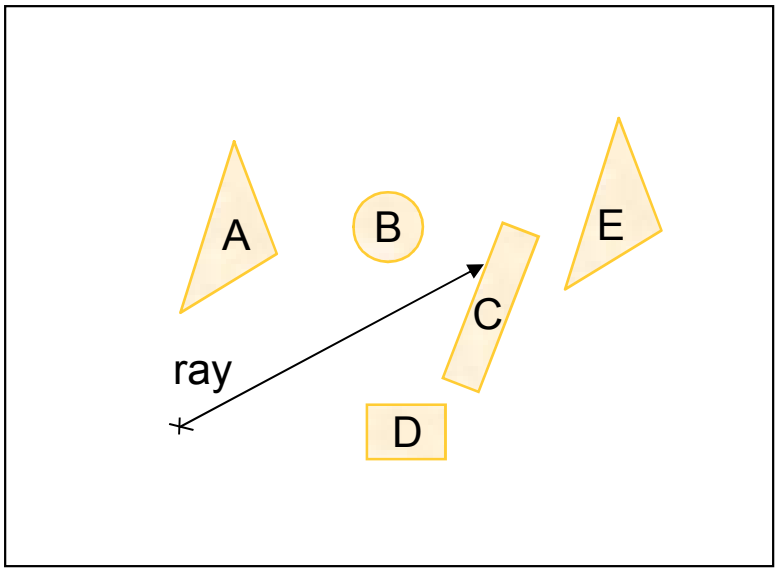
Data Structures for Computer Graphics

Ray Shooting and its Applications I+II

Lectured by Vlastimil Havran

Ray Shooting Algorithm (RSA)

Task: Given a ray, find out
the first object intersected.



Input: a *scene* and a *ray*

Output: the *object C*

Ray Shooting

Find nearest intersection along a ray

Problem	Q	S	A
Ray shooting	**ray**	**{objects}**	**point**
Hidden Surface Removal	{rays}	{objects}	{points}
Visibility culling	{rays}	{objects}	{objects}
Photon maps	point	{points}	{points}
Ray maps	point	{rays}	{rays}
Irradiance caching	point	{spheres}	{spheres}

Ray Shooting

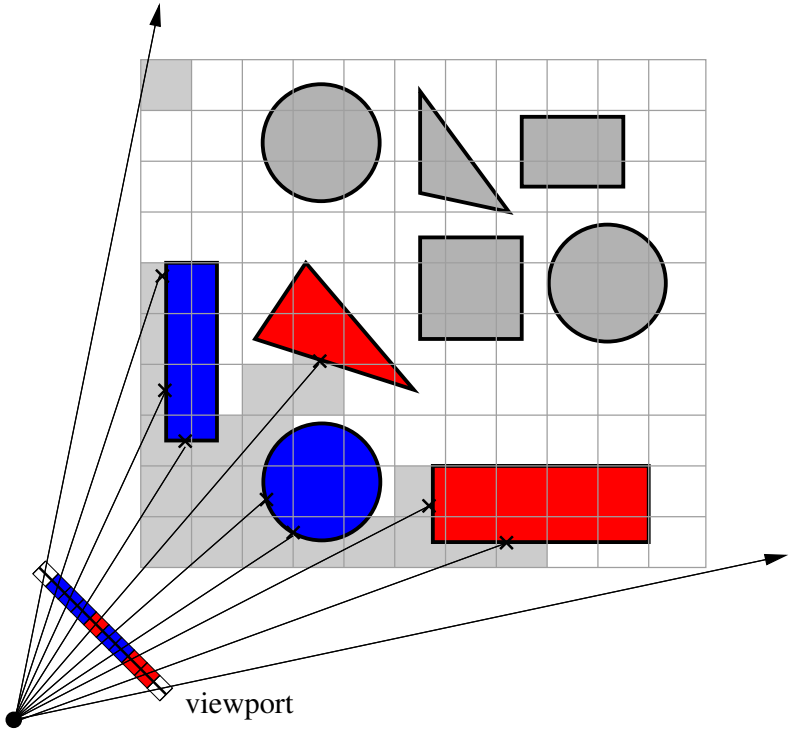
- Ray shooting versus ray tracing
- Applications of ray shooting
- Performance model/studies
- Ray shooting with kd-trees
- Octrees, uniform grids, recursive grids
- Bounding volume hierarchies
- Offline ray shooting
- Special algorithms

Ray Tracing *versus* Ray Shooting

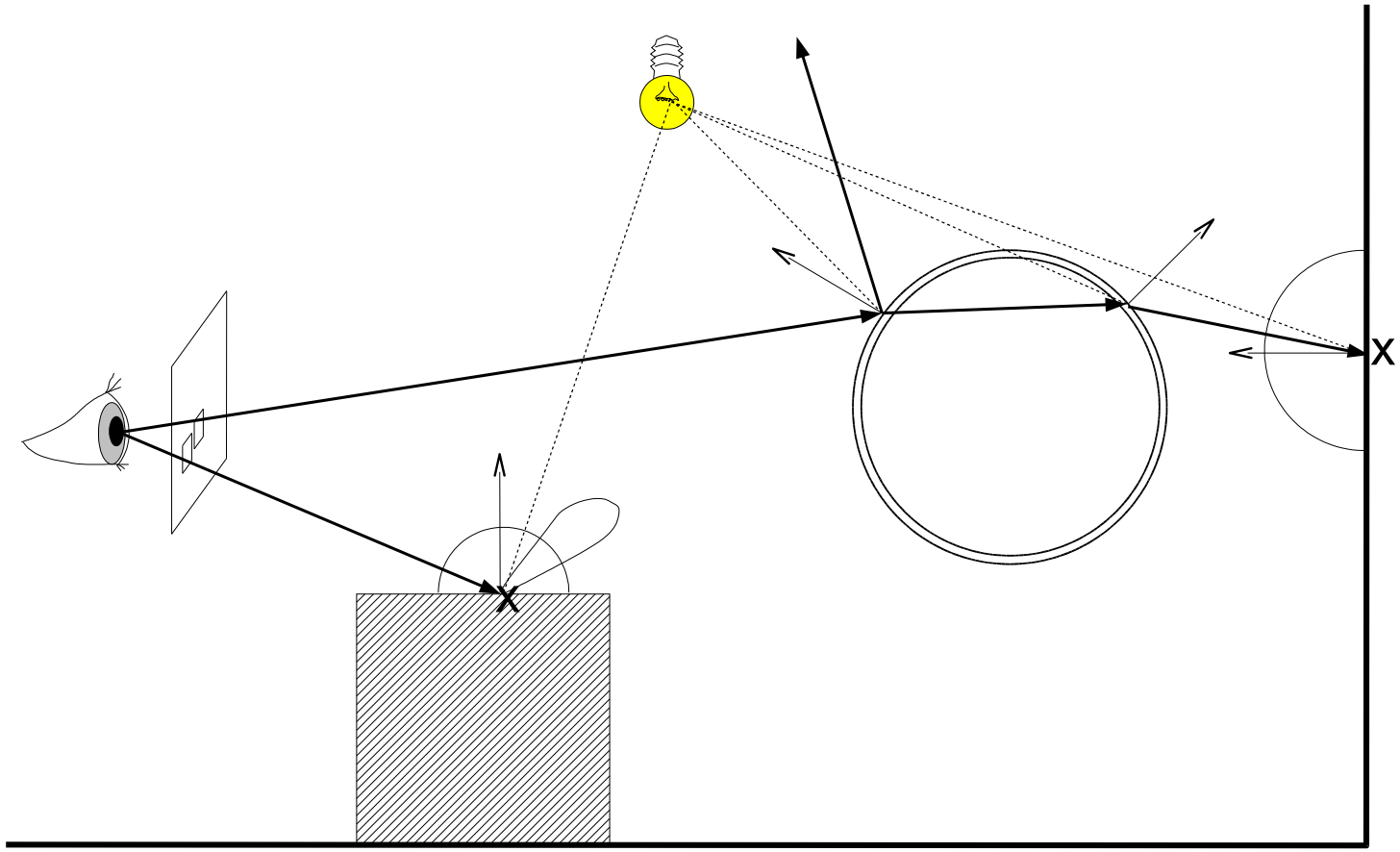
- *Ray shooting* – only a single ray
- *Ray tracing* in computer graphics can be:
  - Ray shooting (only a single ray)
  - Ray casting – only primary rays from camera
  - Recursive ray tracing
  - Distribution ray tracing and others

Ray Casting and Ray Tracing

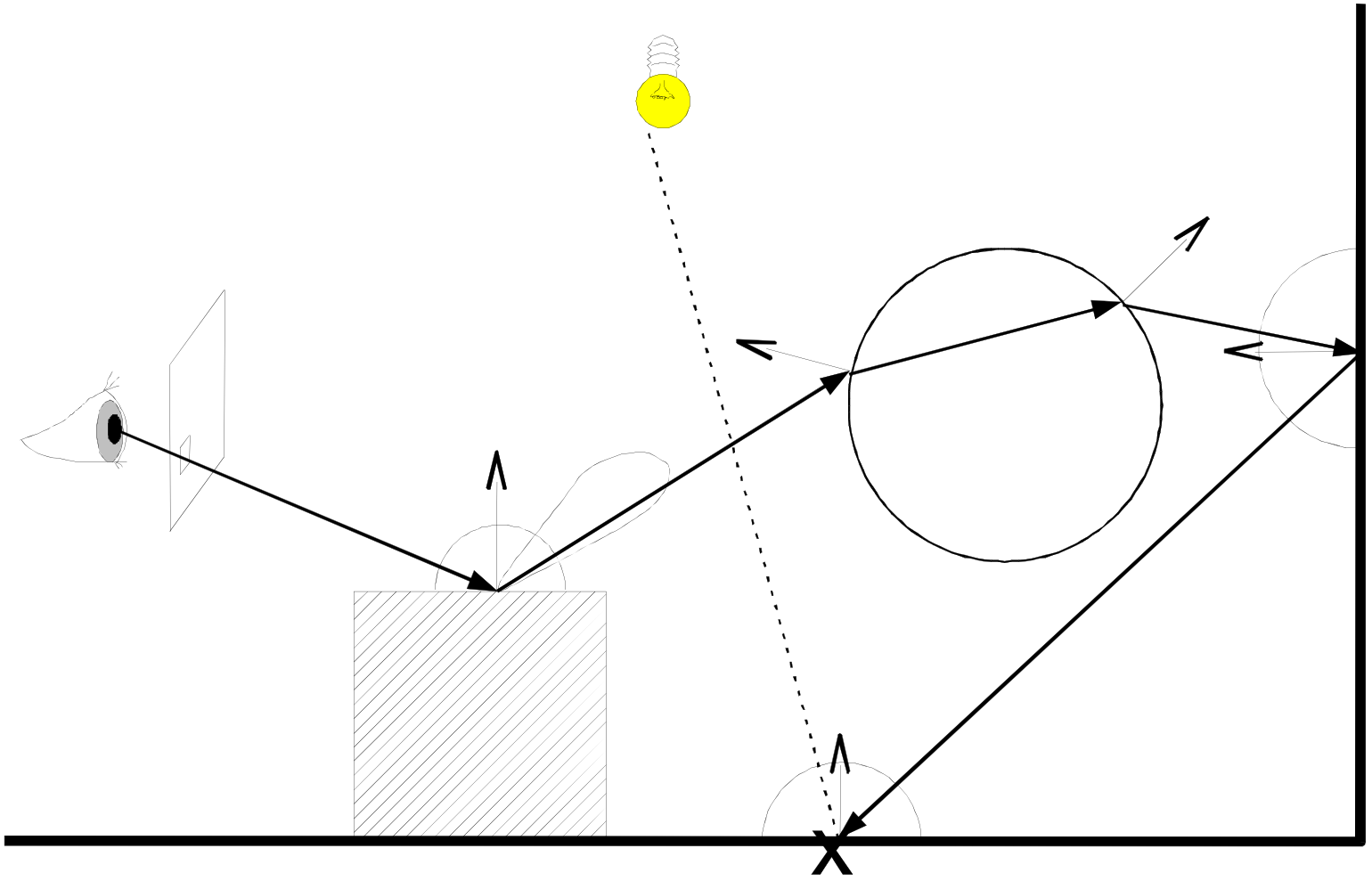
- Cast ray for each pixel
- Step 1: spatial data structure (XYZ)
  - Preprocess
  - Trees ~ quick sort
  - Grid ~ distribution sort
- Step 2: search for nearest intersection
  - Min selection with early termination



(Recursive Backward) Ray Tracing



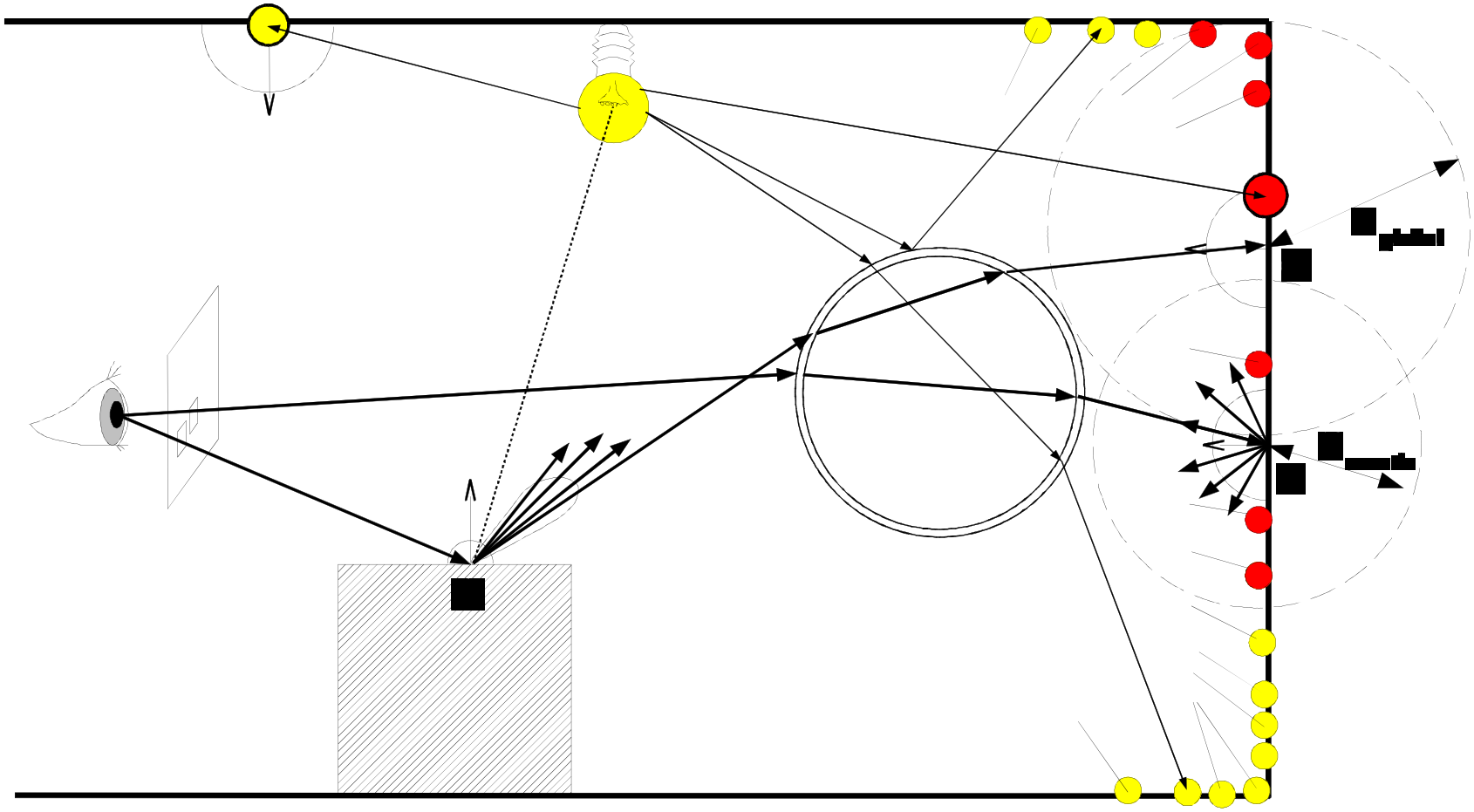
Path Tracing



Photon Mapping

Phase I: photon shooting

Phase II: gathering

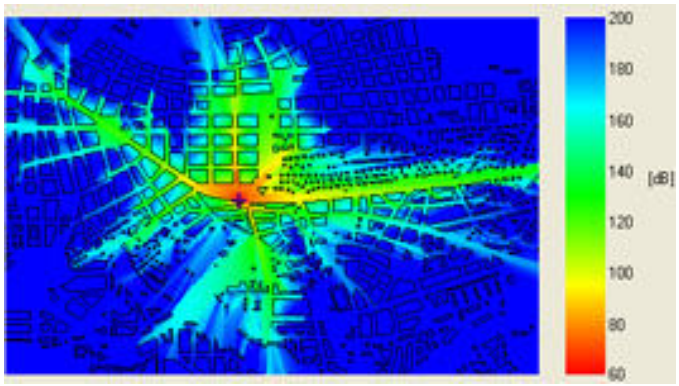


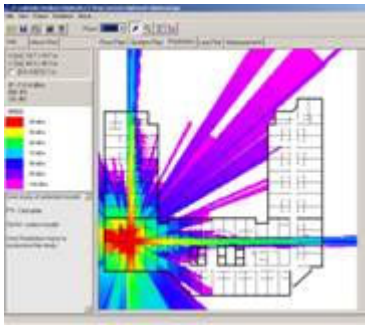
Other Ray Shooting Applications

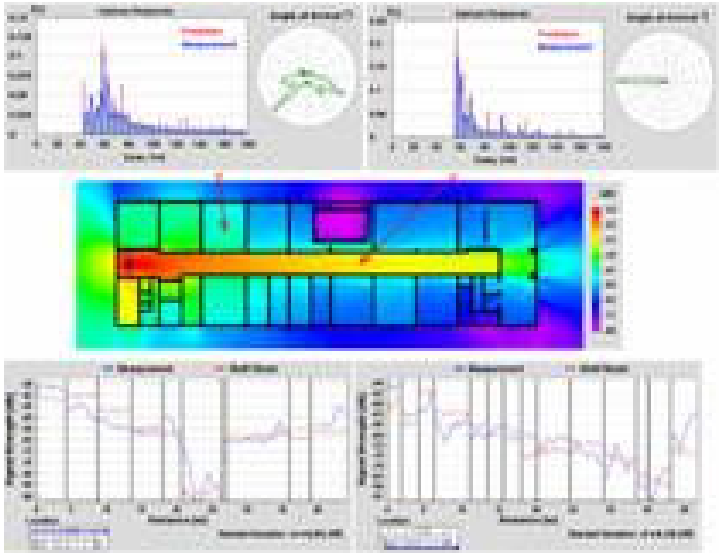
- Simple collision detection: a player is approximated by a sphere, collision is approximated by tens of rays
- Visibility preprocessing
- Radio waves propagation
  - Prediction of radio wave propagation
  - Optimization of radio wave propagation: where to put antennas/ transmitters in a city to maximize its utilization (mobile phones, television, radio waves etc.)
- Optical design: lenses, telescopes, camera objectives (different kinds of distortion, attenuation, reflection)
- Vehicle design for army: minimize the impact of bullet penetration (ballistic analysis) (US army research labs)
- Artificial body garment fitting for games

