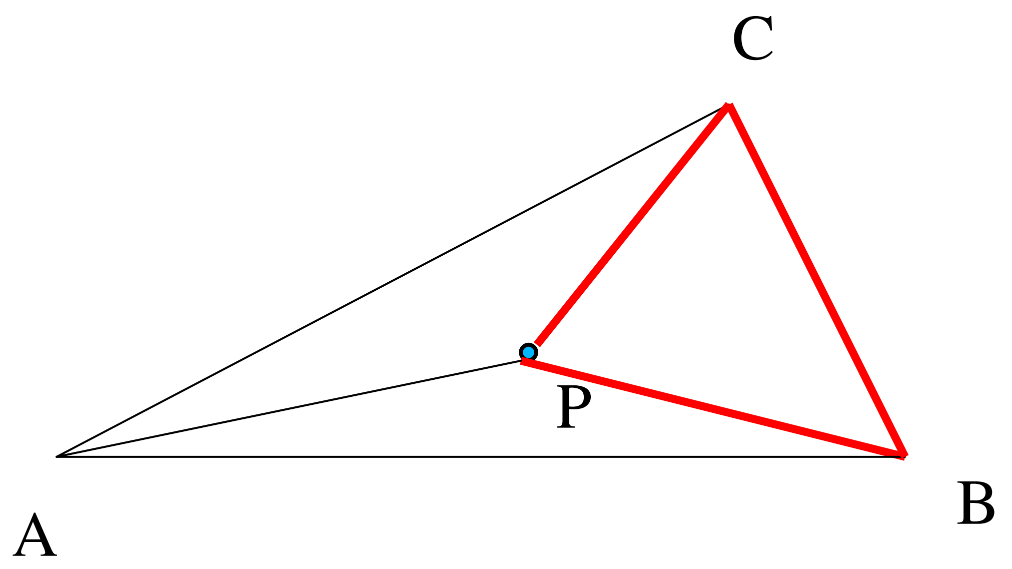
- **Last lecture**
  - Ray tracing I
    - Basic ray tracing
    - What is possible?
    - Recursive ray tracing algorithm
    - Intersection computations
- **Today**
  - Advanced acceleration structures
    - Theoretical Background
    - Hierarchical Grids, kd-Trees, Octrees
    - Bounding Volume Hierarchies
  - Dynamic changes to scenes
  - Ray bundles
- **Next lecture**
  - Realtime ray tracing

## Barycentric Coordinates



$$P = \lambda_1 A + \lambda_2 B + \lambda_3 B$$

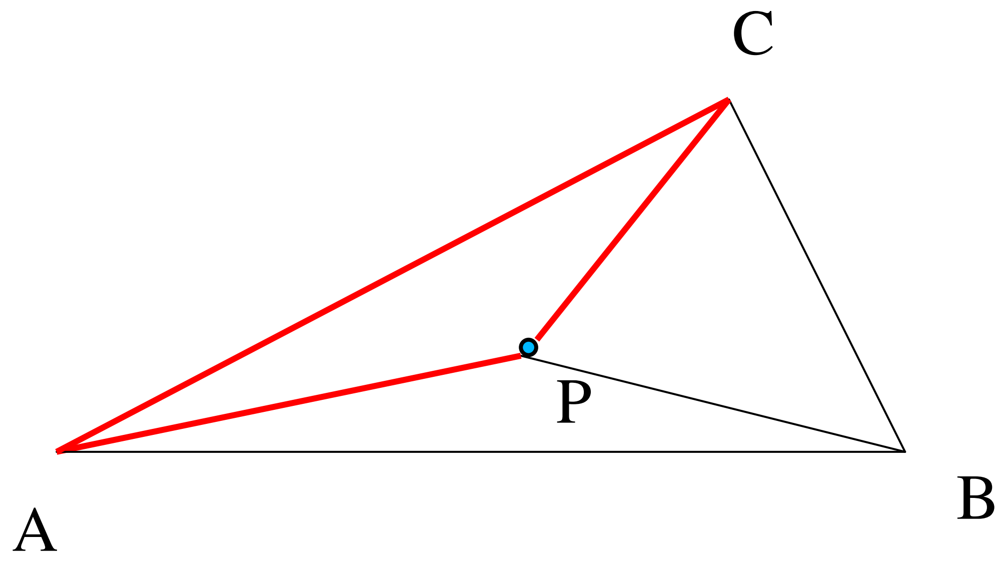
## Barycentric Coordinates



$$P = \lambda_1 A + \lambda_2 B + \lambda_3 B$$

$$\lambda_1 = \frac{S_A}{S_\Delta}$$

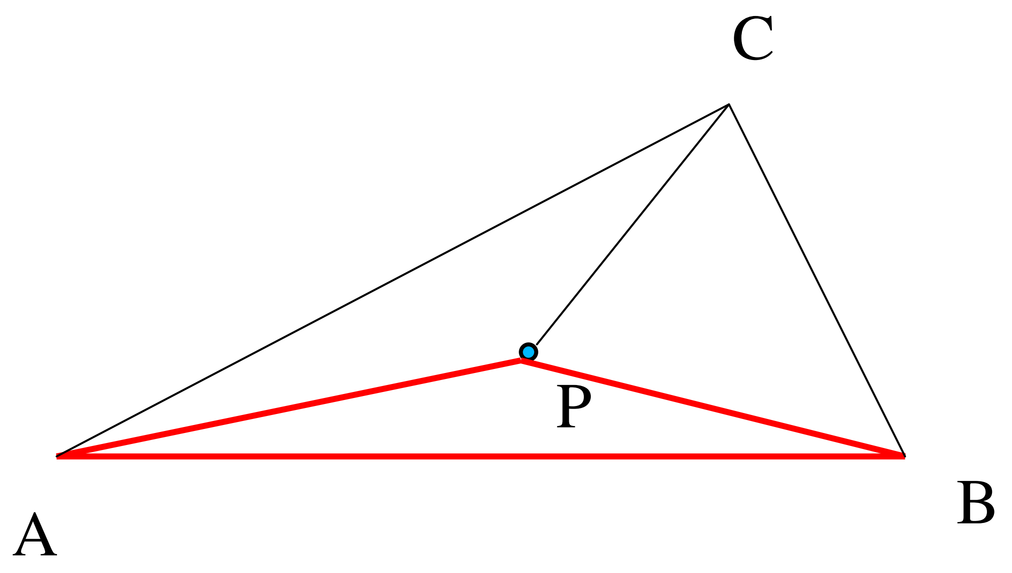
## Barycentric Coordinates



$$P = \lambda_1 A + \lambda_2 B + \lambda_3 B$$

$$\lambda_2 = \frac{S_B}{S_\Delta}$$

## Barycentric Coordinates

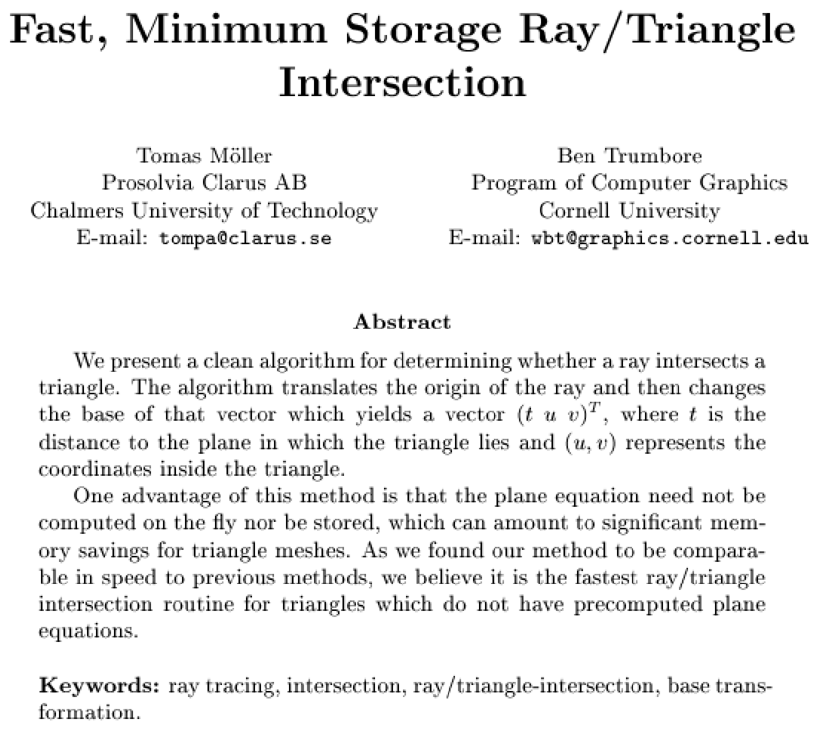


$$P = \lambda_1 A + \lambda_2 B + \lambda_3 B$$

$$\lambda_3 = \frac{S_C}{S_\Delta}$$

## Fast Triangle Intersection

- **see**



Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller
Prosolvia Clarus AB
Chalmers University of Technology
E-mail: `tompa@clarus.se`

Ben Trumbore
Program of Computer Graphics
Cornell University
E-mail: `wbt@graphics.cornell.edu`

**Abstract**

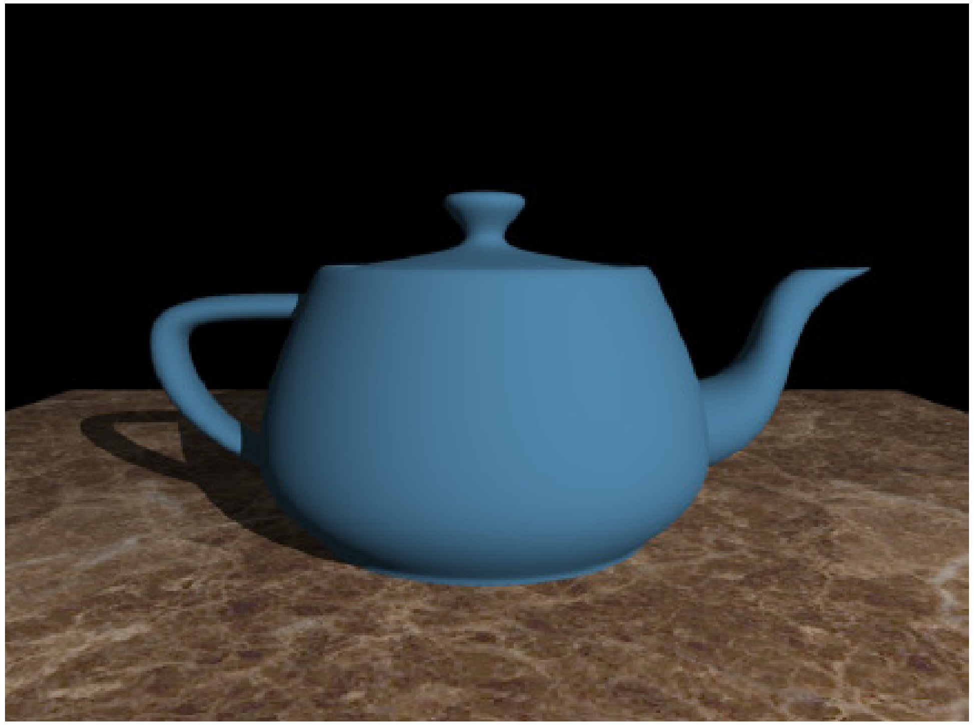
We present a clean algorithm for determining whether a ray intersects a triangle. The algorithm translates the origin of the ray and then changes the base of that vector which yields a vector $(t\ u\ v)^T$, where $t$ is the distance to the plane in which the triangle lies and $(u, v)$ represents the coordinates inside the triangle.

One advantage of this method is that the plane equation need not be computed on the fly nor be stored, which can amount to significant memory savings for triangle meshes. As we found our method to be comparable in speed to previous methods, we believe it is the fastest ray/triangle intersection routine for triangles which do not have precomputed plane equations.

**Keywords:** ray tracing, intersection, ray/triangle-intersection, base transformation.

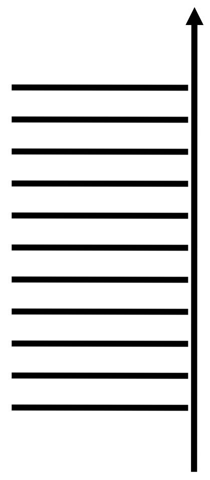
## Cost of Naïve Raytracing

- **Linear complexity**
- **1024 * 768 pixels * 6320 triangles !**



## Theoretical Background

- **Unstructured data results in (at least) linear complexity**
  - Every primitive could be the first one intersected
  - Must test each one separately
  - Coherence does not help
- **Reduced complexity only through pre-sorted data**
  - Spatial sorting of primitives (indexing like for data base)
    - Allows for efficient search strategies
  - Hierarchy leads to O(log n) search complexity
    - But building the hierarchy is still O(n log n)
  - Trade-off between run-time and building-time
    - In particular for dynamic scenes
  - Worst case scene is still linear !!
- **It is a general problem in graphics**
  - Spatial indices for ray tracing
  - Spatial indices for occlusion- and frustum-culling
  - Sorting for transparency

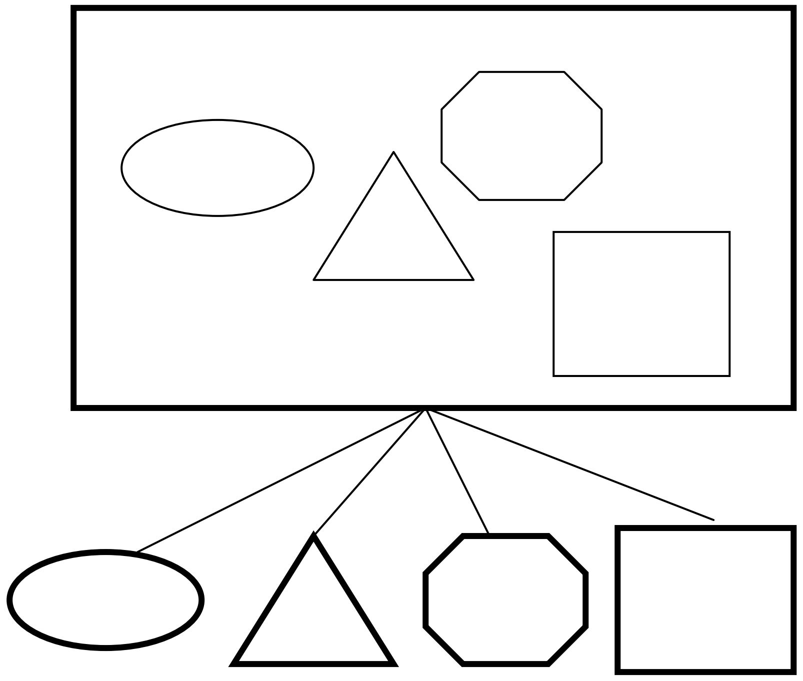

Worst case RT scene:
Ray barely misses
every primitive

## Ray Tracing Acceleration

- **Intersect ray with all objects**
  - Way too expensive
- **Faster intersection algorithms**
  - Still same complexity O(n)
- **Less intersection computations**
  - Space partitioning (often hierarchical)
    - Grid, hierarchies of grids
    - Octree
    - Binary space partition (BSP) or kd-tree
    - Bounding volume hierarchy (BVH)
  - Directional partitioning (not very useful)
  - 5D partitioning (space and direction, once a big hype)
    - Close to pre-compute visibility for all points and all directions
- **Tracing of continuous bundles of rays**
  - Exploits coherence of neighboring rays, amortize cost among them
    - Cone tracing, beam tracing, ...

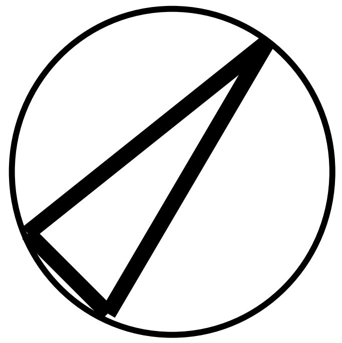
## Aggregate Objects

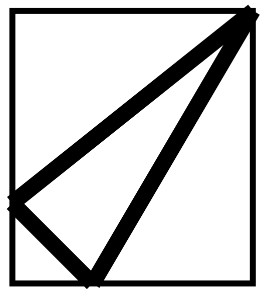
- Object that holds groups of objects
- Stores bounding box and pointers to children
- Useful for instancing and Bounding Volume Hierarchies

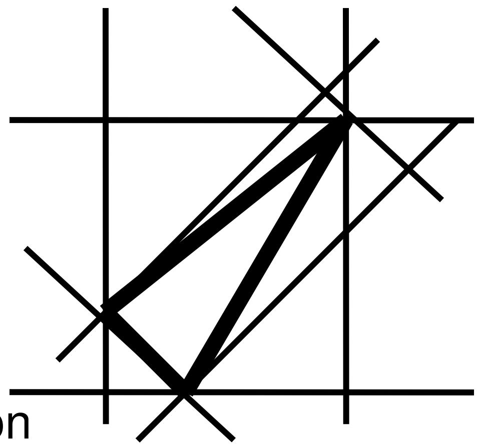


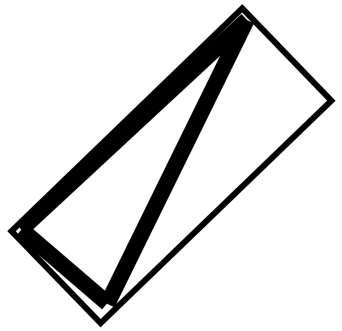
## Bounding Volumes (BV)

- **Observation**
  - Bound geometry with BV
  - Only compute intersection if ray hits BV
- **Sphere**
  - Very fast intersection computation
  - Often inefficient because too large
- **Axis-aligned box**
  - Very simple intersection computation (min-max)
  - Sometimes too large
- **Non-axis-aligned box**
  - A.k.a. „oriented bounding box (OBB)“
  - Often better fit
  - Fairly complex computation
- **Slabs**
  - Pairs of half spaces
  - Fixed number of orientations
    - Addition of coordinates w/ negation
  - Fairly fast computation

