EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Computer Graphics

Computer Graphics (Graphische Datenverarbeitung)

- Ray Tracing II -

WS 2021/22

Hendrik Lensch

Corona

- Regular random lookup of the 3G certificates

- Contact tracing: We need to know who is in the class room
  - New ILIAS group for every lecture slot
  - Register via ILIAS or this QR code (only if you are present in this room)

Overview

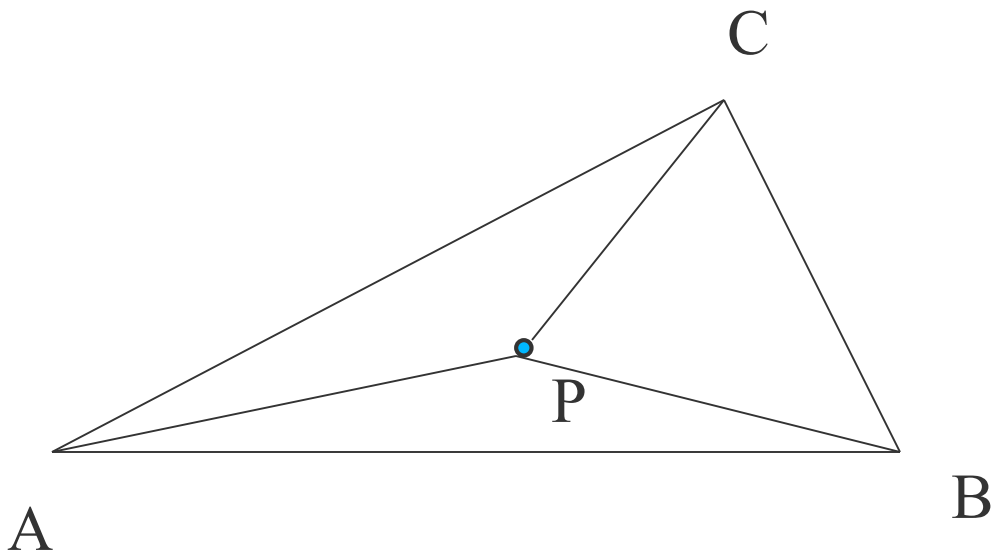
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Last lecture
  - Ray tracing I
    - Basic ray tracing
    - Recursive ray tracing algorithm
    - Intersection computations

- Today
  - Advanced acceleration structures
    - Theoretical Background
    - Hierarchical Grids, kd-Trees, Octrees
    - Bounding Volume Hierarchies
  - Ray bundles

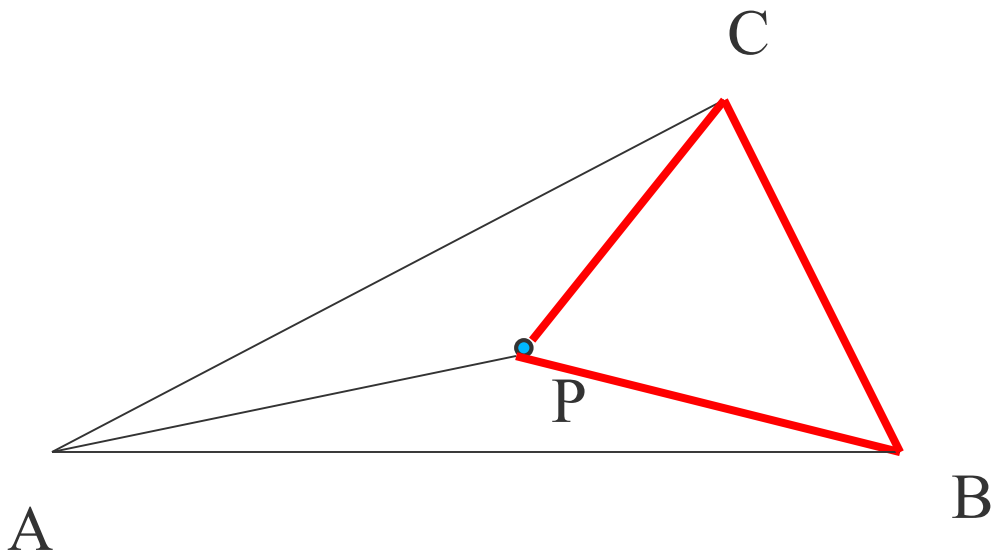
- Next lecture
  - Dynamically changing scenes

Barycentric Coordinates



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

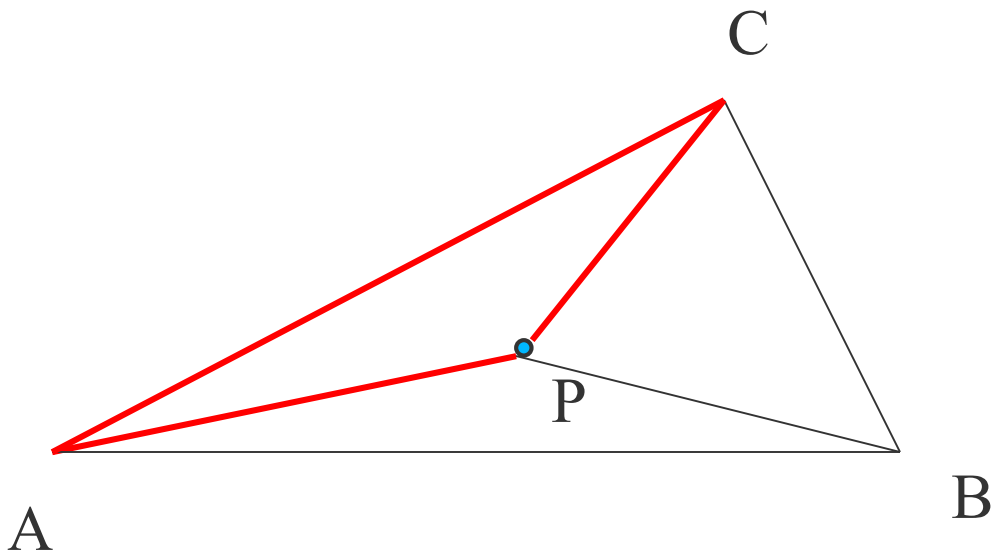
Barycentric Coordinates



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

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

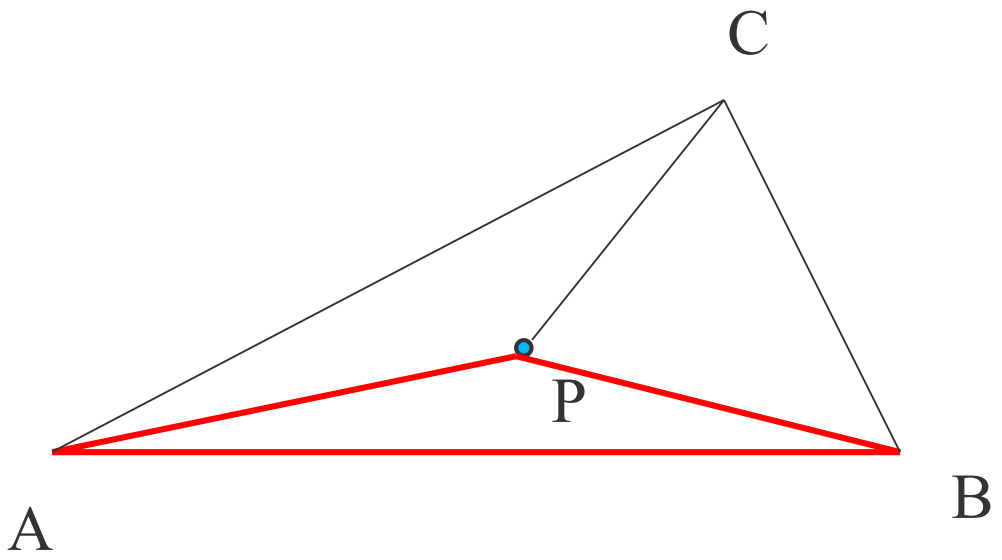
Barycentric Coordinates



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

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

Barycentric Coordinates



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

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

Fast Triangle Intersection

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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.

Fast Triangle Intersection

- Given ray $R(t) = O + tD$
- and triangle V_0, V_1, V_2
- Main idea:
  - compute distance to triangle plane t
  - compute intersection coordinates u, v $(1-u-v), u, v$
- Point on triangle

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2$$

barycentric interpolation of vertices

Fast Triangle Intersection

- Point on triangle

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2$$

- Triangle intersection

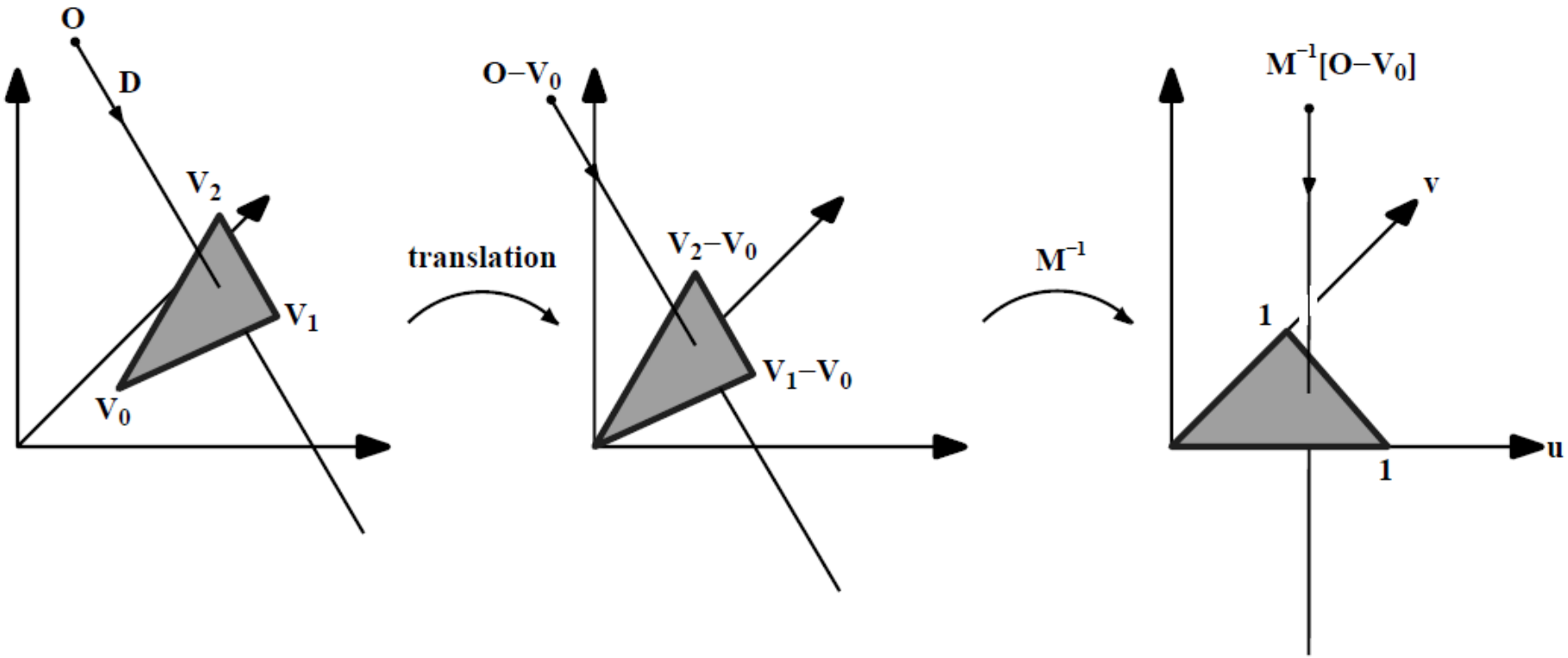
$$O + tD = (1-u-v)V_0 + uV_1 + vV_2$$

- or

$$\begin{pmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = O - V_0$$

Fast Triangle Intersection

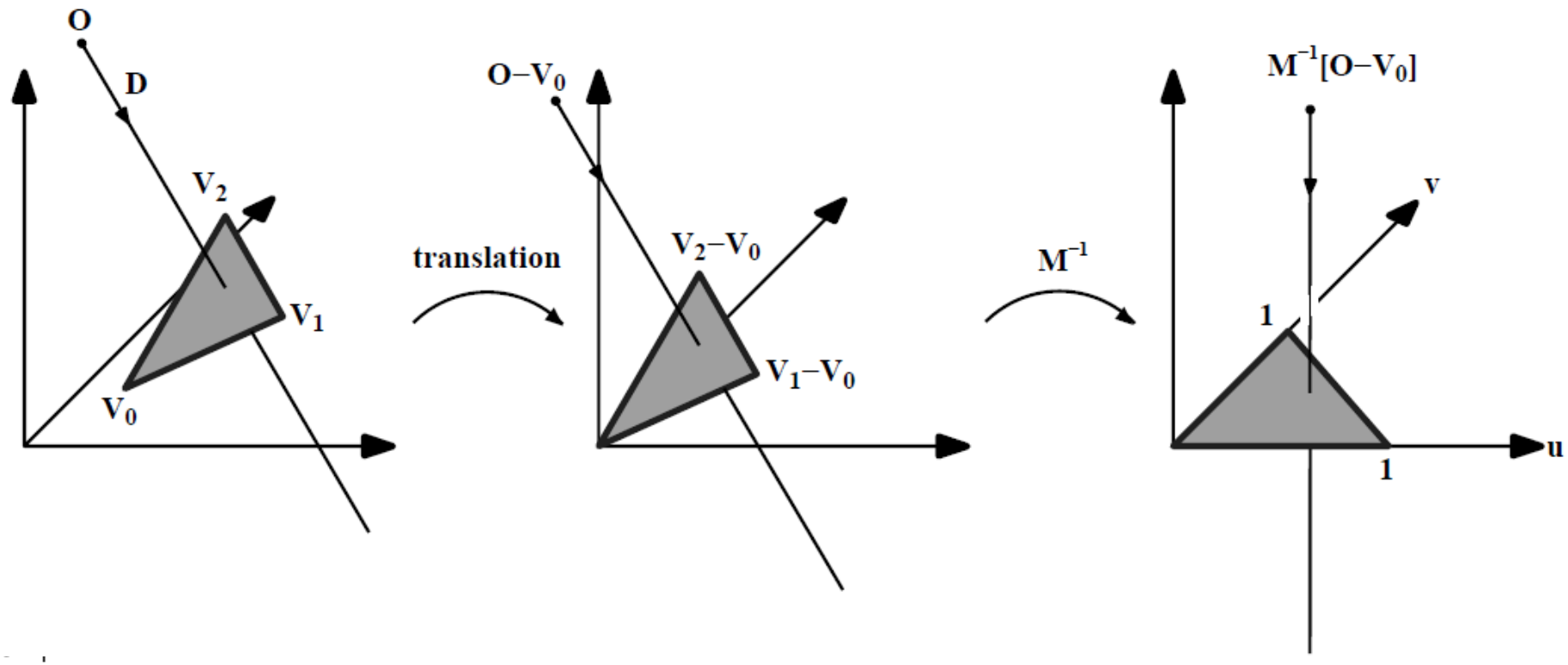
$$\underbrace{\begin{pmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{pmatrix}}_{M} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = O - V_0$$



Fast Triangle Intersection

- compute

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = M^{-1}\left(O - V_0\right)$$



Fast Triangle Intersection

$$\begin{pmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = O - V_0$$

- Inversion using Cramer's rule

- using

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\begin{vmatrix} -D & E_1 & E_2 \end{vmatrix}} \begin{pmatrix} \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{pmatrix}$$

$$E_1 = V_1 - V_0, \quad E_2 = V_2 - V_0, \quad T = O - V_0$$

Fast Triangle Intersection

- … after some further transformations

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{pmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{pmatrix}$$

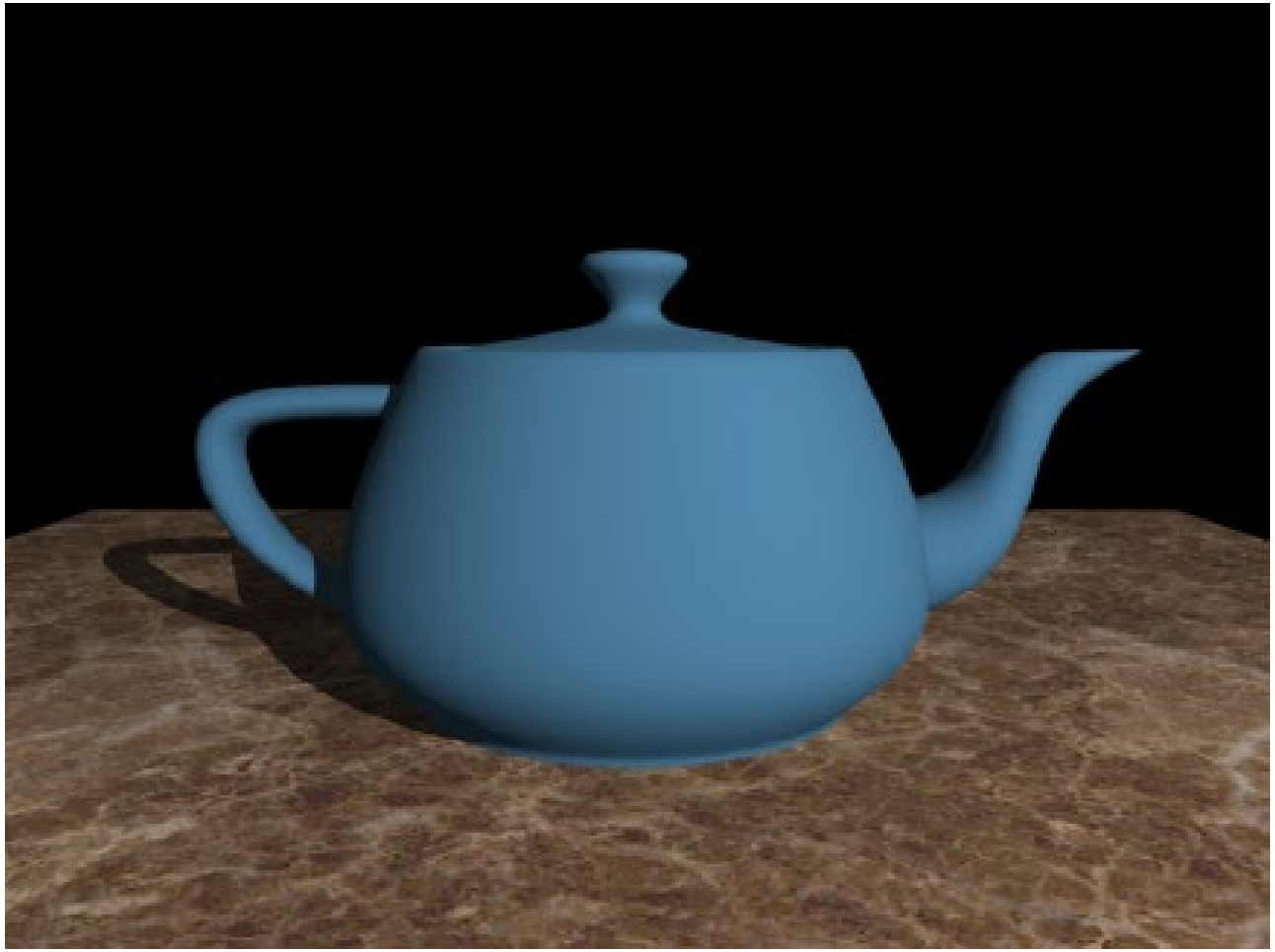
- using $\left|A \quad B \quad C\right| = -(A \times C) \cdot B = -(C \times B) \cdot A$
- Reuse terms and
- Code: see paper

Cost of Ray Tracing

Why can it be sooooo slow?