Radio Signal Propagation

- **Interactive Realistic Simulation of Wireless Networks, RT07**
- **CTU-FEL - K13117 - Department of Electromagnetic Field, http://www.feld.cvut.cz/vv/tymy/radiovlny.html**

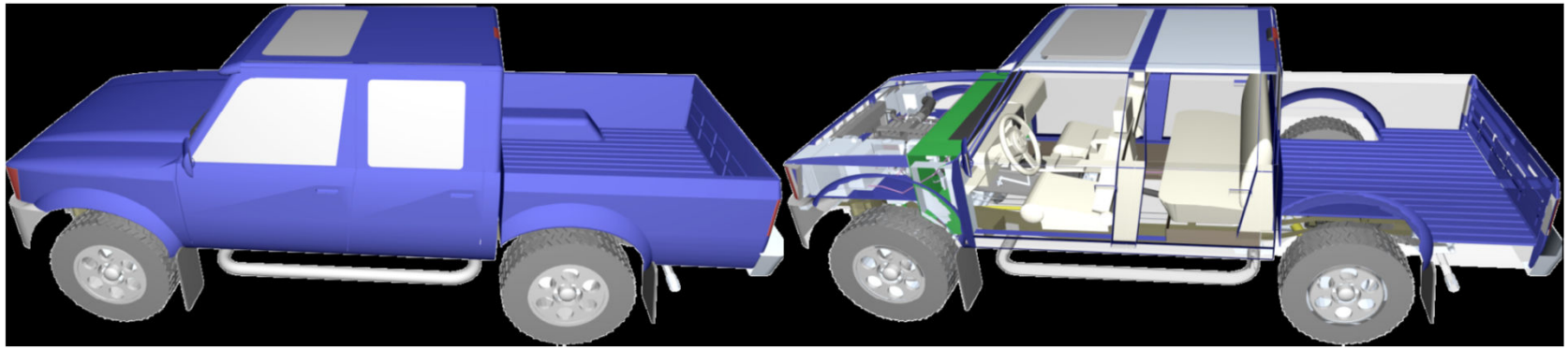


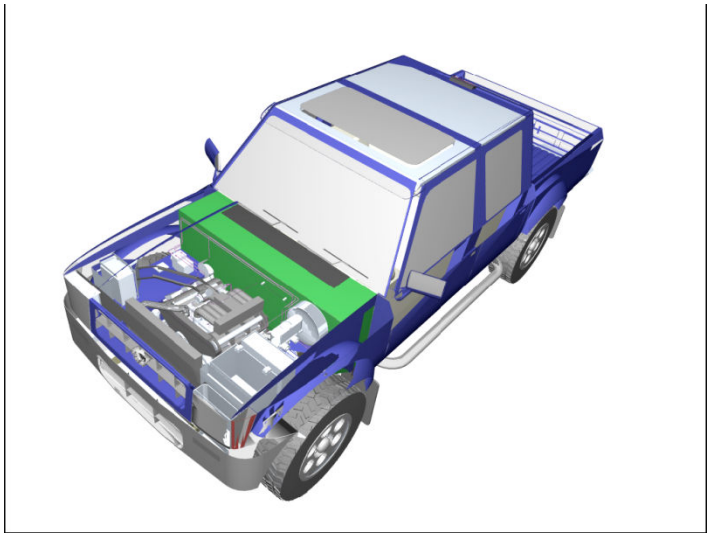




Bullet Ray Vision
– virtually shooting bullets

Images from Bullet Ray Vision by Buttler and Stephens, RT2007





Some Complexity Issues

Computational Geometry

- aims mainly at worst-case complexity
- restriction to certain class of object shape
  (polygons, triangles)
- unacceptable memory requirements, further in slides
  O(log N) query time induces $\Omega(N^4)$ space

Computer Graphics

- aims at average-case complexity
- practical feasibility and robustness
- implementation issues important for performance

Some Complexity Results

Lower bound for worst-case complexity: 1997/98

Laszlo Szirmay-Kalos + Gabor Marton – lower bound for space complexity is $\Omega(N^4)$ for O(log N) search

Applicability of Computational Geometry techniques in CG for ray tracing

- CGE techniques are not general
- limited to small number of primitives
- no real implementations available

Complexity: Why is it so difficult ?

- We do have non-point data !
- For each of **four lines** we find **two lines** which intersect all of them
- Triangles bounded by lines
- How many rays are then formed by N lines: the number of combinations is $K = N! / (4! * (N-4)!)) = \Omega(N^4)$

  - the number of different ray-object sequences
- A data structure based on trees has to distinguish at least $\Omega(N^4)$ possible cases
- This needs $\Omega(N^4)$ space + preprocessing and the search in a tree is computed in O(log N)

Computer Graphics Techniques Overview

Techniques developed: aimed at practical applications, no complexity guarantees, use many "tricks", the analysis difficult or infeasible

B*asic techniques*: bounding volumes, spatial subdivision, ray classification

A*ugmented techniques*: macro regions, pyramid clipping, proximity clouds, directed safe zones

S*pecial tricks*: ray boxing, mailbox, handling CSG primitives, other types of coherence, etc.

RSA Techniques Classification

A) **Subdivision techniques** (top down)

-- binary space partitioning (also kd-trees)

-- octrees

-- uniform and hierarchical grids

-- bounding volume hierarchy

B) **Clustering** (bottom up)

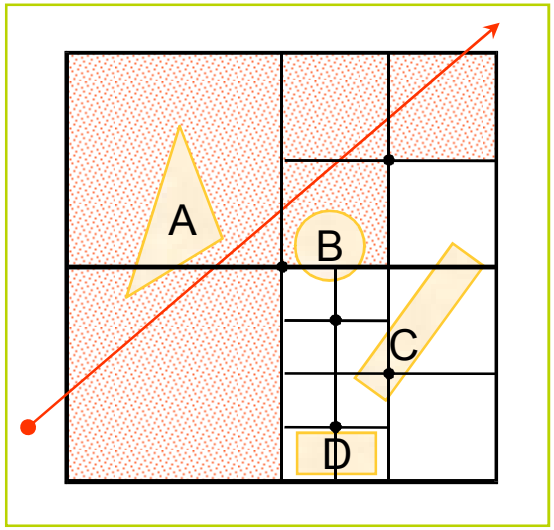
-- bounding volume hierarchy

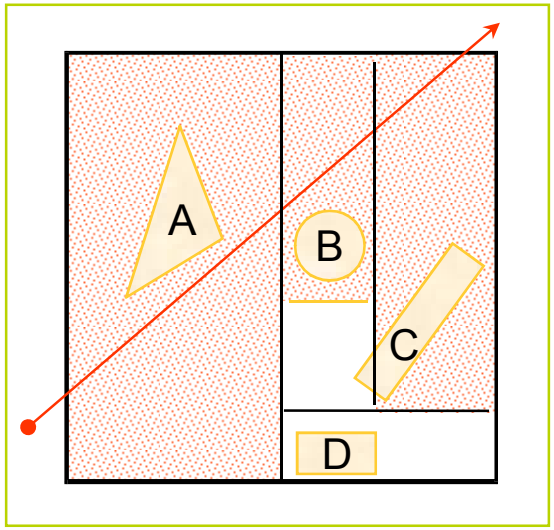
C) **Structures formed by insertion** (incrementally)

-- bounding volume hierarchy

Some RSA Techniques

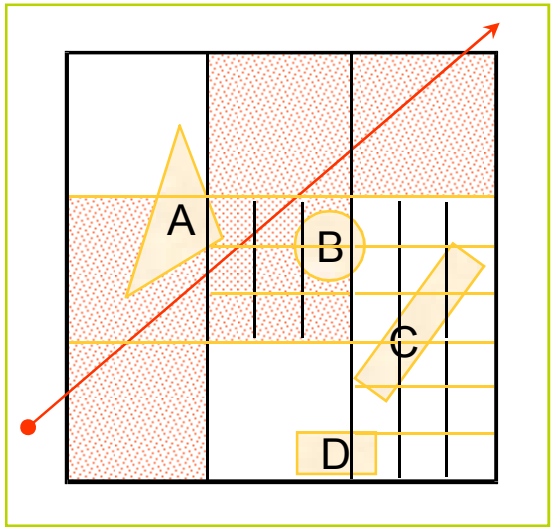
octree

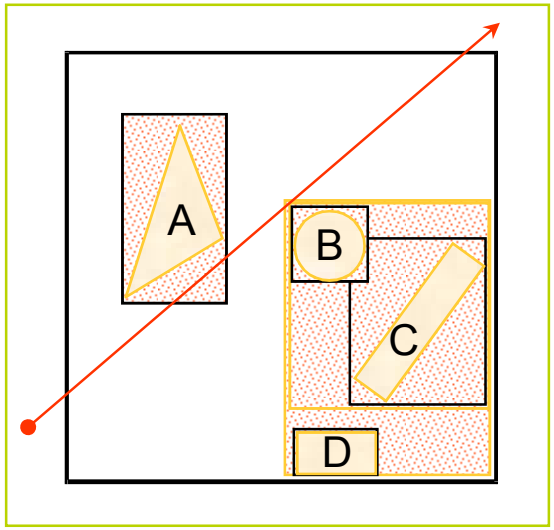




kd-tree

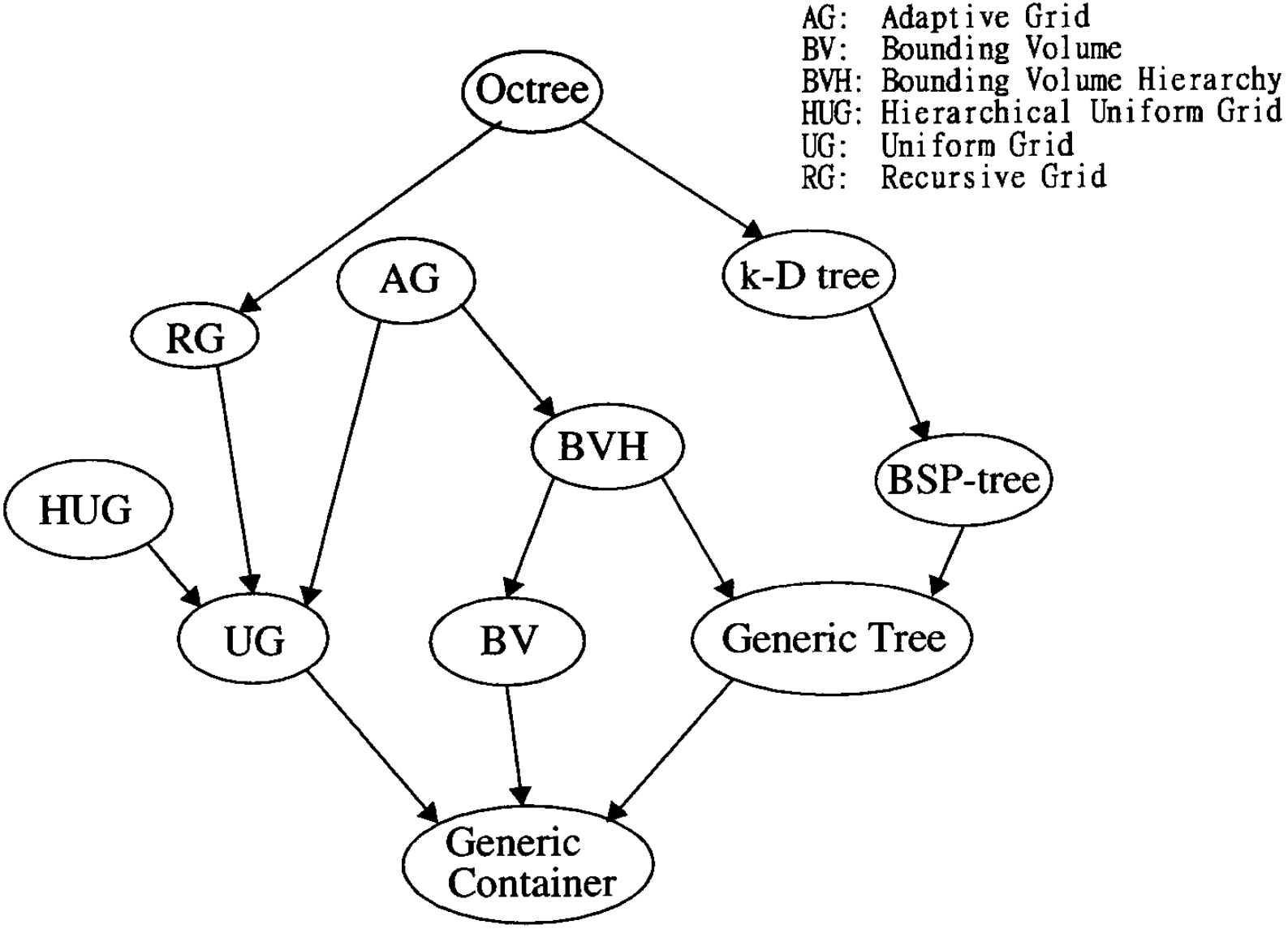
hierarchy of grids





bounding volume hierarchy

Relations between Data Structures



Recall: Search Performance Model

Typical cost model:

$$C = C_T + C_L + C_R$$

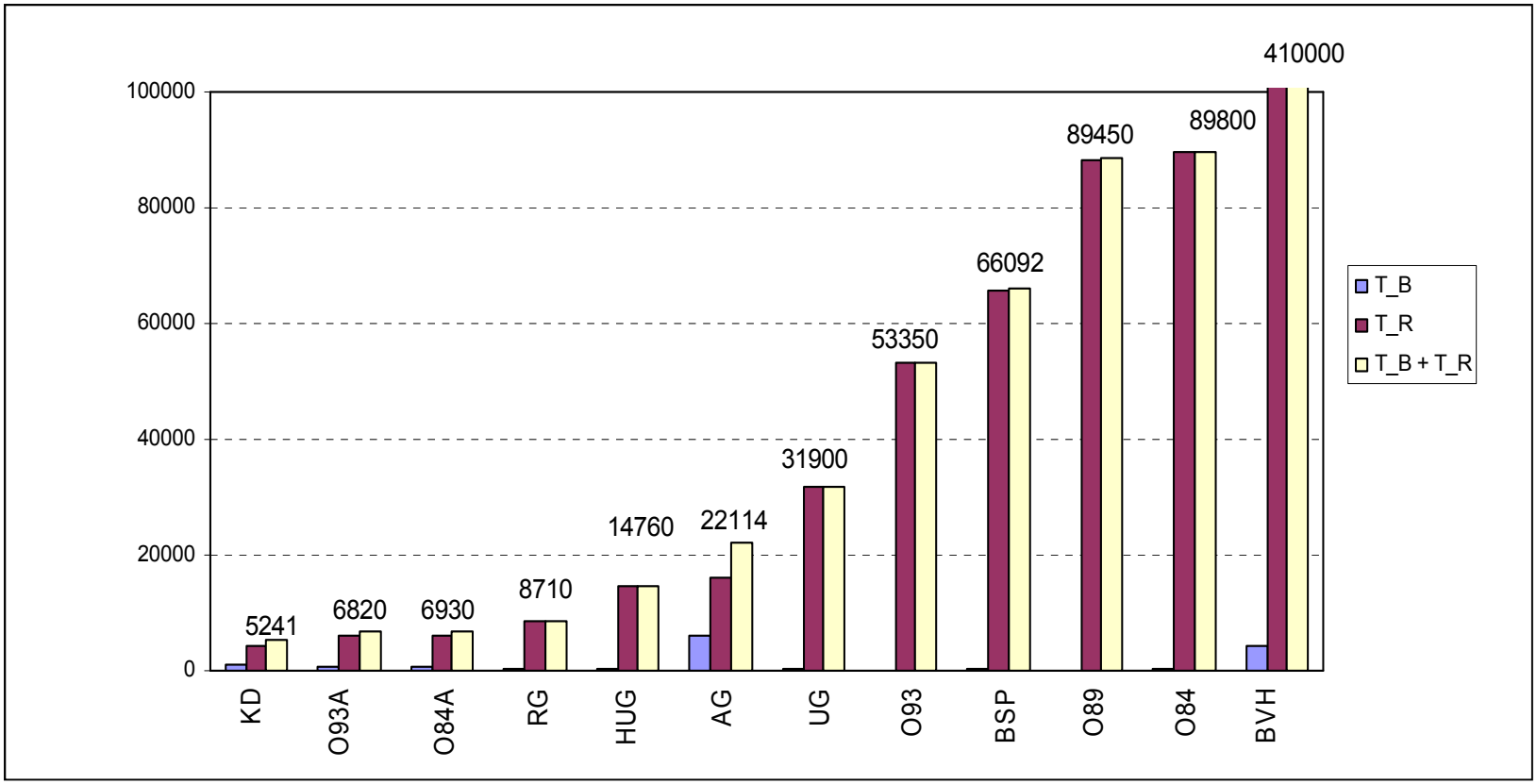
$$C = C_{TS} * N_{TS} + C_{LO} * N_{LO} + C_{Access} * N_{Access}$$

- C_T … cost of traversing the nodes of HDS
- C_L … cost of incidence operation in leaves
- C_R … cost of accessing the data from internal or external memory

RSA Techniques Comparison

30 scenes *times* 12 RSAs *times* 4 ray distribution methods
= 1440 measurements, year 2000-2001

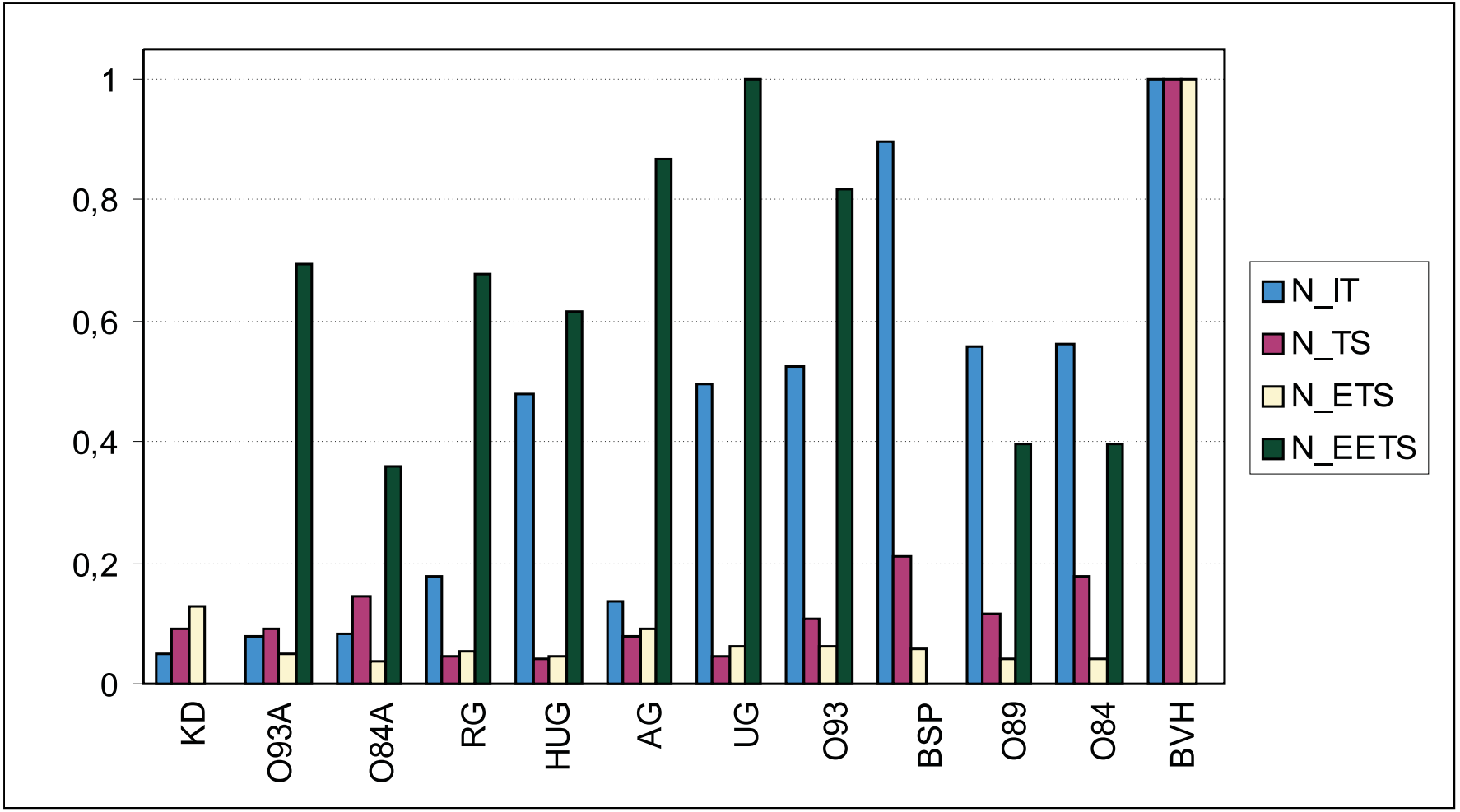
Timings (build time, search time, total time)



Note: In tests BVH constructed by insertion !