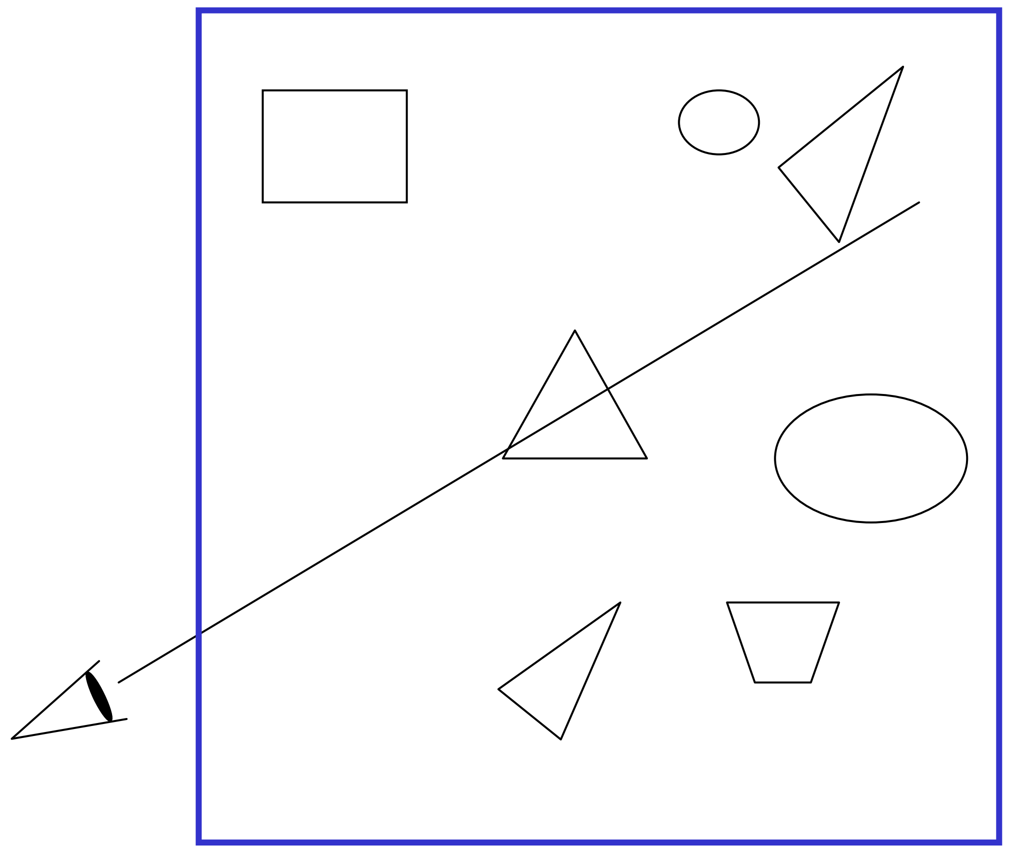






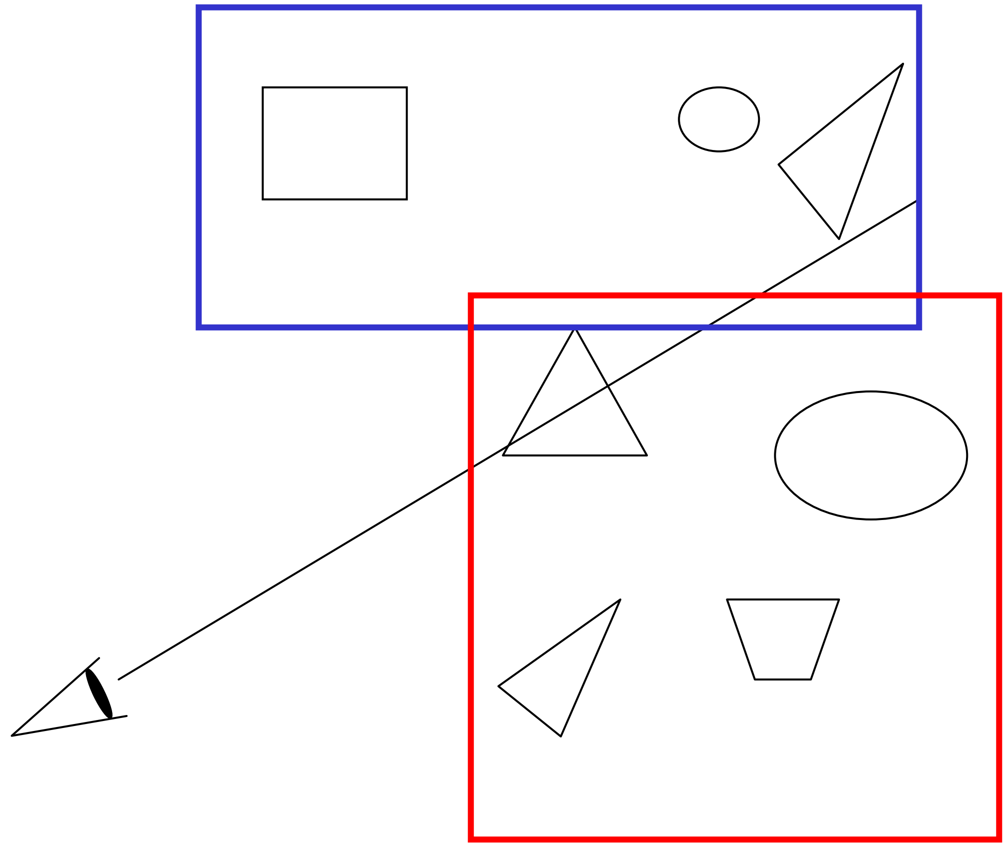


## Bounding Volume Hierarchies



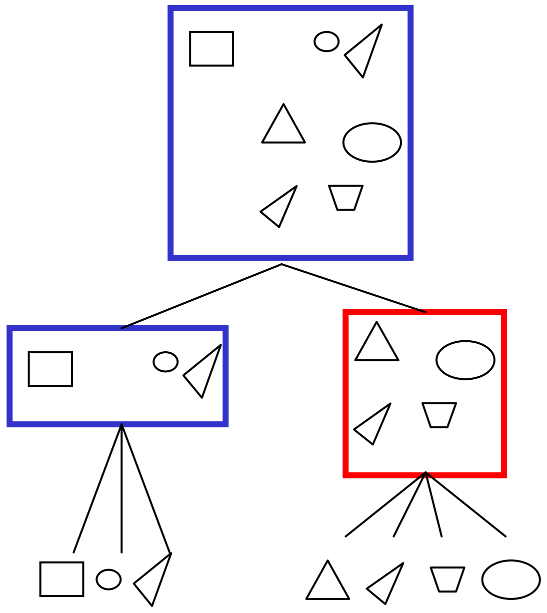
## Bounding Volume Hierarchies

- **Hierarchy of groups of objects**



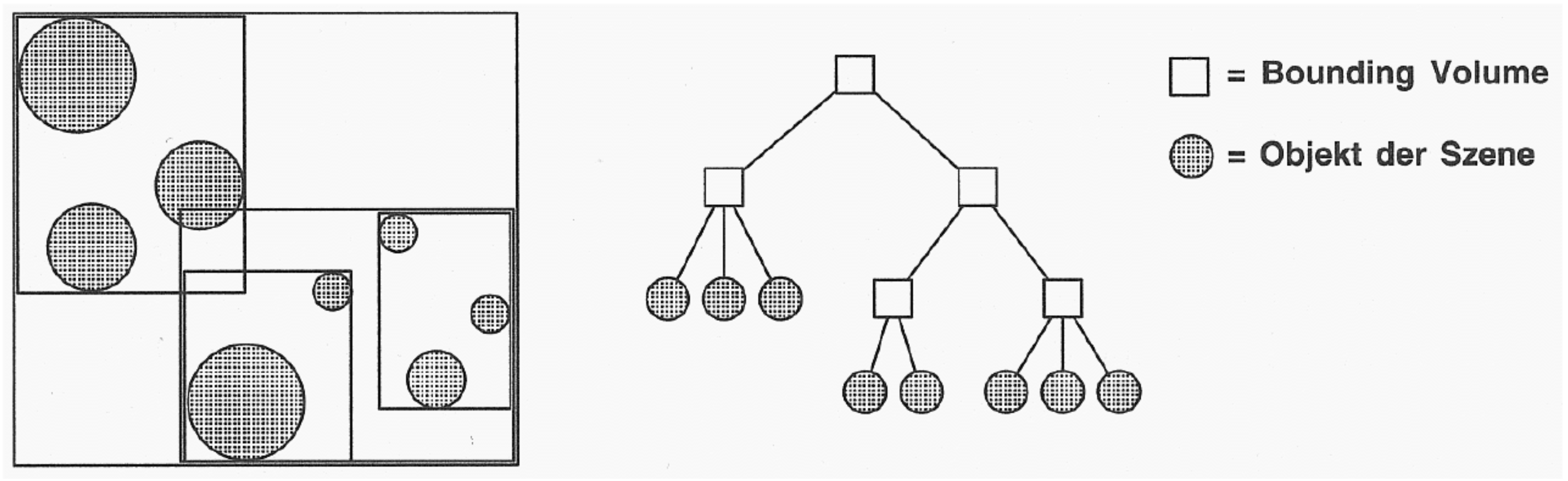
## Bounding Volume Hierarchies

- **Tree structure**
- **Internal nodes are aggregate objects**
- **Leave nodes are geometric objects**
- **Intersection testing involves tree traversal**



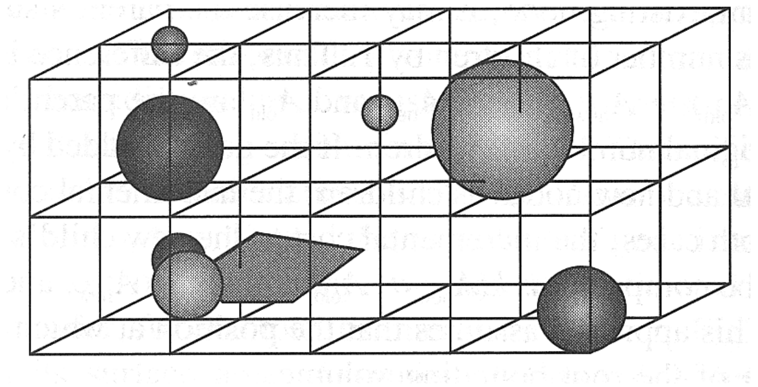
## Bounding Volume Hierarchies

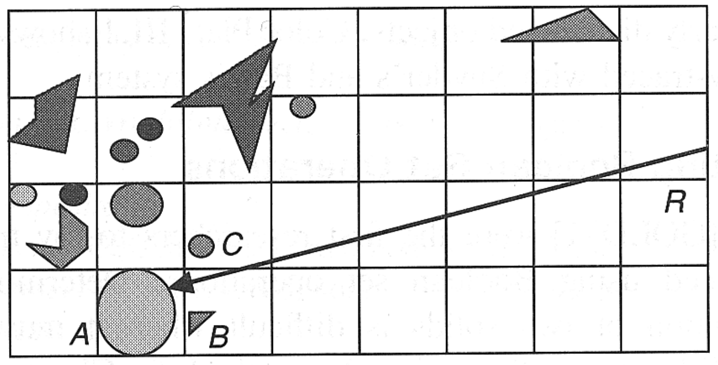
- **Idea:**
  - Organize bounding volumes hierarchically into new BVs
- **Advantages:**
  - Very good adaptivity
  - Efficient traversal O(log N)
  - Often used in ray tracing systems
- **Problems**
  - How to arrange BVs?



# Grid

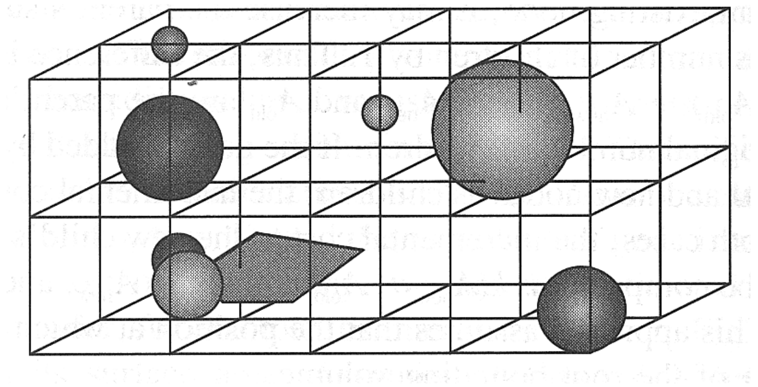
- **Grid**
  - Partitioning with equal, fixed sized „voxels“
- **Building a grid structure**
  - Partition the bounding box (bb)
  - Resolution: often $\sqrt[3]{n}$
  - Inserting objects
    - Trivial: insert into all voxels overlapping objects bounding box
    - Easily optimized
- **Traversal**
  - Iterate through all voxels in order as pierced by the ray
  - Compute intersection with objects in each voxel
  - Stop if intersection found in current voxel

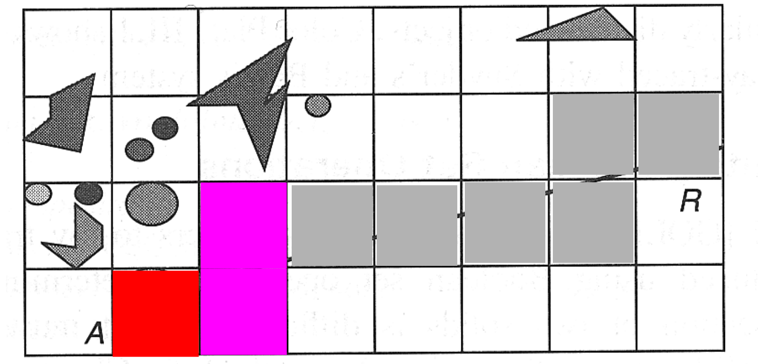




# Grid

- **Grid**
  - Partitioning with equal, fixed sized „voxels“
- **Building a grid structure**
  - Partition the bounding box (bb)
  - Resolution: often $\sqrt[3]{n}$
  - Inserting objects
    - Trivial: insert into all voxels overlapping objects bounding box
    - Easily optimized
- **Traversal**
  - Iterate through all voxels in order as pierced by the ray
  - Compute intersection with objects in each voxel
  - Stop if intersection found in current voxel



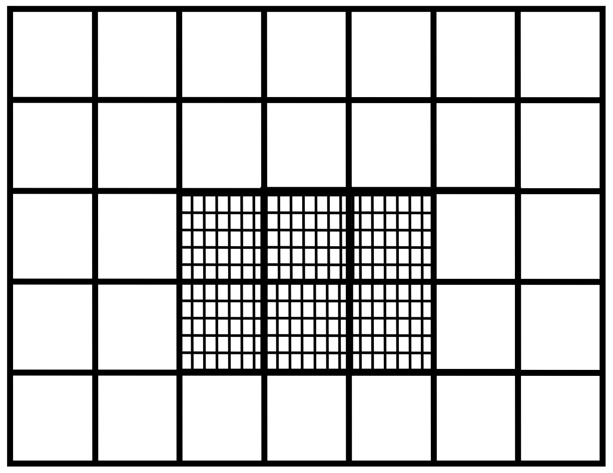
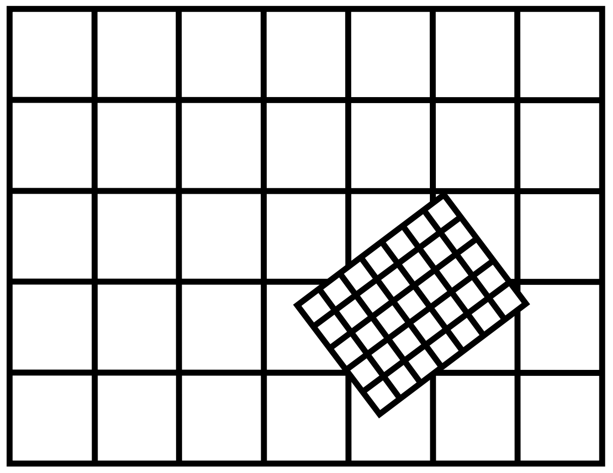


## Grid: Issues

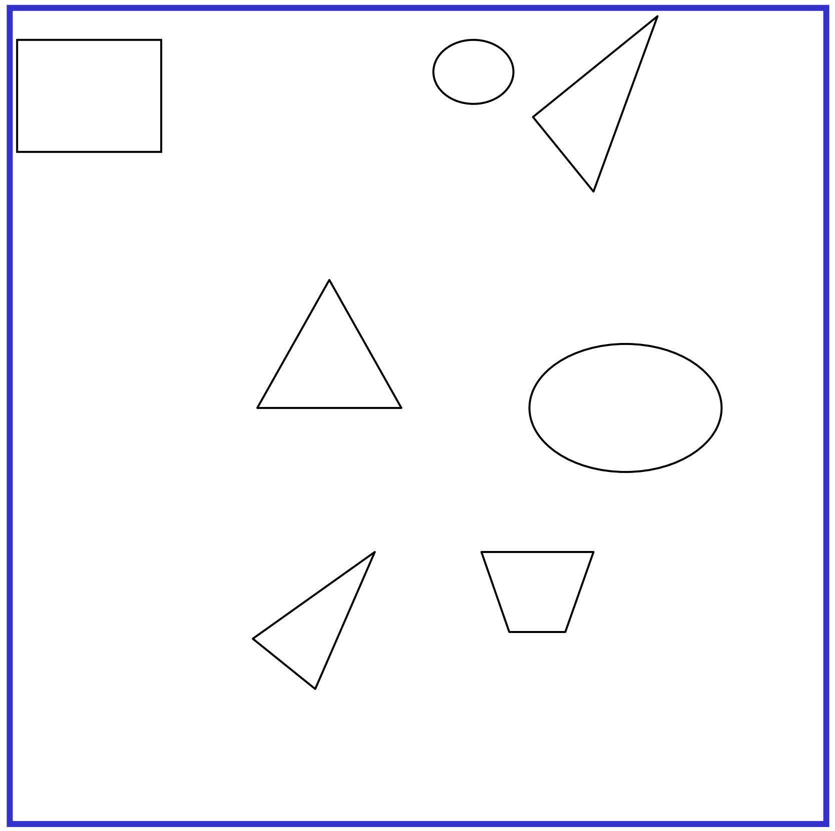
- **Grid traversal**
  - Requires enumeration of voxel along ray
    → 3D-DDA, modified Bresenham (later)
  - Simple and hardware-friendly
- **Grid resolution**
  - Strongly scene dependent
  - Cannot adapt to local density of objects
    - Problem: „Teapot in a stadium“
  - Possible solution: grids within grids ➔ hierarchical grids
- **Objects spanning multiple voxels**
  - Store only references to objects
  - Use mailboxing to avoid multiple intersection computations
    - Store object in small per-ray cache (e.g. with hashing)
    - Do not intersect again if found in cache
  - Original mailbox stores ray-id with each triangle
    - Simple, but likely to destroy CPU caches

# Hierarchical Grids

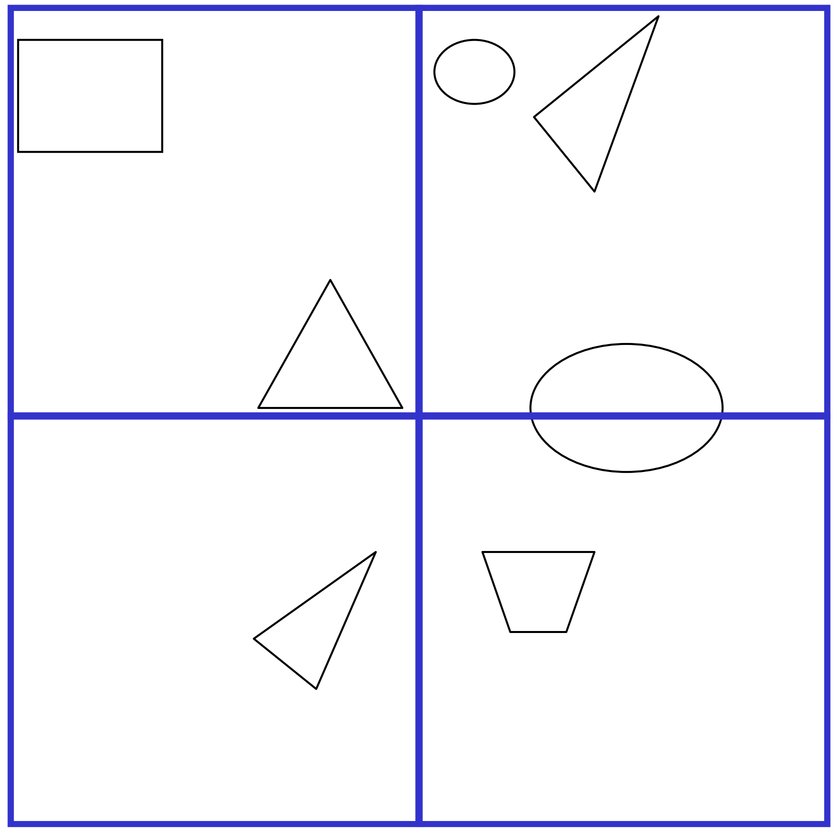
- **Simple building algorithm**
  - Coarse grid for entire scene
  - Recursively create grids in high-density voxels
  - Problem: What is the right resolution for each level?
- **Advanced algorithm**
  - Place cluster of objects in separate grids
  - Insert these grids into parent grid
  - Problem: What are good clusters?