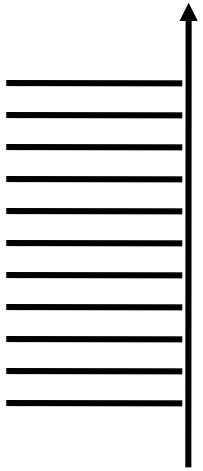
Cost of Naïve Raytracing

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



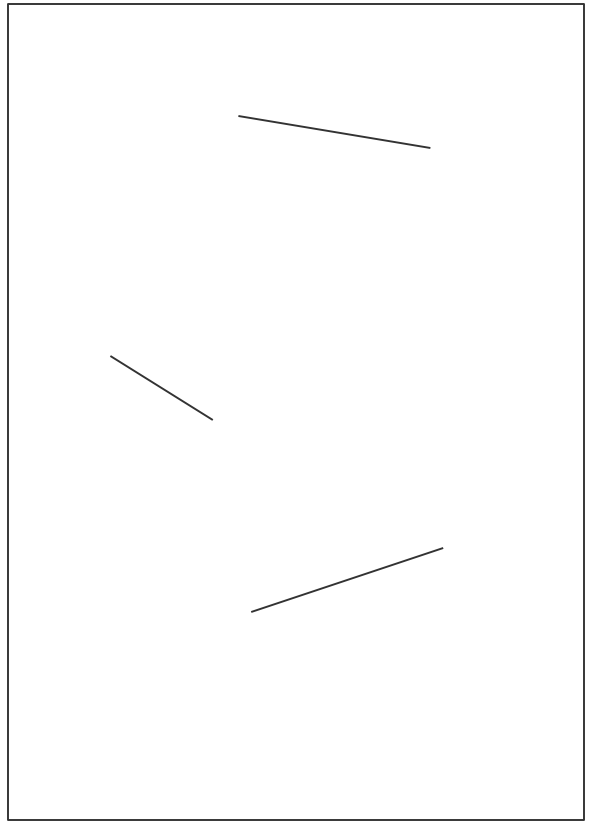
Theoretical Background

- 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

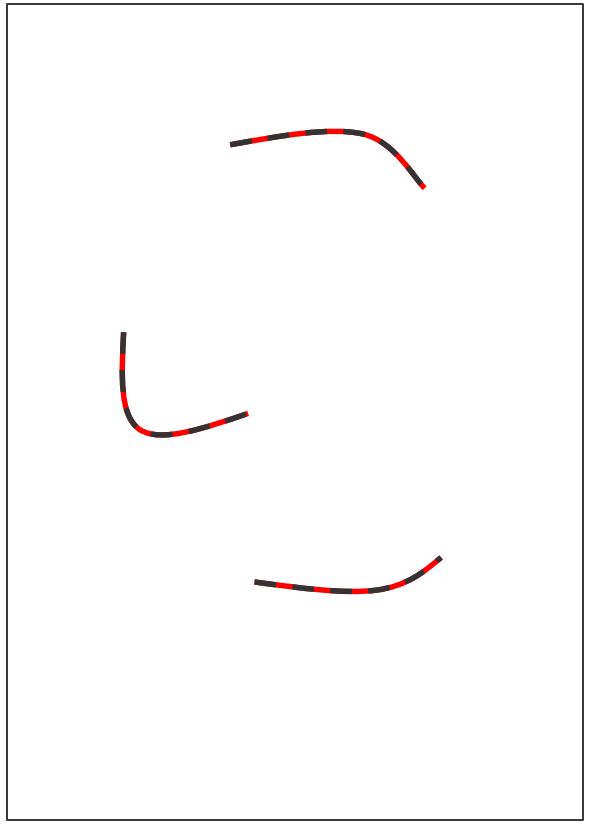


Worst case RT scene:
Ray barely misses
every primitive

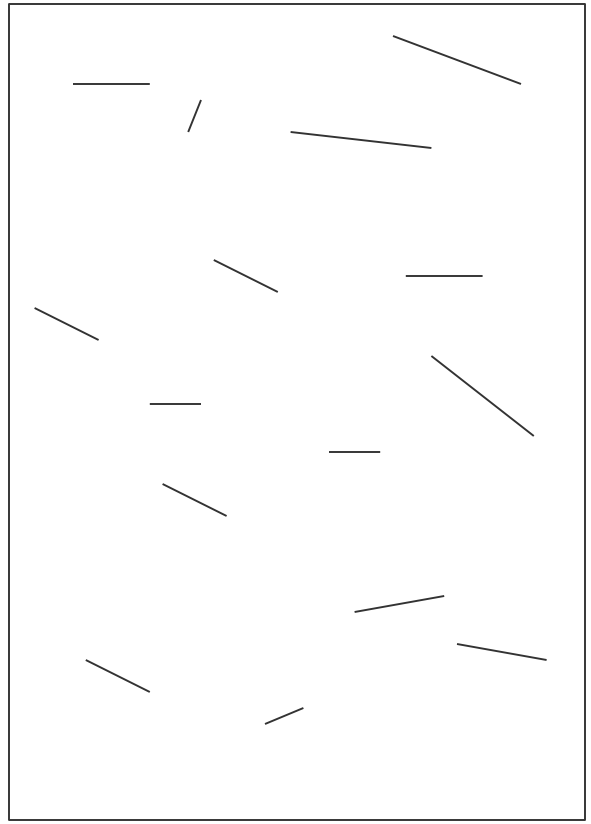
Depth Complexity vs. Polygon Count



few polygons
low depth complexity



many polygons
low depth complexity



many polygons
high depth complexity

Ray Tracing Acceleration

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

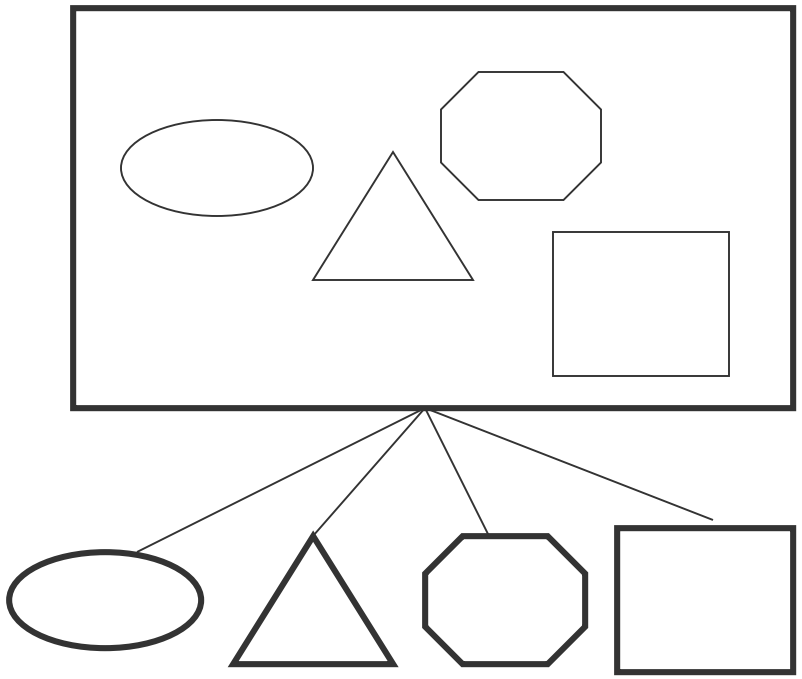
- Intersect ray with all objects
  - Way too expensive
- Faster intersection algorithms
  - Still same complexity O(n)
- Fewer intersection computations
  - Space partitioning (often hierarchical)
    - Grid, hierarchies of grids
    - Octree
    - Binary space partition (BSP) or kd-tree
    - Bounding volume hierarchy (BVH)
  - Directional partitioning (not very useful)
  - 5D partitioning (space and direction)
    - 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, ...

Spatial Acceleration Structures

How to arrange objects for faster ray tracing?

Aggregate Objects

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

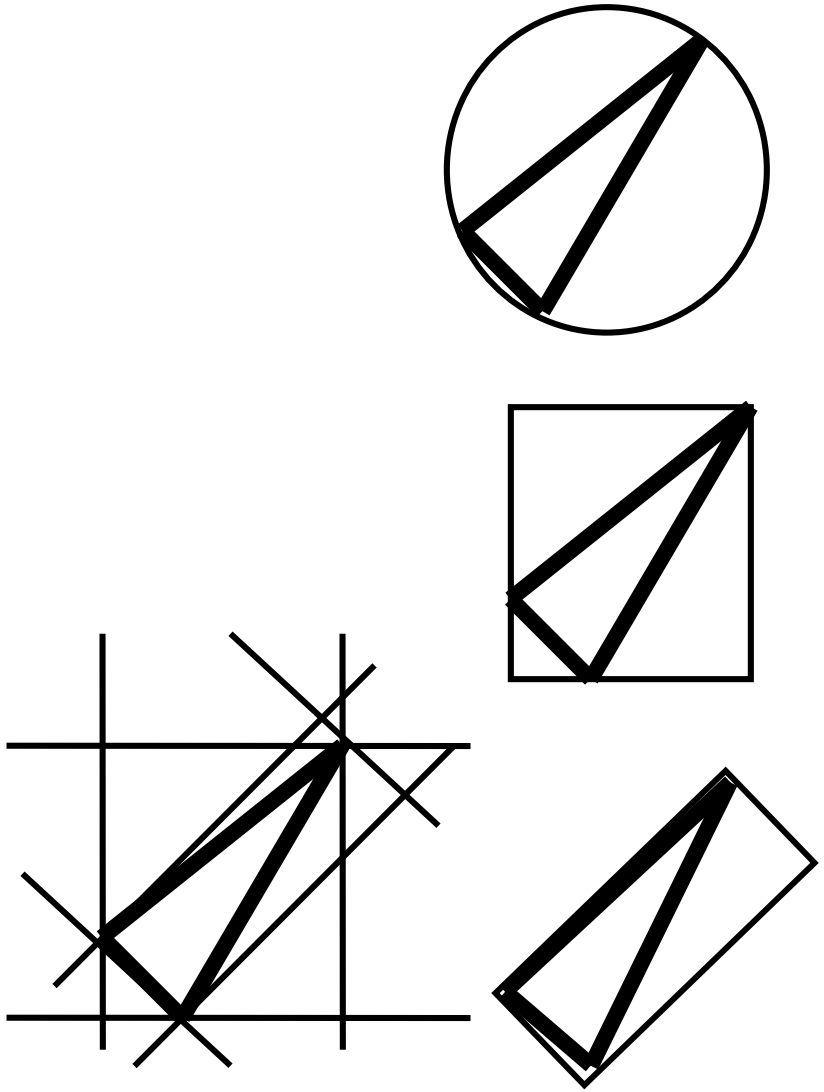


Structures

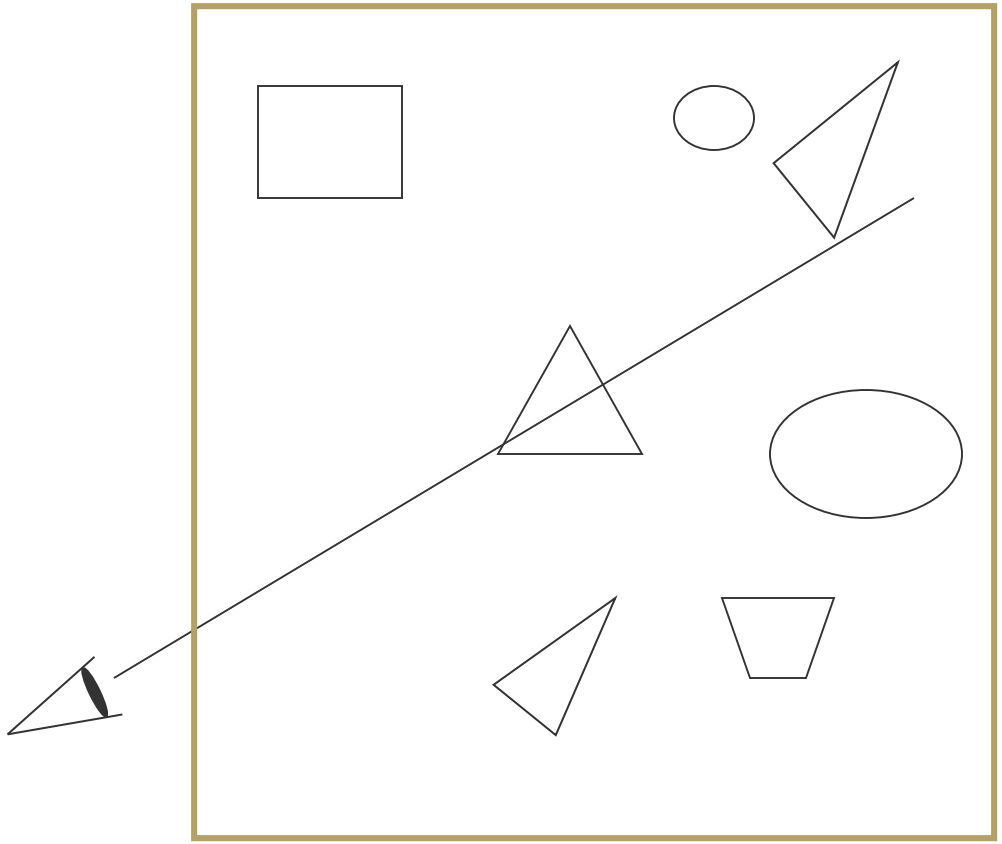
- Bounding Volume Hierarchy (BVH)
- Grid
- Hierarchical Grid
- Quadtree / Octree
- BSP-Tree
- kD-Tree

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 (AABB)
  - 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