RSA Techniques Comparison

Number of operations (ray-object intersections, traversal steps)



Note: values normalized to the worst value.

Notation used in Tables

- T_B … building time for data structures (preprocessing)
- T_R … ray tracing time
- N_{IT} … number of ray-object intersection tests per ray
- N_{TS} … number of traversal steps per ray
- N_{ETS} … number of traversal steps per ray through elementary cells (leaf cells, either empty or full = containing pointers to objects)
- N_{EETS} .. number of traversal steps per ray through empty elementary cells (leaf cells)

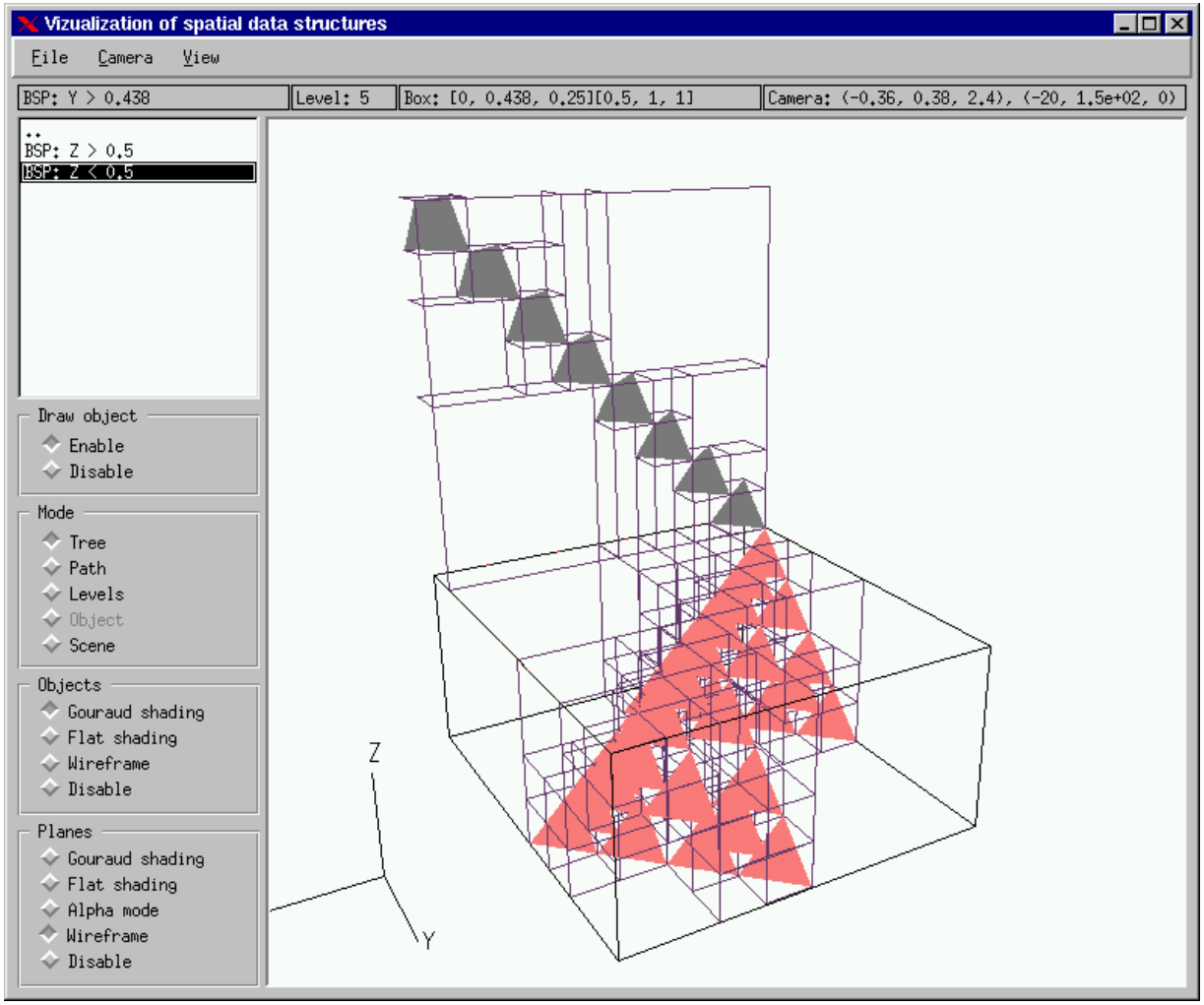
Intro: Ray Shooting with Kd-trees

- Kd-tree is one of the most efficient data structures for ray shooting and also for other problems
- Note that kd-tree is for ray shooting not constructed over point data, but over object data with spatial extent. Some objects can be referenced **several times** in leaves.

Intro: Ray Shooting with Kd-trees cntd.

- Ray shooting is very different searching problem compared to NN-search or k-NN search, or circular range search over point data!
- This requires changes both in the construction algorithm of kd-tree and the traversal algorithm for kd-tree.
- Performance at 2011/12 (CPU 1 thread)
  - ray shooting up to 4M/s for individual coherent rays
  - kd-tree construction time roughly in 80 ms for 100,000 triangles.

Visualisation of Kd-tree



RSA based on Kd-trees

Quite an efficient solution used in practice

Construction (best algorithm in O(N log N) time)

- based on cost function and geometric probability
- automatic termination criteria algorithm
- various efficiency improvements:
  - construction of kd-tree with empty spatial regions
  - reducing objects' axis-aligned bounding boxes
  - preferred ray sets

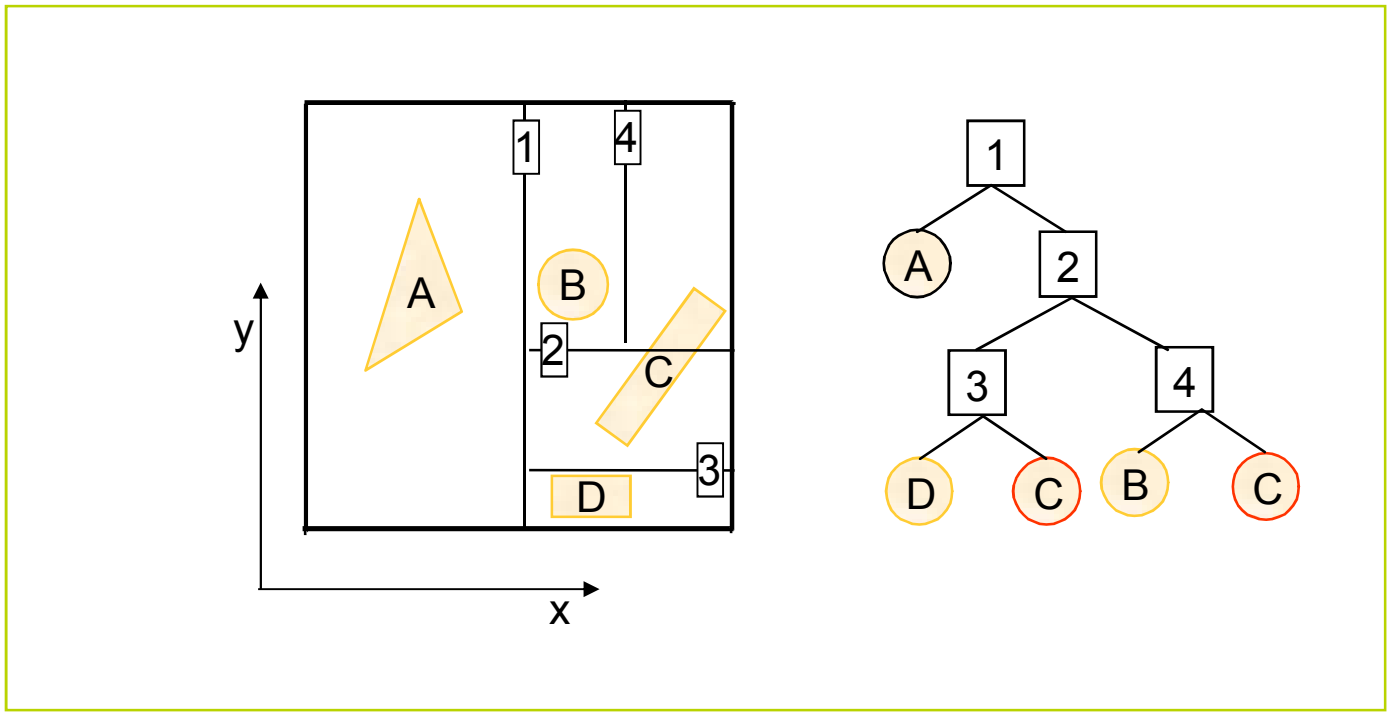
Ray traversal

- in practice achieves expected O(log N) time
- robust recursive ray traversal algorithm

Kd-tree Construction

- Spatial median splitting
  - the node is subdivided in a geometric center, changing orientation of axis in x,y,z,x,y…
  - the resulting kd-tree is rather inefficient
- **Cost model** in a priori setting for splitting
  - the application of the cost model based on local greedy heuristics
  - can improve the performance of ray shooting by two orders of magnitude
  - it is absolutely necessary for scenes with skewed distribution for performance reasons.

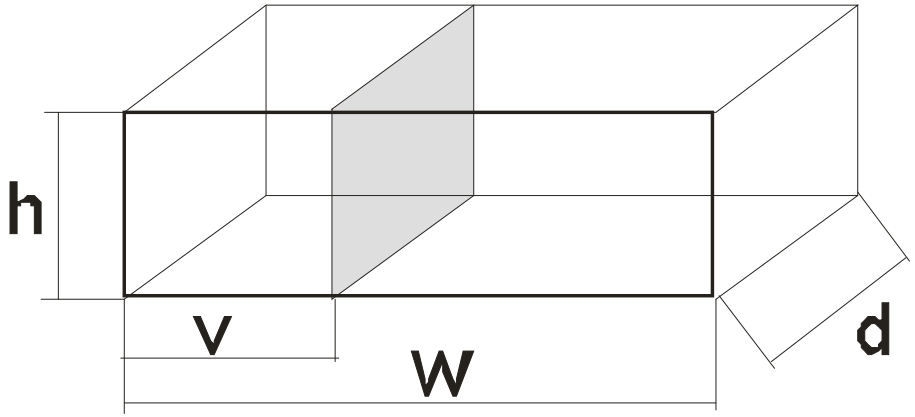
Kd-tree Construction for Ray Shooting

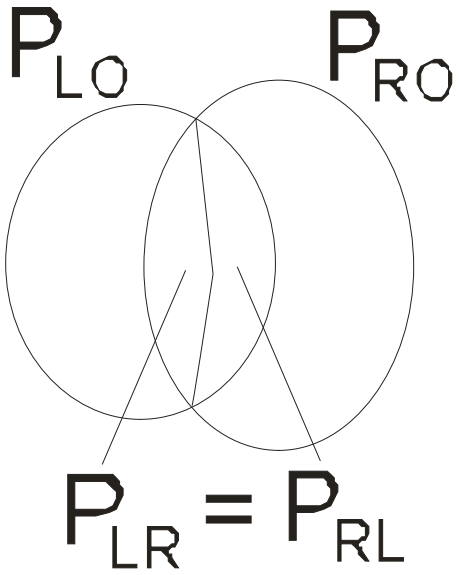


Note that object C is referenced in two leaves !

Geometric Probability of Ray Intersecting a Subdivided Box

$$probability_{LEFT} = S_L / S = P_{LO} + P_{LR} + P_{RL}$$
$$probability_{RIGHT} = S_R / S = P_{RO} + P_{LR} + P_{RL}$$





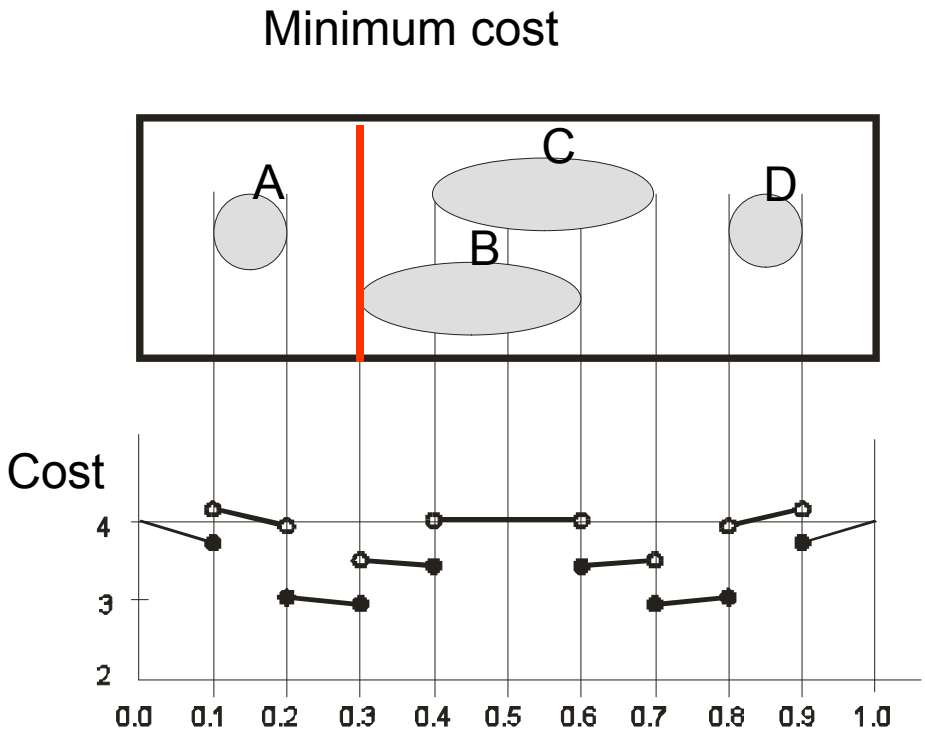
Probability computed from surface area of the box

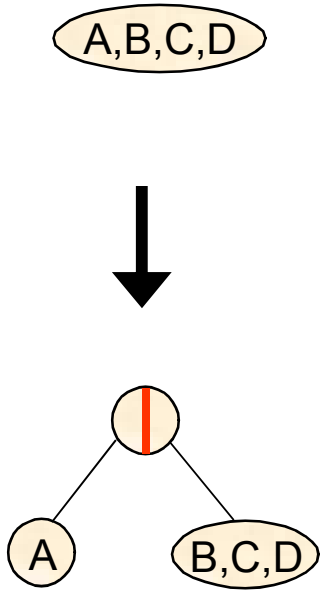
Condition: uniform ray distribution

Kd-tree Construction Based on Cost Function with Greedy Heuristics

$$\text{Cost} = \text{probability}_{LEFT} * \text{Cost}_{LEFT} + \text{probability}_{RIGHT} * \text{Cost}_{RIGHT}$$

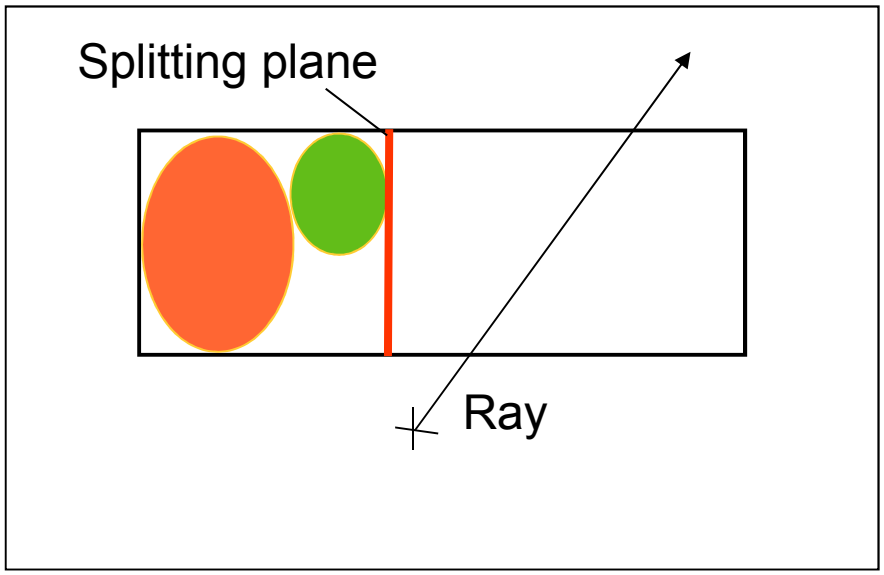
Local greedy: $\text{Cost}_{LEFT} = N_{LEFT}$ and $\text{Cost}_{RIGHT} = N_{RIGHT}$