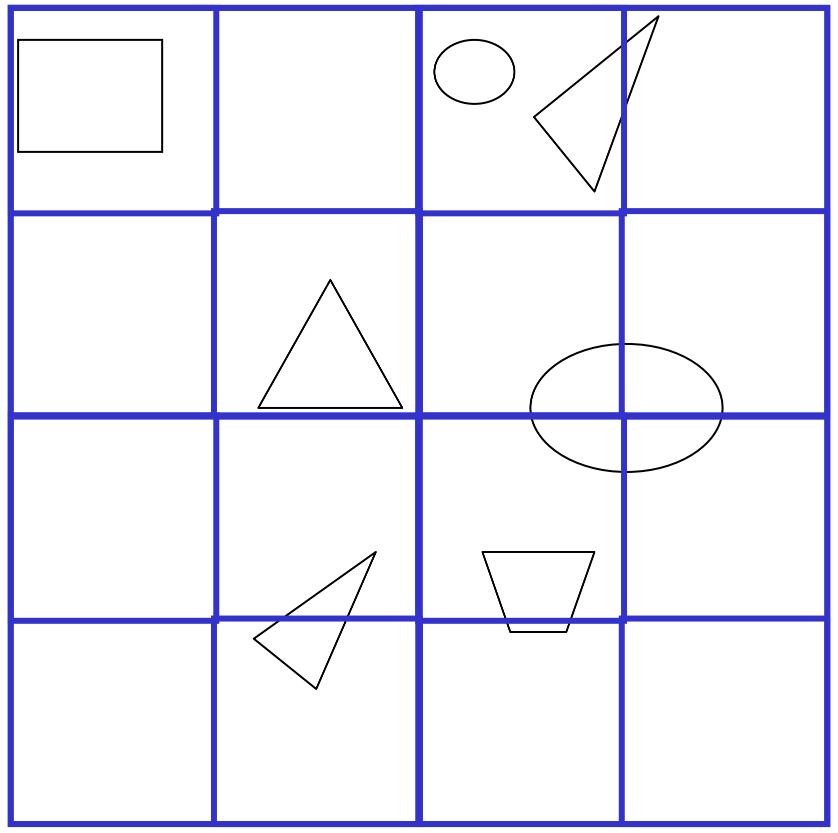
## Quadtree – 2D example



## Quadtree – 2D example

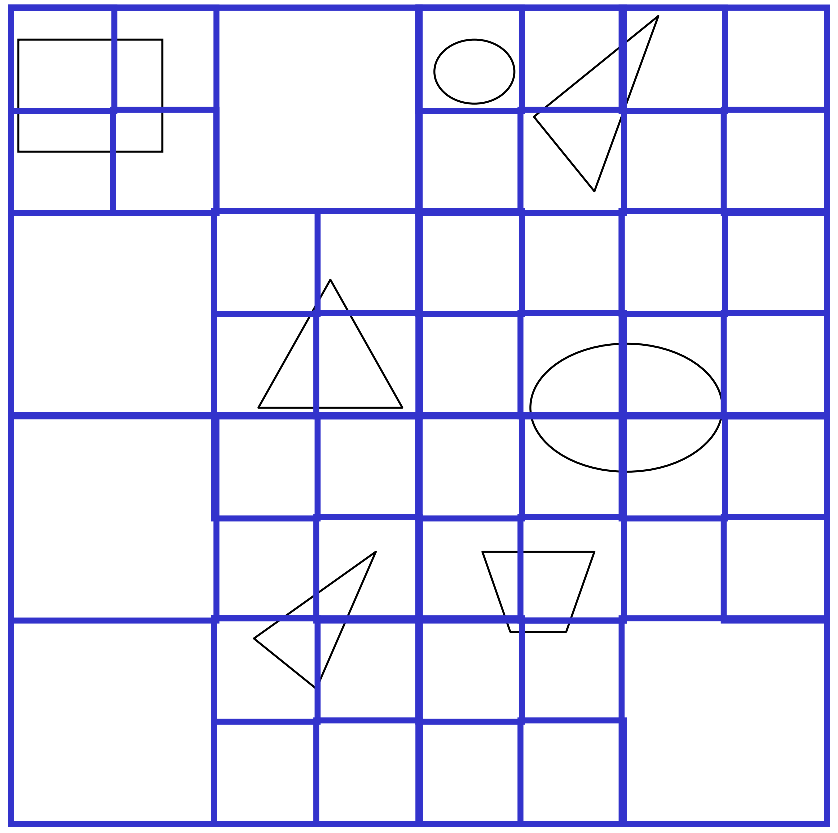


## Quadtree – 2D example



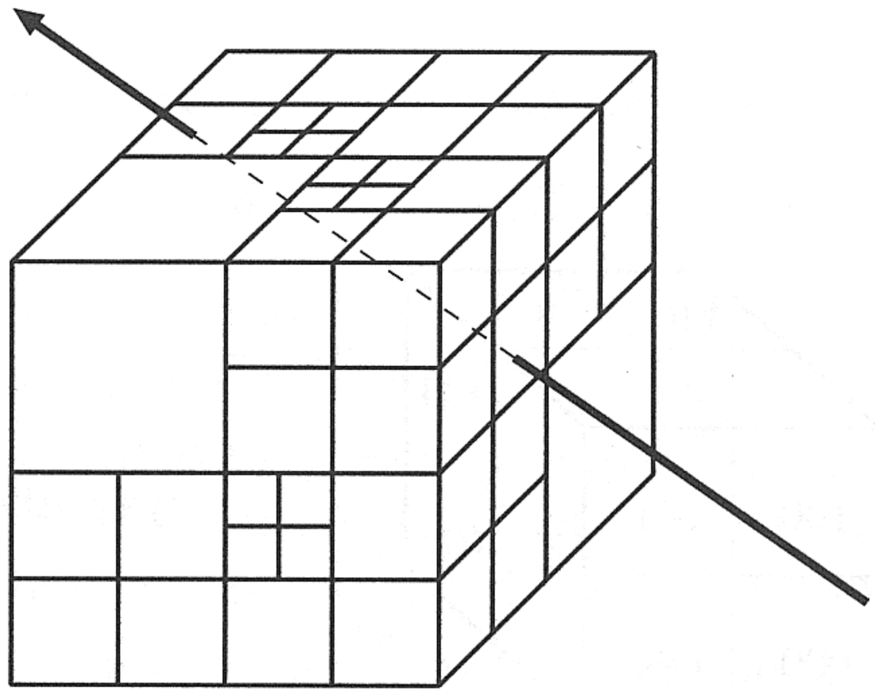
## Quadtree – 2D example

- **Hierarchical subdivision**
- **subdivide unless cell empty (or less then N primitives)**



## Octree

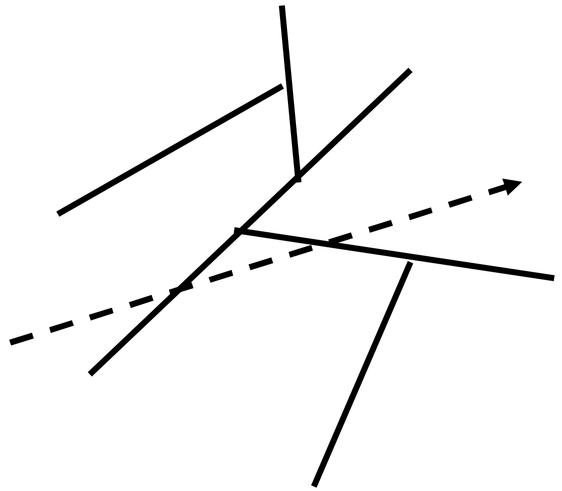
- **Hierarchical space partitioning**
  - Start with bounding box of entire scene
  - Recursively subdivide voxels into 8 equal sub-voxels
  - Subdivision criteria:
    - Number of remaining primitives and maximum depth
  - Result in adaptive subdivision
    - Allows for large traversal steps in empty regions
- **Problems**
  - Pretty complex traversal algorithms
  - Slow to refine complex regions
- **Traversal algorithms**
  - HERO, SMART, ...
  - Or use kd-tree algorithm …

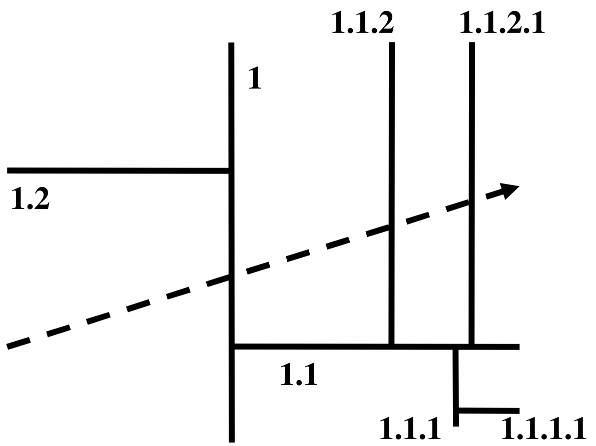


# BSP- and Kd-Trees

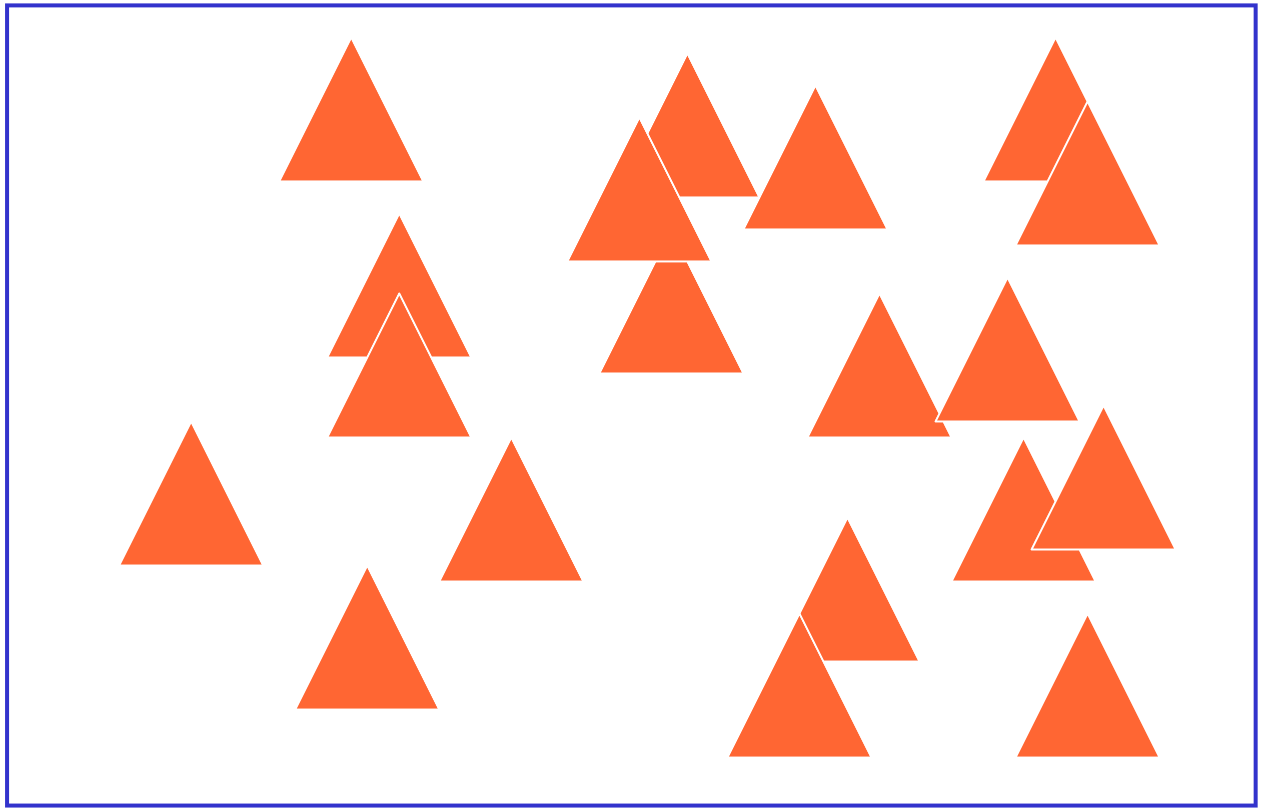
- **Recursive space partitioning with half-spaces**
- **Binary Space Partition (BSP):**
  - Recursively split space into halves
  - Splitting with half-spaces in arbitrary position
    - Often defined by existing polygons
  - Often used for visibility in games (→ Doom)
    - Traverse binary tree from front to back
- **Kd-Tree**
  - Special case of BSP
    - Splitting with axis-aligned half-spaces
  - Defined recursively through nodes with
    - Axis-flag
    - Split location (1D)
    - Child pointer(s)

  - See separate slides for details



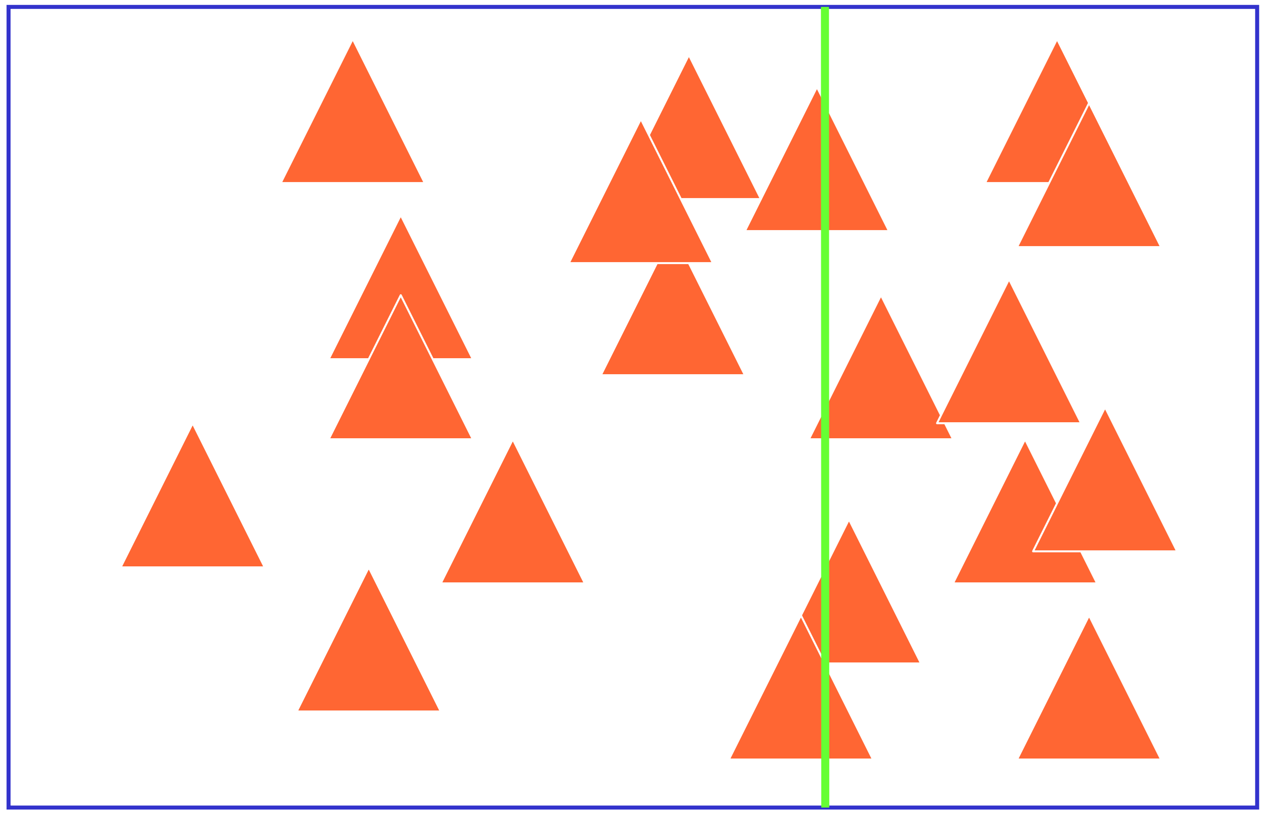


## kD-Trees

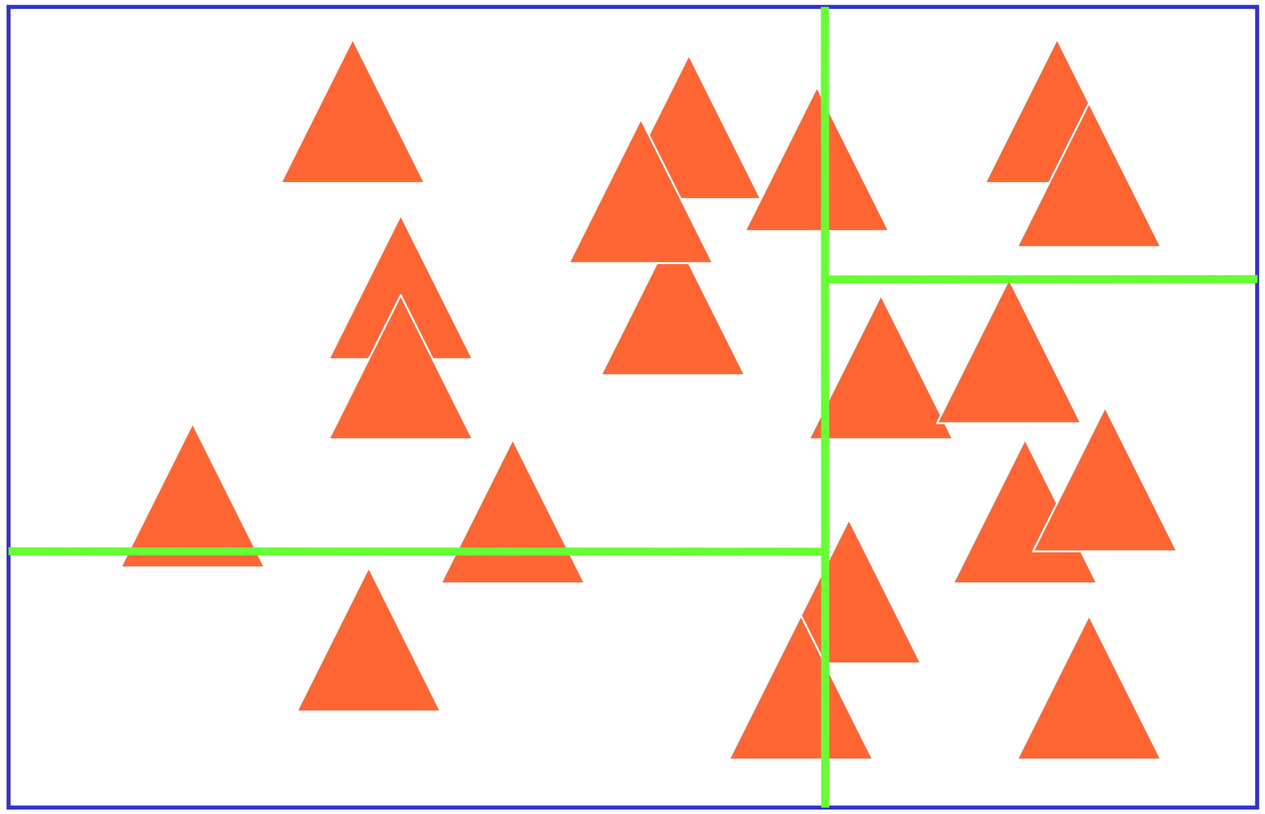


[following slides from Gordon Stoll – SIGGRAPH Course 2005]