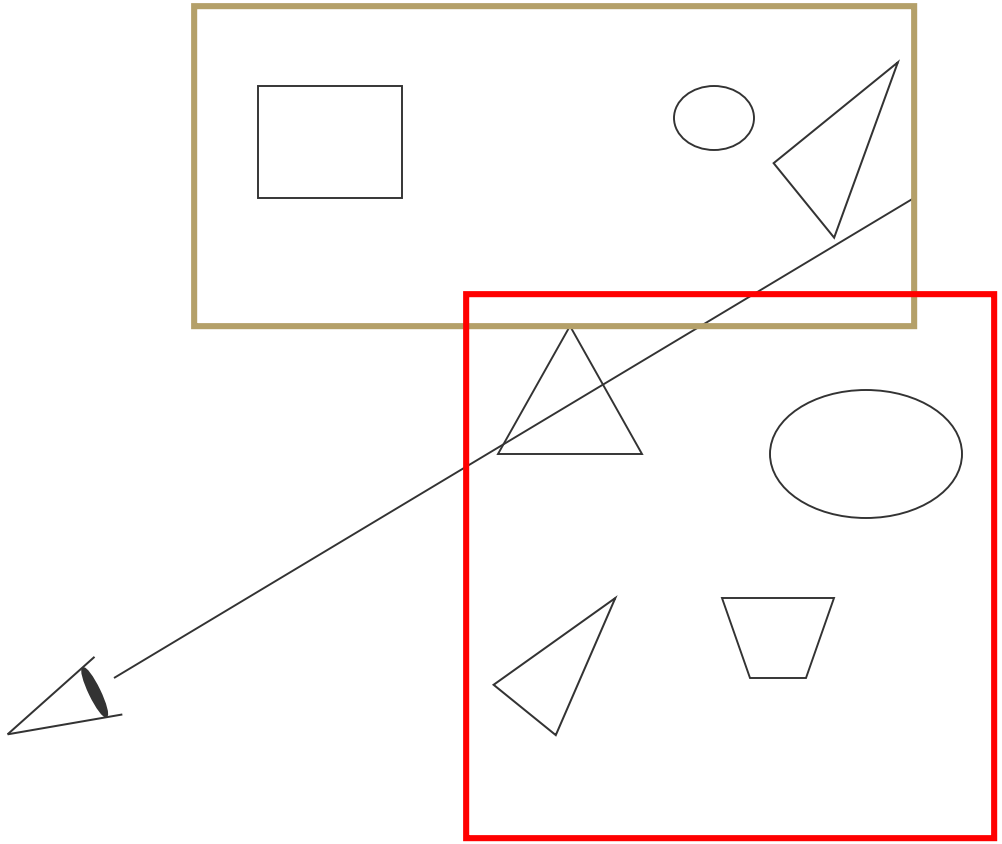


Bounding Volume Hierarchies



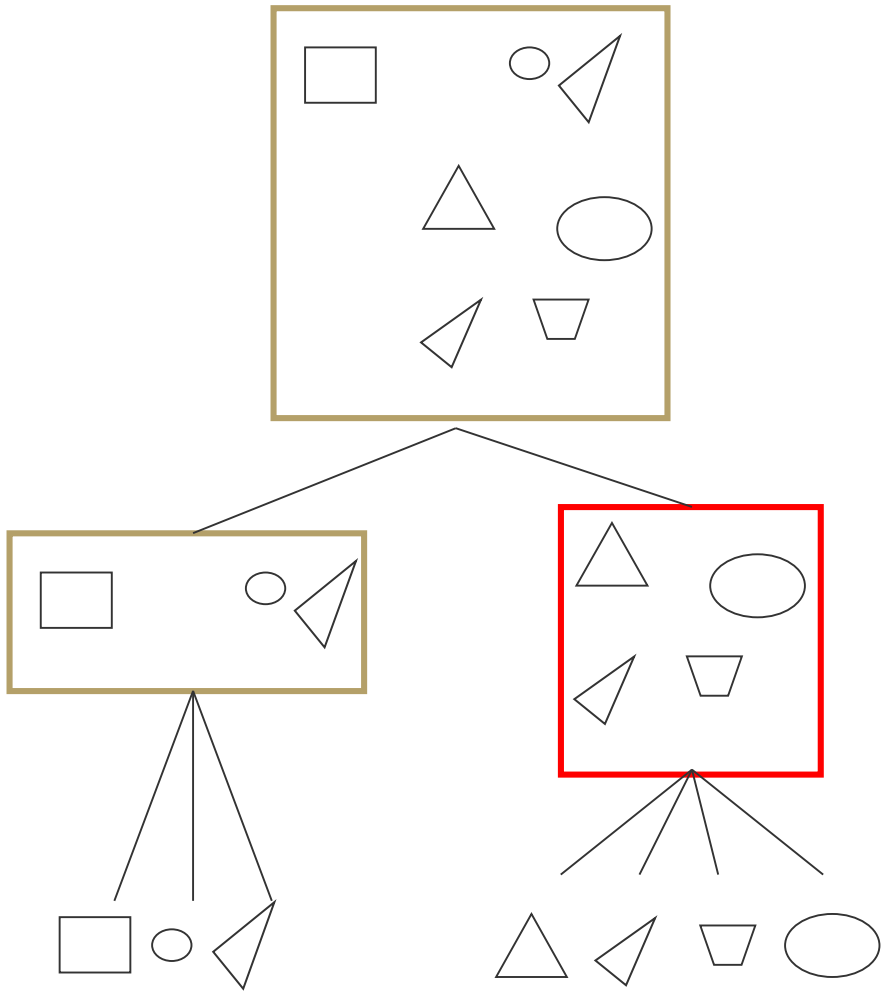
Bounding Volume Hierarchies

- Hierarchy of groups of objects



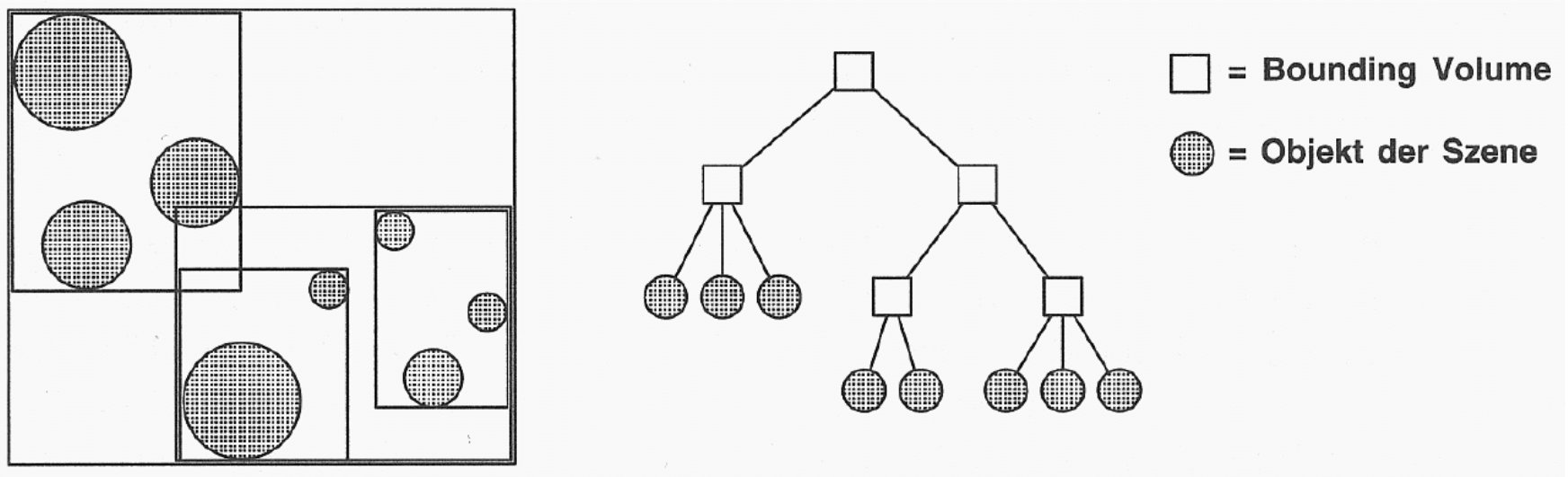
Bounding Volume Hierarchies

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



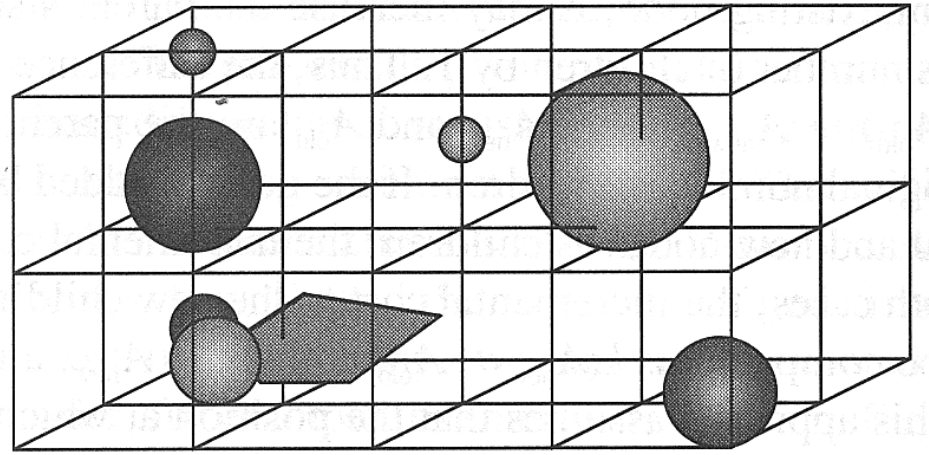
Bounding Volume Hierarchies

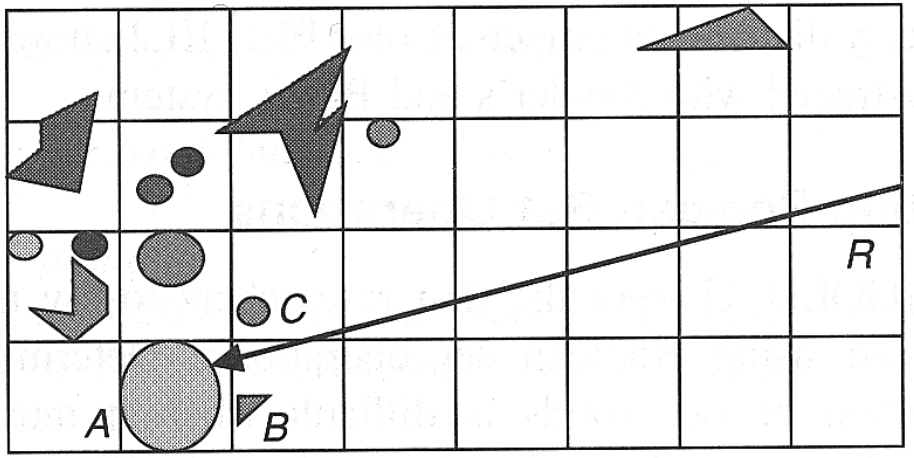
- Idea:
  - Organize bounding volumes hierarchically into new BVs
- Advantages:
  - Very good adaptivity
  - Efficient traversal O(log N)
  - Often used in ray tracing systems
- Problems
  - How to arrange BVs?



Grid

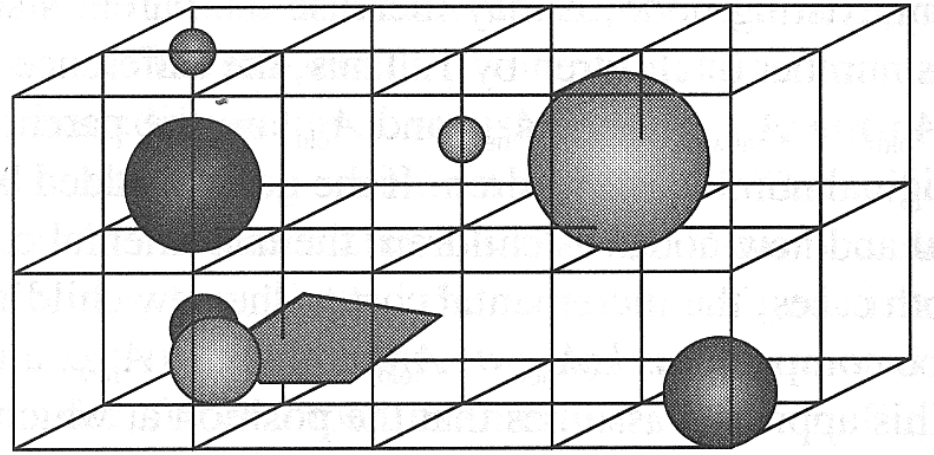
- Grid
  - Partitioning with equal, fixed sized „voxels“
- Building a grid structure
  - Partition the bounding box (bb)
  - Resolution: often $\sqrt[3]{n}$
  - Inserting objects
    - Trivial: insert into all voxels overlapping objects bounding box
    - Easily optimized
- Traversal
  - Iterate through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

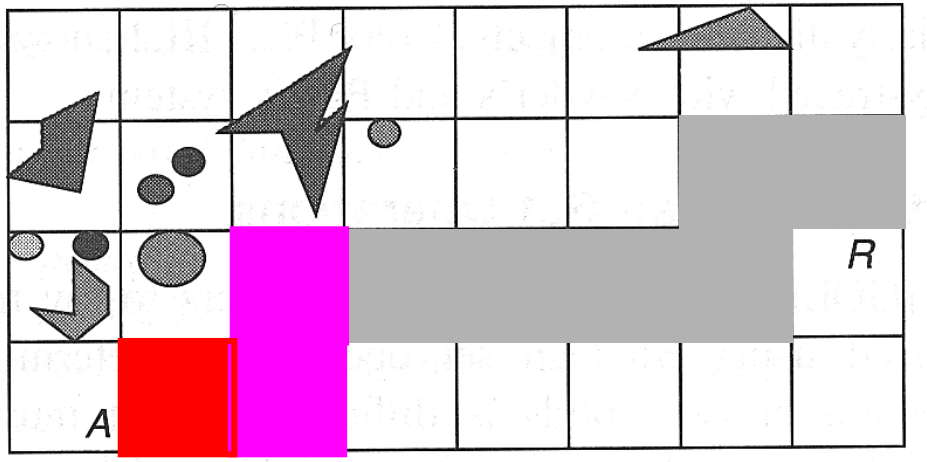




Grid

- 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



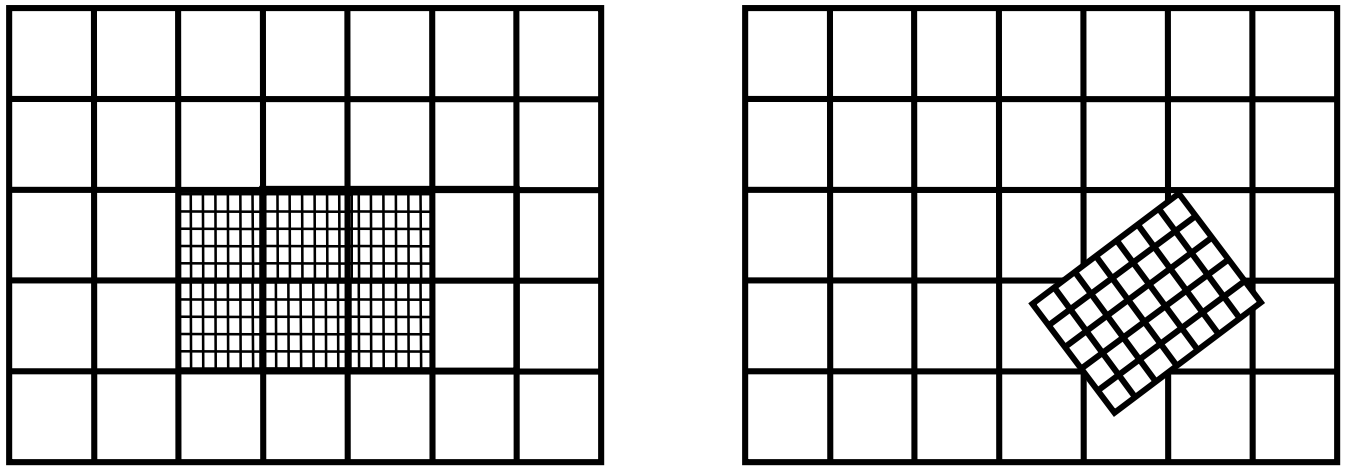


Grid: Issues

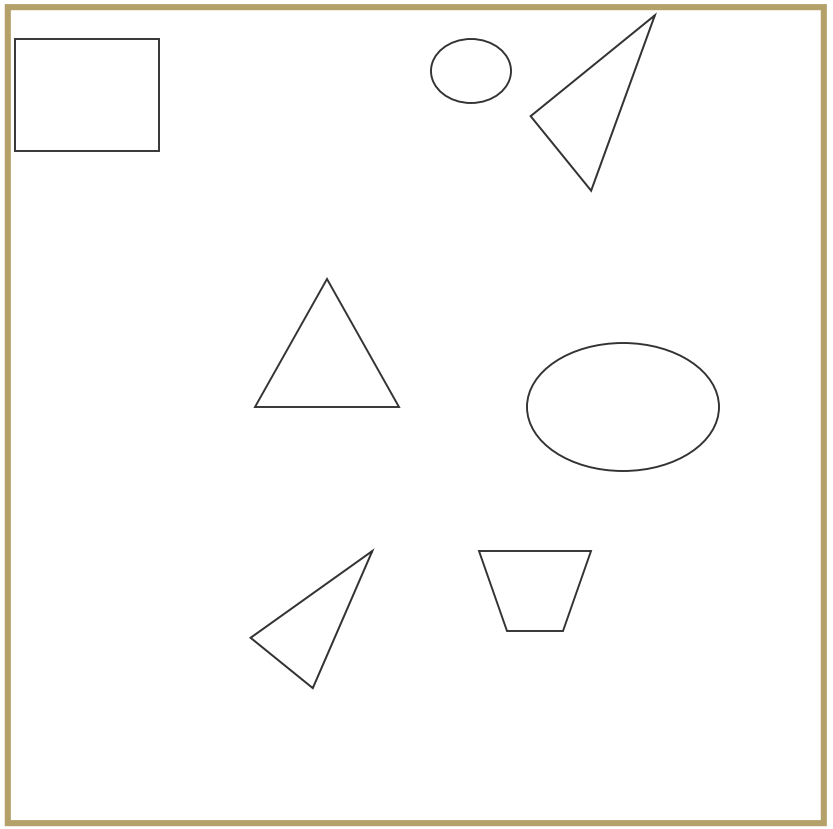
- Grid traversal
  - Requires enumeration of voxel along ray
    - → 3D-DDA, modified Bresenham (later)
  - Simple and hardware-friendly
- Grid resolution
  - Strongly scene dependent
  - Cannot adapt to local density of objects
    - Problem: „Teapot in a stadium“
  - Possible solution: grids within 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

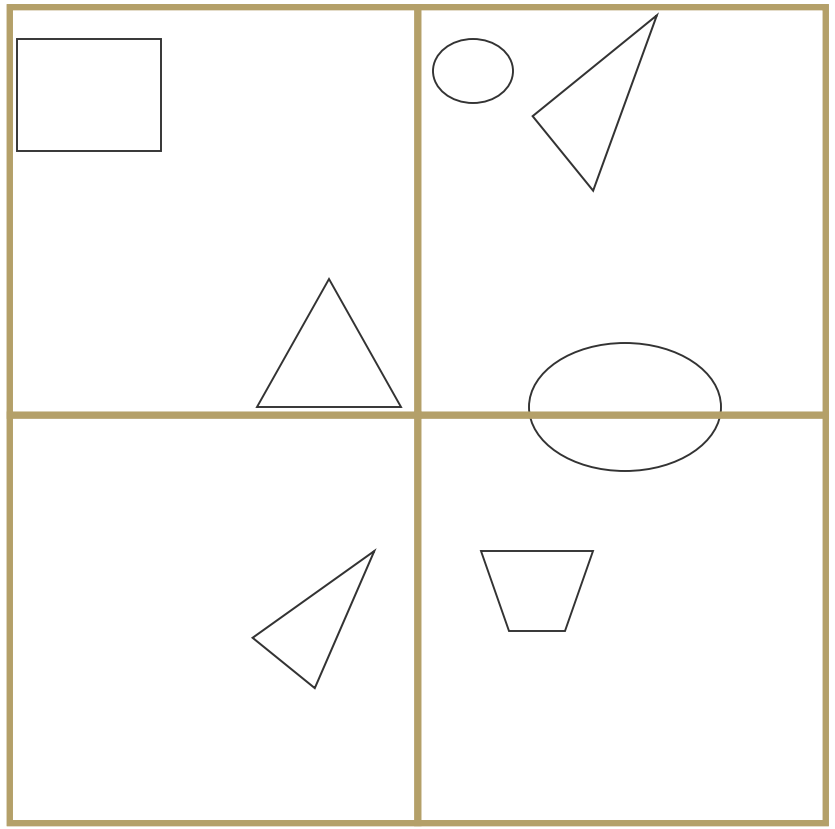
- Simple building algorithm
  - Coarse grid for entire scene
  - Recursively create grids in high-density voxels
  - Problem: What is the right resolution for each level?
- Advanced algorithm
  - Place cluster of objects in separate grids
  - Insert these grids into parent grid
  - Problem: What are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