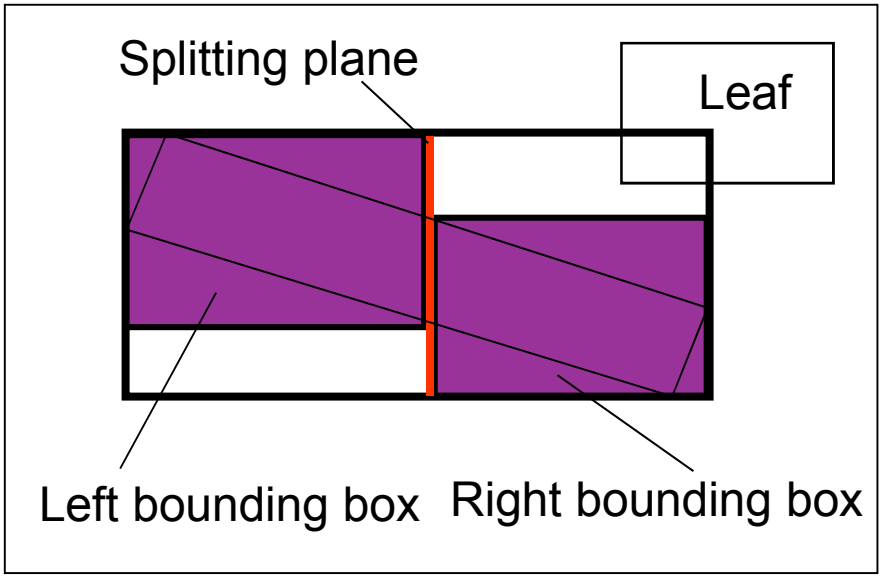


Kd-tree Efficiency Improvements

Cutting off empty space



Reducing objects' axis-aligned bounding boxes



Termination Criteria for Construction

- *Local: using a stack*
  - *Simple local*: maximum depth + number of objects (usually 1 or 2)
  - *More complicated local*: a maximum number of cost improvement failures + maximum estimated depth + number of objects
- *Global: using a priority queue*
  - maximum memory used
  - maximum memory used + maximum leaf cost

Note: maximum depth is good to select as dmax = k1 + k2 * log (N),

for example k1=3, k2 = 1.25

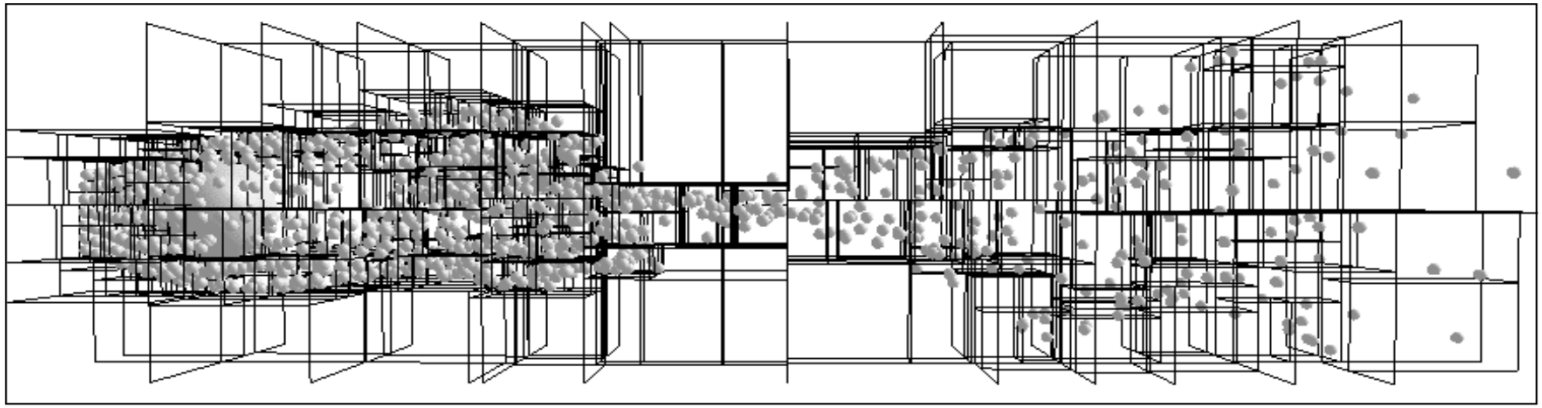
Construction in O(N log N)

There are two methods assuming N objects.

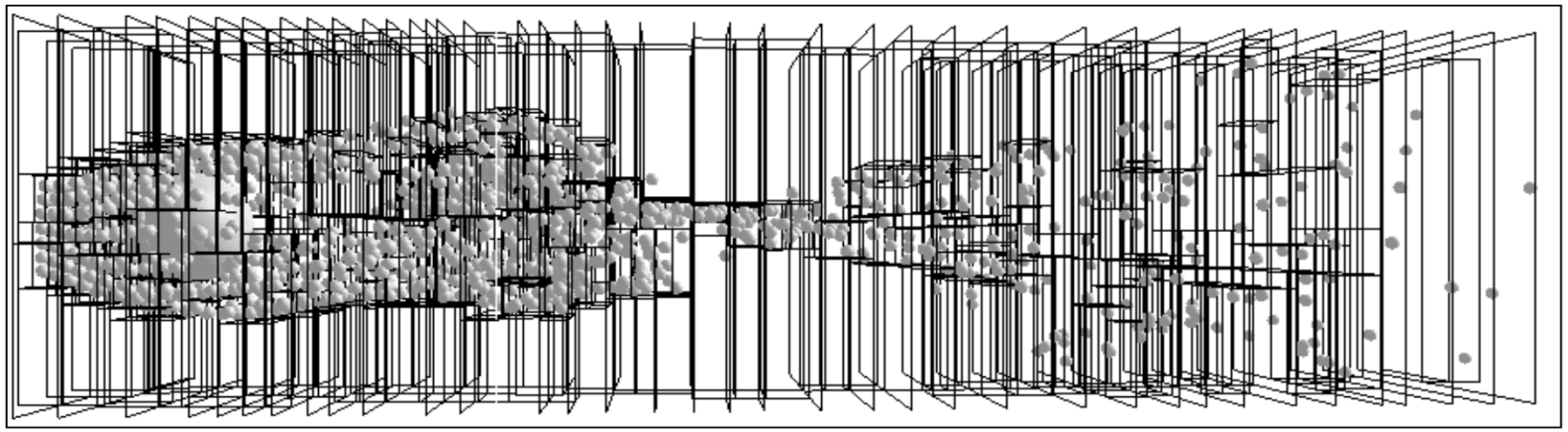
- **Presorting**
  - three sorted lists of boundaries, each list in one axis contains 2*N boundaries
  - for each list we compute cost function and select the position with minimum cost
  - the three lists are subdivided into six lists, three lists for left child, three lists for right child (requires to copy the boundaries of objects that straddle the splitting plane)
- **Discretized sampling**
  - preselect several positions and compute cost function only for them
  - the complexity is O(N * (k+log N)), where “k” is the number of preselected positions (k = 16 or so).

Kd-tree Construction for Preferred Ray Sets

Idea: different than uniform distribution of rays, gain 2 times up to 3 times

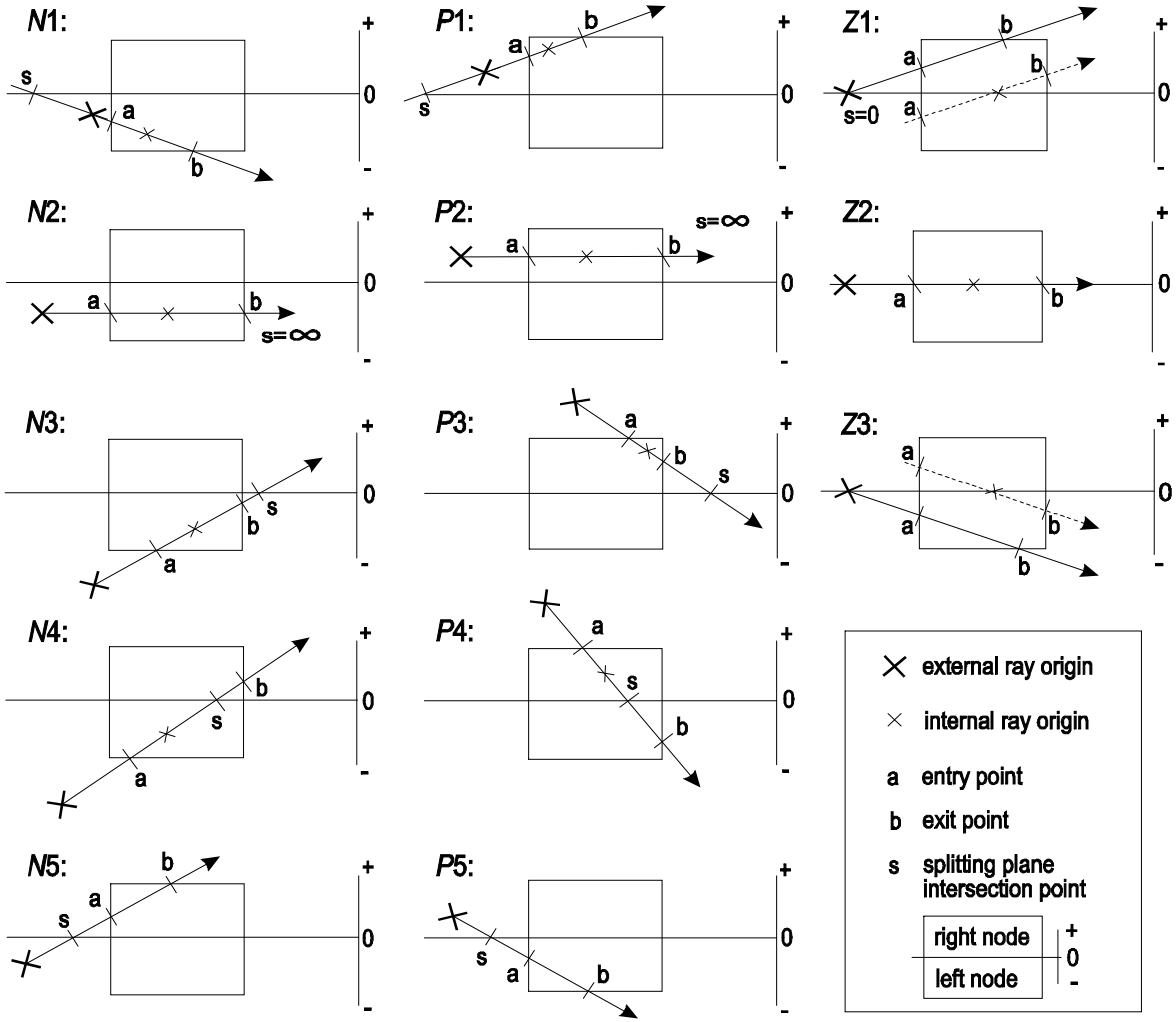


Uniform

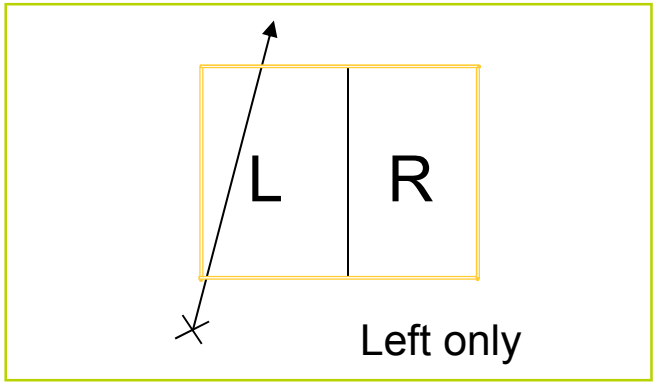


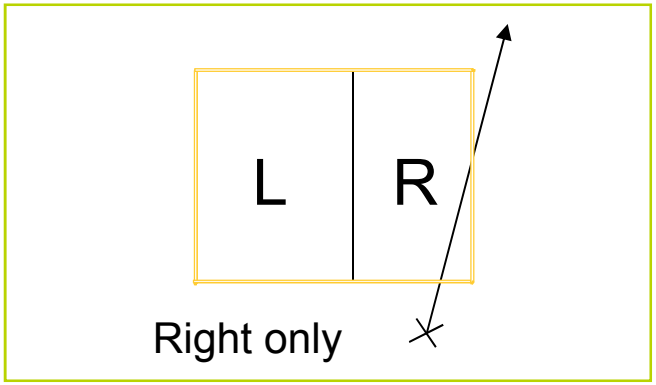
Preferred

Recursive Ray Traversal Algorithm for Kd-tree

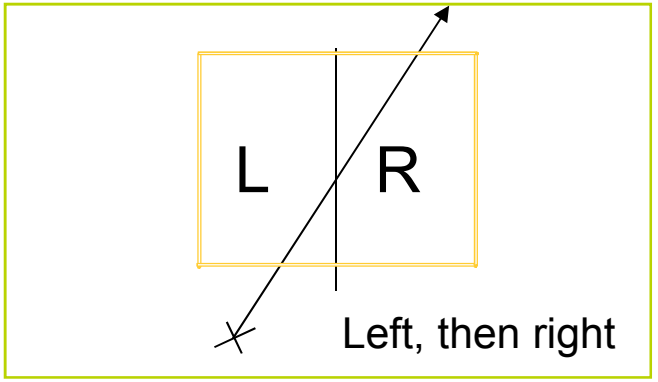


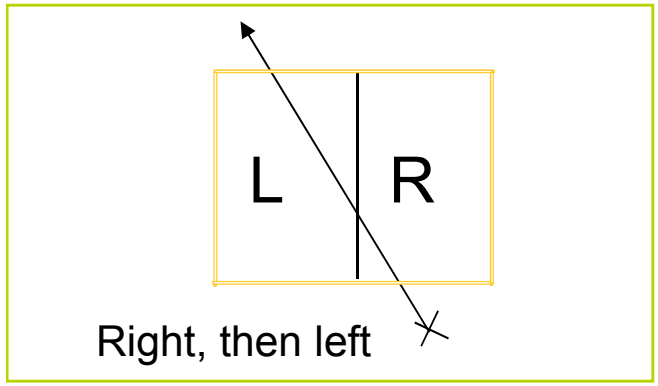
Recursive Ray Traversal
Basic Cases Classification





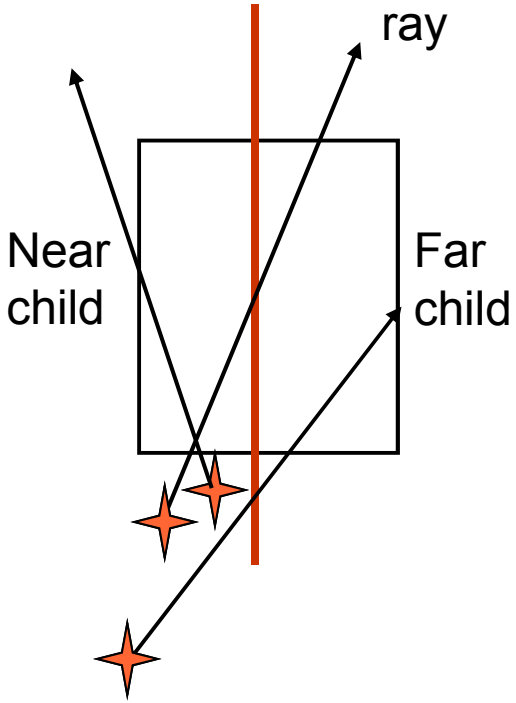
Interior node of kd-tree



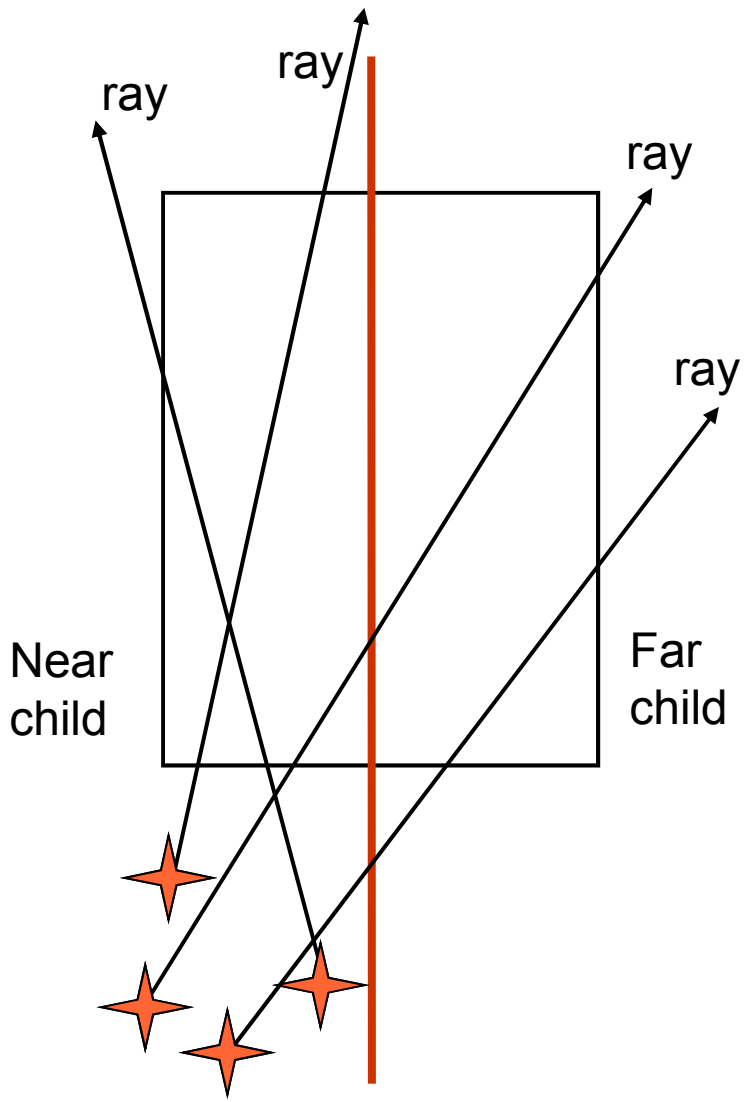


Traversal pseudocode for kd-trees, list of variables, naïve implementation

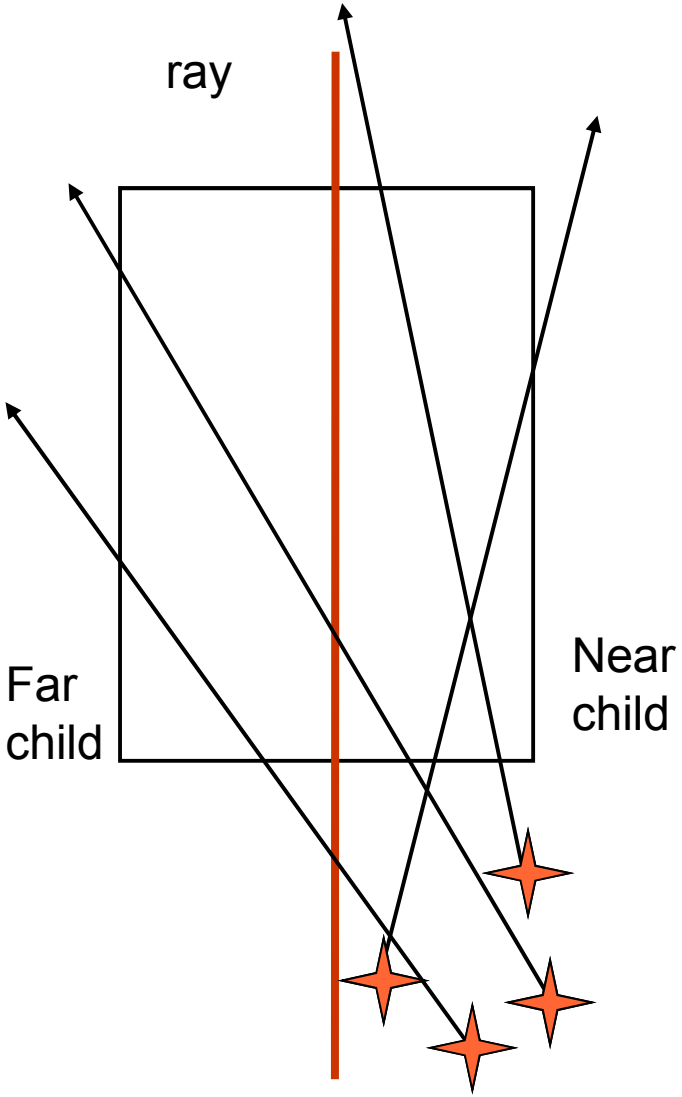
```
struct Skdnode {
    int axis; // 0-x, 1-y, 2-z, 3-leaf
    float value;
    Skdnode *leftChild, *rightChild;
    vector<CObject> *listOfobjects;
};
struct SStackEntry {
   Skdnode *node;
   float mint, maxt;
};
stack<SStackEntry> stacktrav;
float mint, maxt; // the signed entry and exit distance to the box
float t; // the signed distance along ray to the splitting plane
Skdnode *nearChild, *farChild; // the pointer to the near child and far child
```