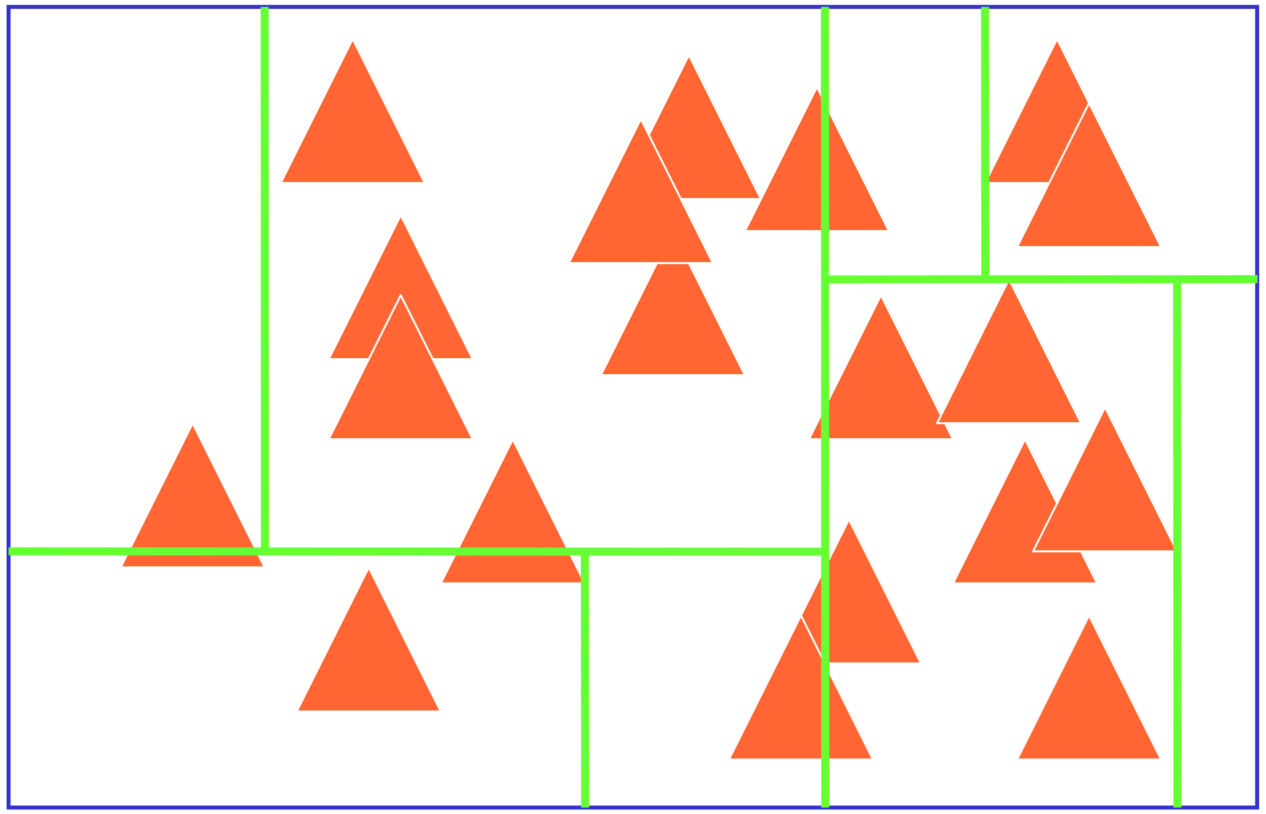
## kD-Trees



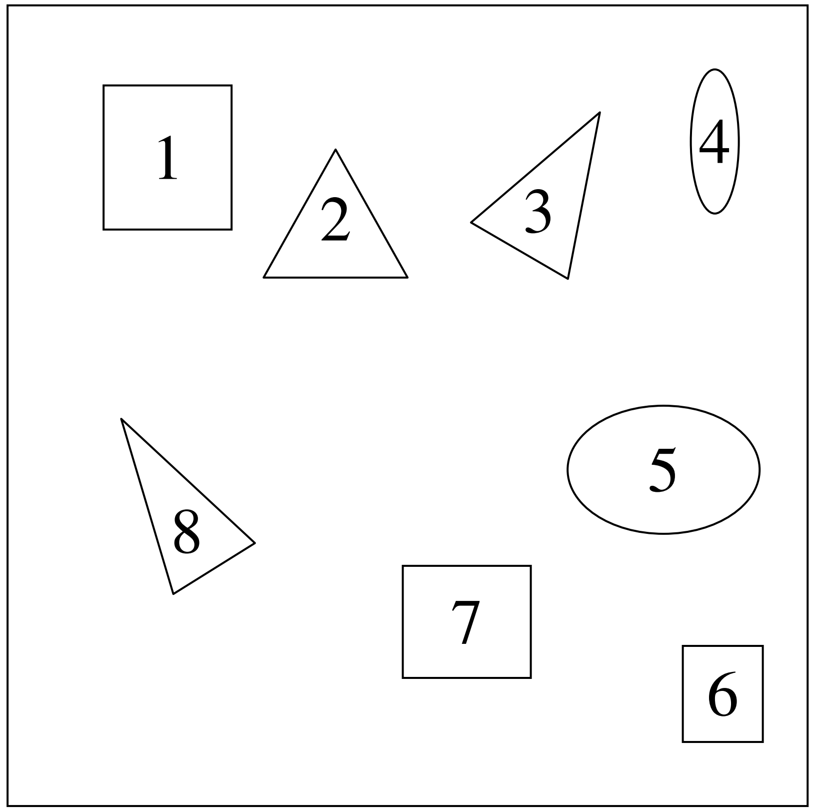
# kD-Trees



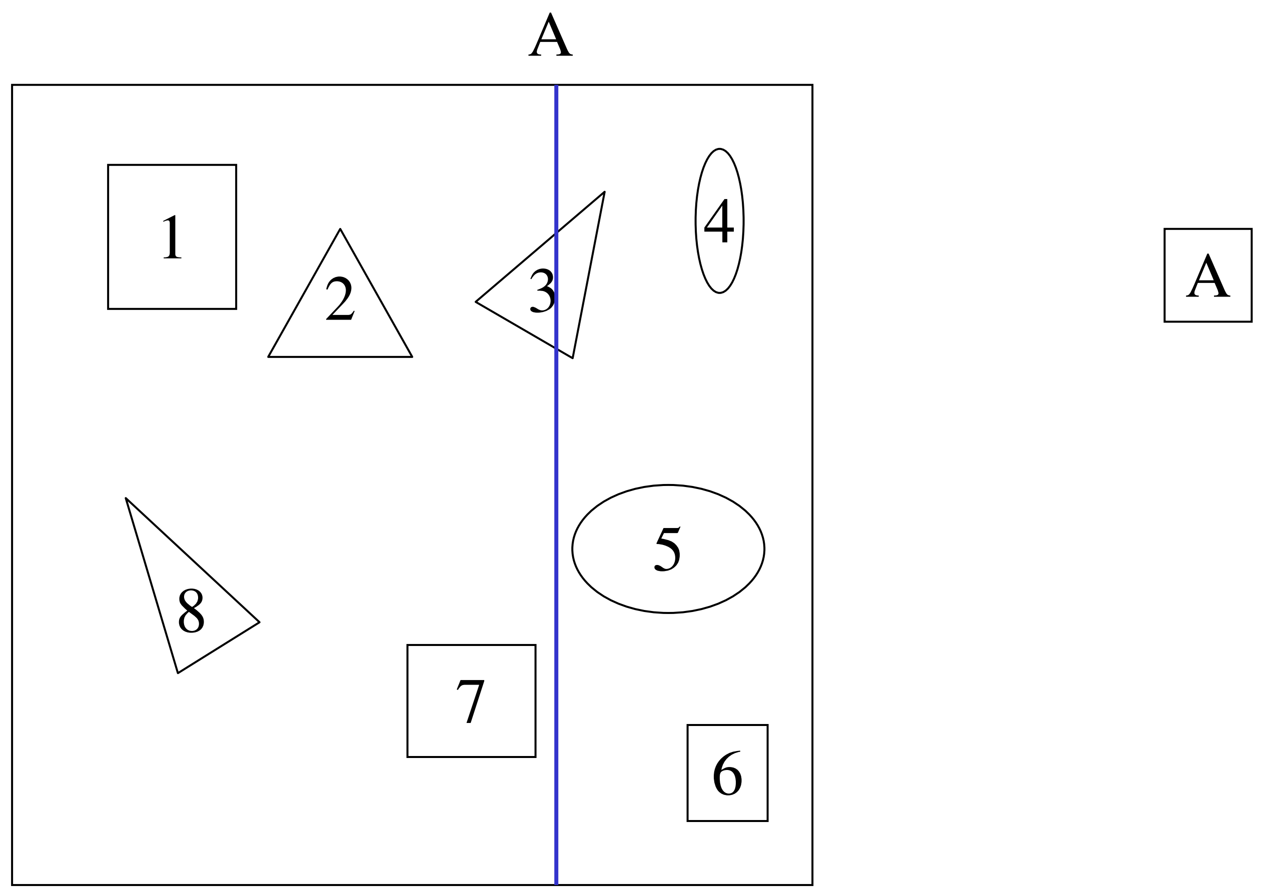
## kD-Trees



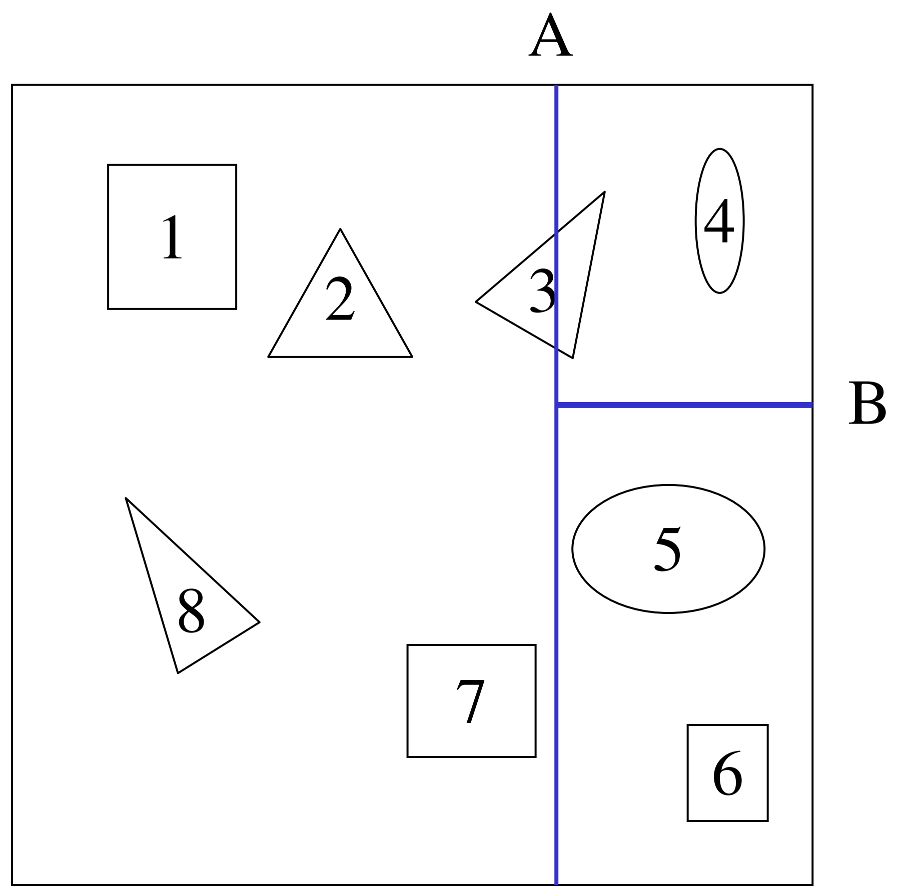
## KD-Tree (explicit example)

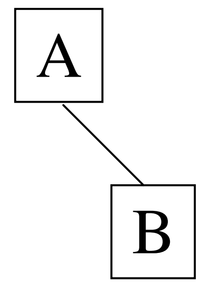


# KD-Tree (explicit example)

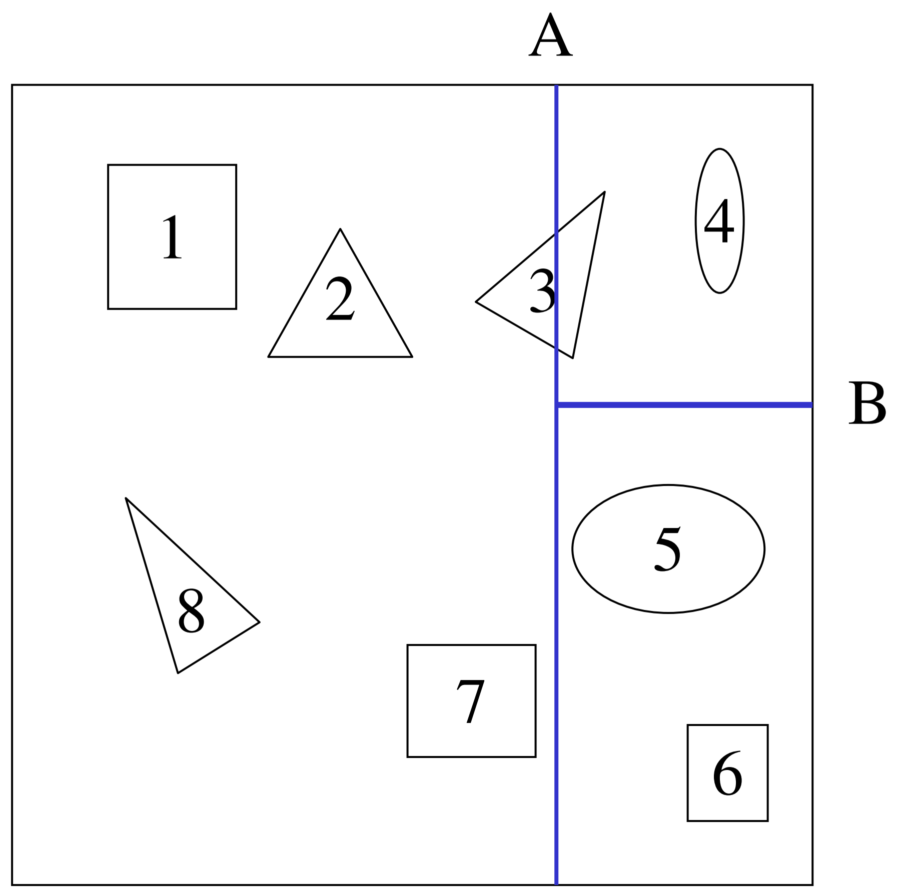


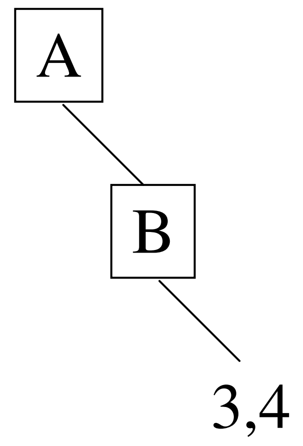
# KD-Tree (explicit example)



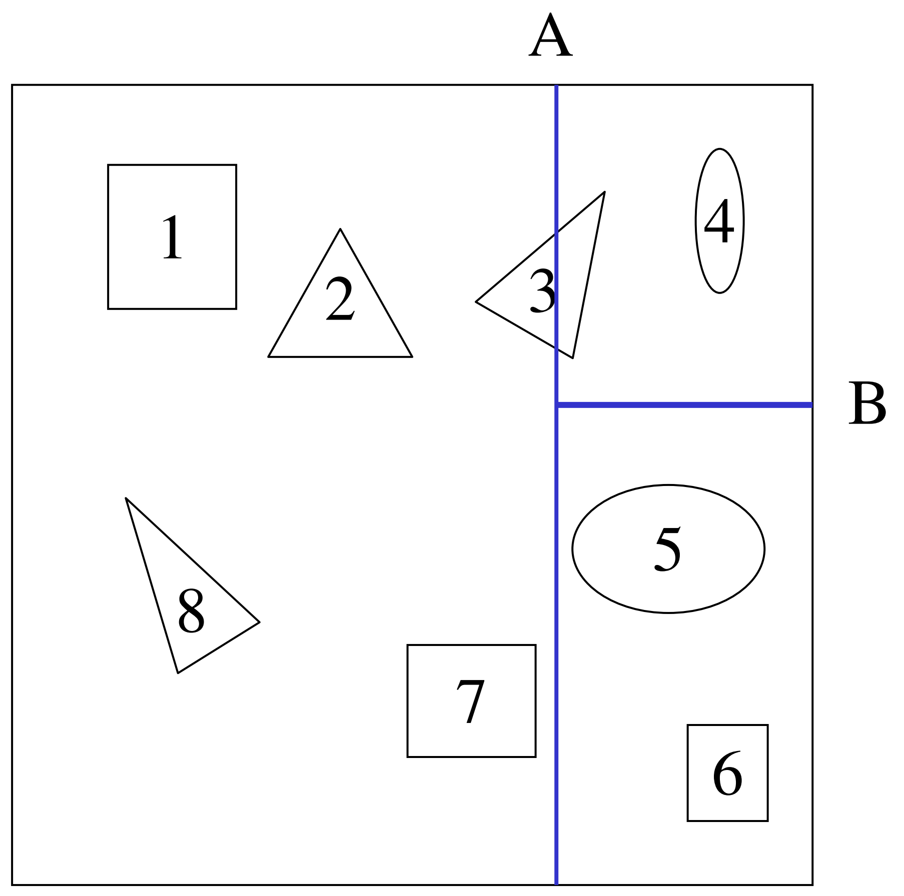


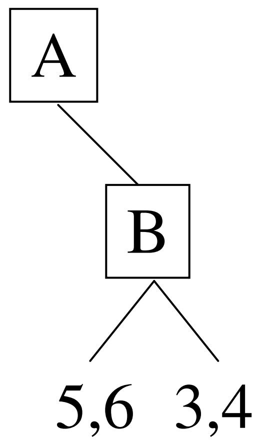
## KD-Tree (explicit example)



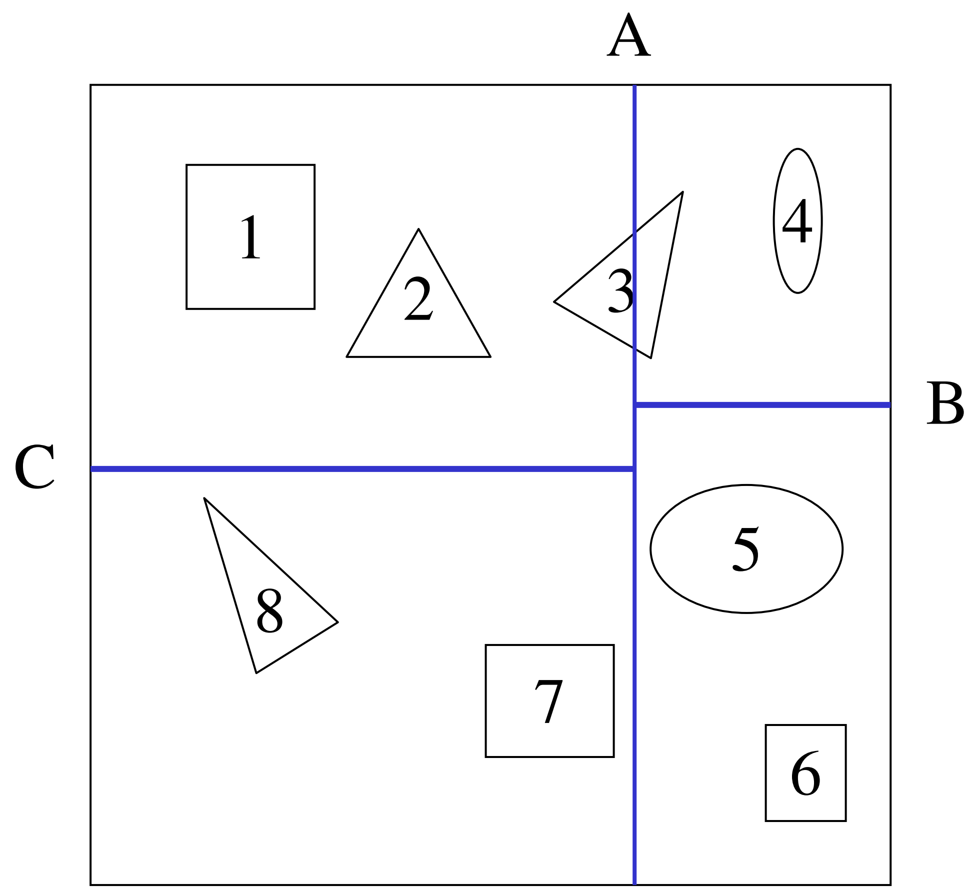


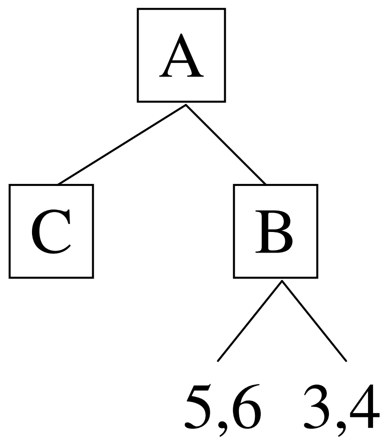
## KD-Tree (explicit example)