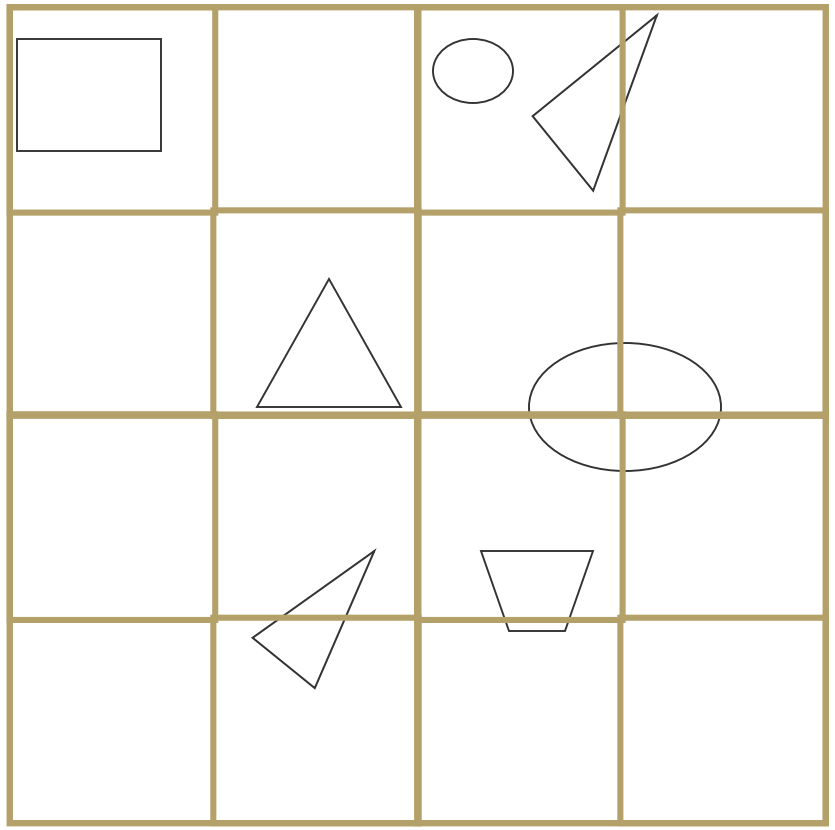
Quadtree – 2D example



Quadtree – 2D example

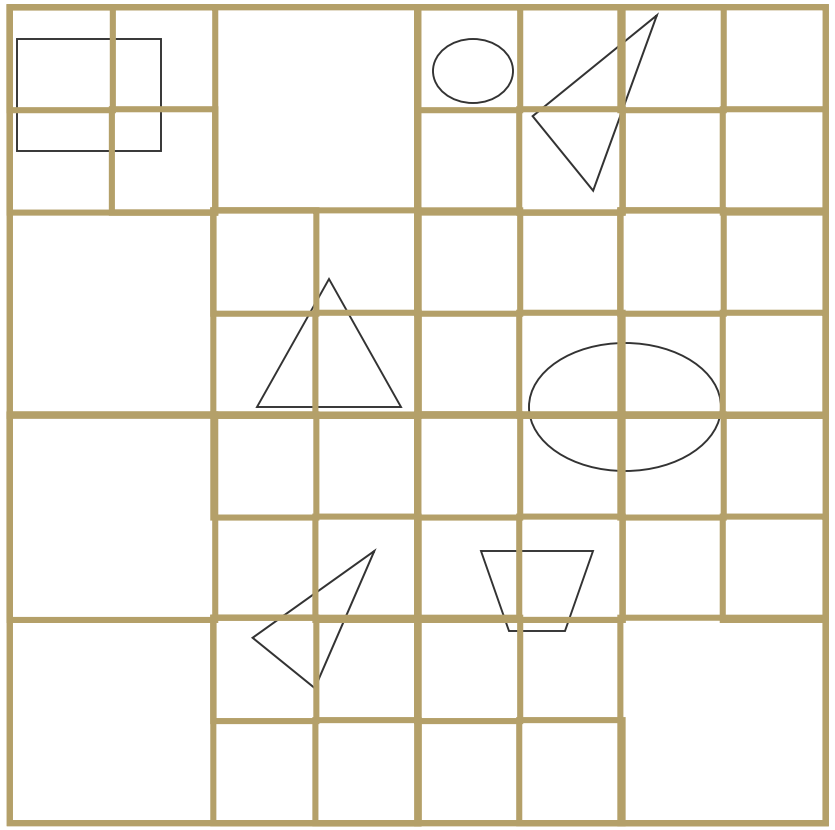


Quadtree – 2D example



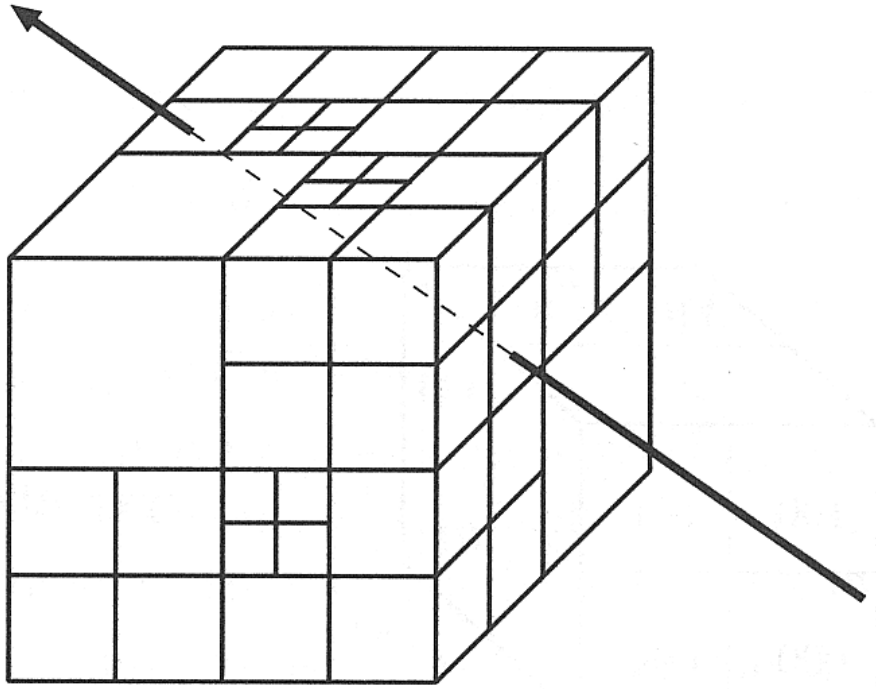
Quadtree – 2D example

- 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 …



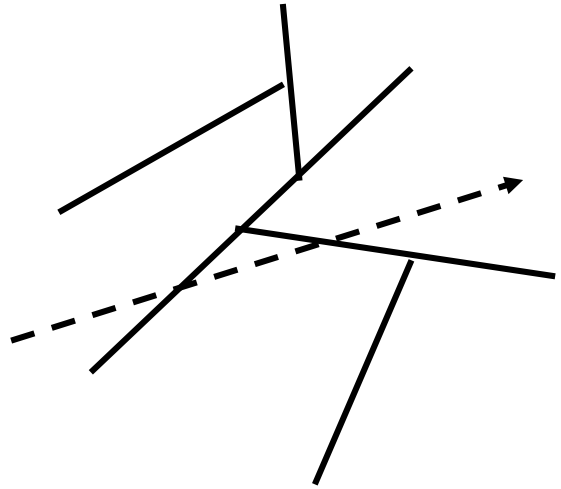
Higher Dimensions?

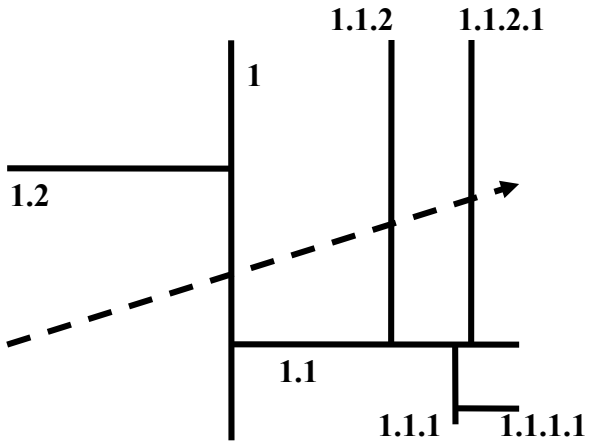
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Similar tree structure possible, but:
- In a d-dimensional space, each node will be split into 2^d children.

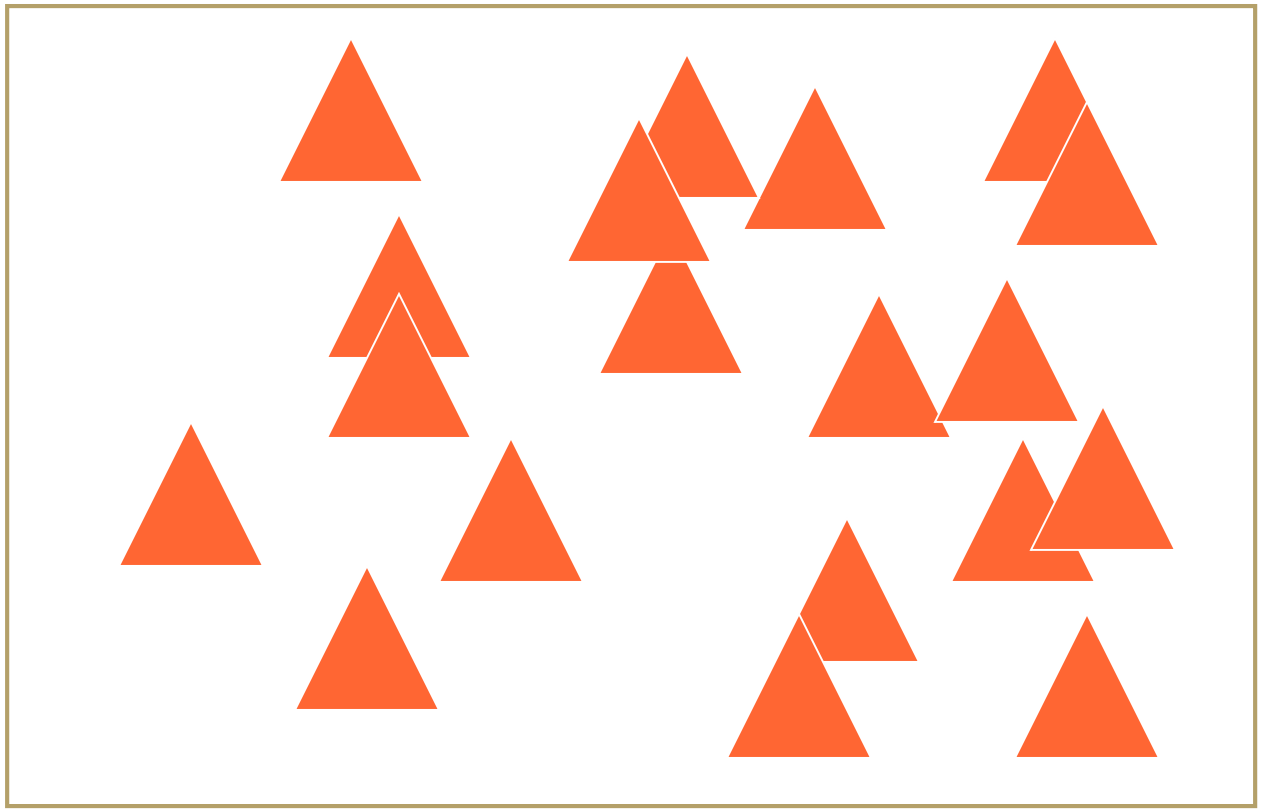
BSP- and kD-Trees

- 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 following slides for details



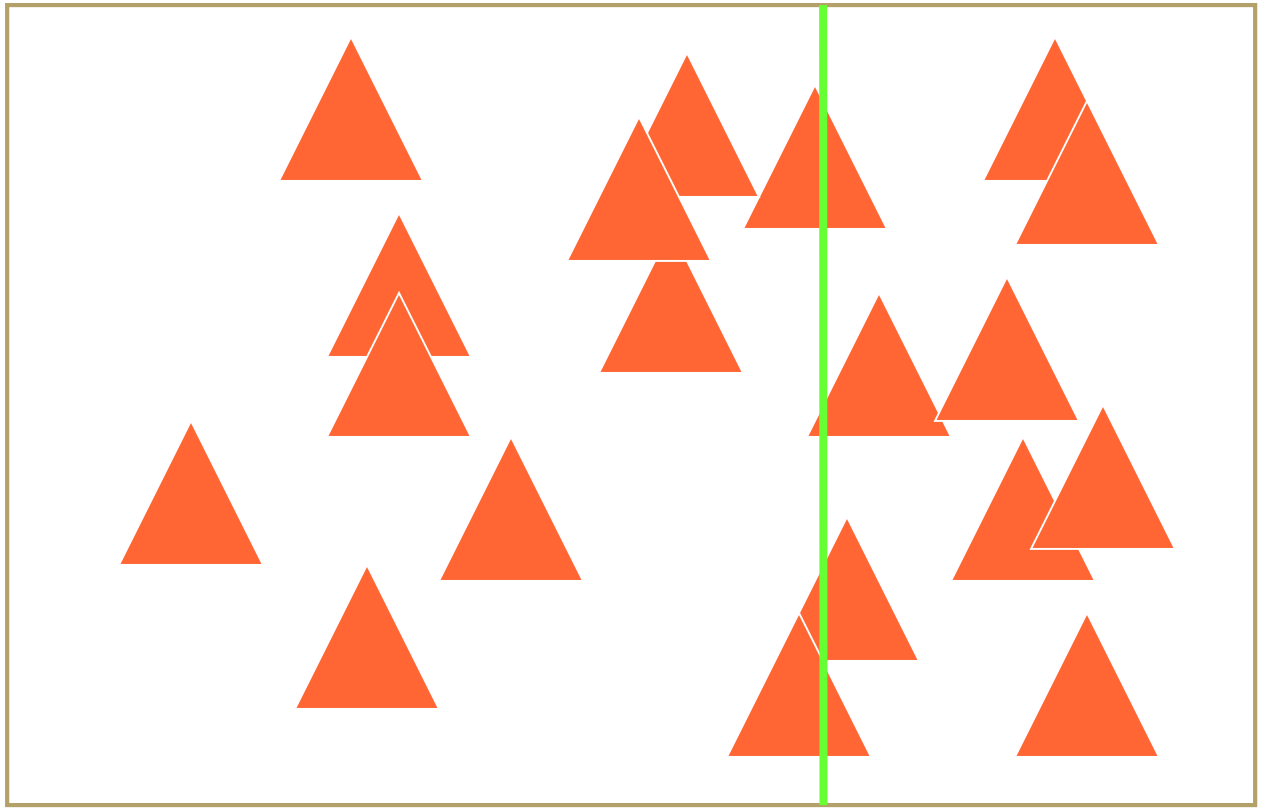


kD-Trees

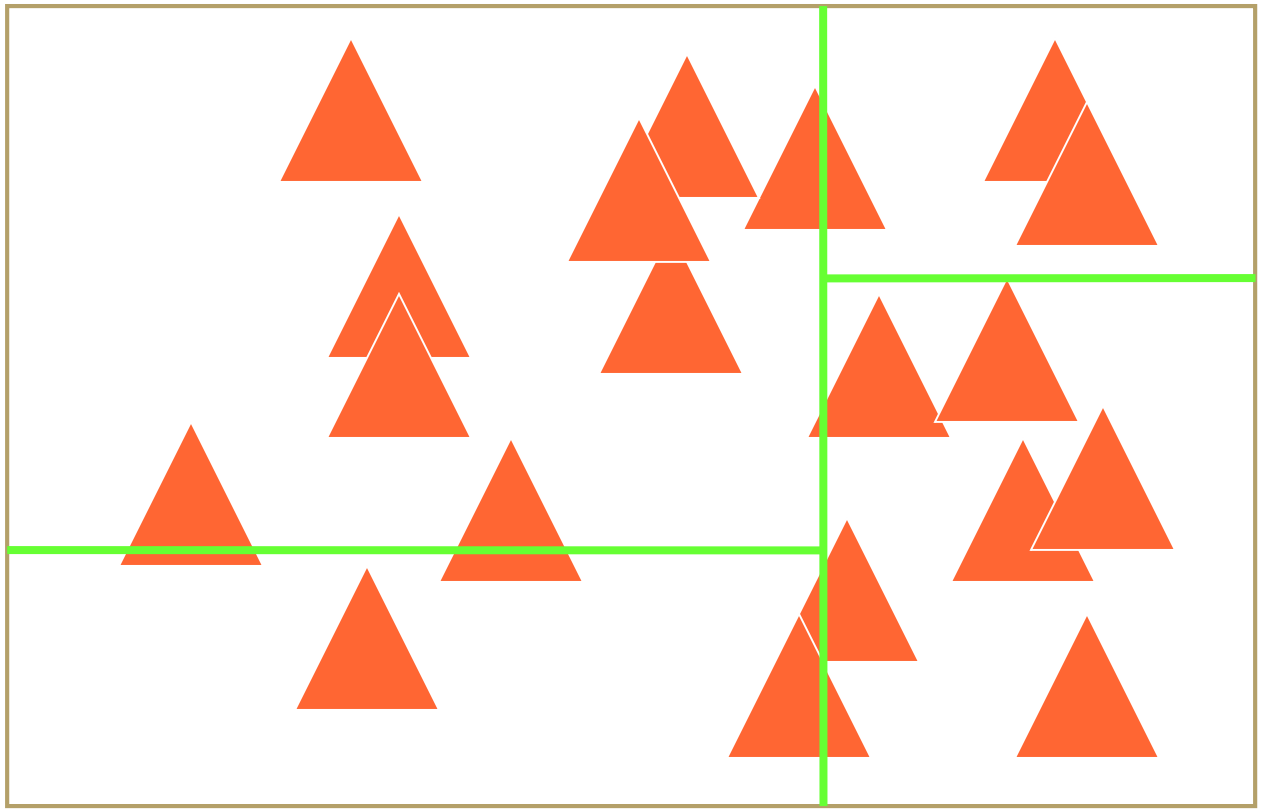


[following slides from Gordon Stoll – SIGGRAPH Course 2005]