Possible Traversal Cases for kd-tree



Ray.origin[axis] < SplittingPlanePosition

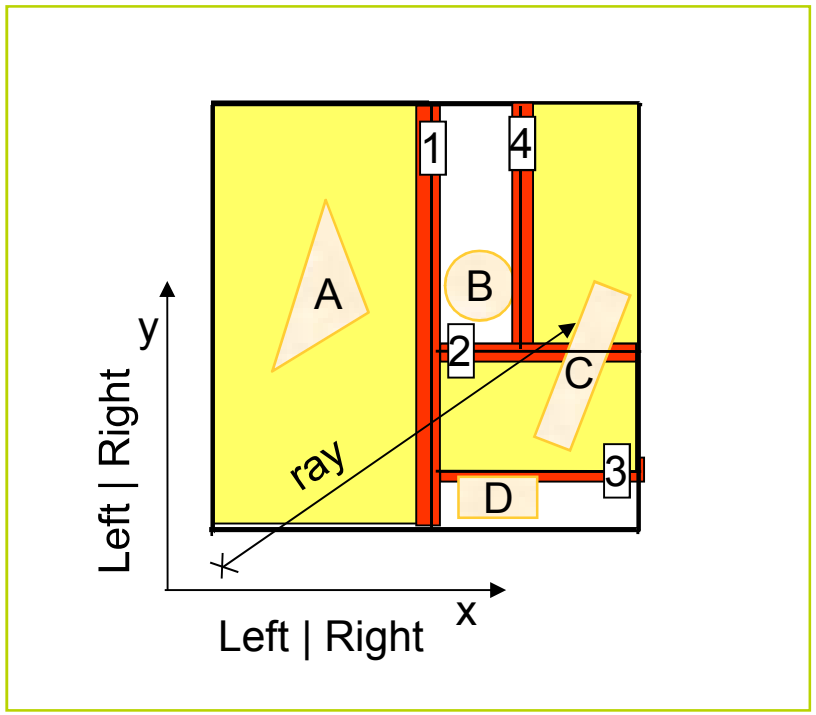


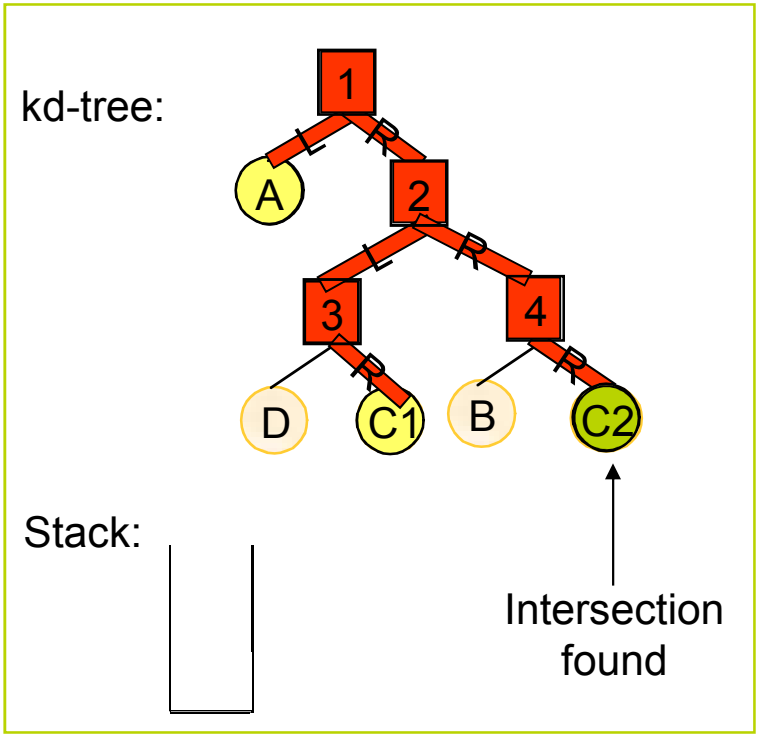
Ray.origin[axis] > SplittingPlanePosition

Traversal Pseudo-Code for kd-trees

```
Compute mint and maxt given the scene box and the ray;
stacktrav.Push(root, mint, maxt); // Init a stack with root node, mint, maxt
while stacktrav is not empty do {
  stacktrav.Pop(w, mint, maxt);
  while w is not a leaf do {
    int axis = w->GetAxis();
    float value = w->GetValue();
    if (value > ray.origin(axis)) { nearChild = w->rightChild; farChild = w->leftChild;  }
    else { nearChild = w->leftChild; farChild = w->rightChild;  }
    float t = (value - ray.origin(axis)) / ray.dir(axis);
    if  ((t < 0) || ( t > maxt) ) then
      w = nearChild; // visit only nearest child node
    else
      if (t > mint) then
        w = w->GetFarthestChild(ray); // visit only farthest child node
      else {
        stacktrav.Push(GetFarthestChild(ray), t, maxt); // visit farther child later
        w = w->GetNearestChild(ray); // visit first near child
        maxt = t;
       }
  } // while
  Compute intersection between ray and all objects referenced in a leaf node w;
  if any ray-object intersection lies between mint and maxt, then return the object; // FINISHED
} // while
return no object;  // no object is intersected by ray - FINISHED
```

Recursive Ray Traversal Algorithm



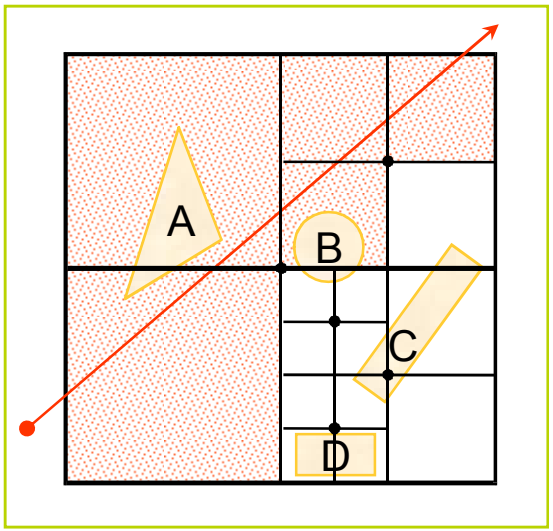


Note: more variants of traversal algorithms for kd-trees in the survey paper by

M. Hapala, V. Havran: *"Review: Kd-tree Traversal Algorithms for Ray Tracing"*, in journal Computer Graphics Forum, Vol. 30, Issue 1, pages 199–213, 2011

Ray Shooting with Octrees

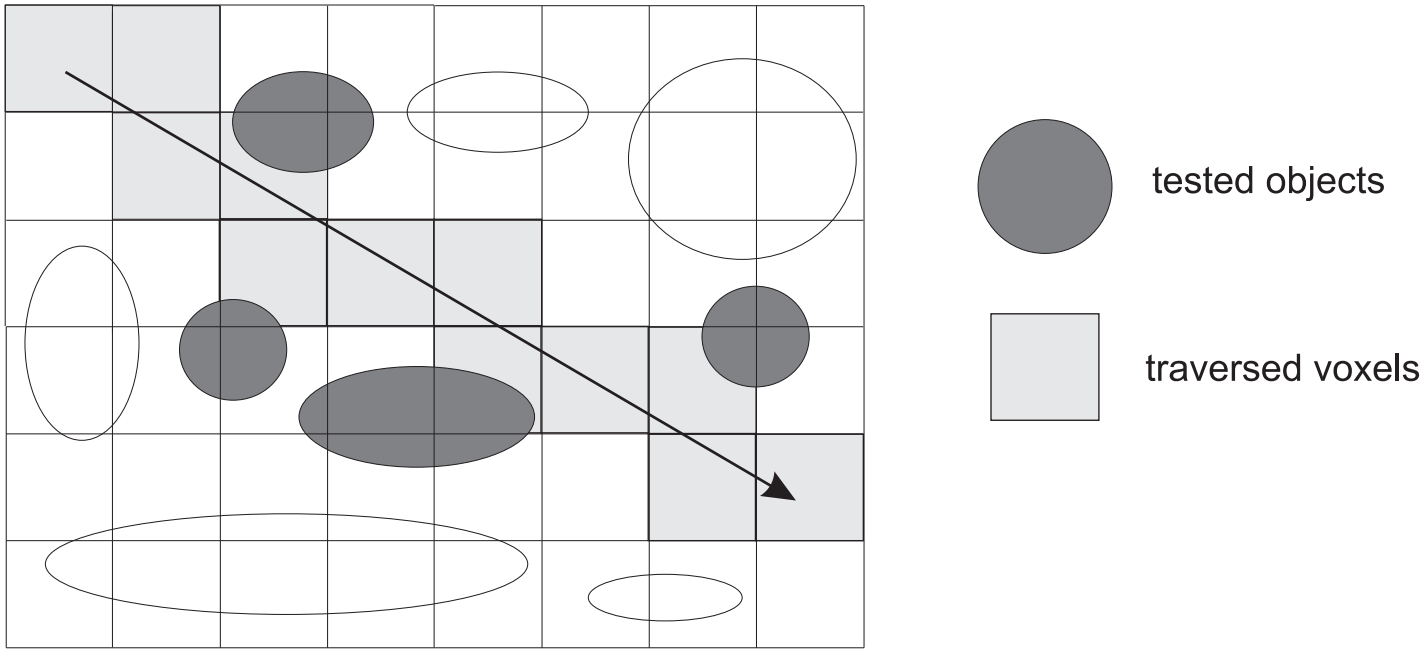
- Interior node arity (branching factor) is eight
- Up to four child nodes can be traversed in an interior node
- Traversal algorithm necessarily more complicated than for kd-trees
- About 26 papers about ray tracing with octrees were published
- Octrees are less adaptive to the scene object distributions than kd-trees
- Geometric probability can be used in the same way as for kd-trees (Octree-R)
- According to the experiments, octrees are less efficient than kd-trees even if we use the most efficient traversal algorithm for octree.



Note: octrees can be simulated by kd-trees

Ray Shooting with Uniform Grids

- Arity (branching factor) of a node proportional to the number of objects
- Traversal method based on 3D discrete differential analyzer (3DDDA)
- For skewed distributions of objects in the scene it is inefficient
- For highly and moderately uniform distributions of objects it is slightly more efficient than kd-trees
- Construction time is only O(N).



Trick: Mailboxing

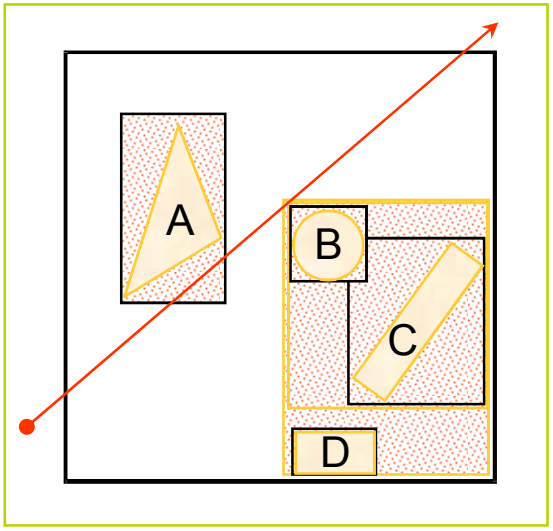
- Some objects are contained in more than one cell
- An object that was tested for ray-object intersection can be tested again
- Idea: save already computed results of ray-object intersections. Instead of computing it again, we find the result in the records.
- Suitable for complex object primitives such as NURBS etc.
- Does not pay off for triangles, spheres, and other easy primitives since the computation overhead to maintain the cache is higher than to compute the result repetitively.

Mailboxing variants

1. Each object has its own data for mailboxing – the results are exact, but it requires much more memory (8 Bytes for each object)
2. There is a small array cache of size N (8 or 16 entries) and by hashing function we record by simple hashing function (triangle ID mod N) the result to the cache (triangle ID + result)

Ray Shooting with Bounding Volume Hierarchy (BVH)

- Each interior node is fully described by a bounding box
- The number of child nodes is usually two for top-down construction (more for bottom-up construction)
- The nodes can overlap
- Each object is referenced only once (no mailboxing) and the number of nodes is limited



- The storage of a node is high … the memory consumption is higher than for kd-trees, typically 32 Bytes
- Efficient traversal algorithm is very similar to kd-trees, but we cannot finish the traversal upon the first intersection
- Kd-trees can be emulated by BVHs.

Three Variants of BVHs Construction

- Constructed in top-down fashion
- Constructed incrementally by insertion
- Constructed by clustering
- Constructed in any way and further optimized, (article "Optimizing the Cost of Bounding Volume Hierarchies via Rotations", 2008 IEEE Symposium on Interactive Ray Tracing (2008) DOI: 10.1109/RT.2008.4634624)

1) BVH Constructed in Top-Down Fashion

- Very similar to kd-trees
- Each interior node has two children.
- Compute the cost function as for kd-trees, but two child nodes can overlap.
- The ray traversal algorithm can first visit closer cells, then farther cells as for kd-trees.
- The performance of ray-tracing with these BVHs is comparable with kd-trees. O(N log N) complexity. The analogy is *quicksort*.

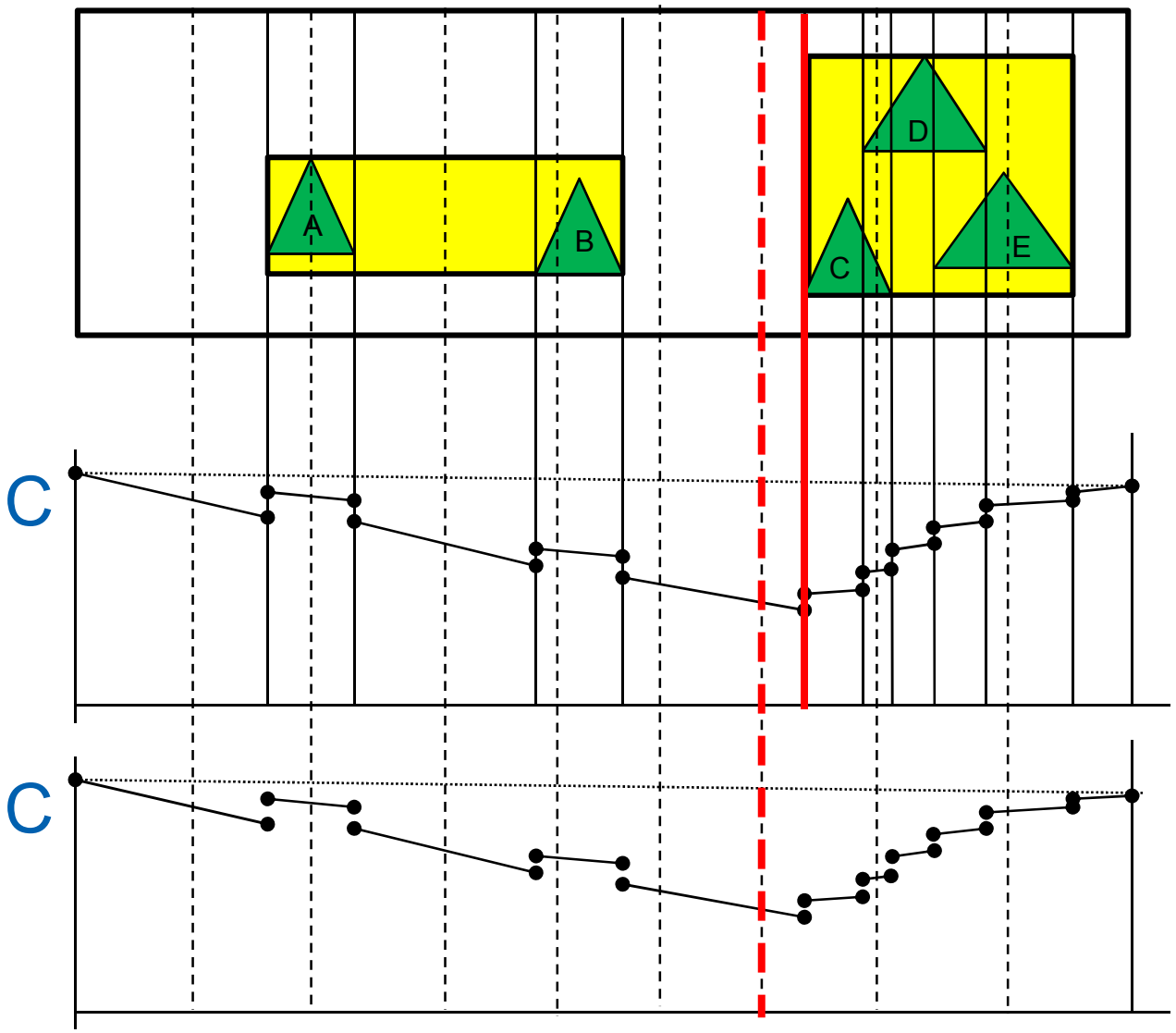
2) BVH Constructed Incrementally by Insertion

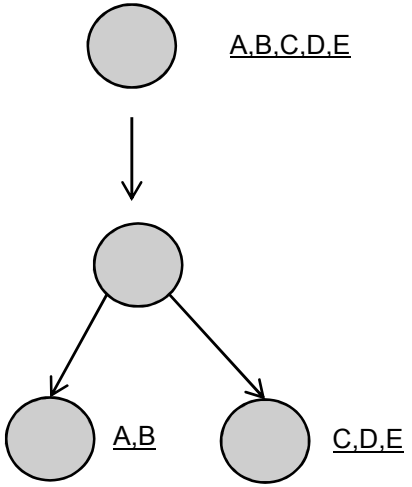
- We insert each object into partially constructed tree based on the cost function. Upon insertion either the new child is created, or a child node becomes an interior node with two children. Generally, the interior node can have several children.
- The cost of such BVH is much worse than for top-down construction.
- The traversal algorithm has to visit all interior nodes intersected by a ray. O(N log N) complexity. The analogy is *insertion sort*.