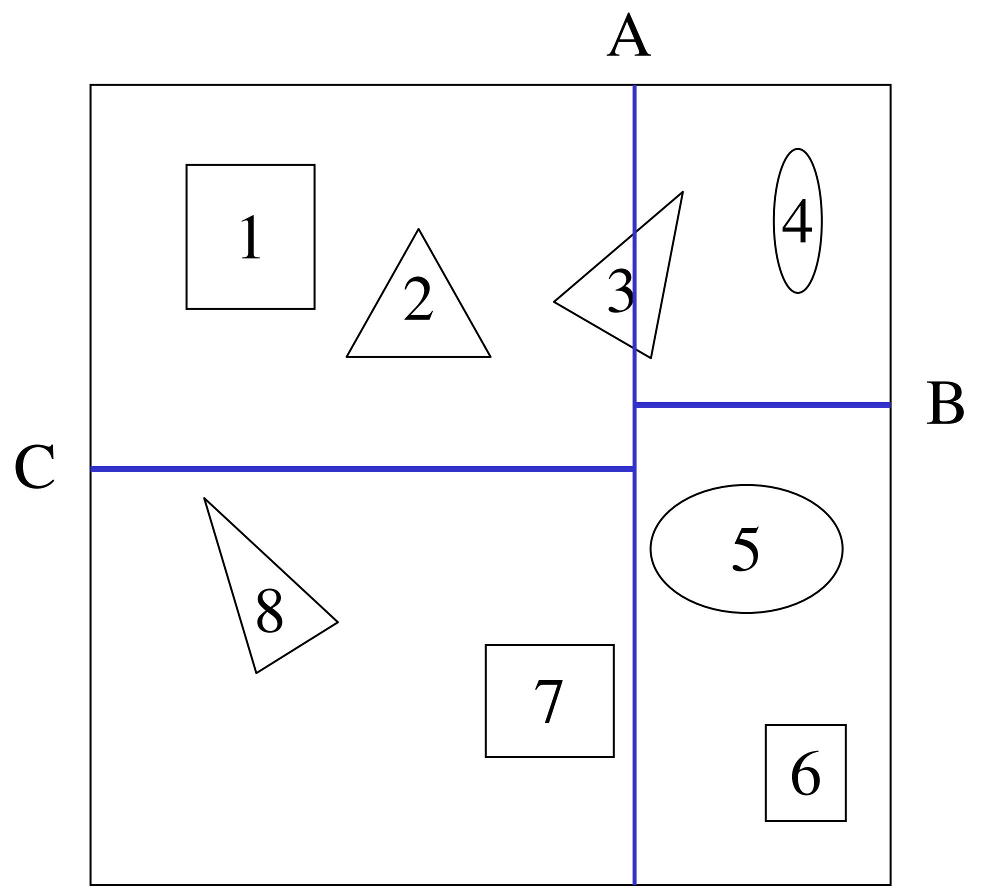


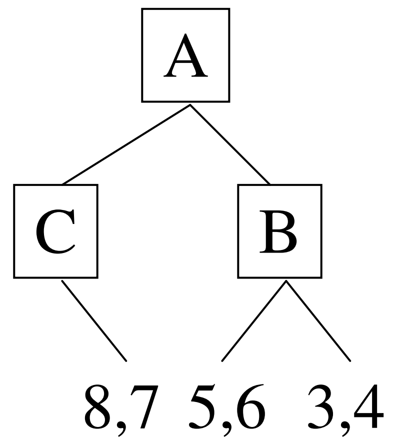
## KD-Tree (explicit example)



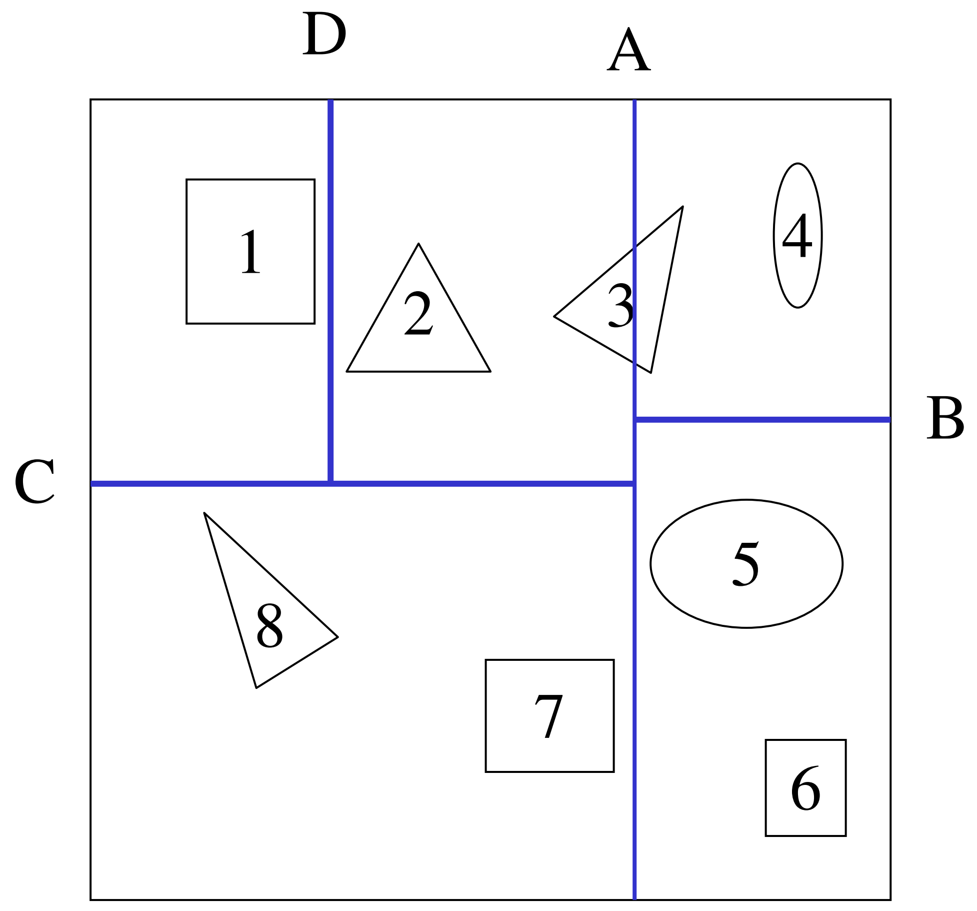


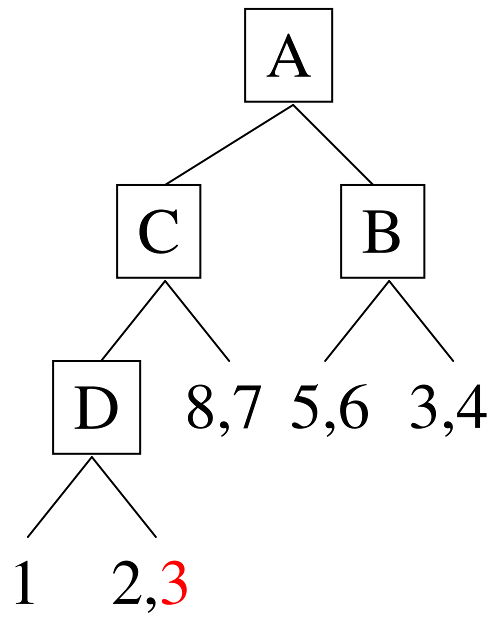
# KD-Tree (explicit example)





# KD-Tree (explicit example)

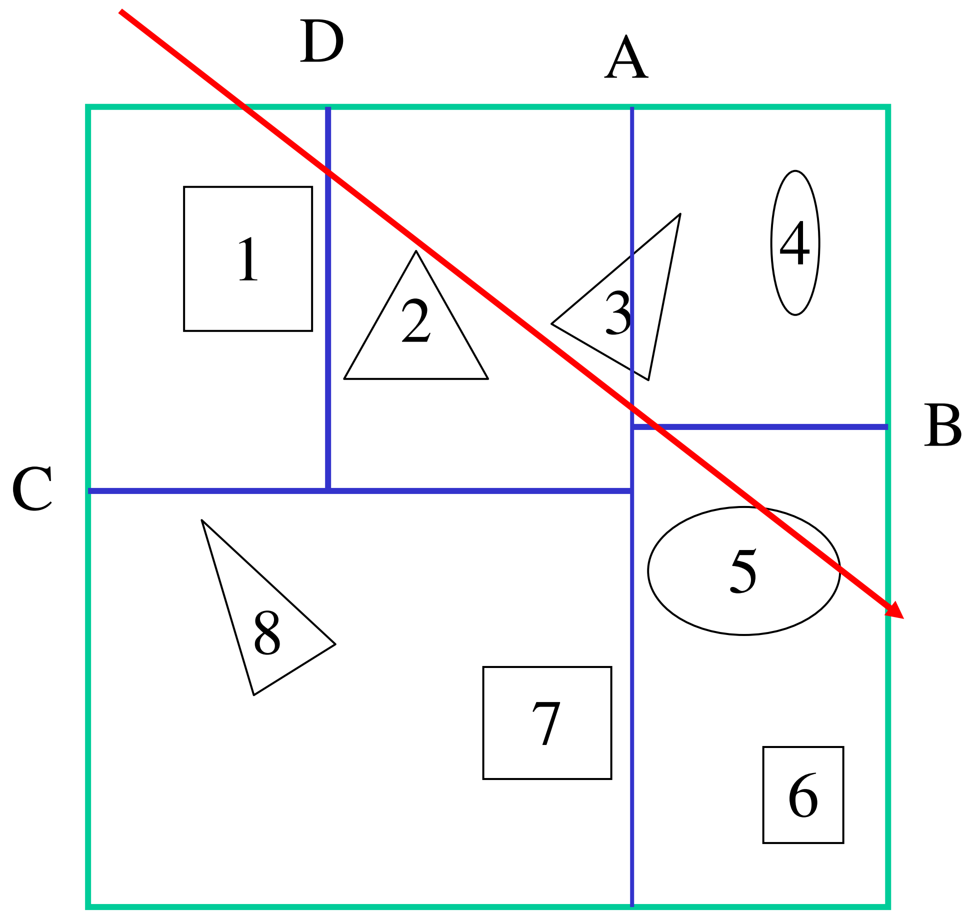


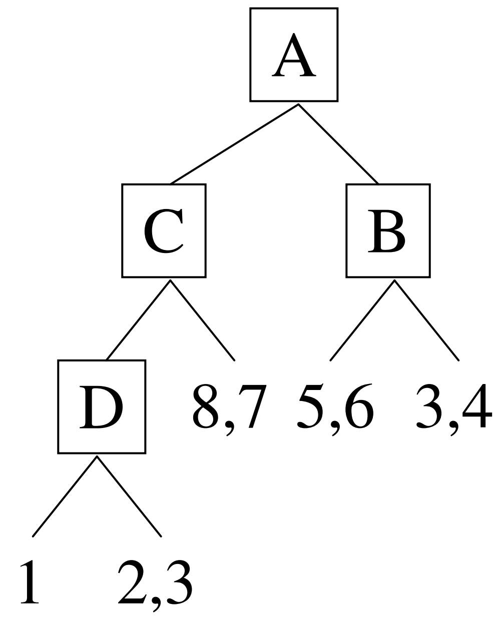


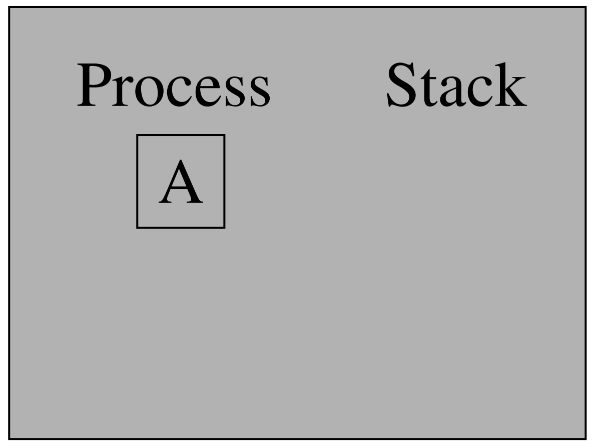
## BSP-Tree Traversal

- **“Front-to-back” traversal**
- **Traverse child nodes in order along rays**
- **Stop traversing as soon as surface intersection is found**
- **Maintain stack of subtrees to traverse**
  - More efficient than recursive function calls