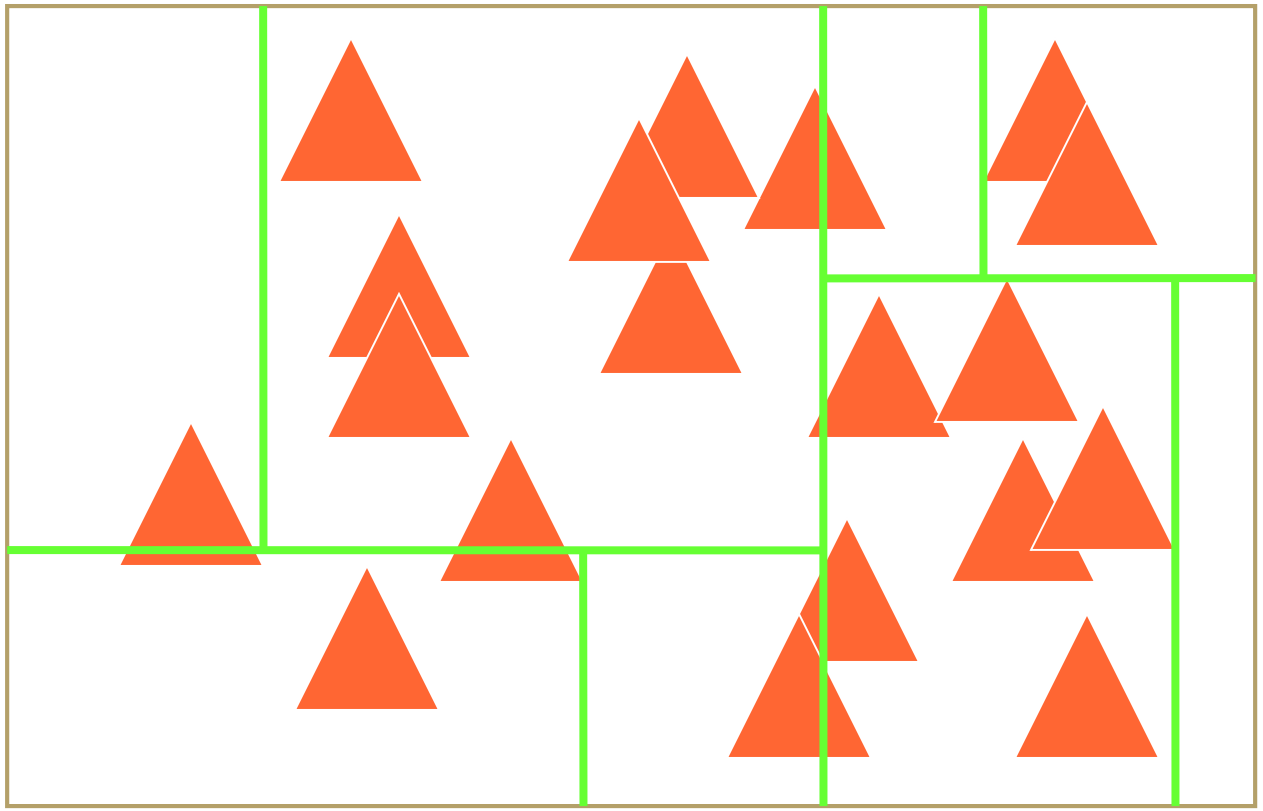
kD-Trees



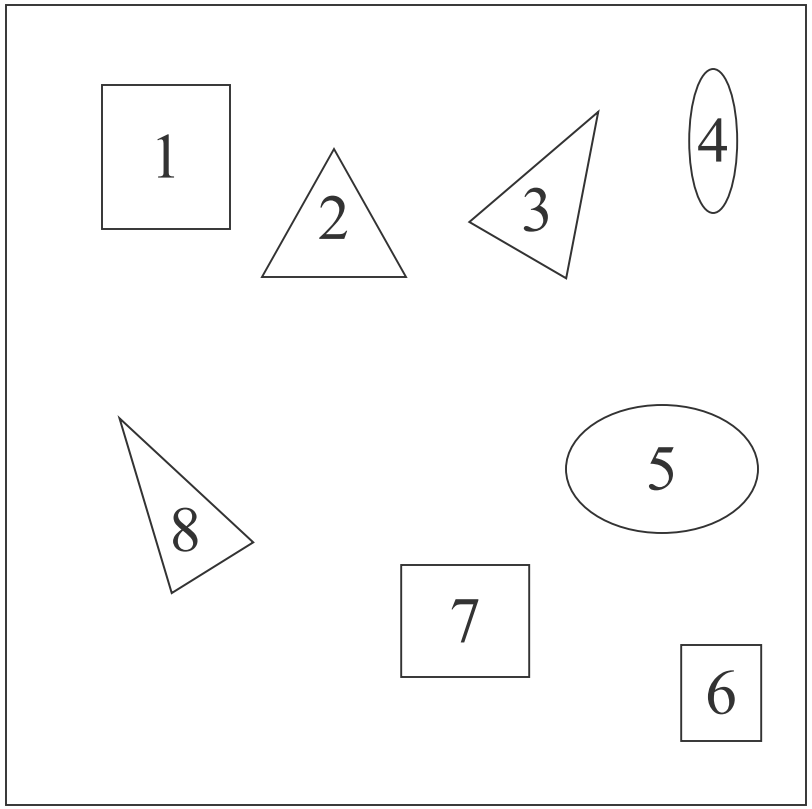
kD-Trees



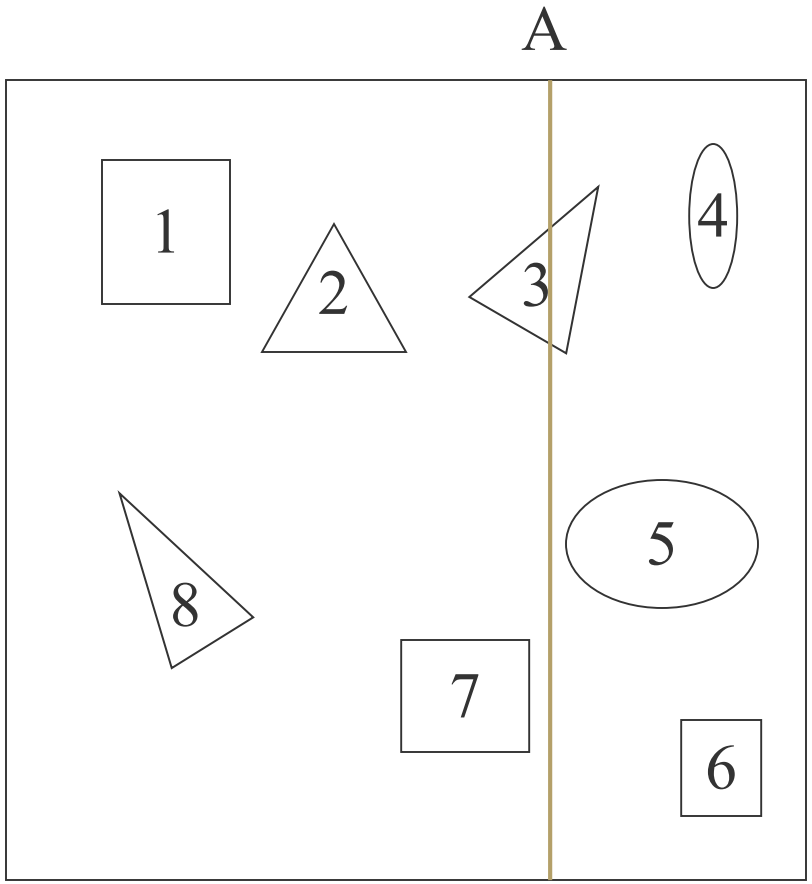
kD-Trees



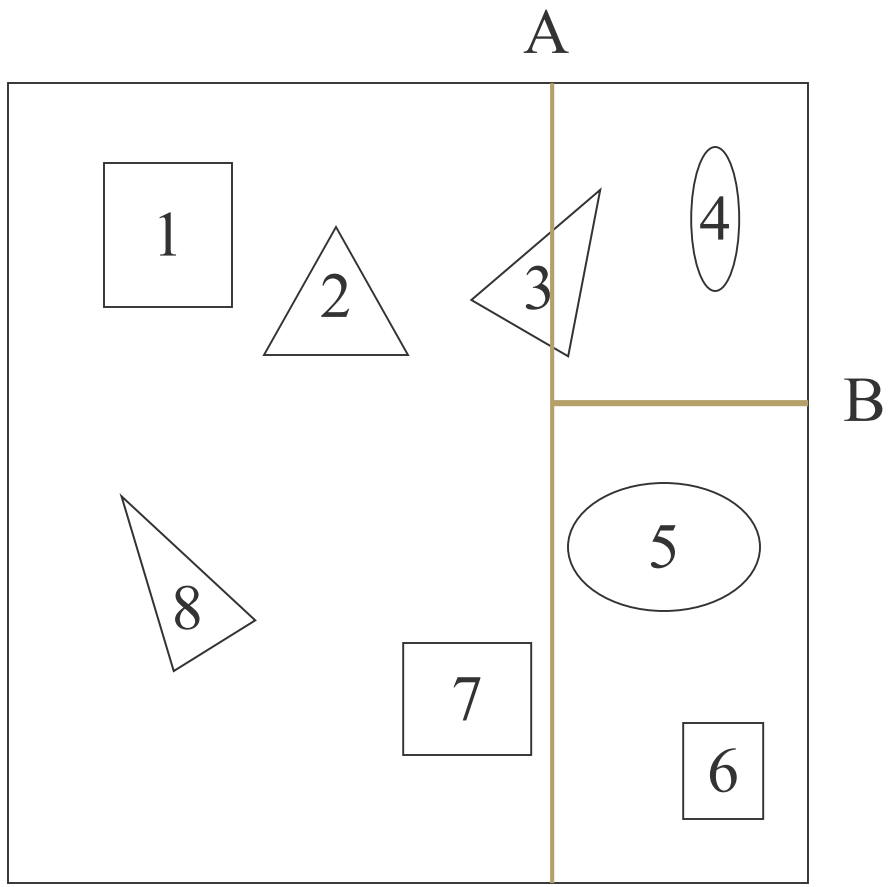
KD-Tree (explicit example)

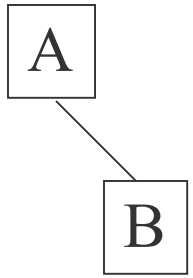


KD-Tree (explicit example)

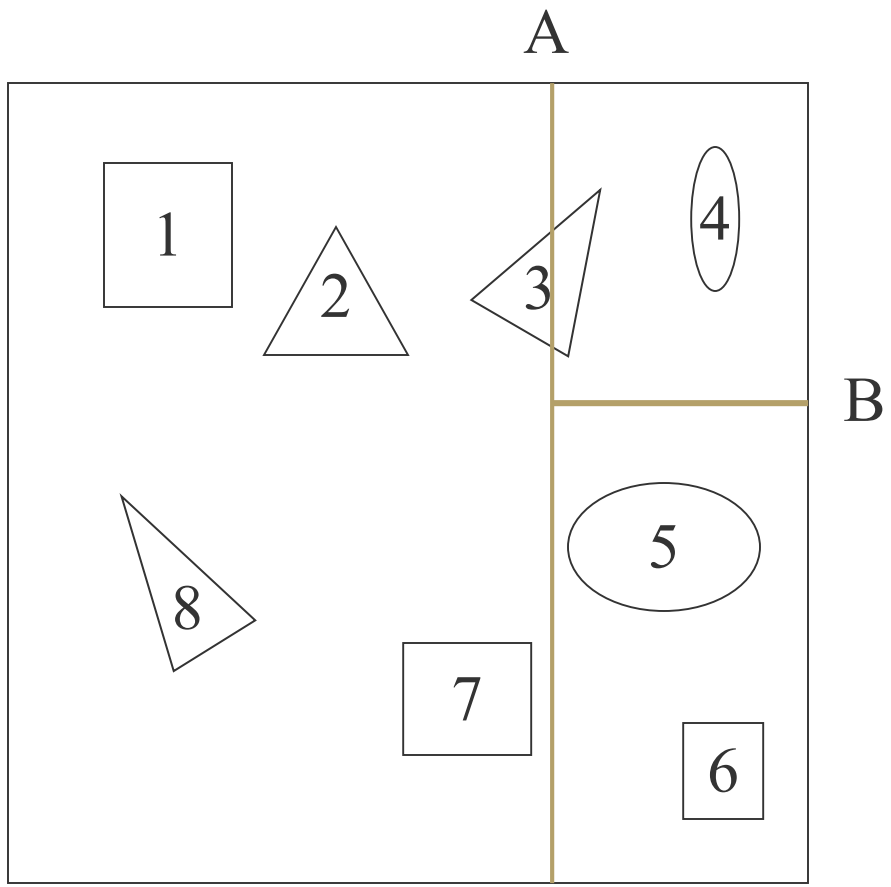


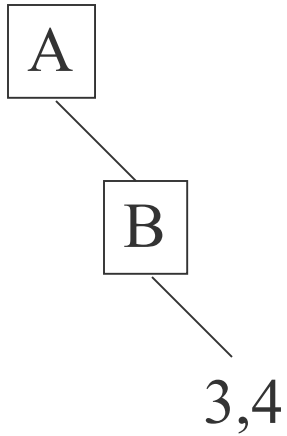
KD-Tree (explicit example)



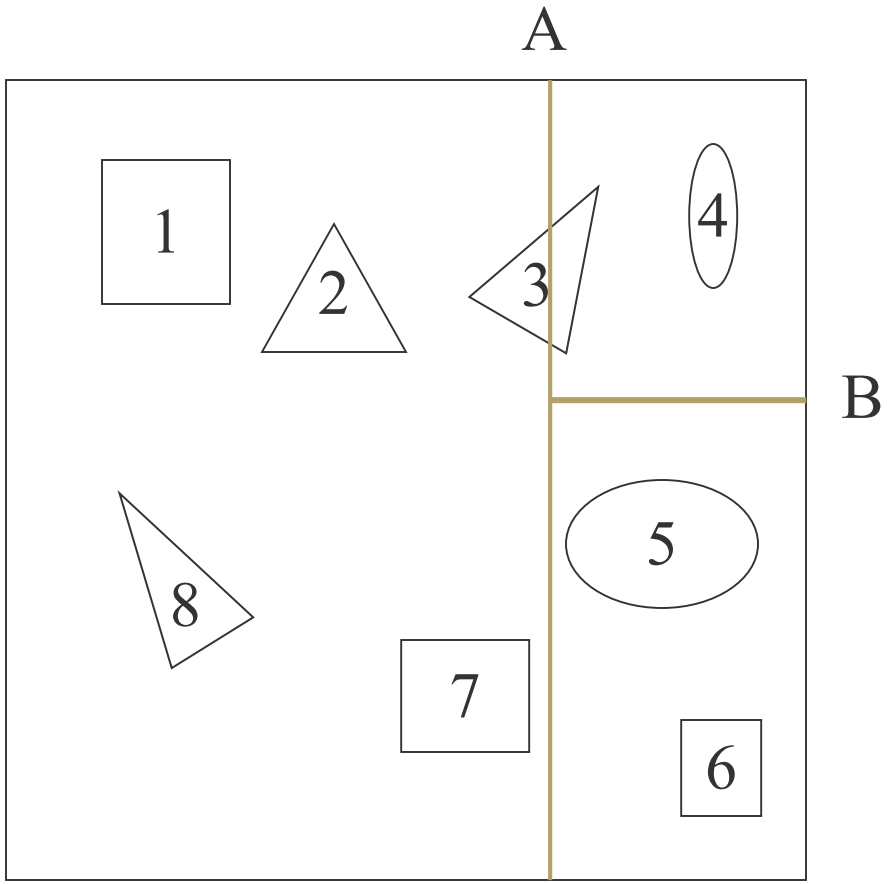


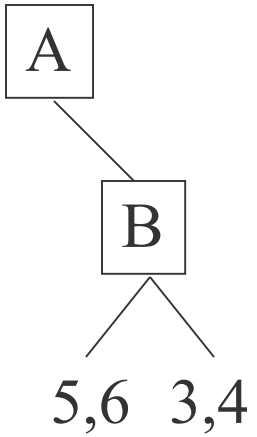
KD-Tree (explicit example)



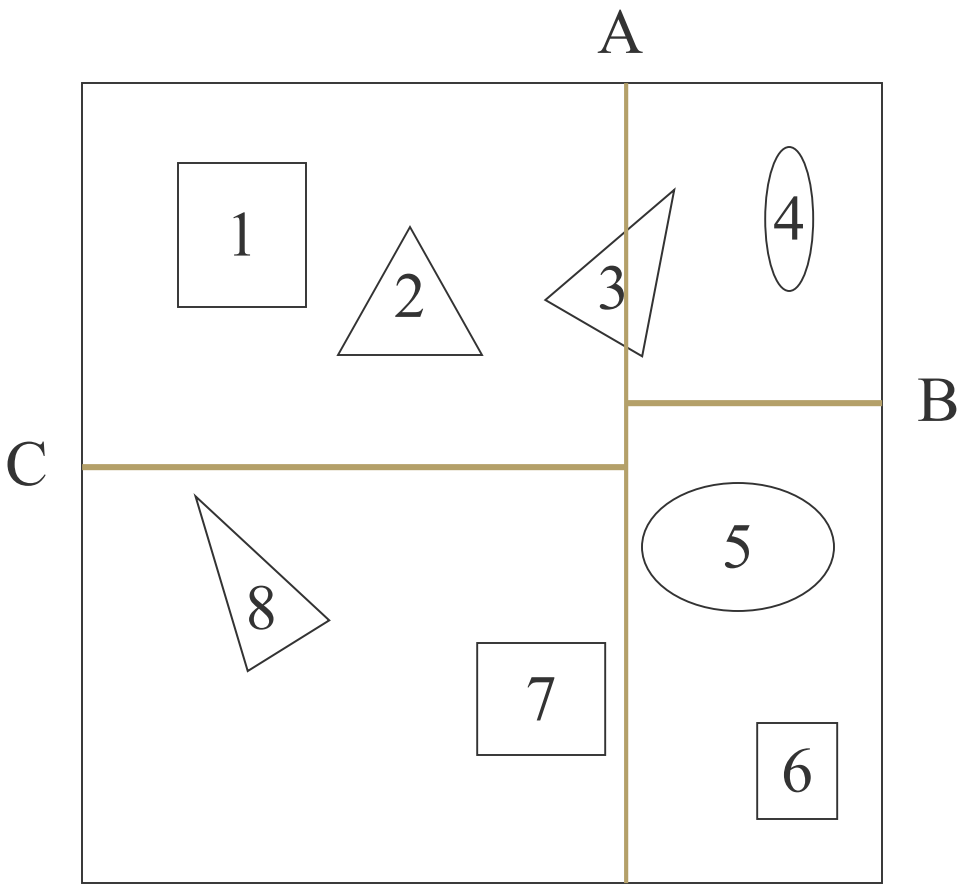


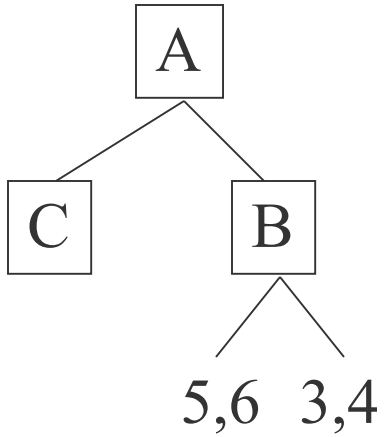
KD-Tree (explicit example)



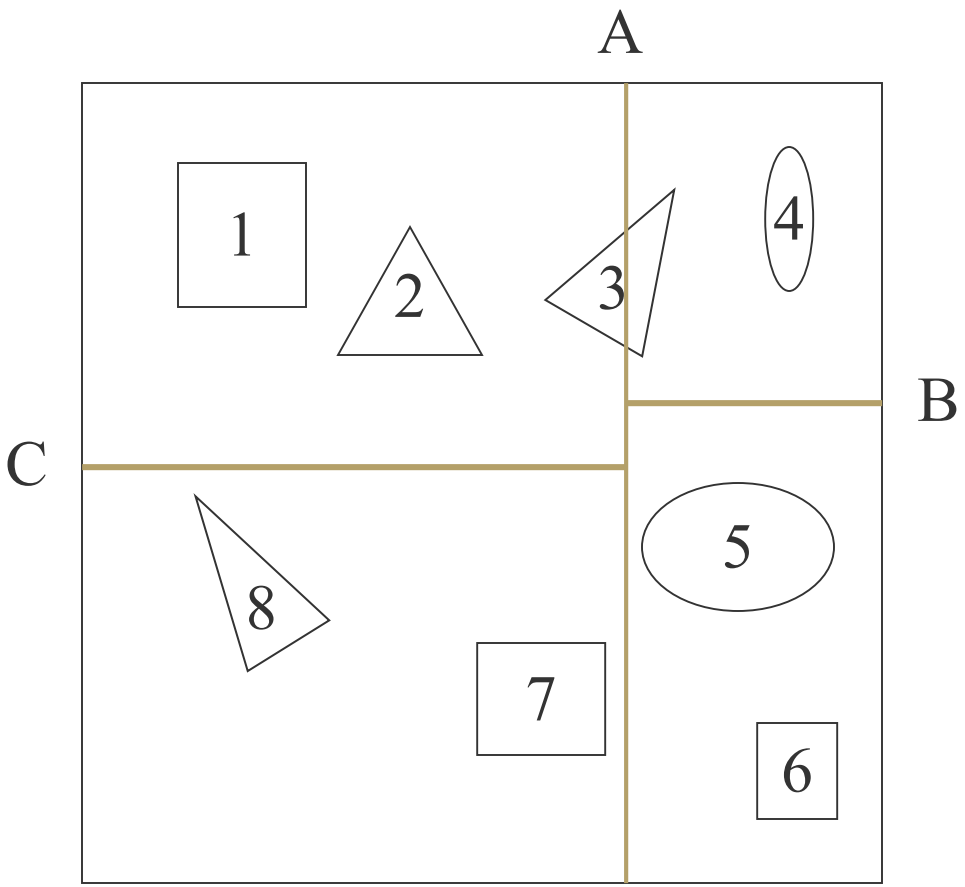


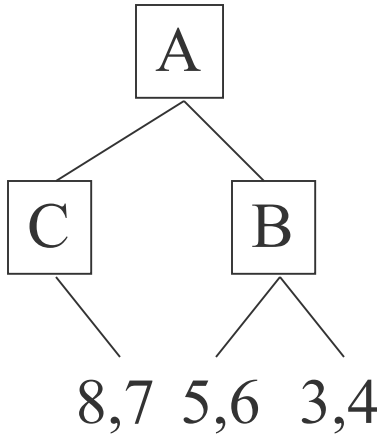
KD-Tree (explicit example)