3) BVH Constructed by Clustering

- Up to $O(N^2)$ complexity, similar properties to variant 2), no ordering of child nodes in an interior node, it uses only a distance. Note: analogy is *merge sort*. RT 2008 paper – O(N log N) by locally organized agglomerative clustering.

Exact versus Approximate Cost Evaluation

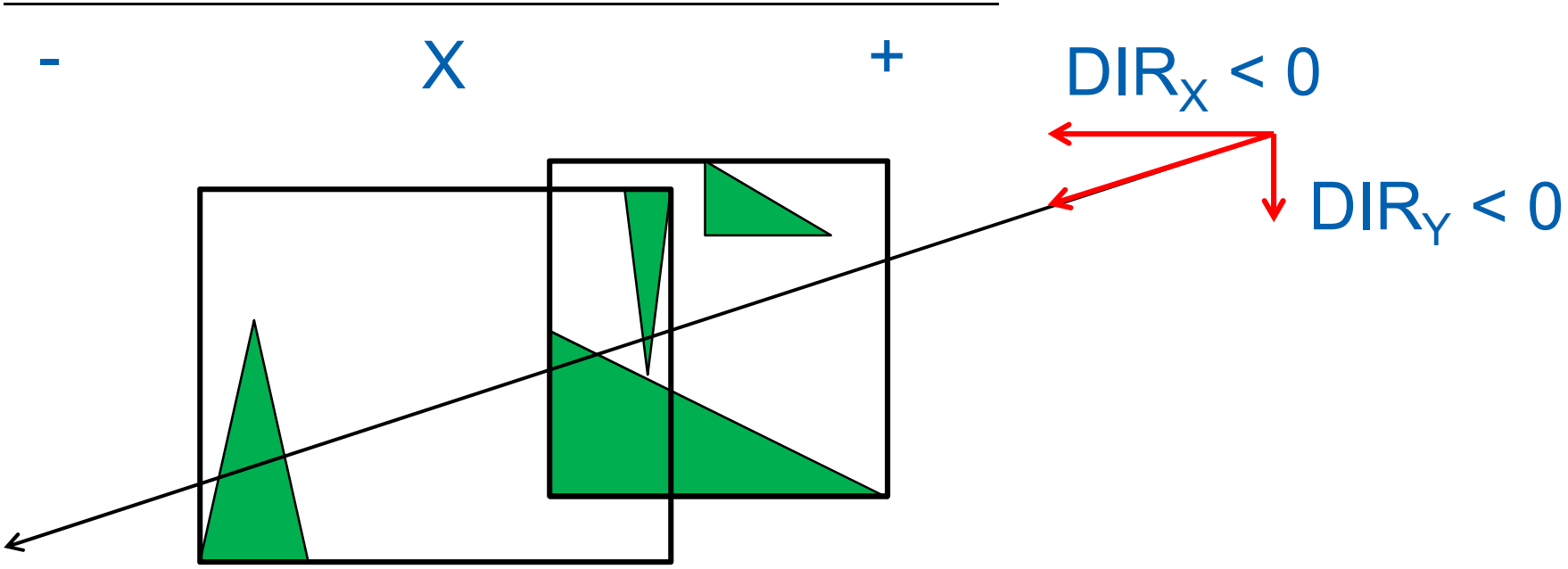




- Exact using boundaries
- Approximate with 8 samples

BVH Traversal Algorithm

- Similar, but the ray has to be checked along its traversed path until the first intersection found
- The bounding boxes in principle arbitrary, in practice a single axis orientation is encoded as for kd-trees in 2 bits



Two-Step Algorithm Building Data Structures for Ray Shooting

- Motivation
  - uniform grids are constructed in O(N) time. Uniform grids are efficient for uniform distribution of objects in the scene, but do not work well for skewed distribution.
- Algorithm
  - Construct uniform grid in O(N) time
  - Estimate if the grid will be fast enough (compute statistical characterization of scene known as variance, skewness, and kurtosis or shoot few sample rays)
  - If the uniform grid is estimated to be efficient, use it. Otherwise, discard the uniform grid and construct a standard kd-tree or BVH.

Statistical Scene Characterization

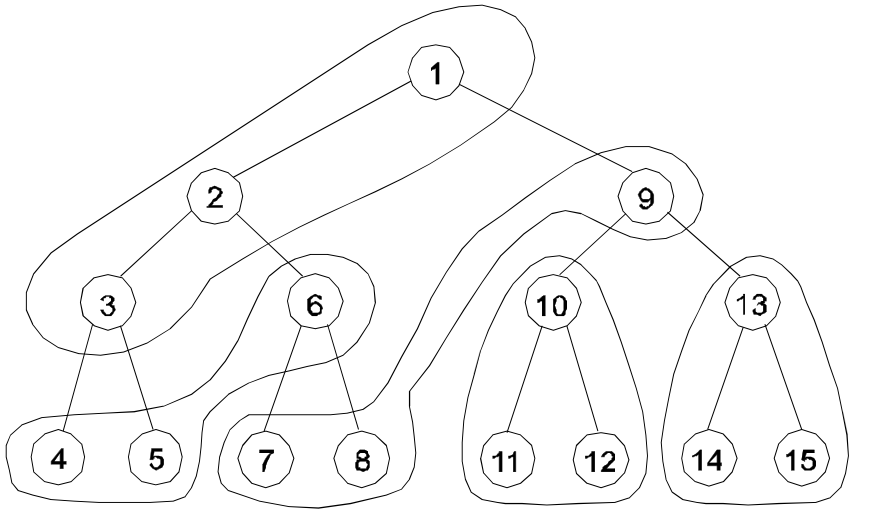
- Initially – distribute the scene to a uniform grid cells
  - N_v … number of cells (proportional to number of objects)
  - n_i … number of objects associated with a cell “i”
- Variance: $v = 1/N_v \cdot \Sigma (n_i - n_{avg})^2$,

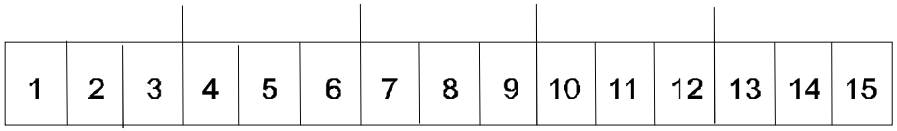
  where mean n_{avg} is a mean: $n_{avg} = 1/N_v \cdot \Sigma n_i$
- Standard deviation $\sigma = \text{sqrt}(v)$
- Skewness: $s = 1/N_v \cdot \Sigma ((n_i - n_{avg})/\sigma)^3$
- Curtosis: $k = 1/N_v \cdot \Sigma ((n_i - n_{avg})/\sigma)^4 - 3$

Data Layout of Trees in Memory

Inorder, preorder (depth-first-search), heap (breadth-first-search), van Emde Boas

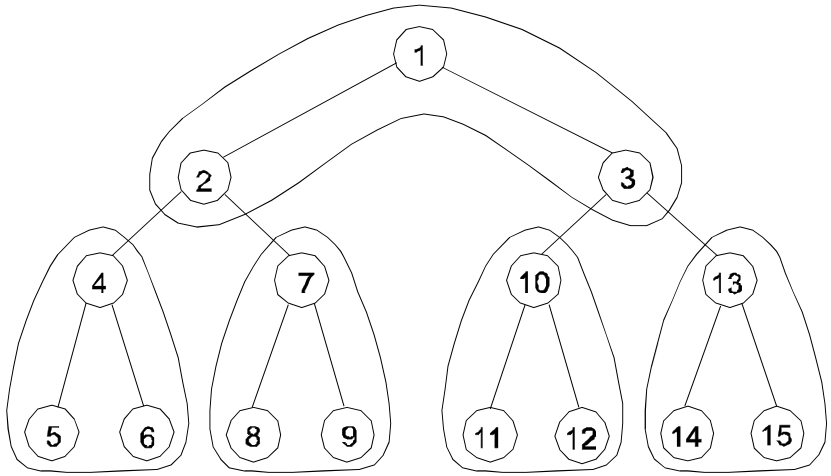
By standard memory allocator

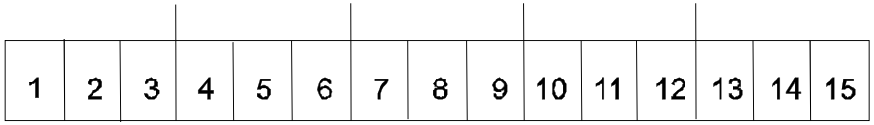




Depth-first-search (DFS) layout

Needs rewriting





van Emde Boas layout

Performance Model of Ray Shooting

Total cost for RSA =

cost for ray-object intersection tests +
cost for ray traversal of kd-tree +
cost for data move from memory to CPU

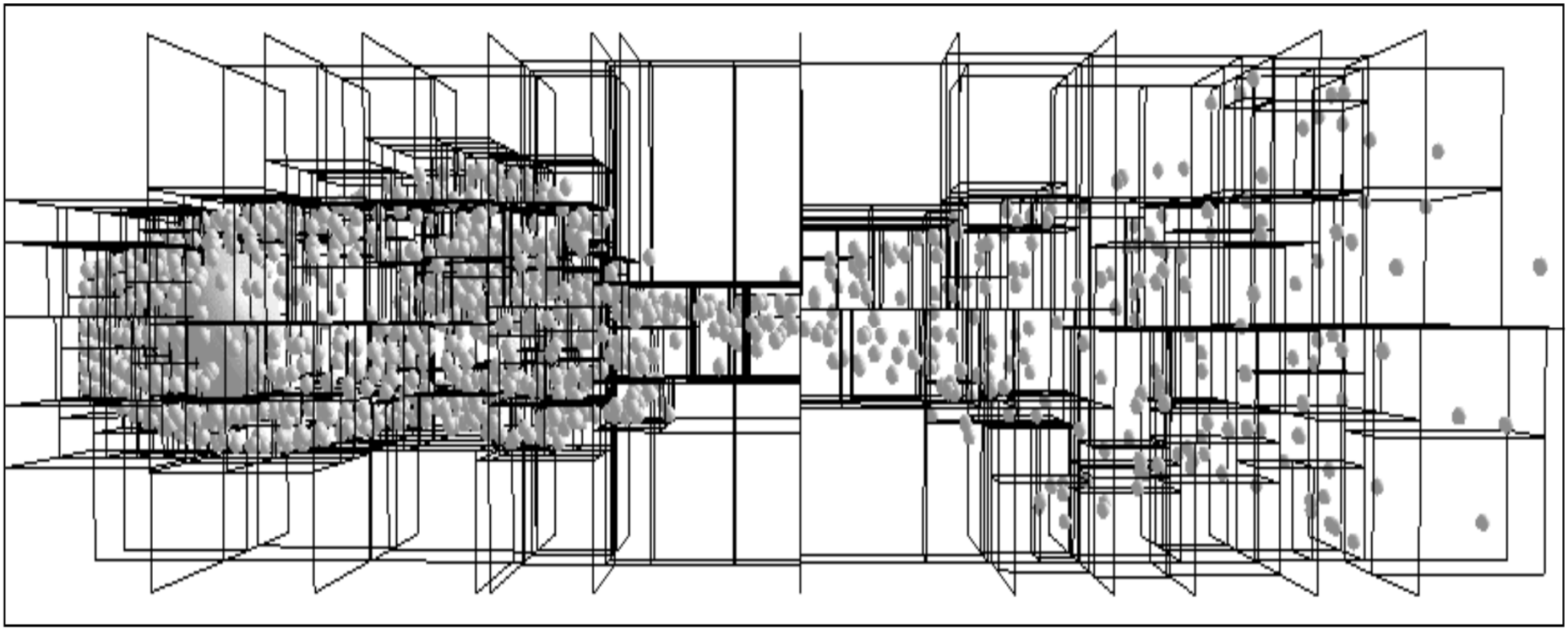
- Faster ray-object intersection tests
- Decreasing number of ray-object intersection tests
- Faster traversal step
- Decreasing number of traversal steps
- Reducing CPU-memory traffic

Offline Ray Shooting

- Shooting several rays at once
- Rays are formed by camera, by viewing frustum or by point light sources
- Rays are **coherent** = similar in direction and origin
- Problem can be formulated as offline setting of searching
- We can amortize the cost of traversal operations though the data structure … the number of traversal steps is decreased typically by 60-70%
- Solving by LCTS – longest common traversal sequence

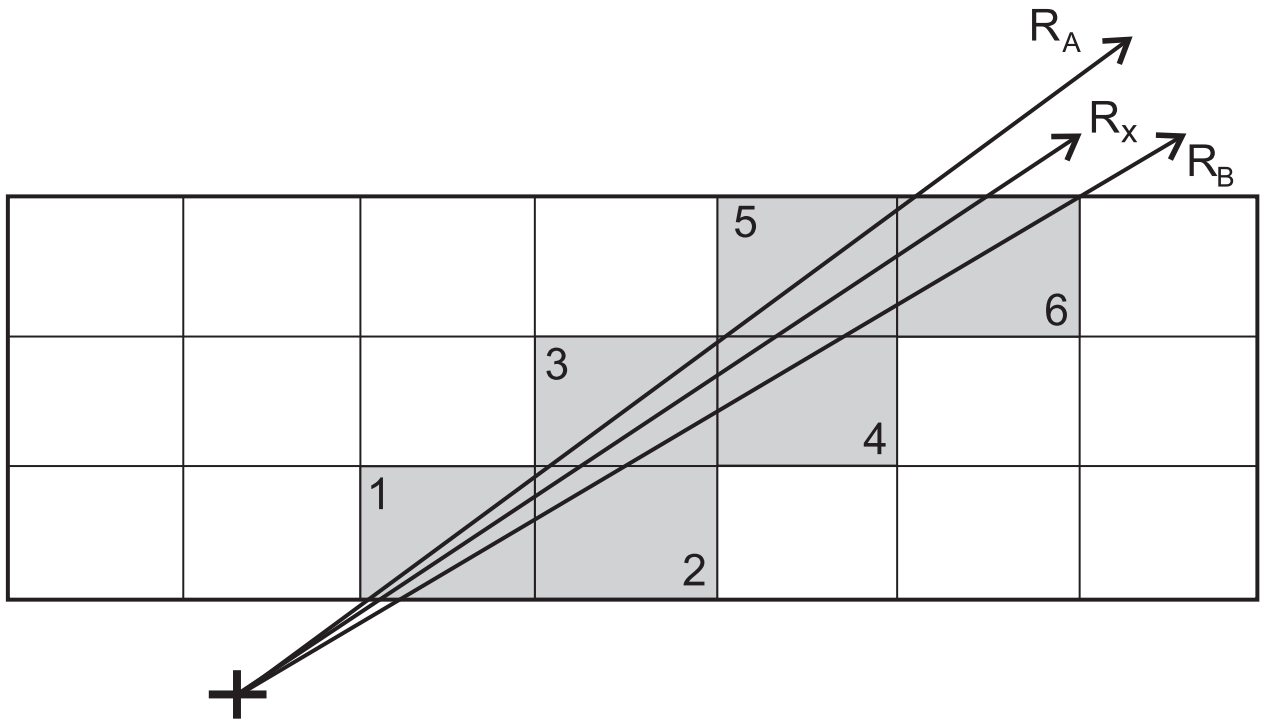
How the cells look at the projection to the viewer ?

Constructed for uniform ray distribution

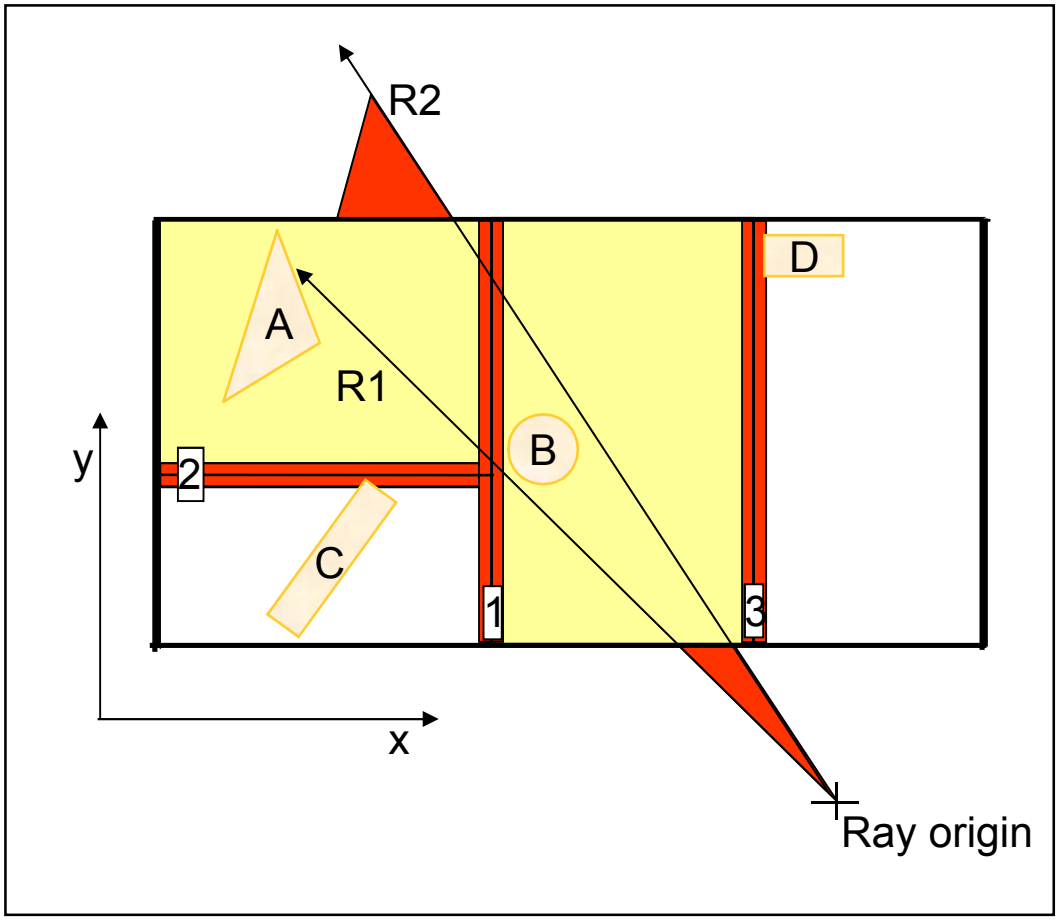


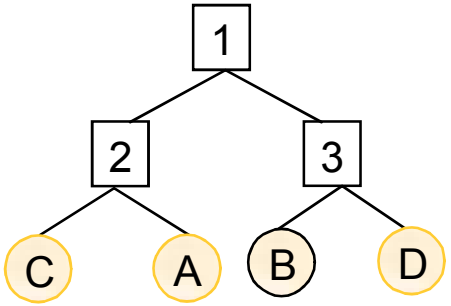
Offline Ray Shooting: Coherence

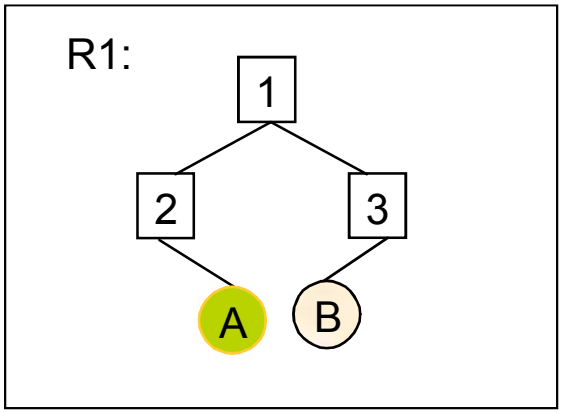
- If boundary rays traverse the same sequence S of leaves, then all rays in between also traverse the same sequence.
- Proof by convexity (convex leaves, convex shaft)