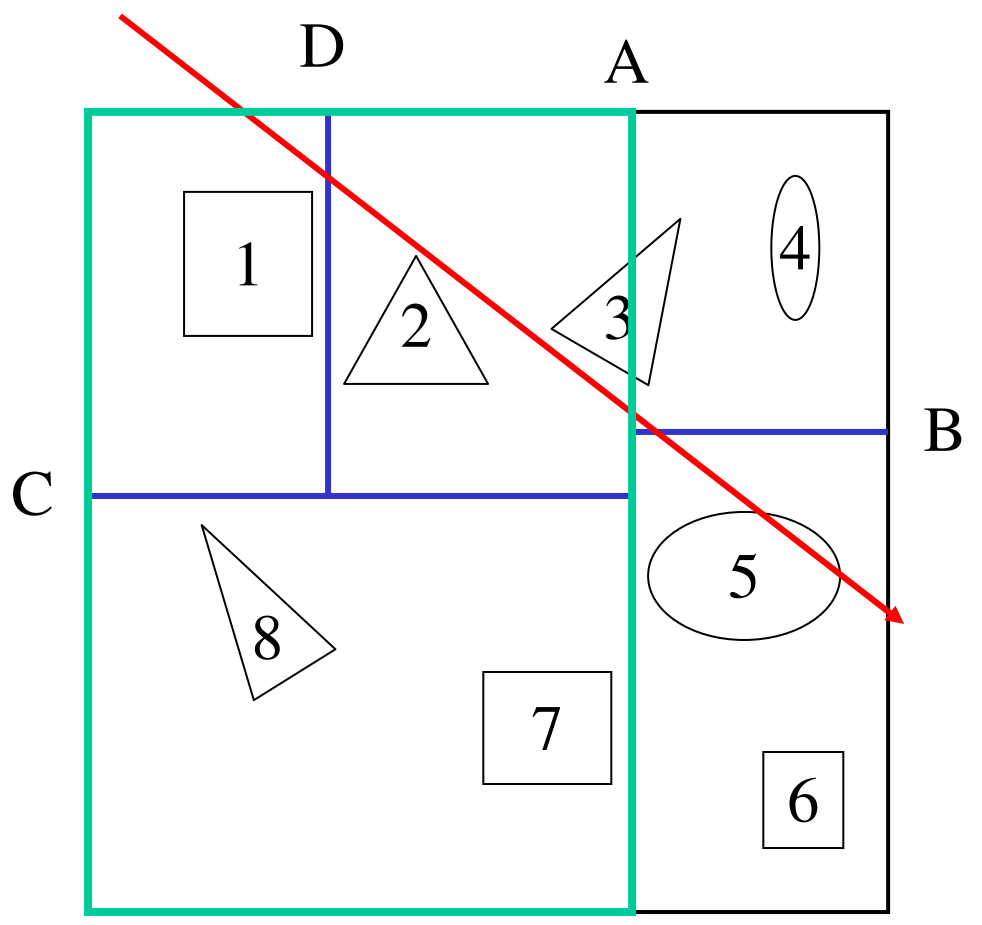
# BSP-Tree Traversal

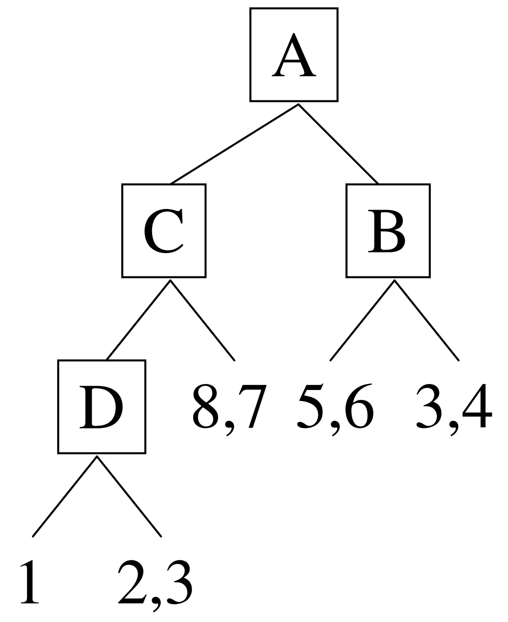






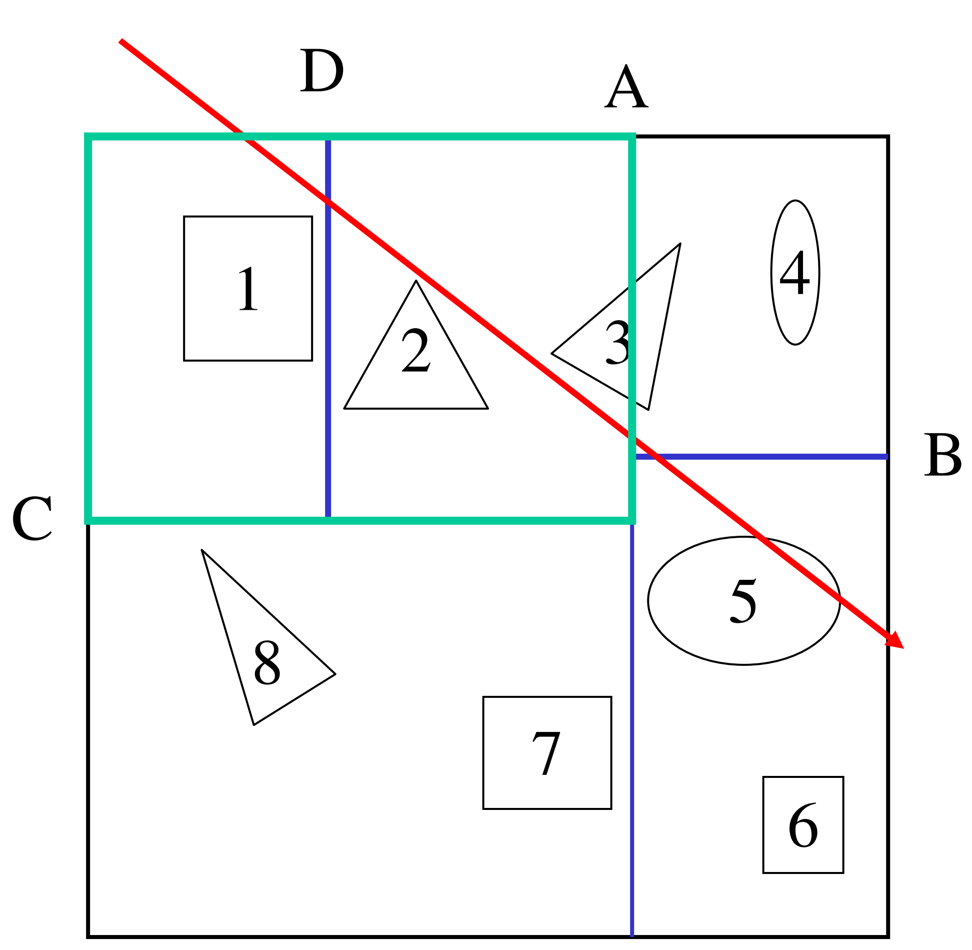
# BSP-Tree Traversal

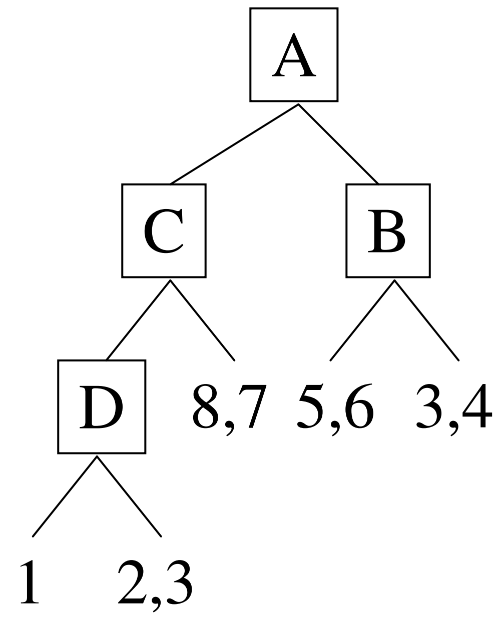




| Process | Stack |
|---|---|
| C | B |

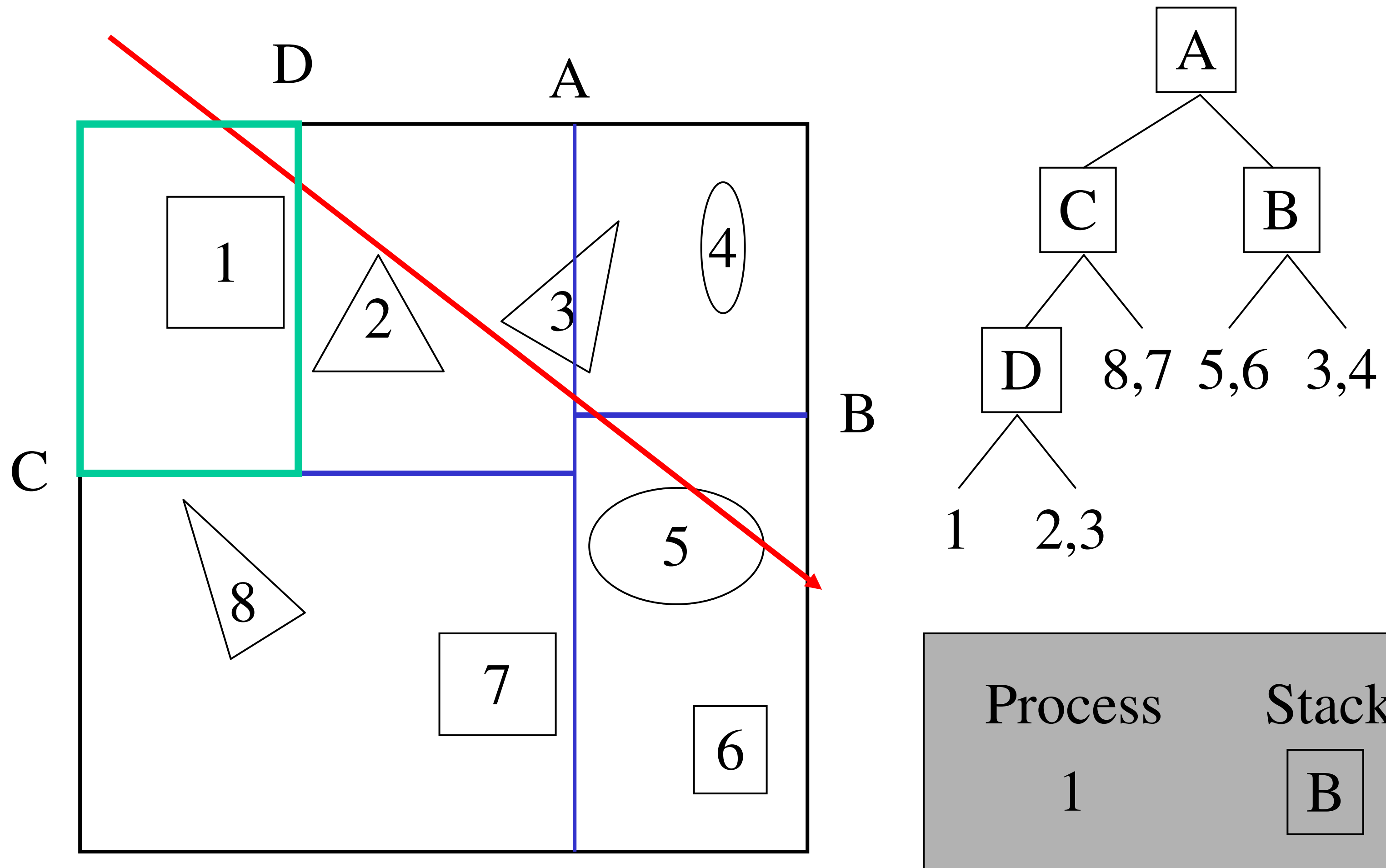
# BSP-Tree Traversal





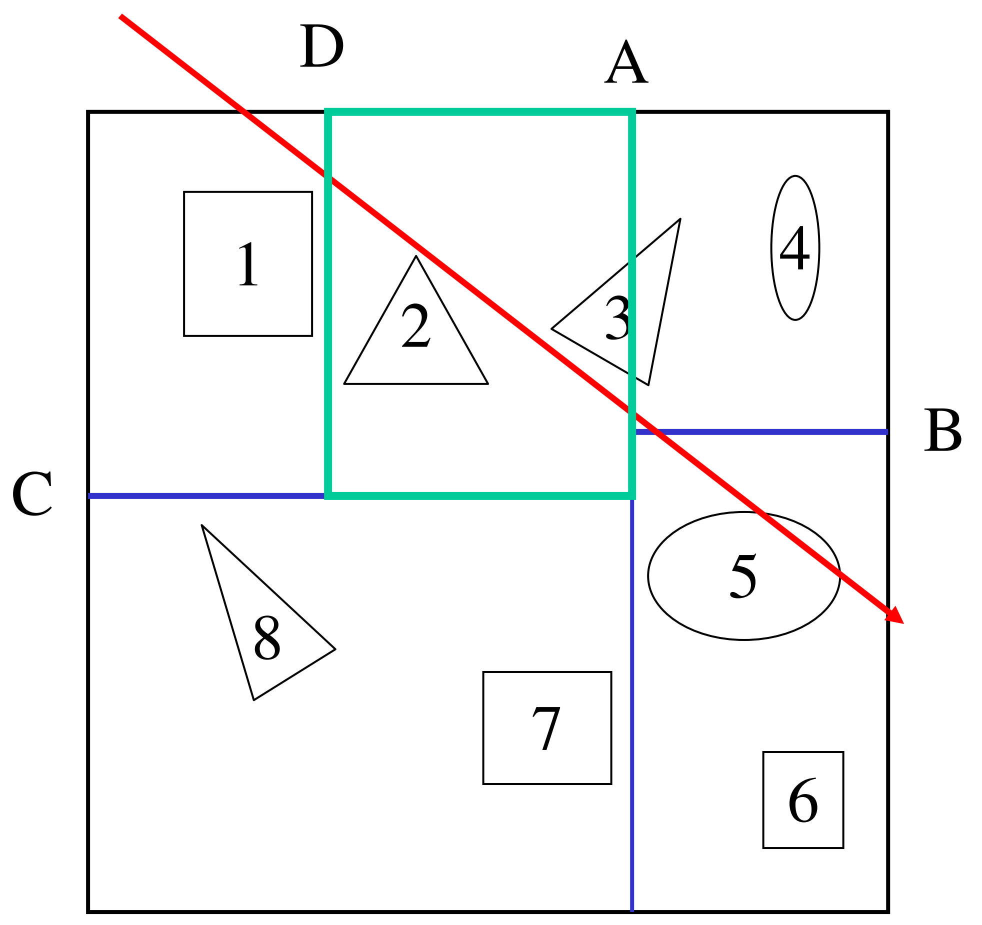
| Process | Stack |
|---|---|
| D | B |

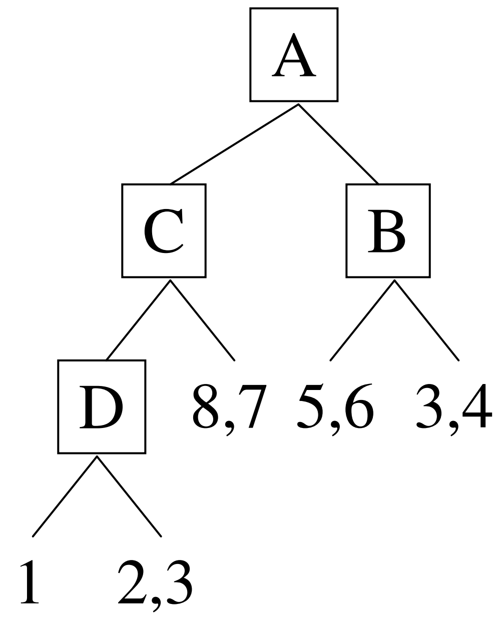
# BSP-Tree Traversal



| Process | Stack |
|---|---|
| 1 | $\boxed{B}$ |
| | 2,3 |

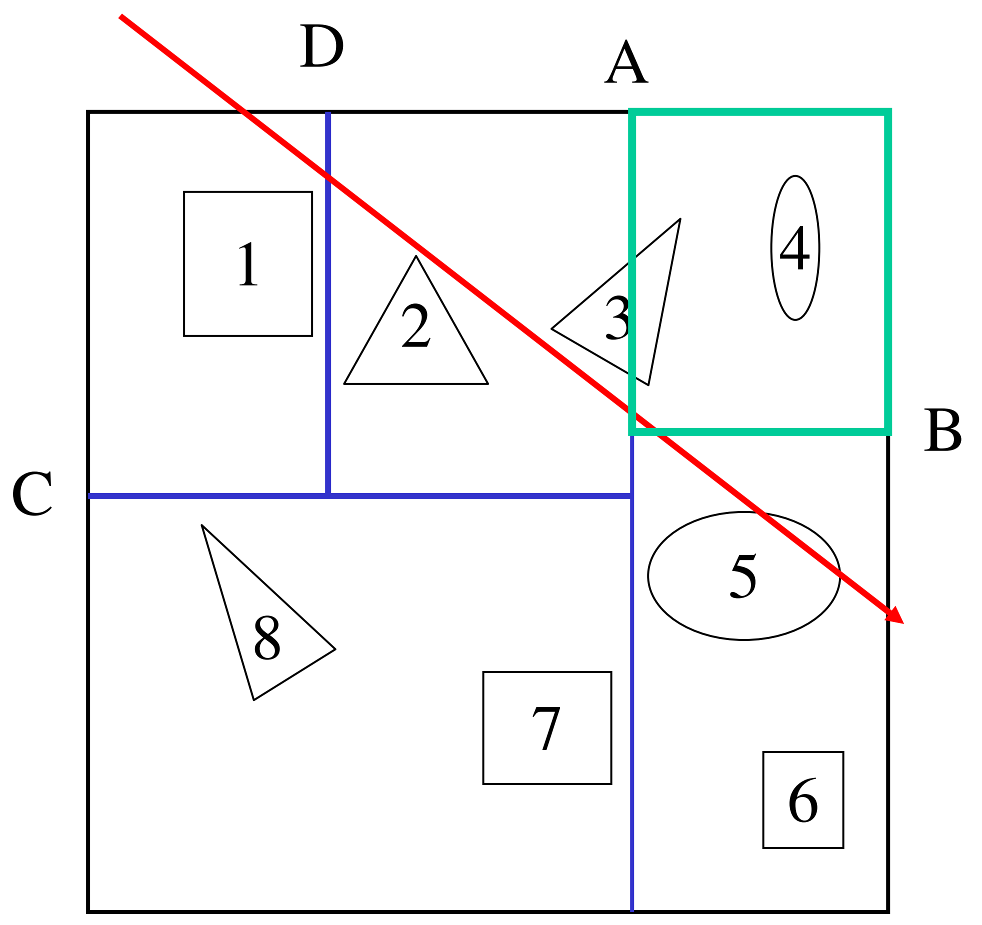
# BSP-Tree Traversal

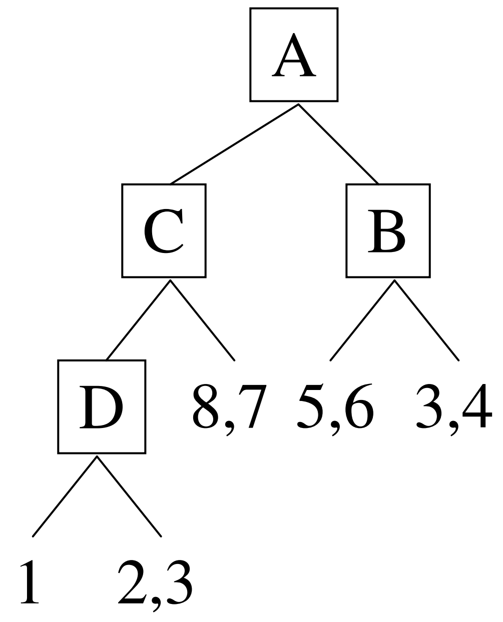




| Process | Stack |
|---|---|
| 2,3 | $\boxed{B}$ |

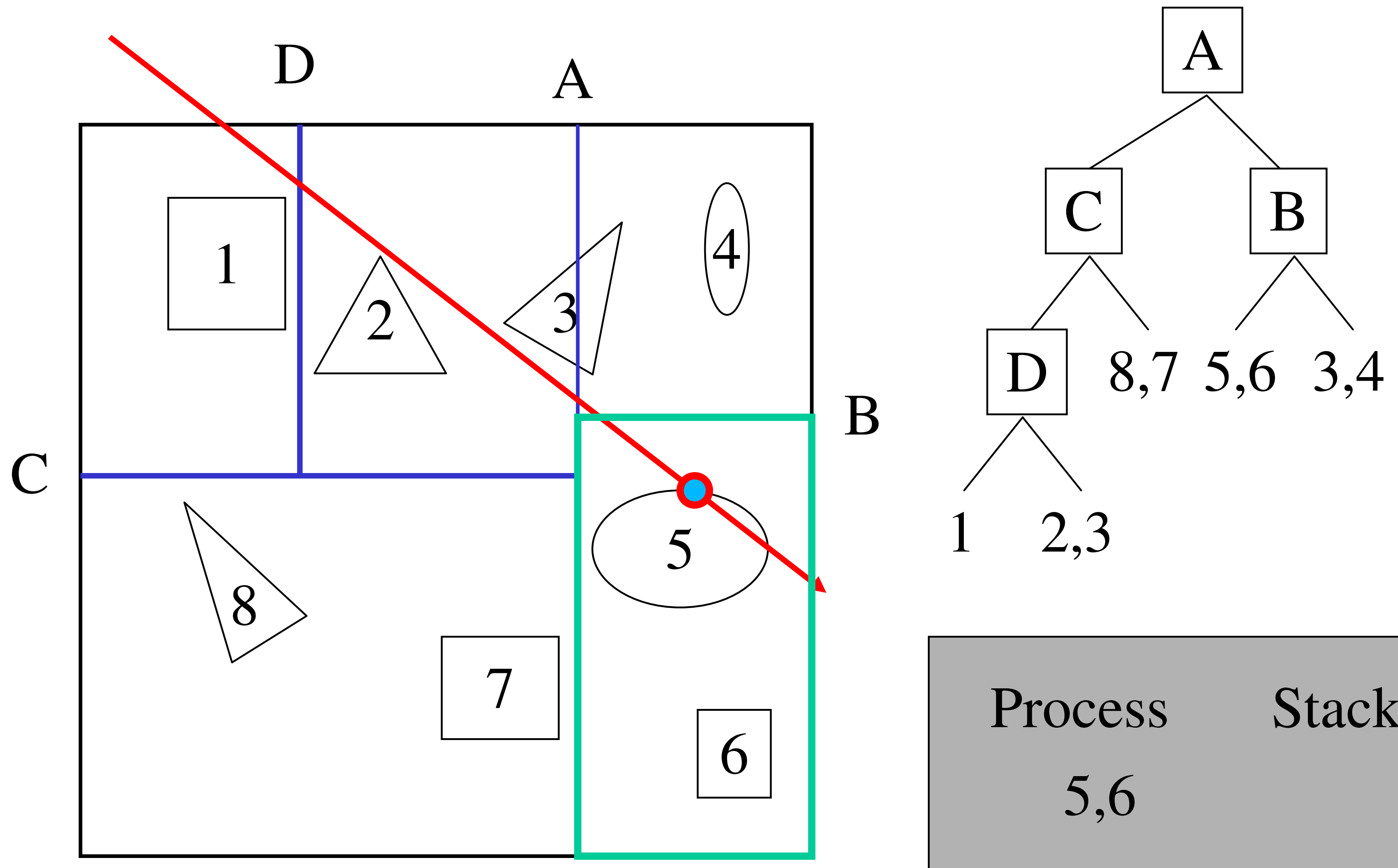
# BSP-Tree Traversal





| Process | Stack |
|---|---|
| 3,4 | 5,6 |

# BSP-Tree Traversal



| Process | Stack |
|---|---|
| 5,6 | |

## Advantages of kD-Trees

- **Adaptive**
  - Can handle the “Teapot in a Stadium”
- **Compact**
  - Relatively little memory overhead
- **Cheap Traversal**
  - One FP subtract, one FP multiply

## Take advantage of advantages

- **Adaptive**
  - You have to build a good tree
- **Compact**
  - At least use the compact node representation (8-byte)
  - You can’t be fetching whole cache lines every time
- **Cheap traversal**
  - No sloppy inner loops! (one subtract, one multiply!)

## Fast Ray Tracing w/ kD-Trees

- Adaptive
- Compact
- Cheap traversal

## Building kD-trees

- **Given:**
  - axis-aligned bounding box (“cell”)
  - list of geometric primitives (triangles?) touching cell
- **Core operation:**
  - pick an axis-aligned plane to split the cell into two parts
  - sift geometry into two batches (some redundancy)
  - recurse

## Building kD-trees

- **Given:**
  - axis-aligned bounding box (“cell”)
  - list of geometric primitives (triangles?) touching cell
- **Core operation:**
  - pick an axis-aligned plane to split the cell into two parts
  - sift geometry into two batches (some redundancy)
  - recurse
  - termination criteria!