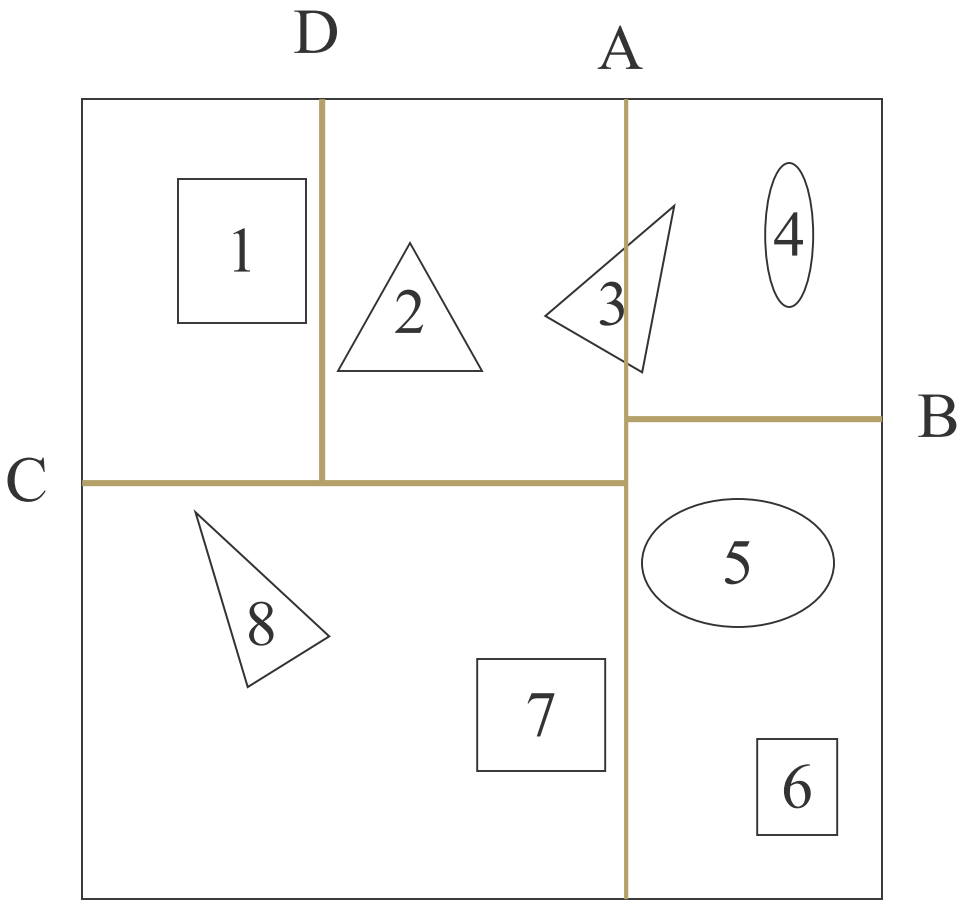


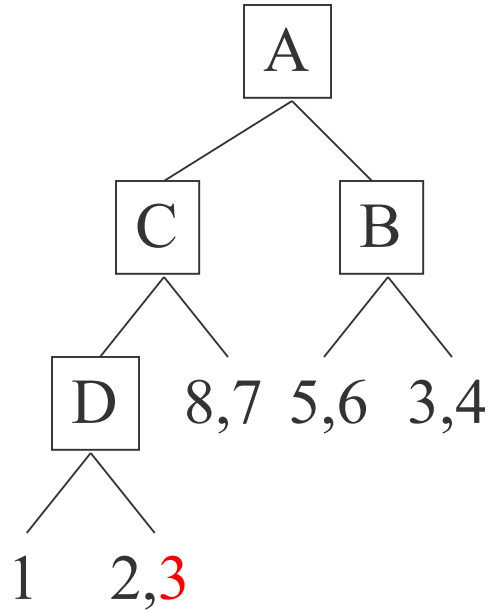
KD-Tree (explicit example)





KD-Tree (explicit example)





KD-Tree Traversal

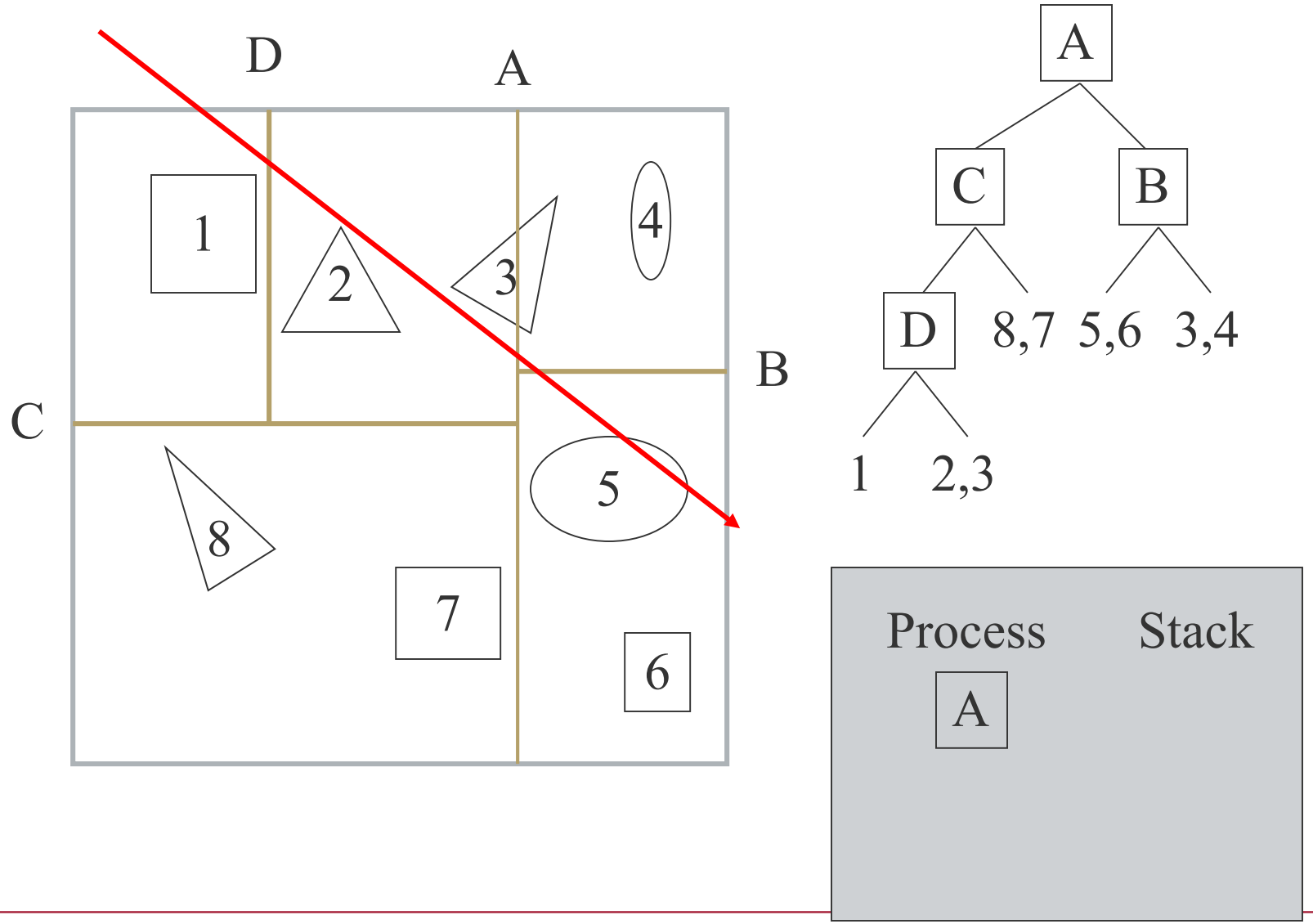
How to find the first intersection in a BSP / KD-Tree?

BSP-Tree Traversal

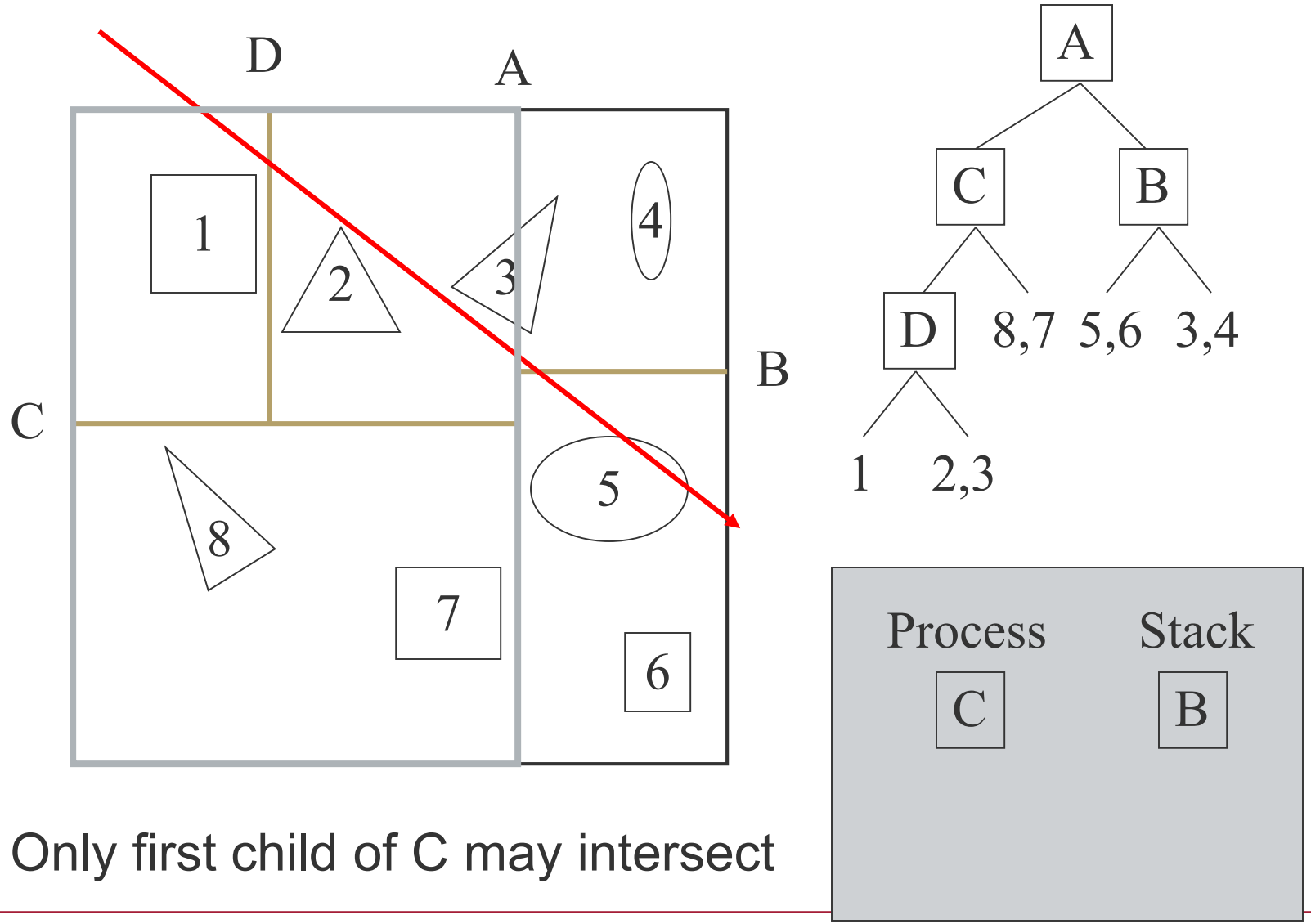
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- “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