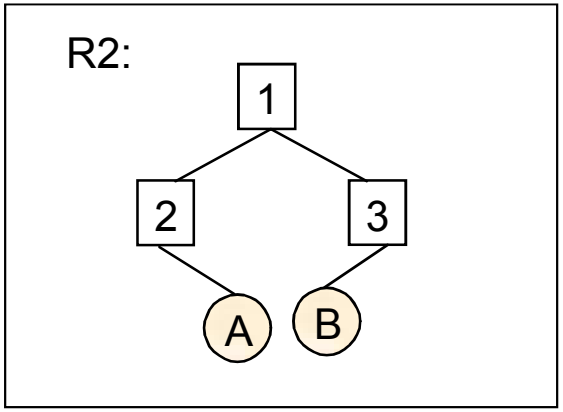


Offline Ray Shooting in HDS: Principle



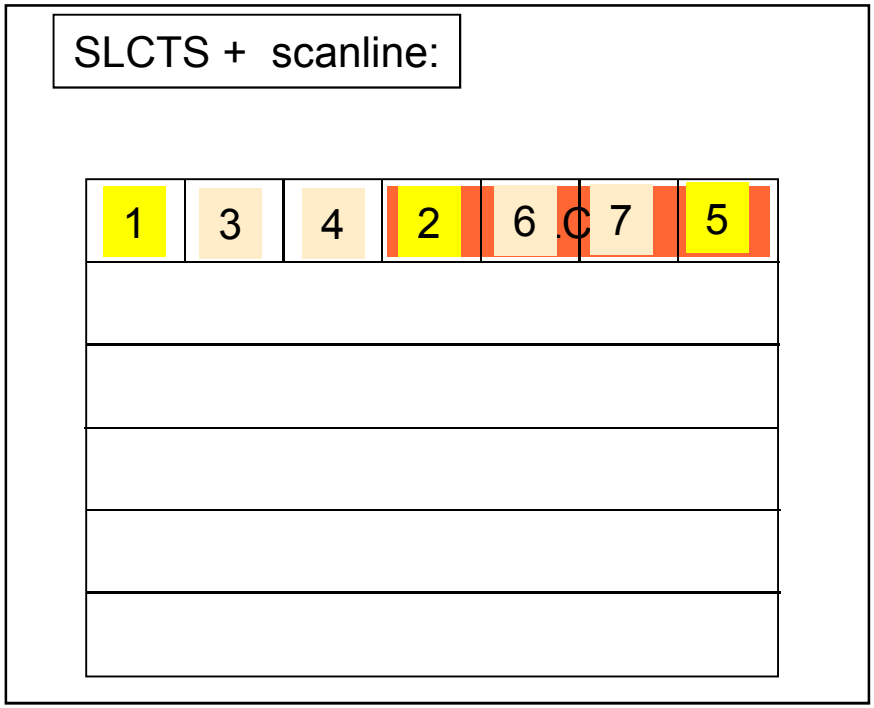


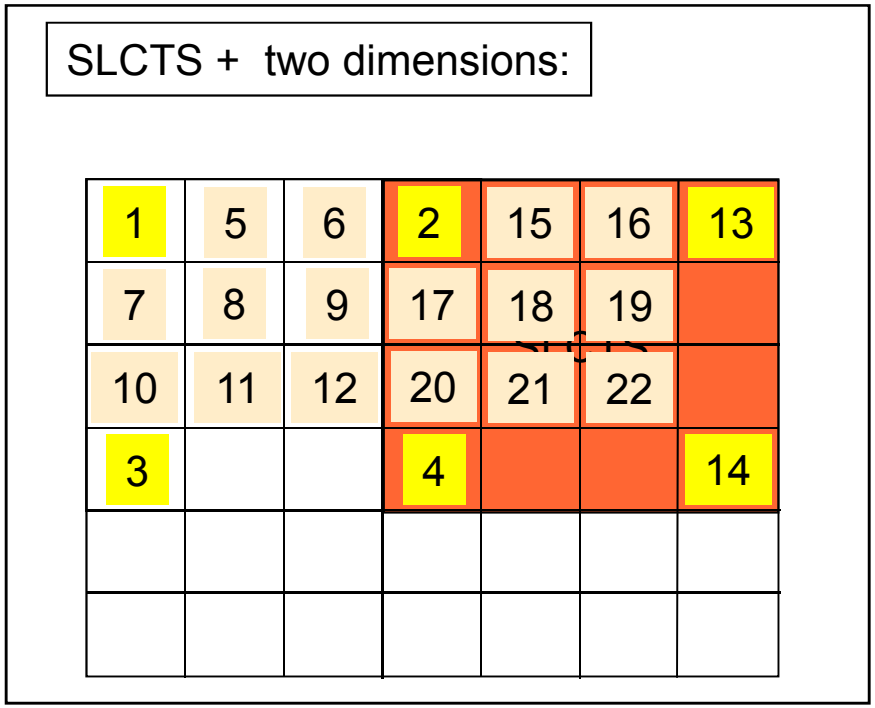




Sampling in Image Space

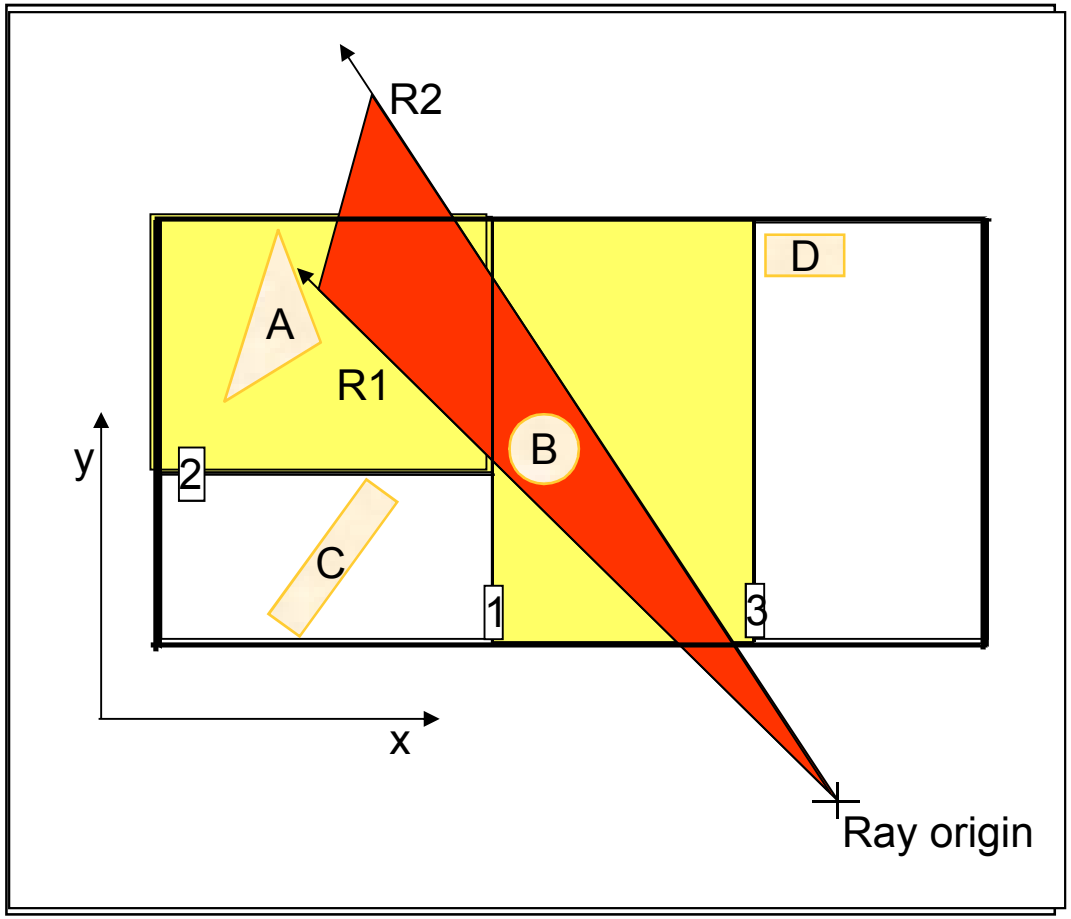
Hidden surface removal based on LCTS concept in one or two dimensions.

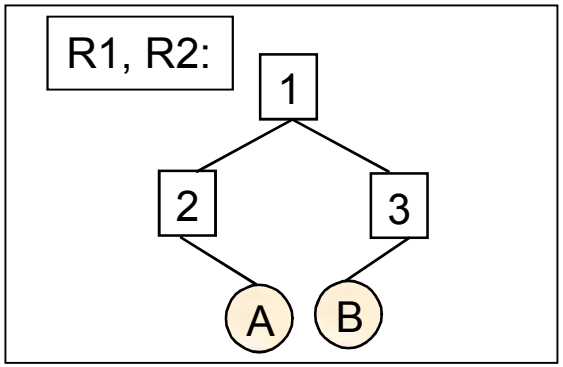


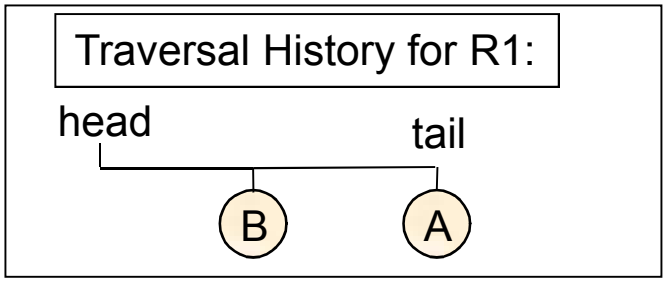


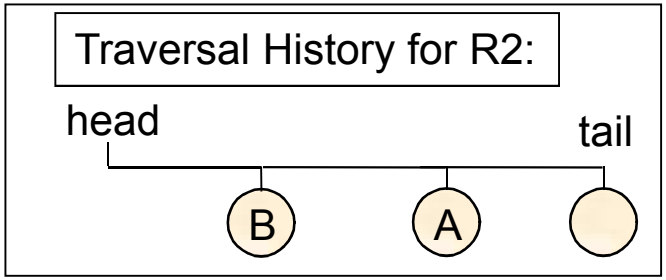
Other schemes: hierarchical image sampling

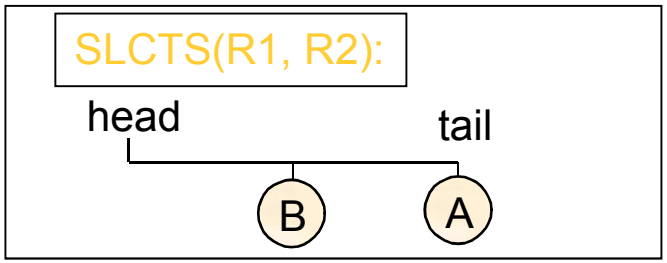
Simple LCTS = Sequence of Leaves







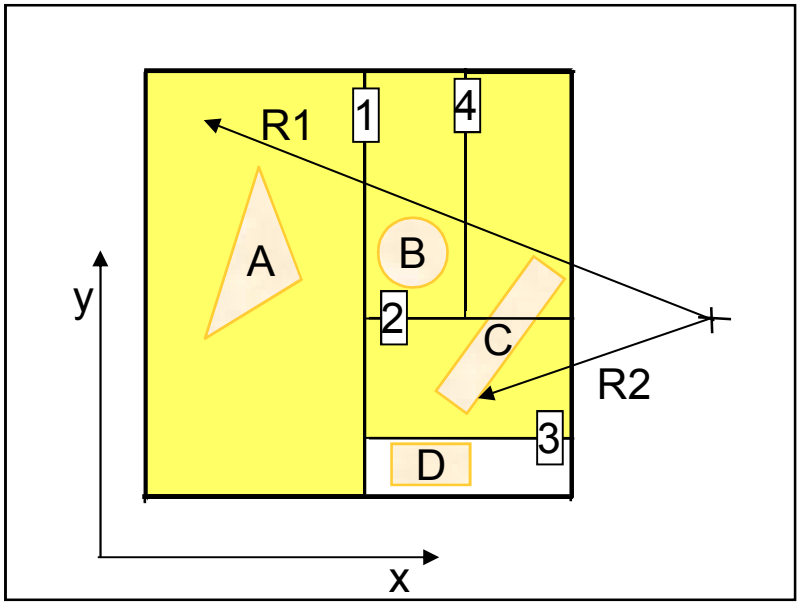


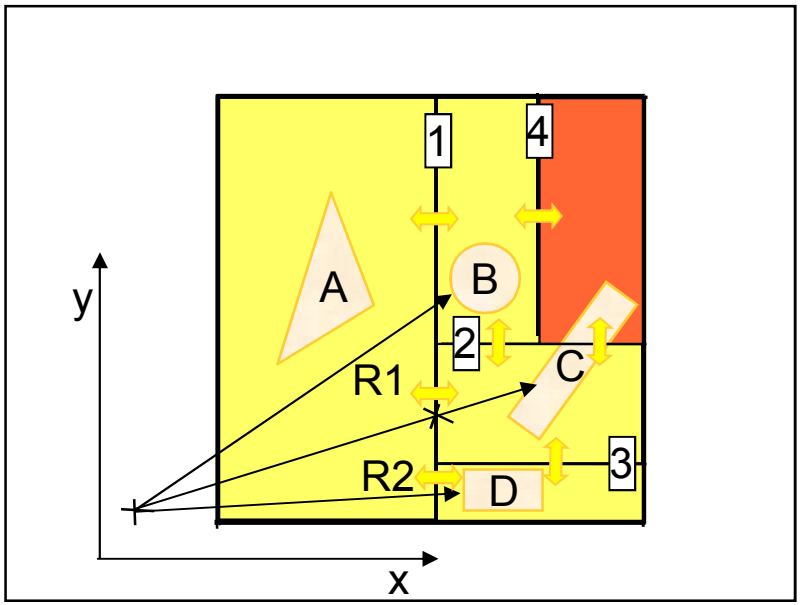


Simple LCTS - Problems

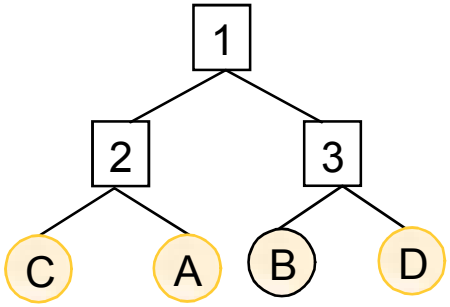
1) No common sequence of leaves exists.

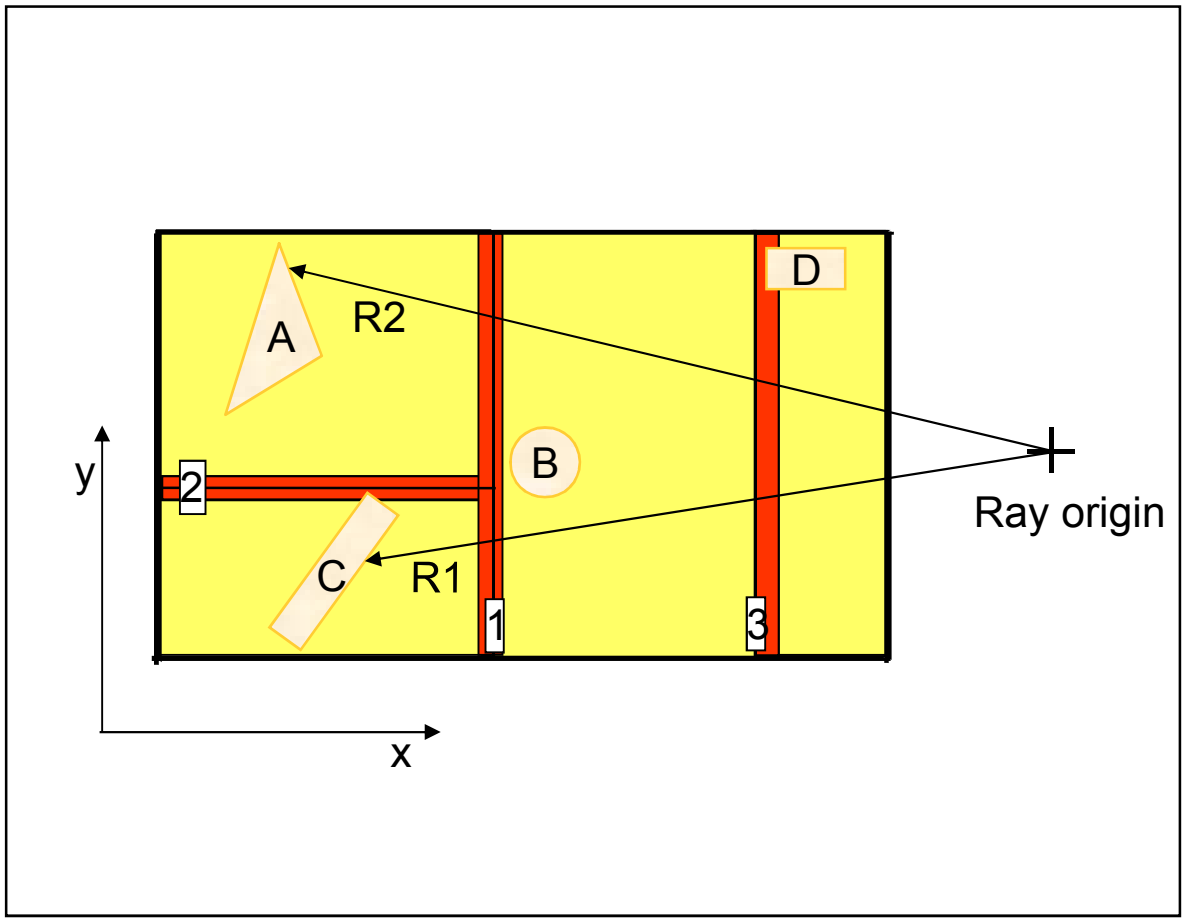
2) When accessing SLCTS, object was not found, and traversal has to continue further.