## “Intuitive” kD-Tree Building

- **Split Axis**
  - Round-robin; largest extent
- **Split Location**
  - Middle of extent; median of geometry (balanced tree)
- **Termination**
  - Target # of primitives, limited tree depth

## “Hack” kD-Tree Building

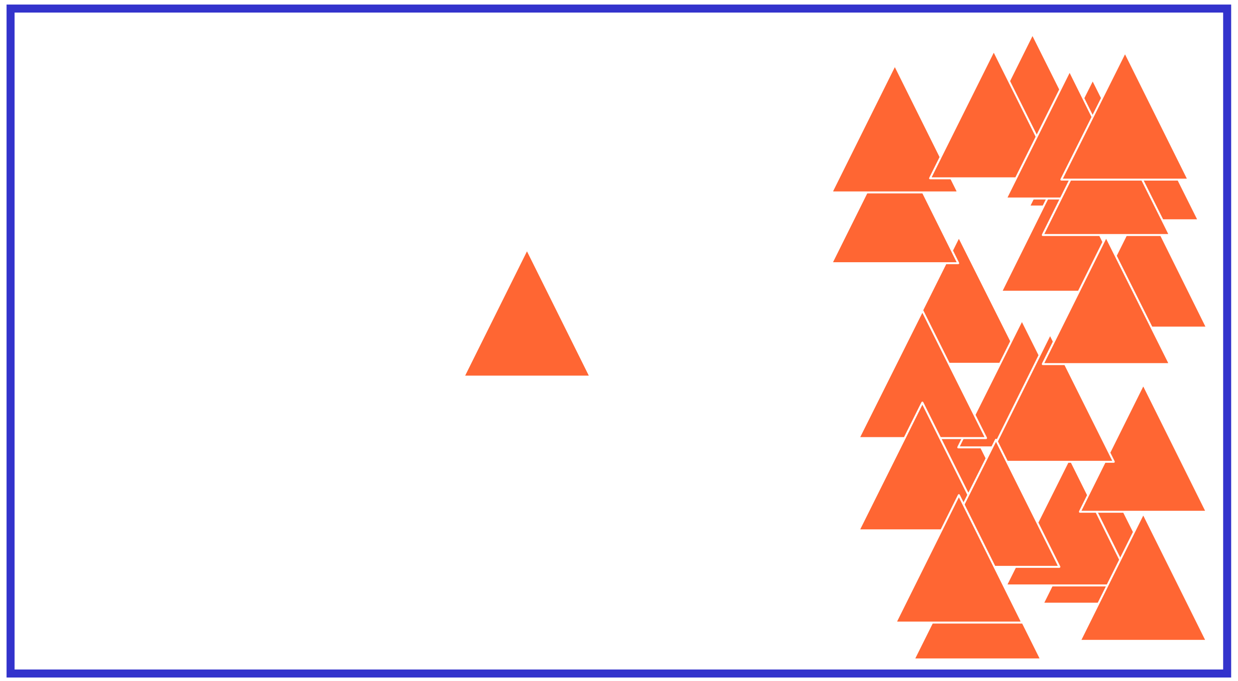
- **Split Axis**
  - Round-robin; largest extent
- **Split Location**
  - Middle of extent; median of geometry (balanced tree)
- **Termination**
  - Target # of primitives, limited tree depth
- **All of these techniques are not very clever**

## Building good kD-trees

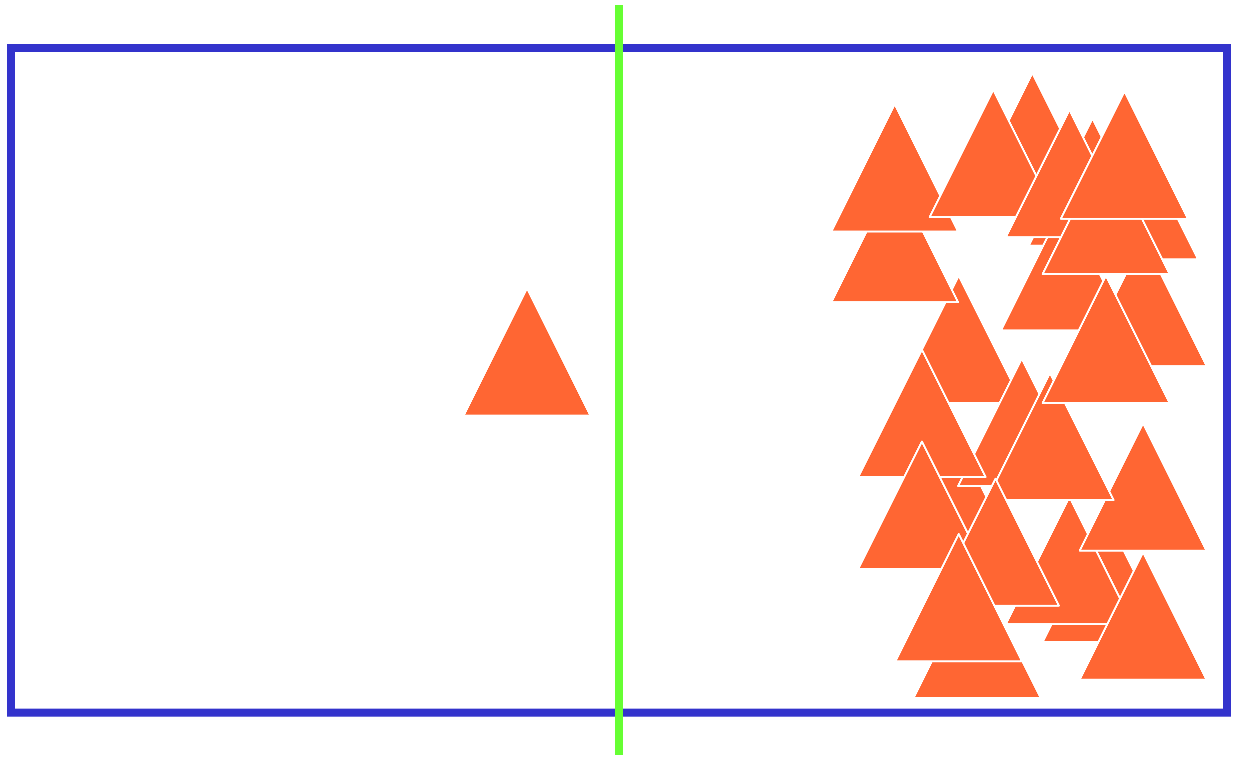
- **What split do we really want?**
  - Clever Idea: The one that makes ray tracing cheap
  - Write down an expression of cost and minimize it
  - *Cost Optimization*
- **What is the cost of tracing a ray through a cell?**

Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

## Splitting with Cost in Mind

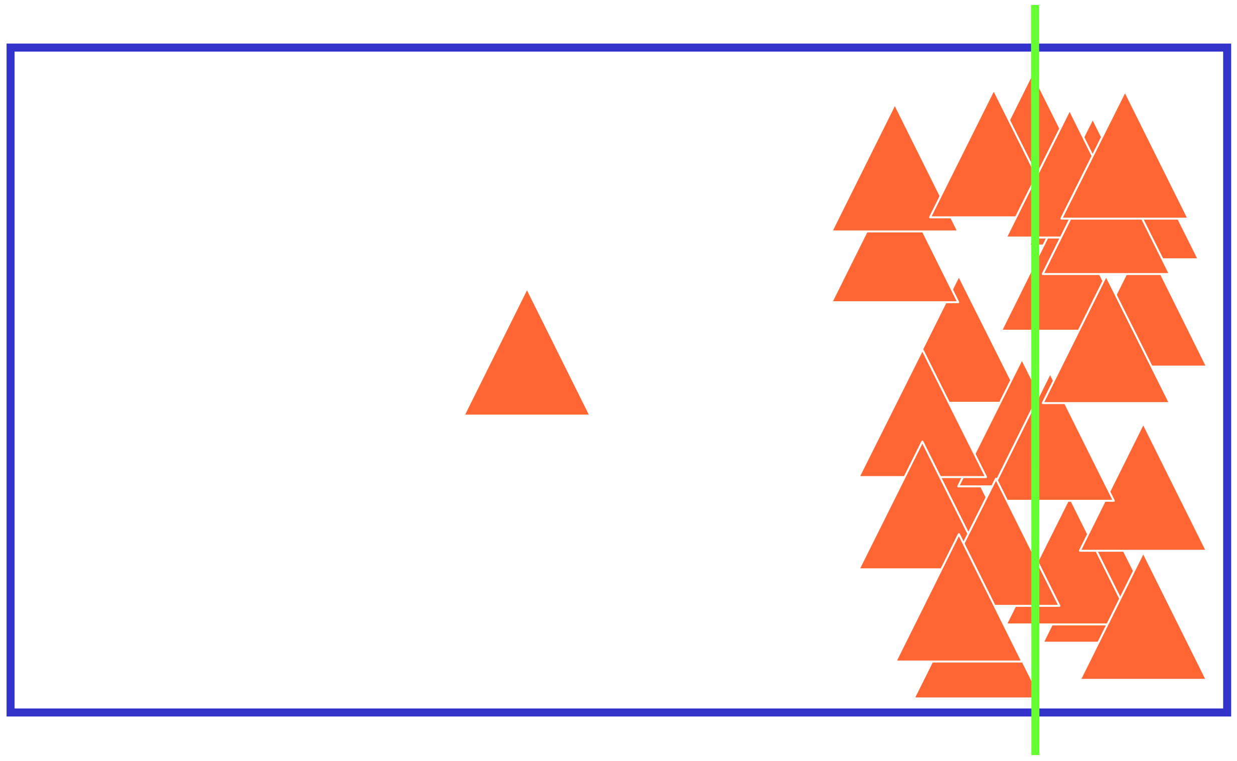


## Split in the middle



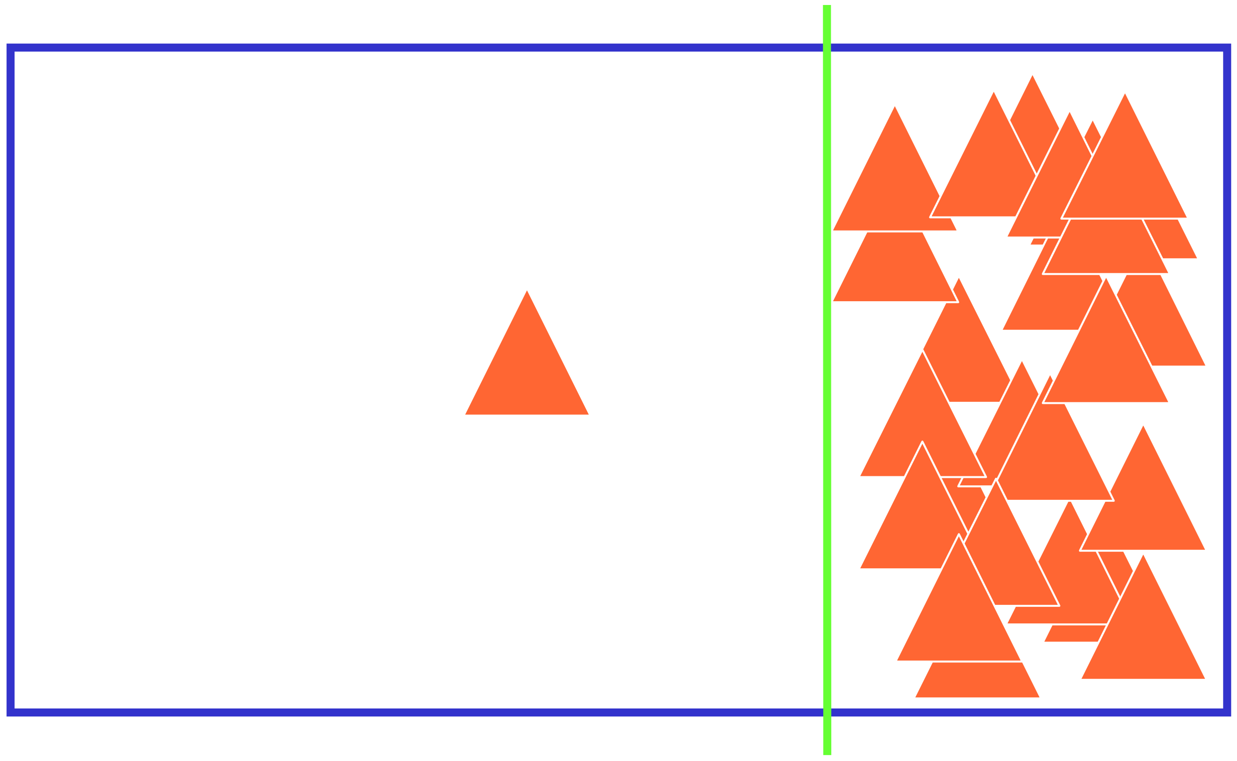
- Makes the L & R probabilities equal
- Pays no attention to the L & R costs

## Split at the Median



- Makes the L & R costs equal
- Pays no attention to the L & R probabilities

## Cost-Optimized Split



- Automatically and rapidly isolates complexity
- Produces large chunks of empty space

## Building good kD-trees

- **Need the probabilities**
  - Turns out to be proportional to surface area
- **Need the child cell costs**
  - Simple triangle count works great (very rough approx.)

Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

= C_trav + SA(L) * TriCount(L) + SA(R) * TriCount(R)

## Termination Criteria

- **When should we stop splitting?**
  - Another Clever idea:  When splitting isn’t helping any more.
  - Use the cost estimates in your termination criteria
- **Threshold of cost improvement**
  - Stretch over multiple levels
- **Threshold of cell size**
  - Absolute probability so small there’s no point

## Building good kD-trees

- **Basic build algorithm**
  - Pick an axis, or optimize across all three
  - Build a set of “candidates” (split locations)
    - BBox edges or exact triangle intersections
  - Sort them or bin them
  - Walk through candidates or bins to find minimum cost split
- **Characteristics you’re looking for**
  - long and thin
  - depth 50-100,
  - ~2 triangle leaves,
  - big empty cells

## Building kD-trees quickly

- **Very important to build good trees first**
  - otherwise you have no basis for comparison
- **Don’t give up cost optimization!**
  - Use the math, Luke…
- **Luckily, *lots* of flexibility…**
  - axis picking (“hack” pick vs. full optimization)
  - candidate picking (bboxes, exact; binning, sorting)
  - termination criteria (“knob” controlling tradeoff)

## Building kD-trees quickly

- **Remember, profile first!  Where's the time going?**
  - split personality
    - memory traffic all at the top (NO cache misses at bottom)
      - sifting through bajillion triangles to pick one split (!)
      - hierarchical building?
    - computation mostly at the bottom
      - lots of leaves, need more exact candidate info
      - lazy building?
    - change criteria during the build?

## Fast Ray Tracing w/ kD-Trees

- **adaptive**
  - build a cost-optimized kD-tree w/ the surface area heuristic
- **compact**
- **cheap traversal**

## What’s in a node?

- **A kD-tree internal node needs:**
  - Am I a leaf?
  - Split axis
  - Split location
  - Pointers to children

## Compact (8-byte) nodes

- **kD-Tree node can be packed into 8 bytes**
  - Leaf flag + Split axis
    - 2 bits
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - One 32-bit pointer

## Compact (8-byte) nodes

- **kD-Tree node can be packed into 8 bytes**
  - Leaf flag + Split axis (3+1)
    - 2 bits
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - One 32-bit pointer
- **So close!  Sweep those 2 bits under the rug…**

## No Bounding Box!

- **kD-Tree node corresponds to an AABB**
- **Doesn’t mean it has to *contain* one**
  - 24 bytes
  - 4X explosion (!)

## Memory Layout

- **Cache lines are much bigger than 8 bytes!**
  - advantage of compactness lost with poor layout
- **Pretty easy to do something reasonable**
  - Building depth first, watching memory allocator

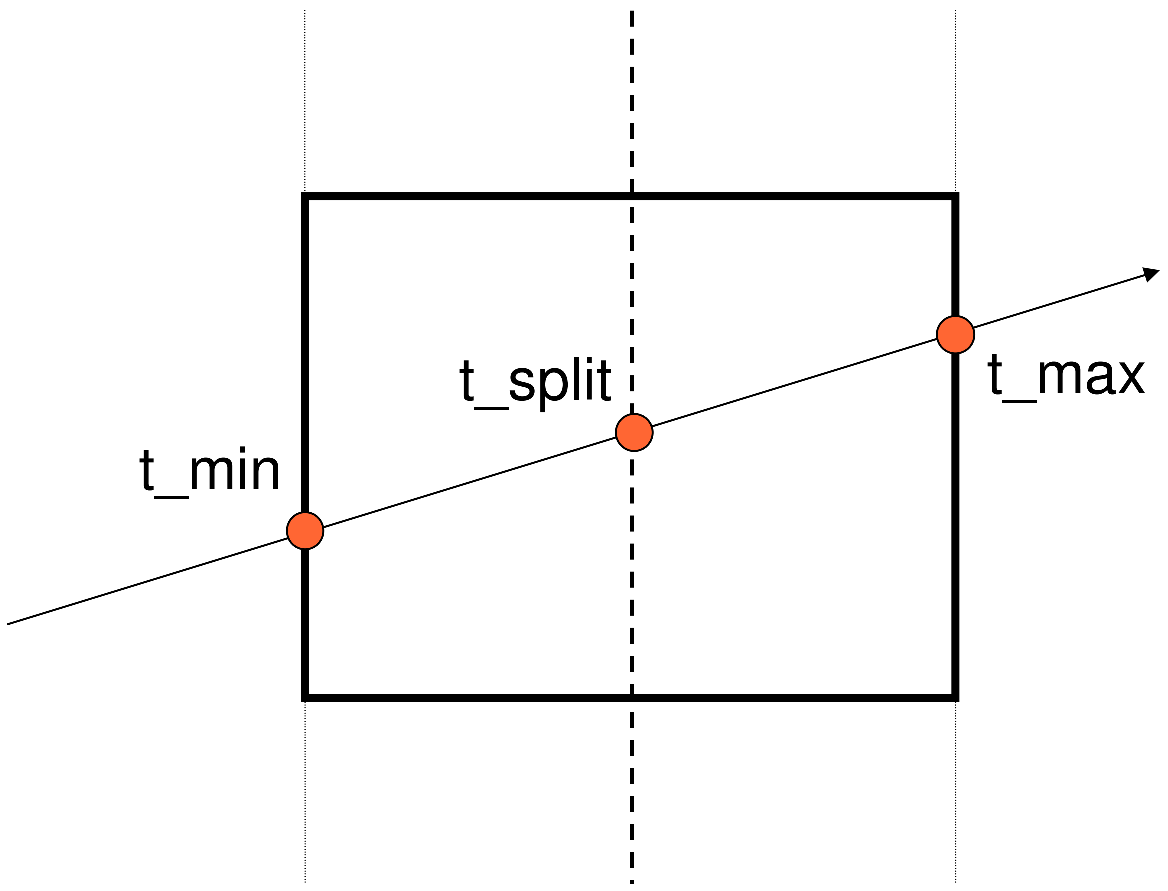
## Other Data

- **Memory should be separated by rate of access**
  - Frames
  - << Pixels
  - << Samples [ Ray Trees ]
  - << Rays [ Shading (not quite) ]
  - << Triangle intersections
  - << Tree traversal steps
- **Example: pre-processed triangle, shading info…**

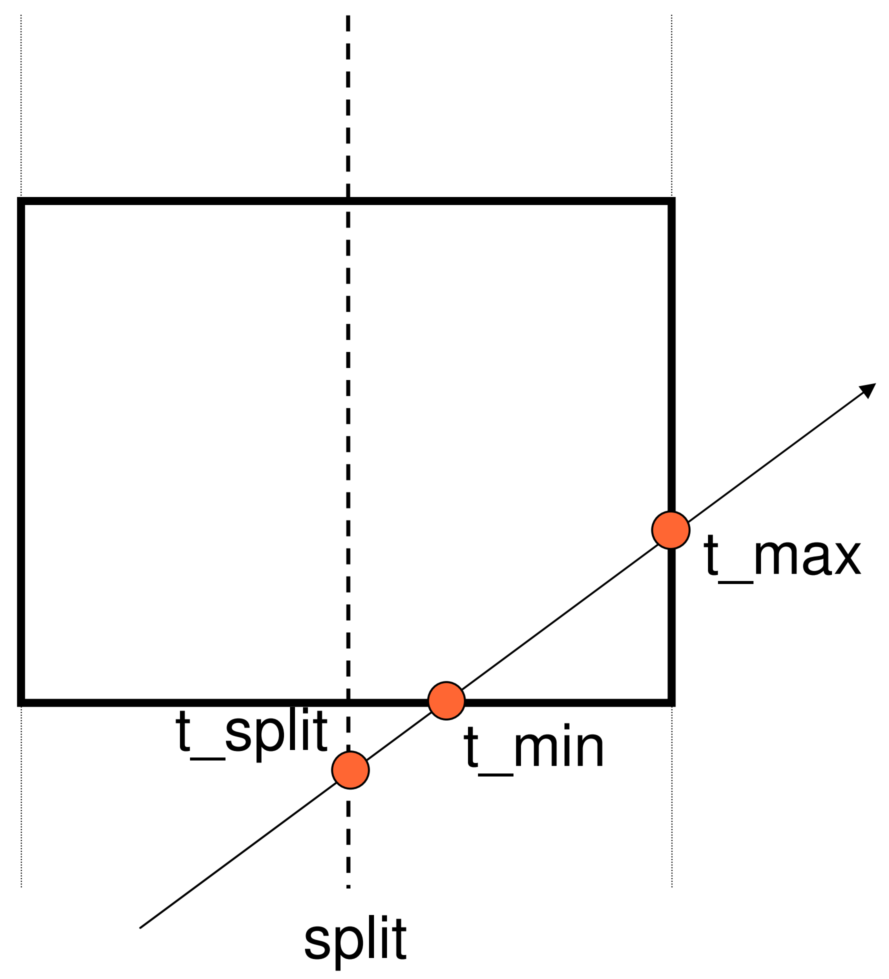
## Fast Ray Tracing w/ kD-Trees

- **adaptive**
  - build a cost-optimized kD-tree w/ the surface area heuristic
- **compact**
  - use an 8-byte node
  - lay out your memory in a cache-friendly way
- **cheap traversal**