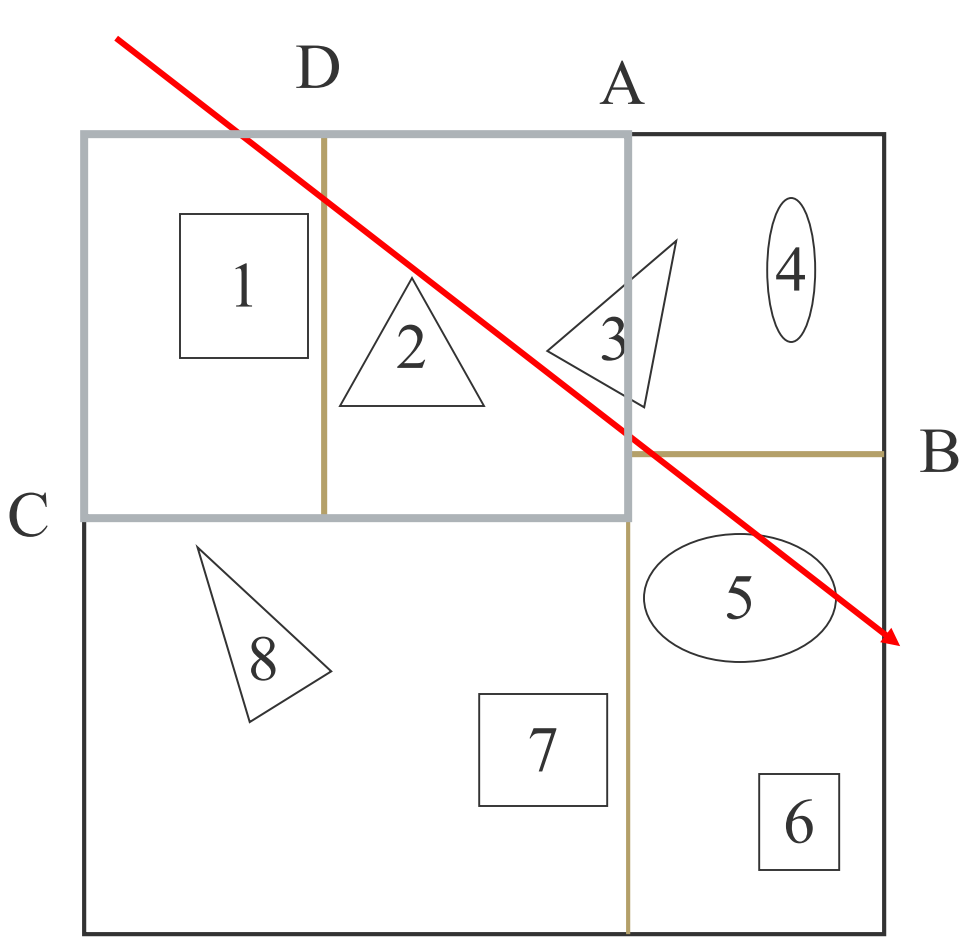


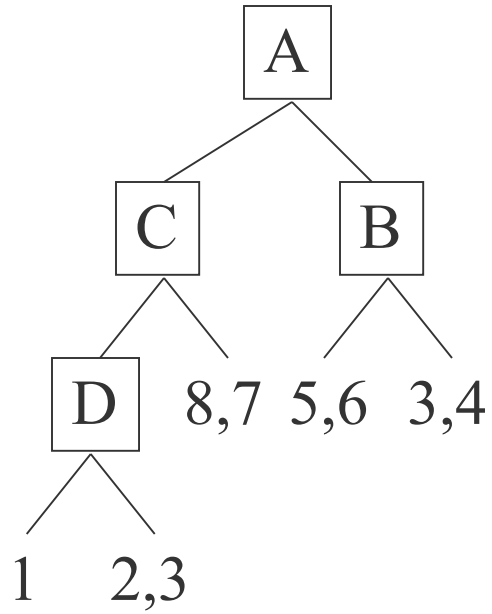
BSP-Tree Traversal

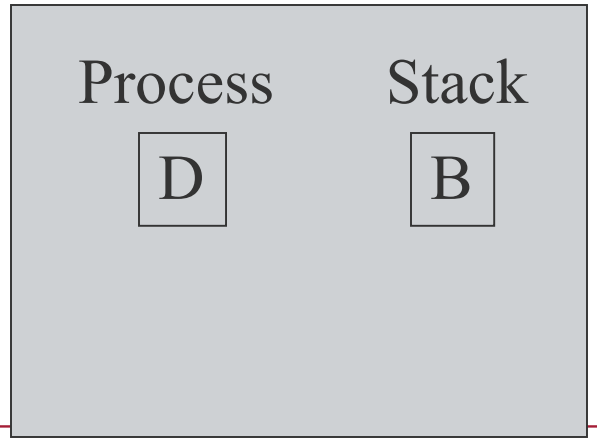


Only first child of C may intersect

BSP-Tree Traversal

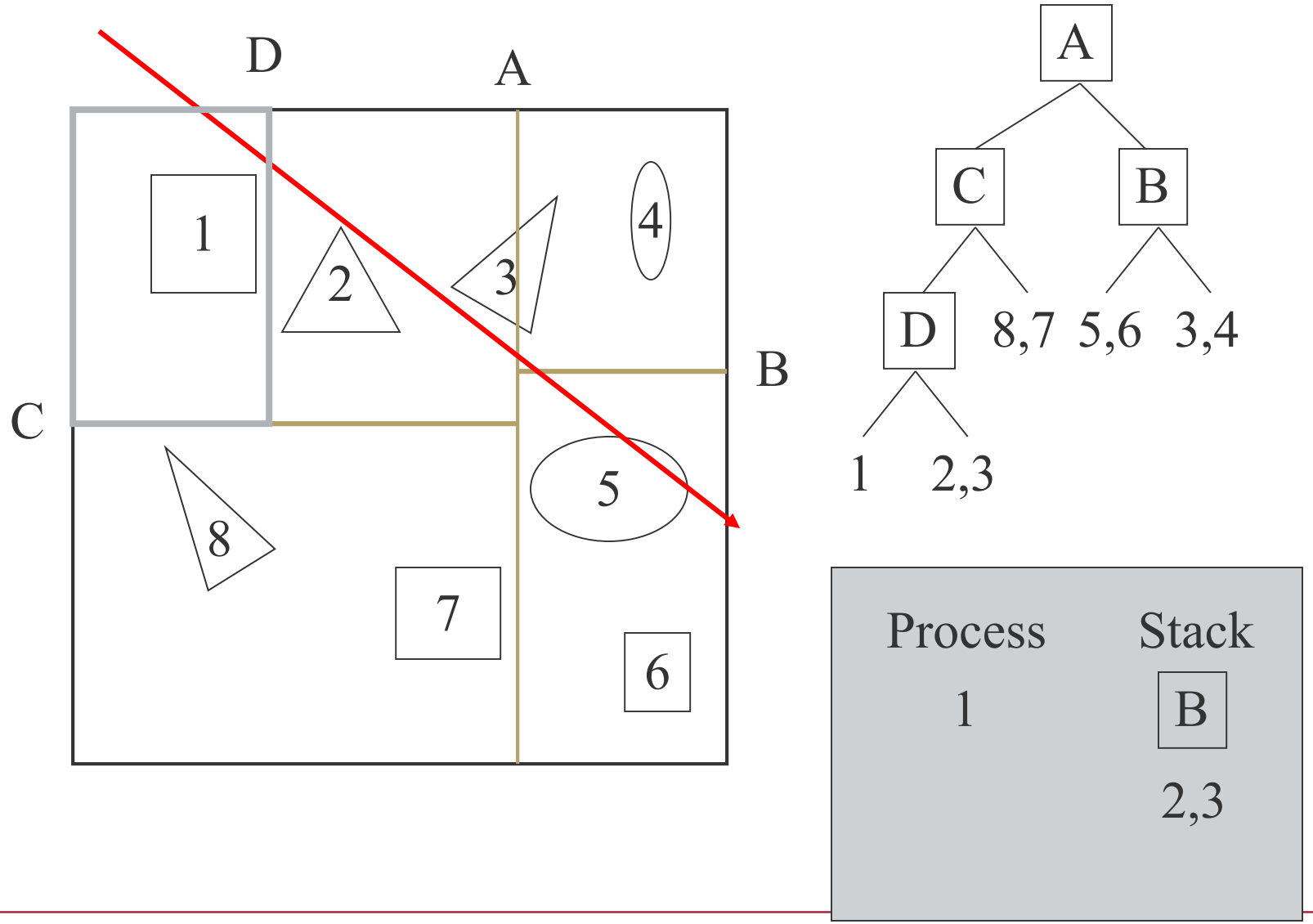




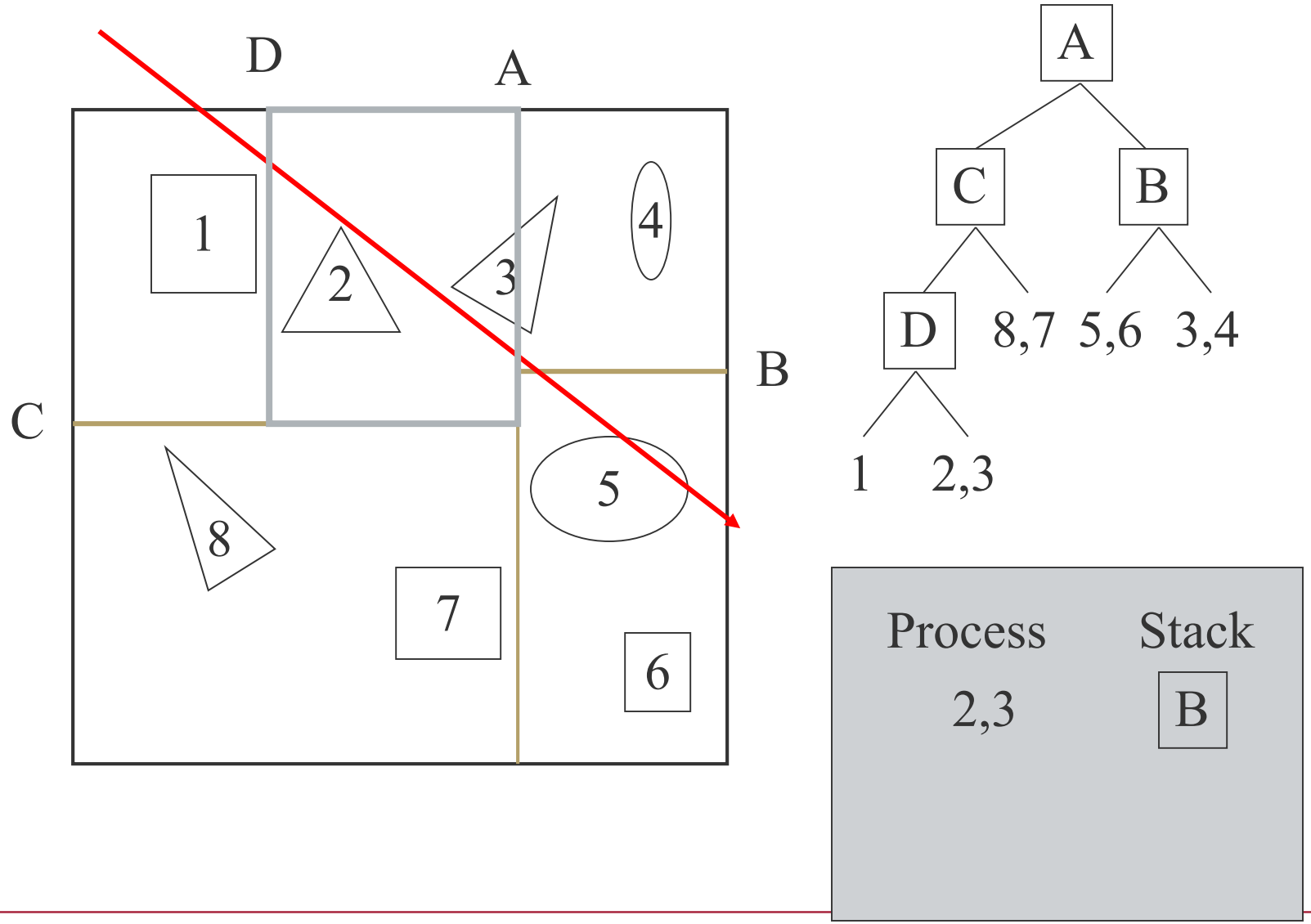


Both children of D may intersect

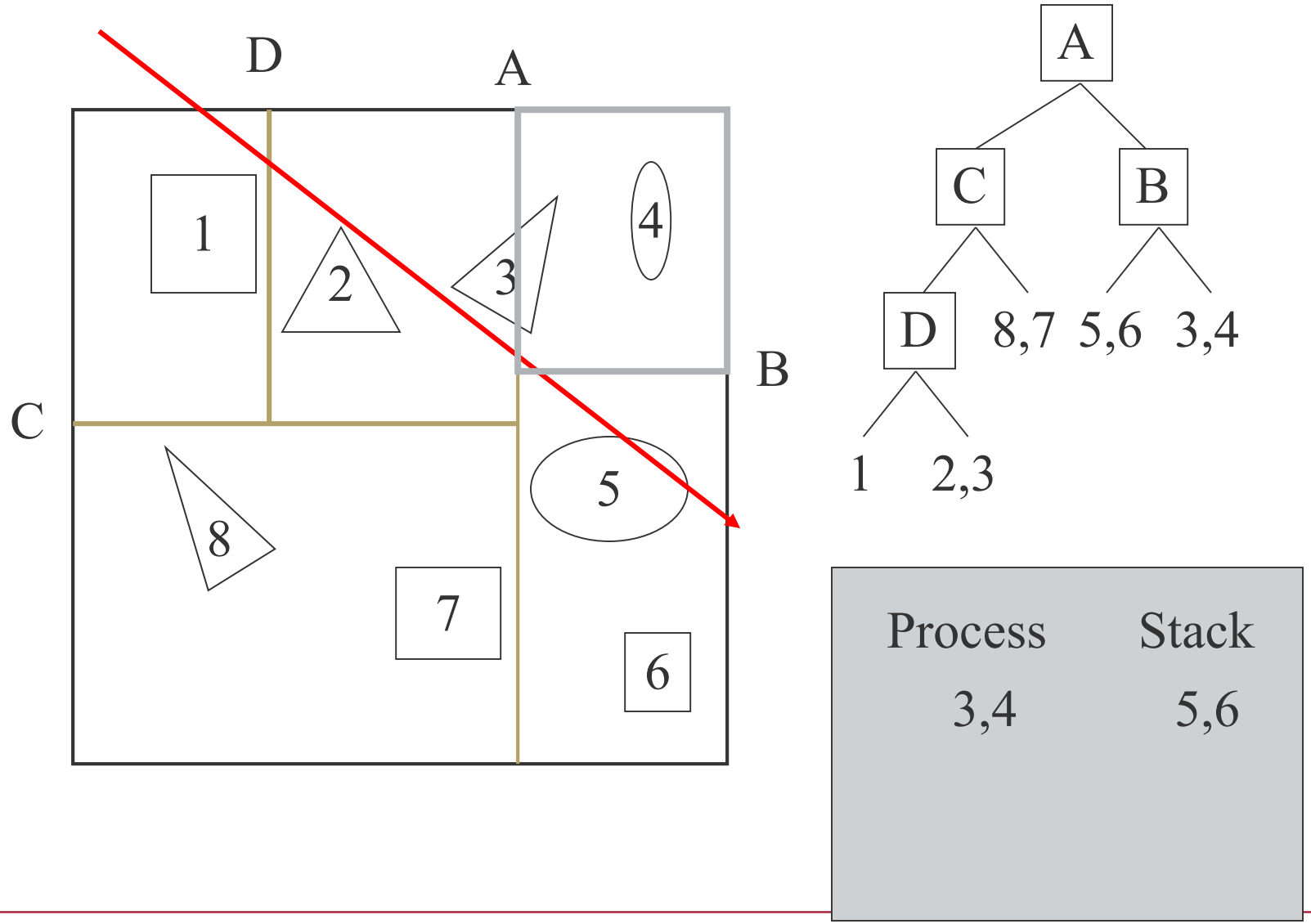
BSP-Tree Traversal



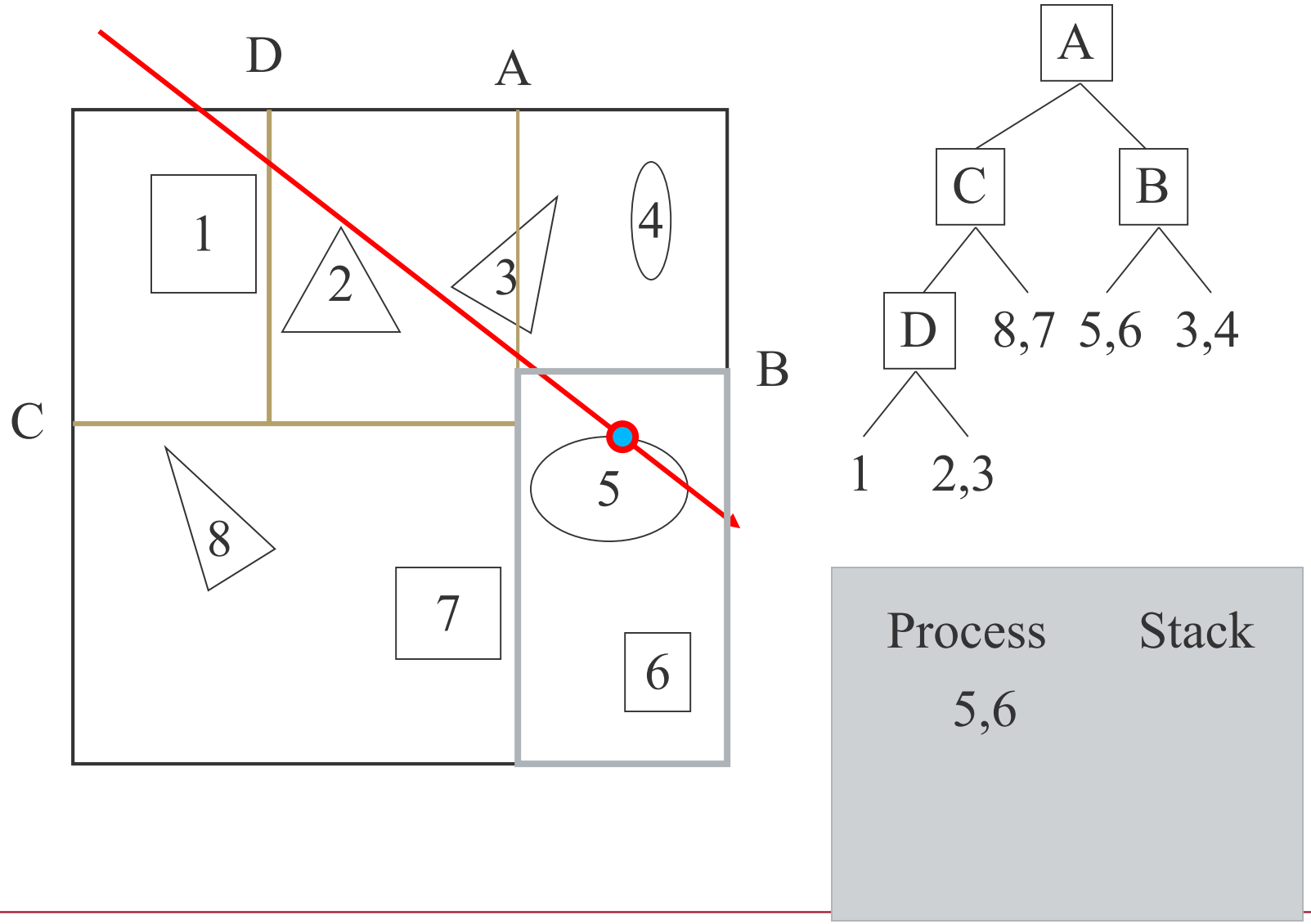
BSP-Tree Traversal



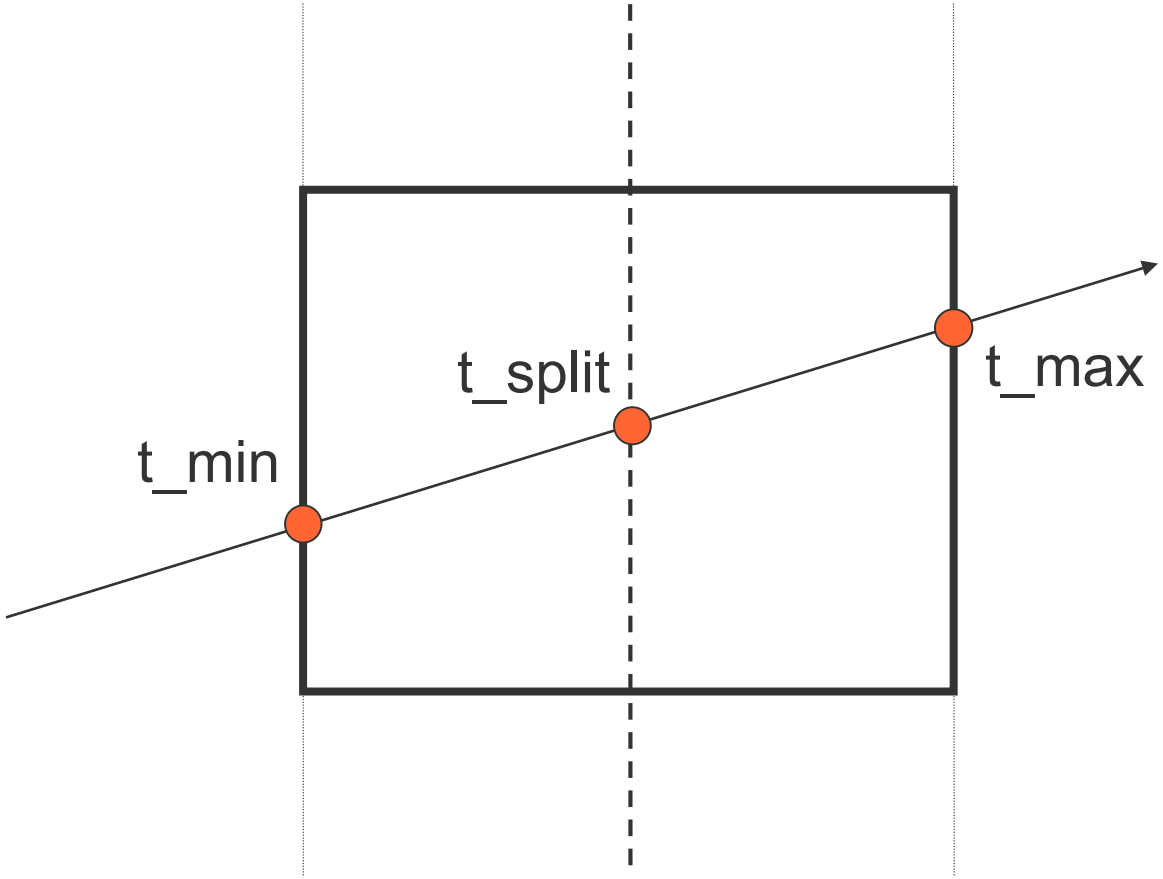
BSP-Tree Traversal



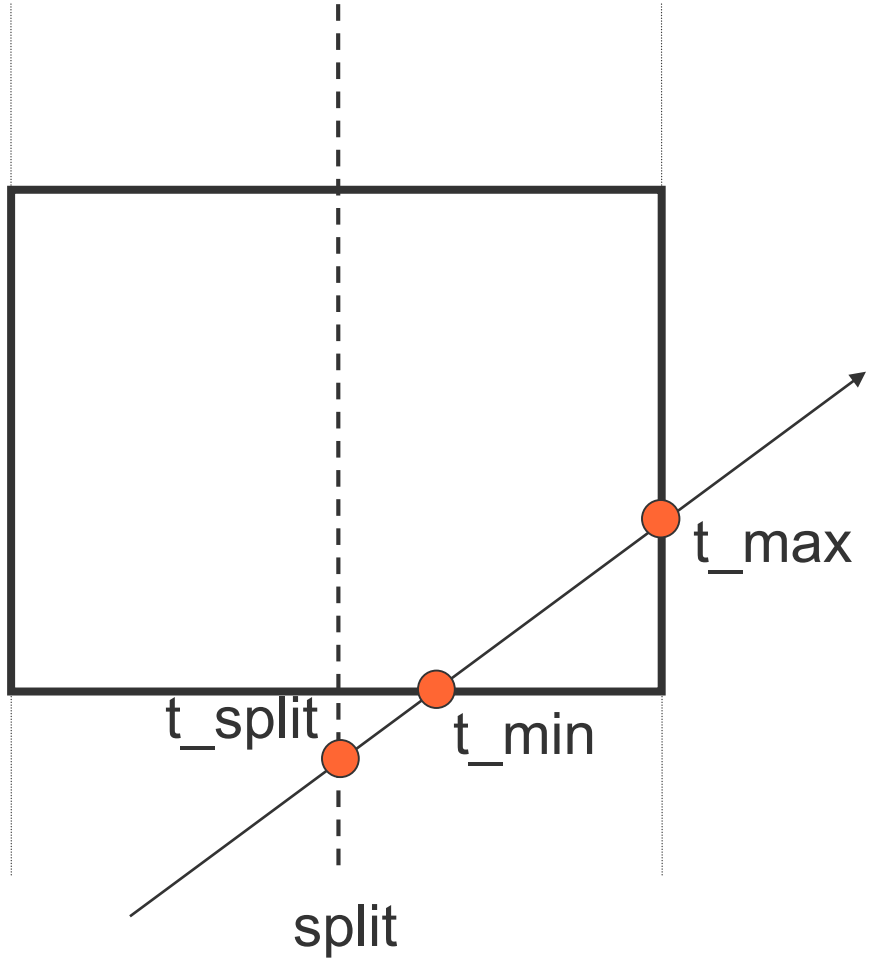
BSP-Tree Traversal



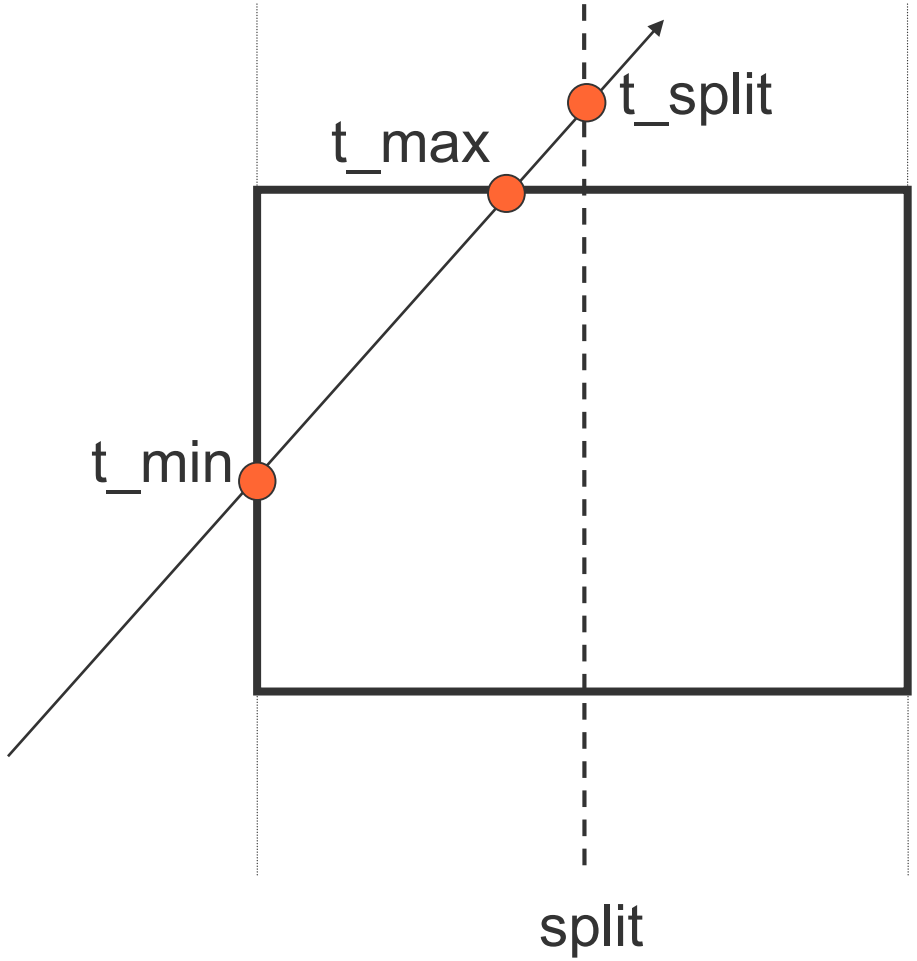
kD-Tree Traversal Step



kD-Tree Traversal Step



kD-Tree Traversal Step



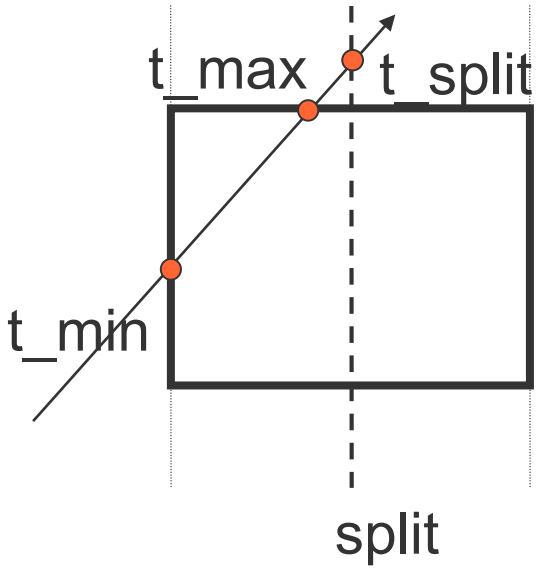
kD-Tree Traversal Step


```
ray: origin P,  direction V (normalized)
t_min, t_max (e.g. from bounding box of the entire scene)

Given:  ray P & V, t_min, t_max, split_location, split_axis

t_split = ( split_location - ray->P[split_axis] ) /
          ray->V[split_axis]

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



KD-Tree Traversal

```
1  Kd-tree Recursive Traversal:
2  begin
3      (entry distance, exit distance) ← intersect ray with root's
       AABB;
4      if ray does not intersect AABB then
5          return no object intersected;
6      end
7      push ( tree root node, entry distance, exit distance) to
       stack ;
8      while stack is not empty do
9          (current node, entry distance, exit distance) ← pop
           stack;
10         while current node is not a leaf do
11             a ← current node's split axis;
12             t ← (current node's split position.a - ray origin.a)
               / ray dir.a;
13             (near, far) ← classify near/far with (split
               position.a > ray origin.a);
14             if t ≥ exit distance or t < 0 then
15                 current node ← near;
16             else if t ≤ entry distance then
17                 current node ← far;
18             else
19                 push ( far, t, exit distance) to stack;
20                 current node ← near;
21                 exit distance ← t;
22             end
23         end
24         if current node is not empty leaf then
25             intersect ray with each object;
26             if any intersection exists inside the leaf then
27                 return closest object to the ray origin;
28             end
29         end
30     end
31     return no object intersected;
32 end
```

[Hapala & Havran, 2011]

Can it go faster?

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

Building a KD-Tree

What are good, what are bad trees?

Advantages of kD-Trees