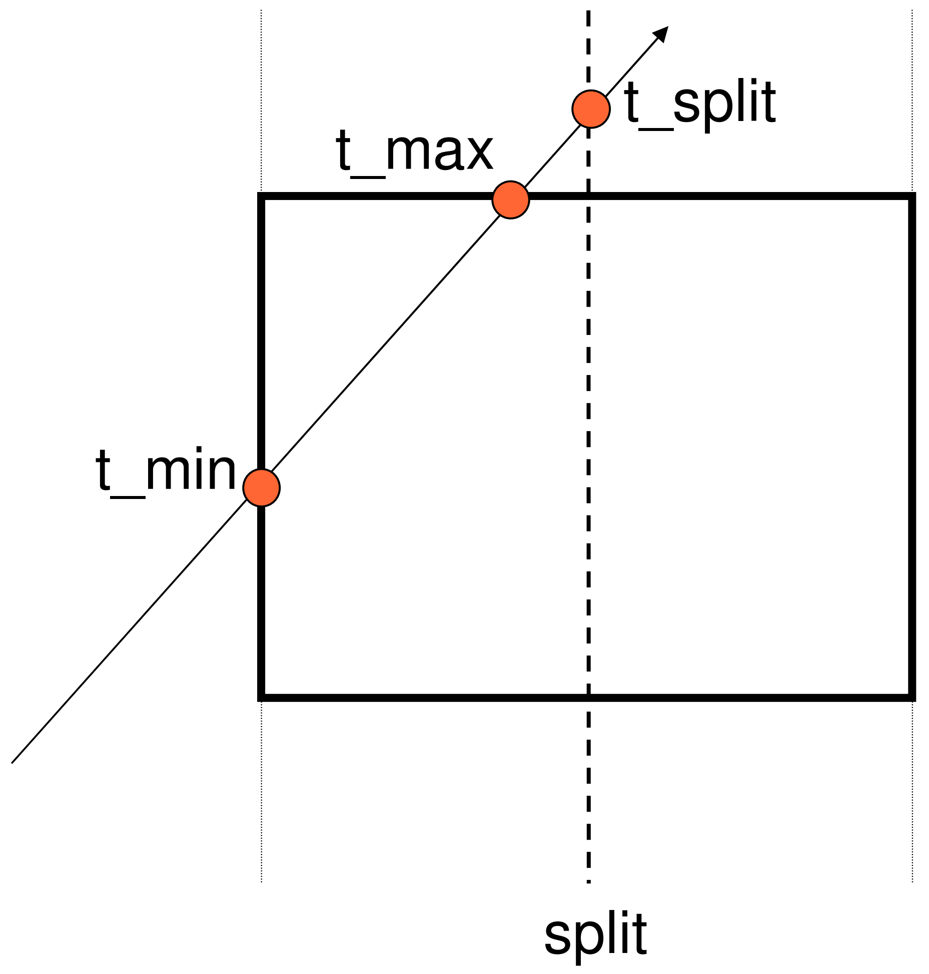
## kD-Tree Traversal Step



## kD-Tree Traversal Step



## kD-Tree Traversal Step



## kD-Tree Traversal Step

```
Given:  ray P & iV=(1/V), t_min, t_max, split_location, split_axis

t_split = ( split_location - ray->P[split_axis] ) * ray->iV[split_axis]

if t_split > t_min
    need to test against near child
If t_split < t_max
    need to test against far child
```

## Optimize Your Inner Loop

- **kD-Tree traversal is the most critical kernel**
  - It happens about a zillion times
  - It’s tiny
  - Sloppy coding *will* show up
- **Optimize, Optimize, Optimize**
  - Remove recursion and minimize stack operations
  - Other standard tuning & tweaking

## kD-Tree Traversal

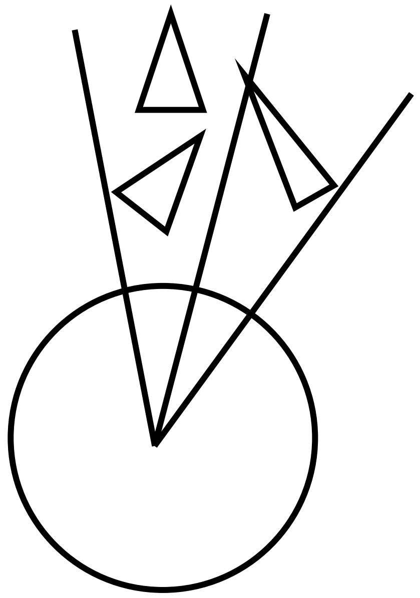
```
while ( not a leaf )
    t_at_split = ( split_location - ray->P[split_axis] ) * ray_iV[split_axis]
    if t_split <= t_min
          continue with far child        // hit either far child or none
    if t_split >= t_max
          continue with near child       // hit near child only
    // hit both children
    push (far child, t_split, t_max) onto stack
    continue with (near child, t_min, t_split)
```

## Can it go faster?

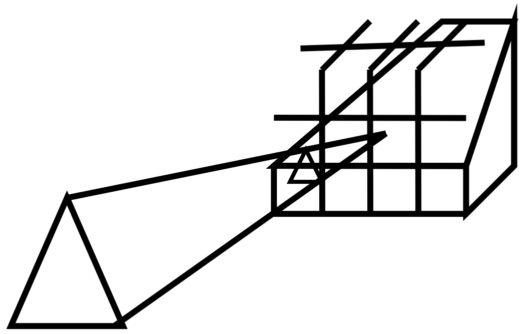
- **How do you make fast code go faster?**
- **Parallelize it!**

## Directional Partitioning

- **Applications**
  - Useful only for rays that start from a single point
    - Camera
    - Point light sources
  - Preprocessing of visibility
  - Requires scan conversion of geometry
    - For each object locate where it is visible
    - Expensive and linear in # of objects
- **Generally not used for primary rays**

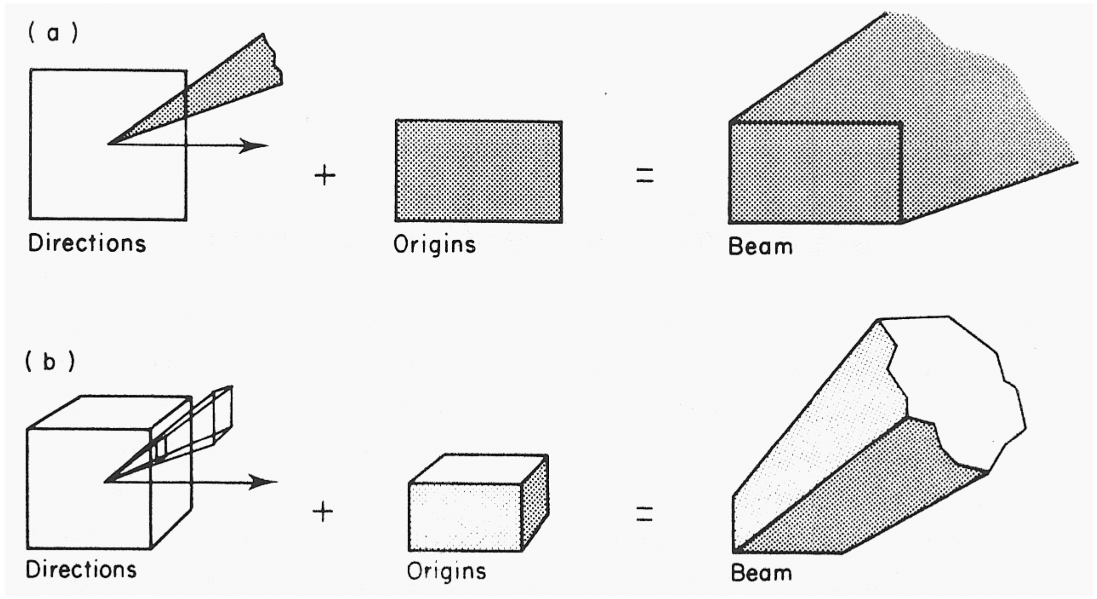


- **Variation: Light buffer**
  - Lazy and conservative evaluation
  - Store occluder that was found in directional structure
  - Test entry first for next shadow test



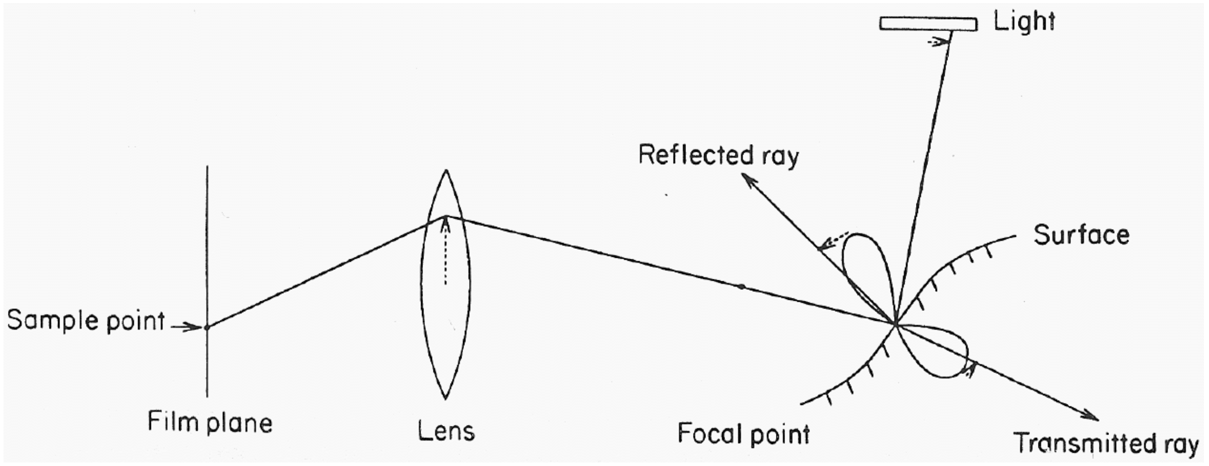
# Ray Classification

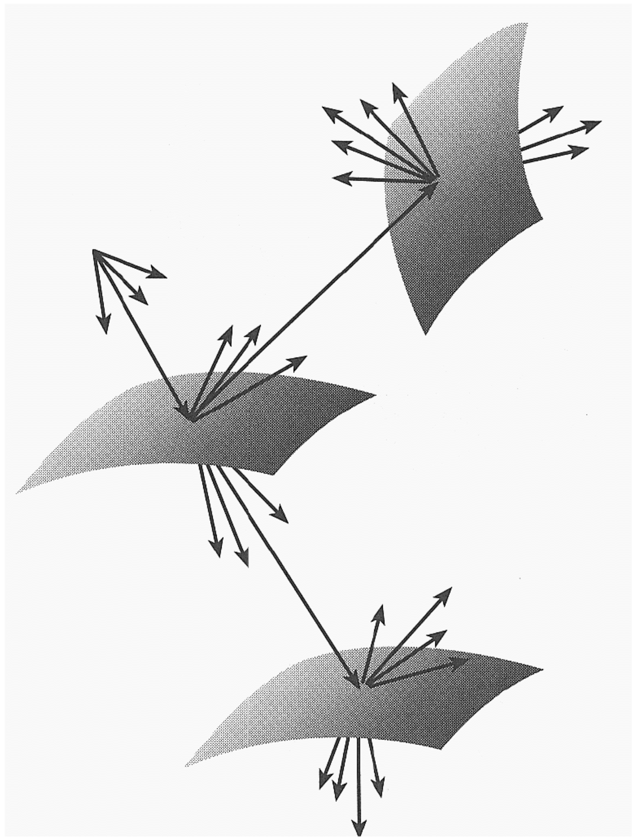
- **Partitioning of space and direction [Arvo & Kirk´87]**
  - Roughly pre-computes visibility for the entire scene
    - What is visible from each point in each direction?
  - Very costly preprocessing, cheap traversal
    - Improper trade-off between preprocessing and run-time
  - Memory hungry, even with lazy evaluation
  - Seldom used in practice



## Distribution Ray Tracing

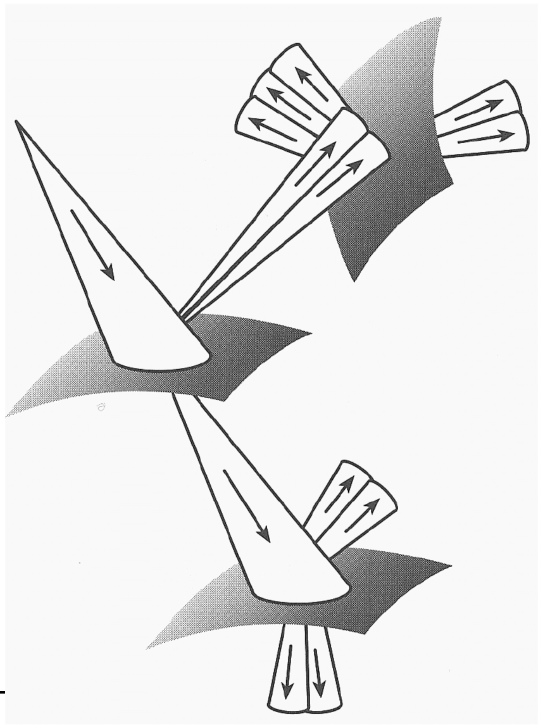
- **Formerly called Distributed Ray Tracing [Cook`84]**
- **Stochastic Sampling of**
  - Pixel: Antialiasing
  - Lens: Depth-of-field
  - BRDF: Glossy reflections
  - Lights: Smooth shadows from area light sources
  - Time: Motion blur



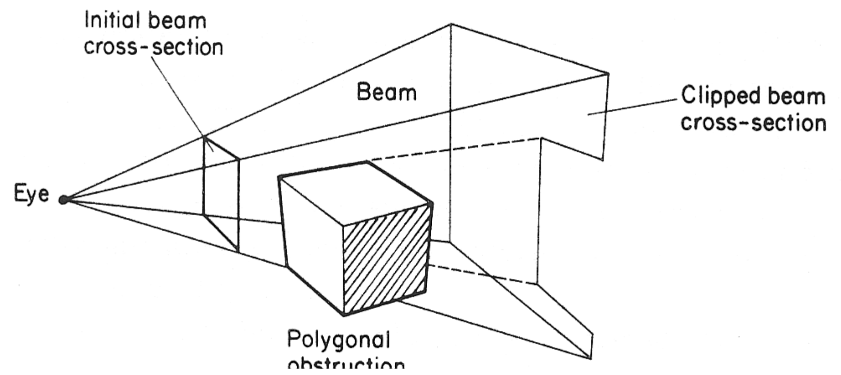


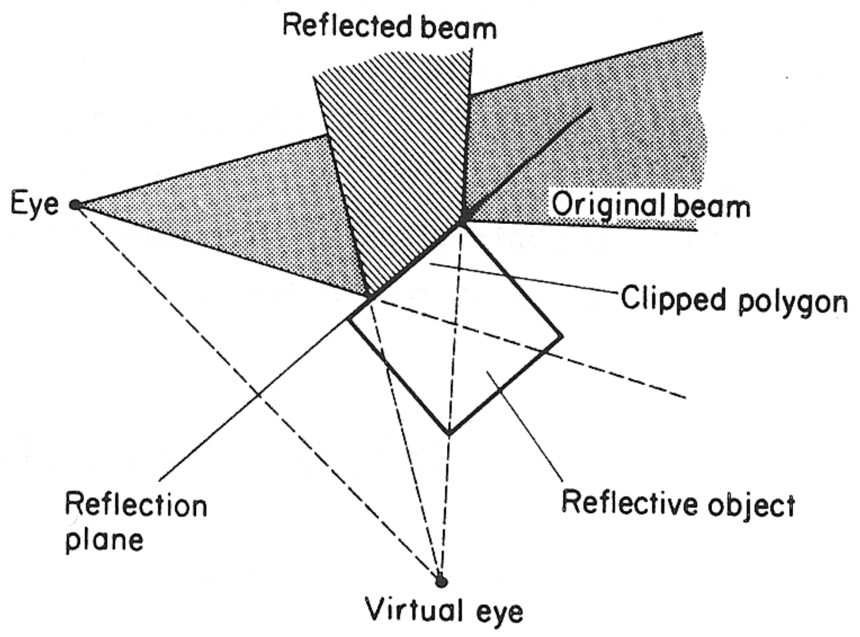
## Beam und Cone Tracing

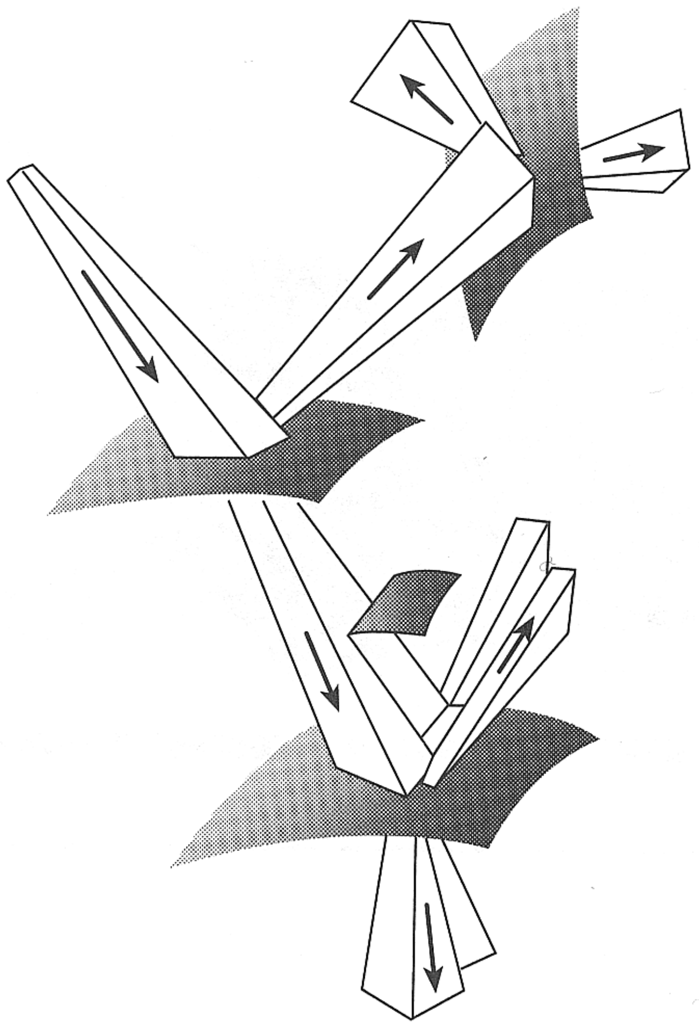
- **General idea:**
  - Trace continuous bundles of rays
- **Cone Tracing:**
  - Approximate collection of ray with cone(s)
  - Subdivide into smaller cones if necessary
- **Beam Tracing:**
  - Exactly represent a ray bundle with pyramid
  - Create new beams at intersections (polygons)
- **Problems:**
  - Clipping of beams?
  - Good approximations?
  - How to compute intersections?
- **Not really practical !!**



# Beam Tracing







## Packet Tracing

- **Approach**
  - Combine many similar rays (e.g. primary or shadow rays)
  - Trace them together in SIMD fashion
    - All rays perform the same traversal operations
    - All rays intersect the same geometry
  - Exposes coherence between rays
    - All rays touch similar spatial indices
    - Loaded data can be reused (in registers & cache)
    - More computation per recursion step → better optimization
  - Overhead
    - Rays will perform unnecessary operations
    - Overhead low for coherent and small set of rays (e.g. up to 4x4 rays)