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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?

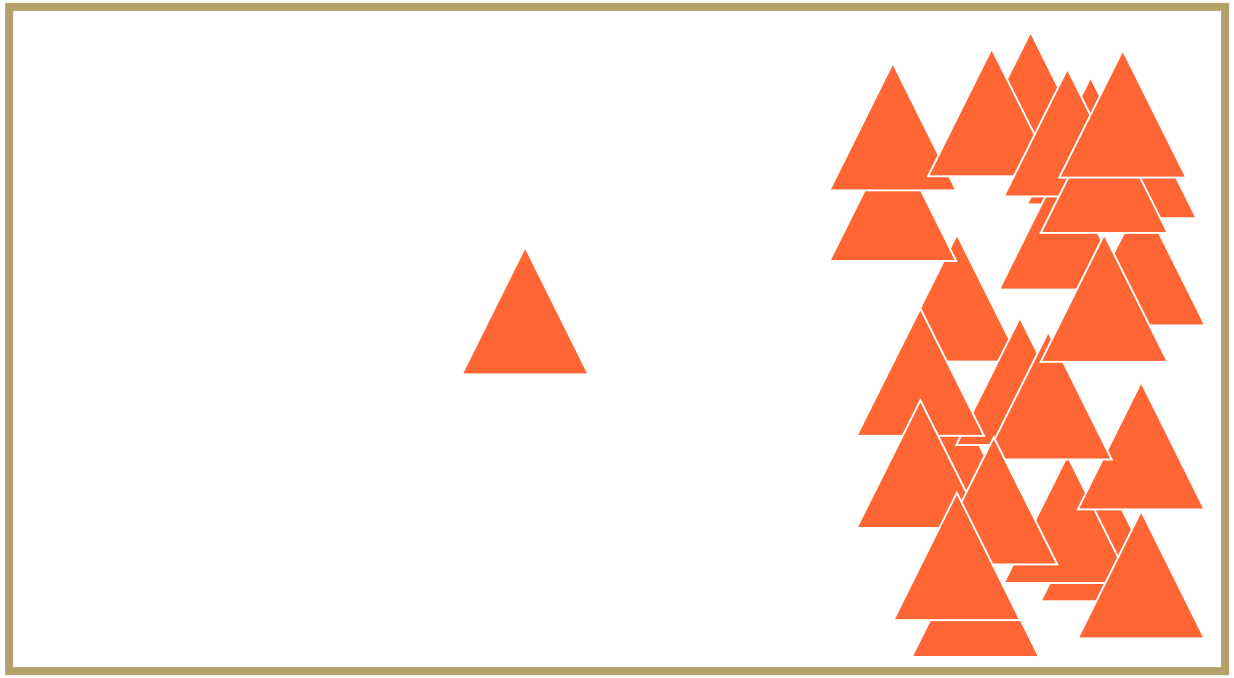
$$Cost(cell) = C\_trav + Prob(hit\ L) * Cost(L) + Prob(hit\ R) * Cost(R)$$

C_trav – cost for traversing another level

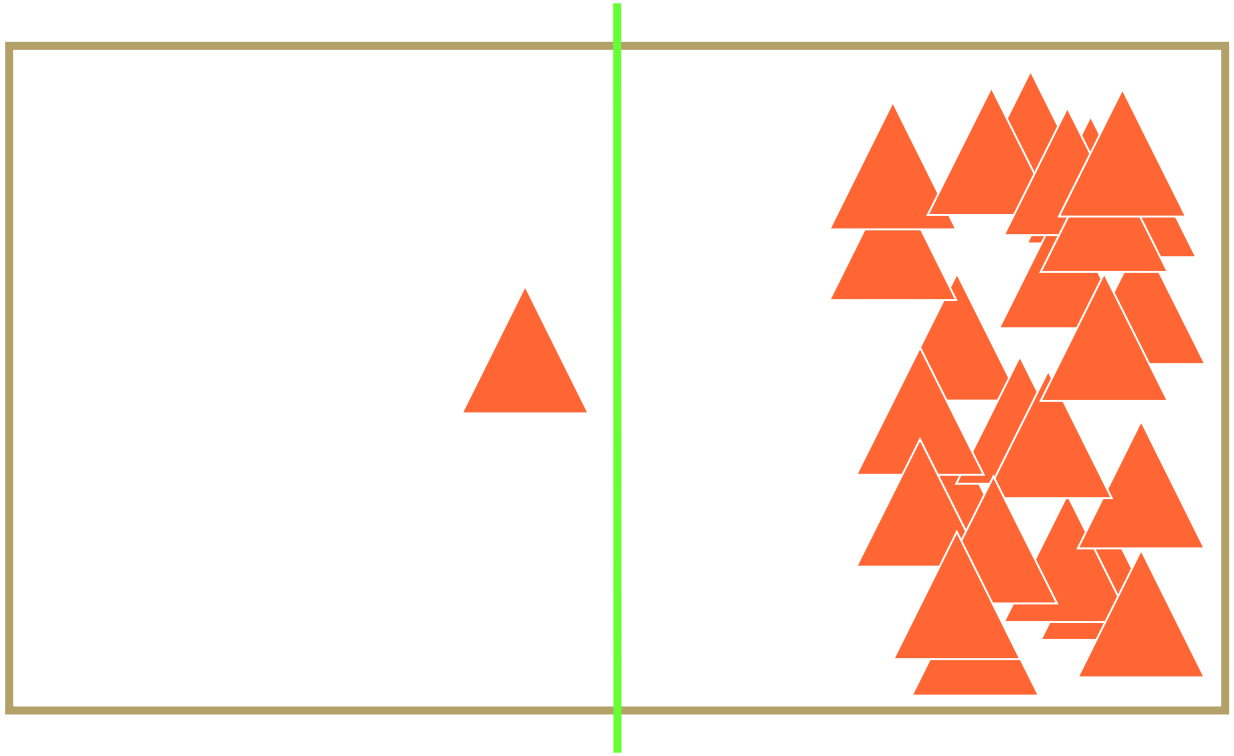
Prob – probability of hitting something (related to surface area)

Cost – cost for doing the intersection testing (related to #objects)

Splitting with Cost in Mind

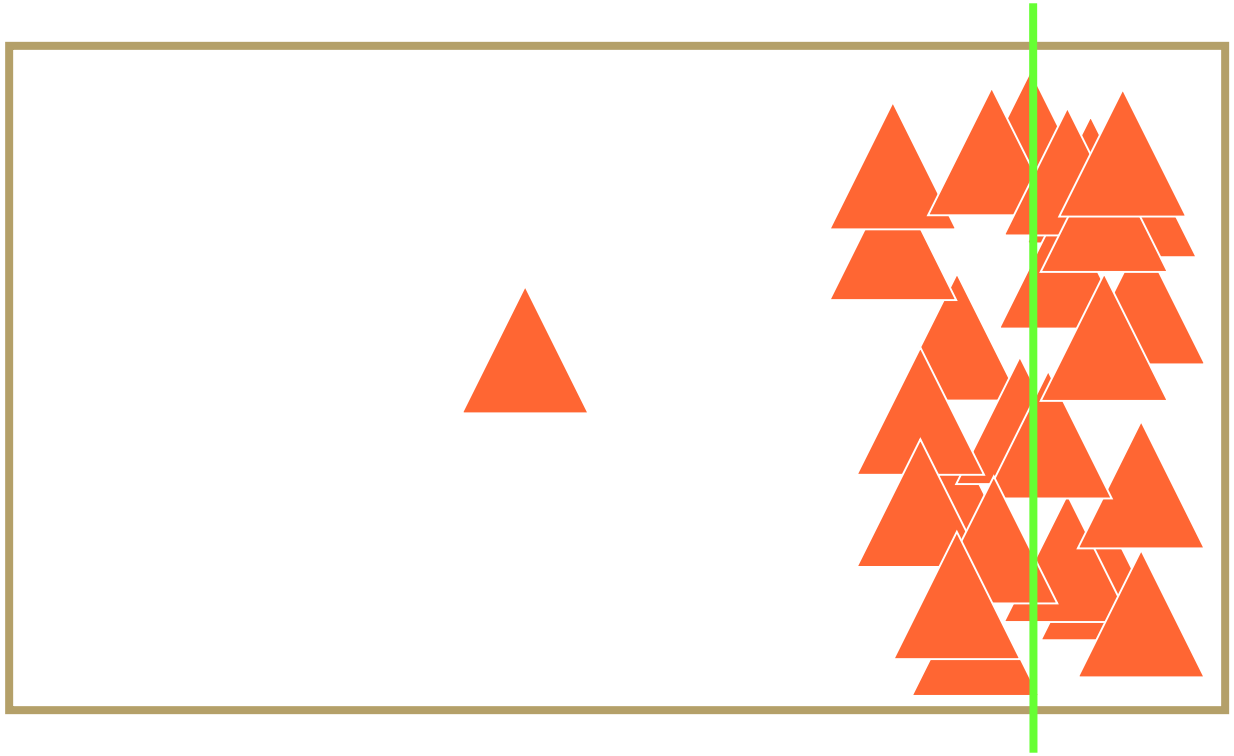


Split in the middle



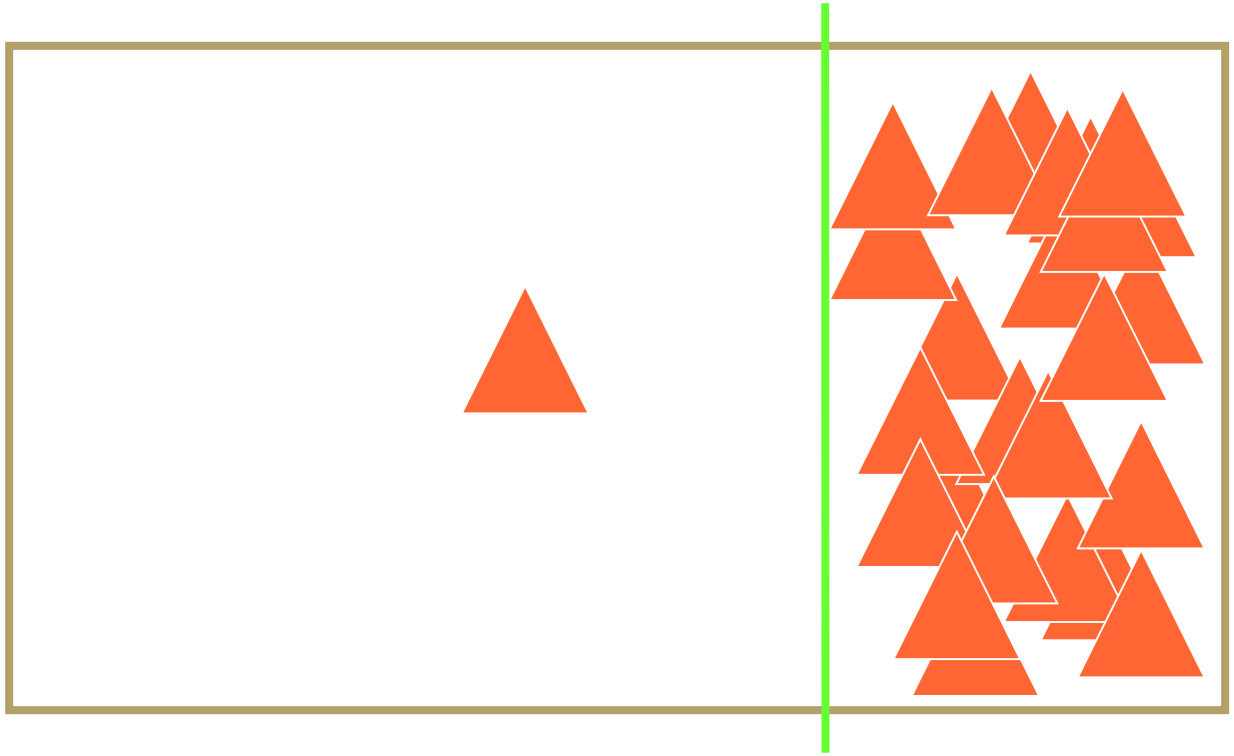
- Makes the L & R probabilities equal
- Pays no attention to the L & R costs

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

Surface Area Heuristic (SAH)

- Building good kD-trees

- Need the probabilities
  - Turns out to be proportional to surface area of bounding box

- Need the child cell costs
  - Simple triangle count works great (very rough approx.)

$$\begin{aligned} \text{Cost(cell)} &= C_{trav} + \text{Prob(hit L)} * \text{Cost(L)} + \text{Prob(hit R)} * \text{Cost(R)} \\ &= C_{trav} + SA(L) * \text{TriCount(L)} + SA(R) * \text{TriCount(R)} \end{aligned}$$

Termination Criteria

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

Fast Ray Tracing w/ kD-Trees

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- kD-Tree node can be packed into 8 bytes
  - Leaf flag + Split axis (3+1 states)
    - 2 bits
  - Split location
    - 32 bit float
  - Always two children, put them side-by-side
    - One 32-bit pointer

Compact (8-byte) nodes

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- kD-Tree node can be packed into 8 bytes
  - Leaf flag + Split axis (3+1 states)
    - 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!

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

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

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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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**

Questions

- Describe and compare quadtrees and grids.
- Describe and compare BVHs and kD-Trees.
- What is the surface area heuristic? What is it used for?
- How do you traverse a kD-Tree? or a BSP-tree?

Wrap-Up

- Hierarchical space partitioning
  - BSP, KD trees
  - Grids
  - Octrees
  - …
- Building
- Traversal

Next Lecture
- Raytracing Dynamic Structures

References

- T. Möller, B. Trumbore, Fast, minimum storage ray-triangle intersection, Journal of Graphics Tools, Volume 2 Issue 1, pges 21-28, 1997

- M. Hapala, V. Havran, Review: Kd-Tree Traversal Algorithms for Ray Tracing, Computer Graphics Forum, volume 30, issue1, 2011.