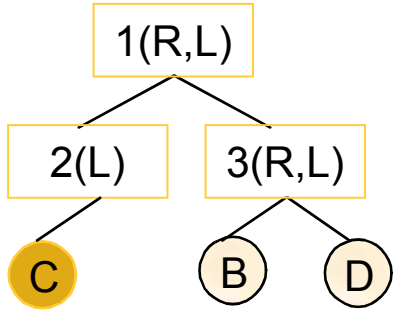


Hierarchical LCTS

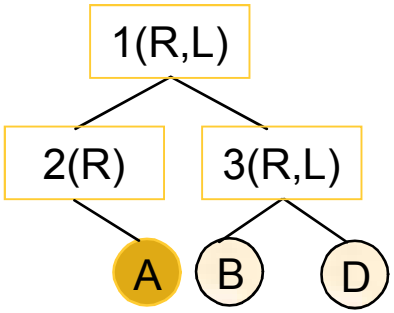




Traversal History for R1:

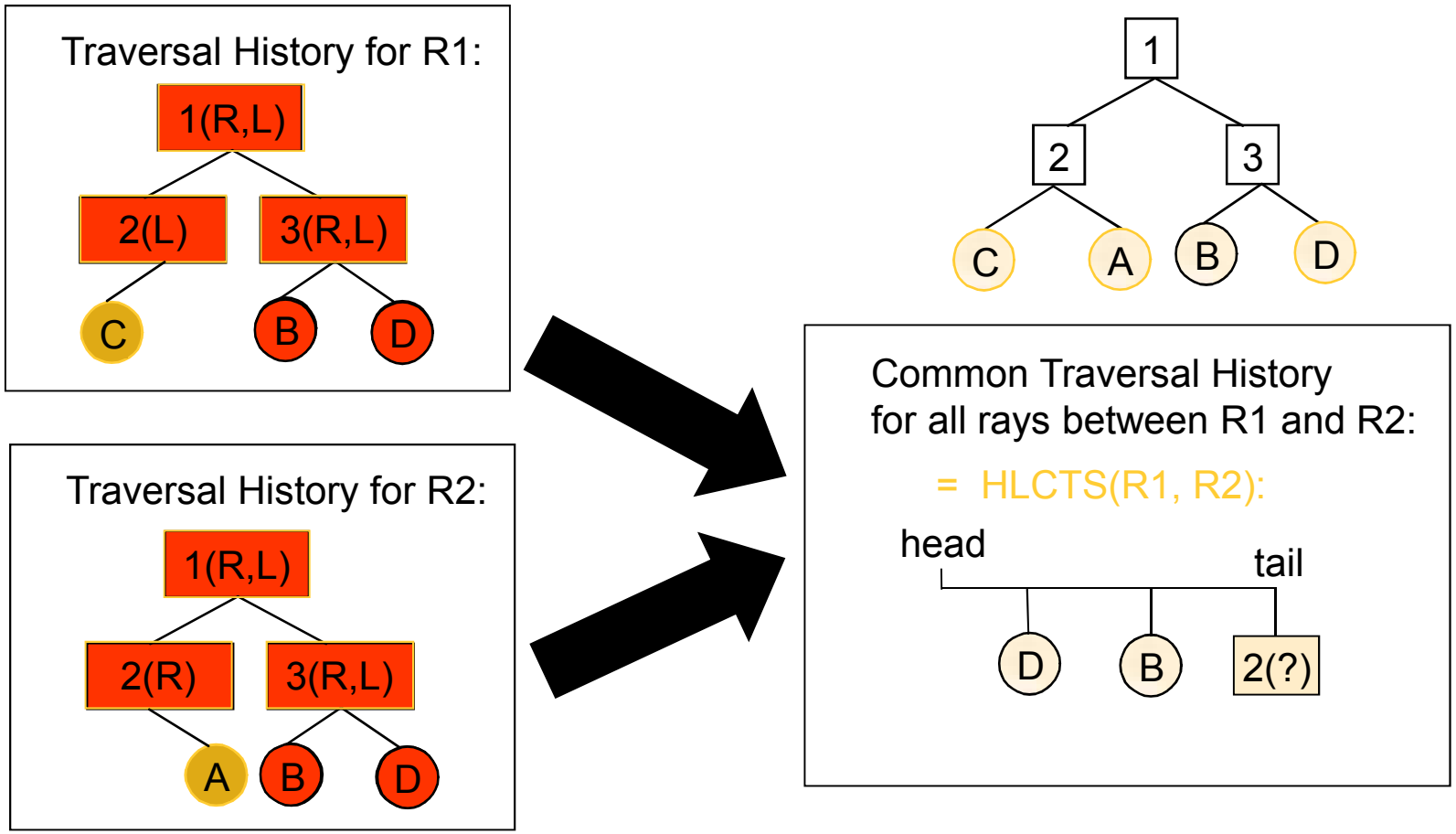


Traversal History for R2:



Hierarchical LCTS contd.

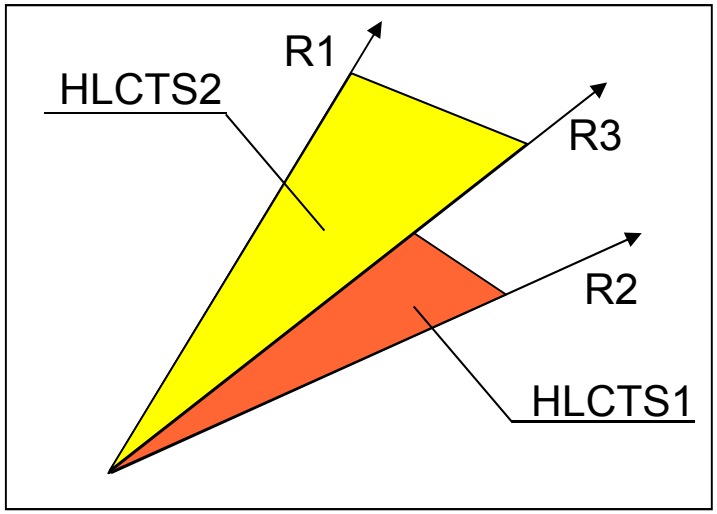
Matching two traversal histories into common one:



Hierarchical LCTS contd.

1) Matching traversal histories for two or more rays.

2) Matching traversal histories for rays with the previously constructed common traversal history.



HLCTS1 - constructed from traversal history of R1 and R2

Ray R3 - traversal uses HLCTS1

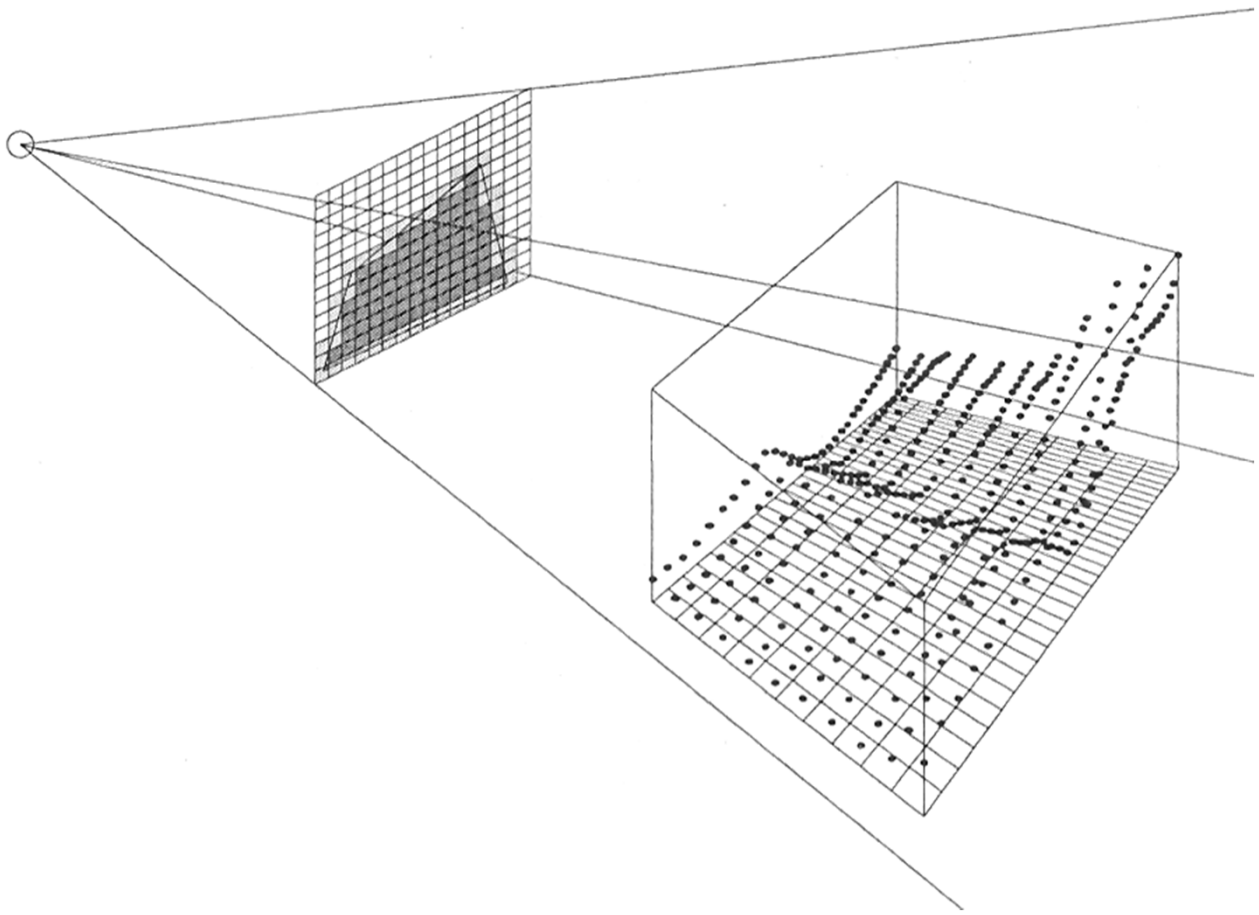
HLCTS2 - constructed from HLCTS1 and traversal history of R3

Special Issues in Ray Tracing

- Ray tracing heightfields
- Volumetric ray tracing
- Approximate ray tracing
- Ray tracing with ray cache for final gathering

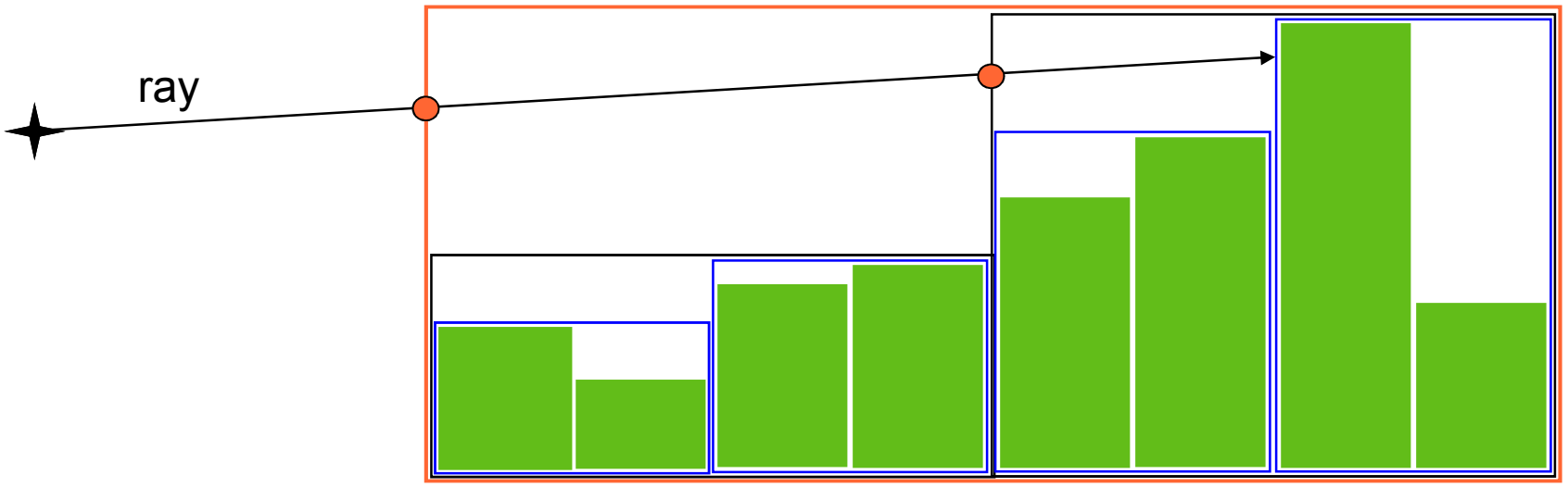
Ray Tracing Heightfields

- Heightfield: 2D array + height for each location in 2D array:



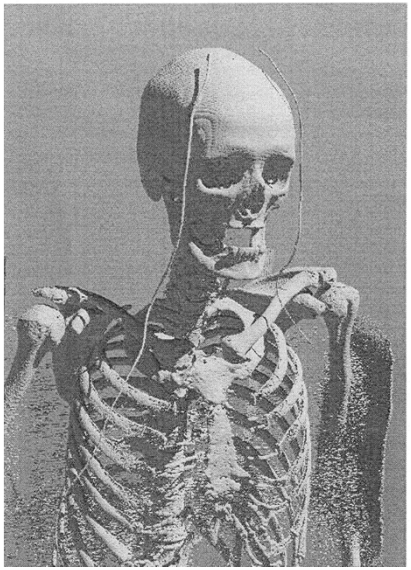
Fast Traversal Algorithm for Heightfields

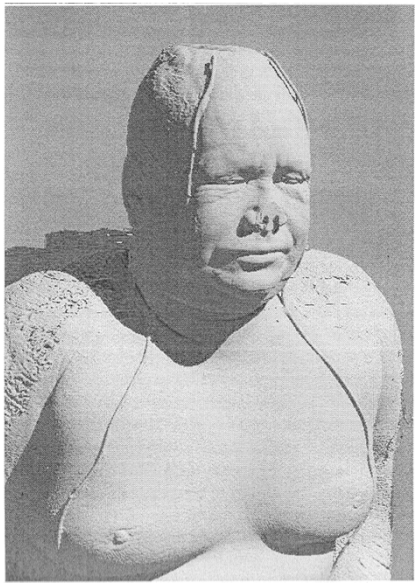
- Construct either quadtree or kd-tree and store in each interior node min/max value of height of all child nodes rooted here.
- During ray traversal operation we skip the parts of the scene that cannot be intersected:



Ray Tracing Volumetric Objects based on Isosurfaces

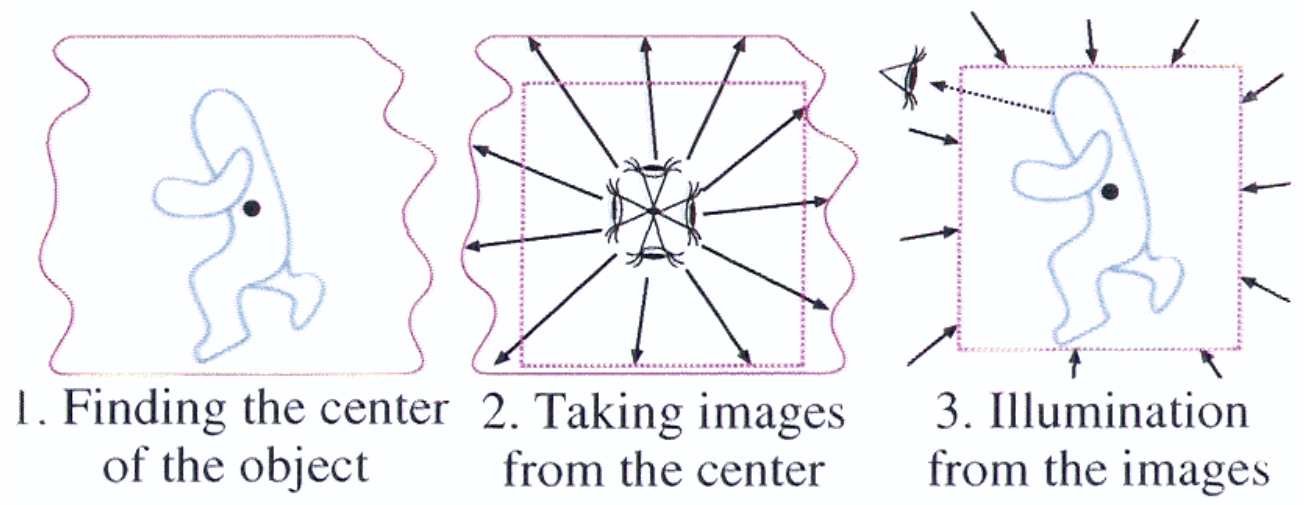
- Source data: computer tomography
- Output of measurement: 3D grid + density for each voxel in the grid – similar to heightfields, but data have one dimension more
- Output of rendering: 2D image, each ray is finished at the voxel where the density is higher than a selected threshold
- Skipping of low-level density regions can be implemented similarly to height-fields. This requires either octrees or kd-trees and remembering minimum and maximum density in all spatial region rooted in interior nodes.





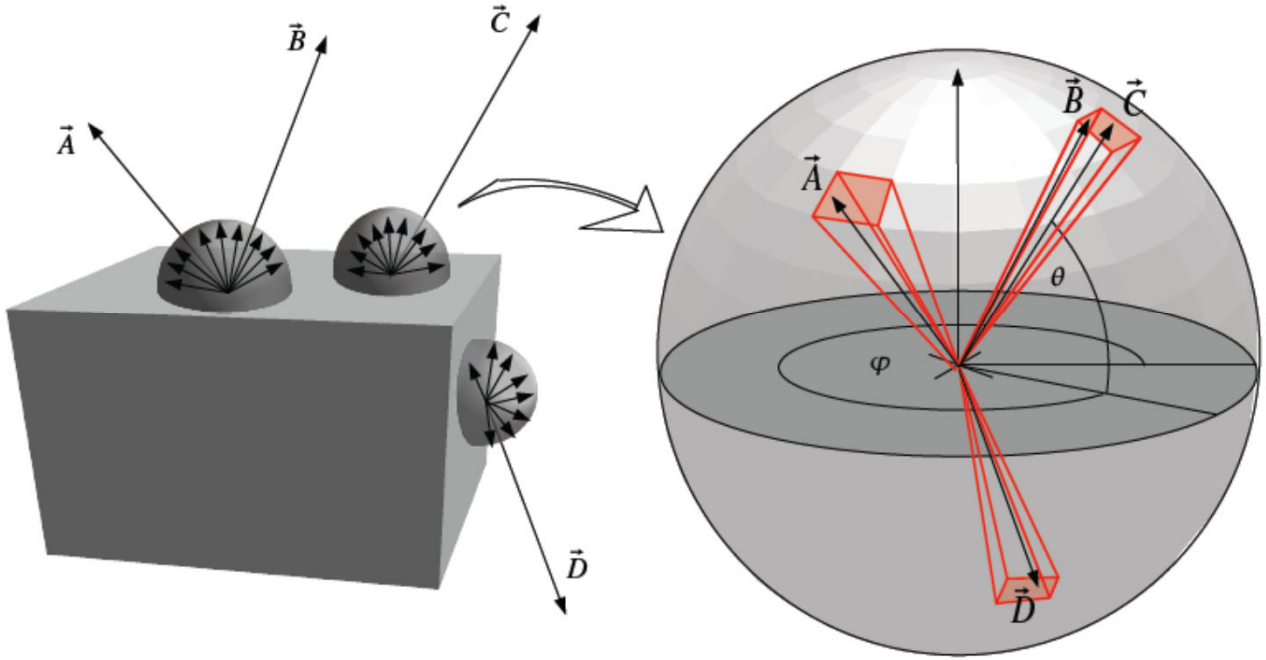
Approximate Ray Tracing with Distance Imposters

- Details in the paper: Szirmay-Kalos et al.: Approximate Ray-Tracing on the GPU with Distance Imposters, 2005.
- Idea: approximate environment with the imposters given the depth from the center of the scene.
- Use iterative algorithm to find out the intersection with high probability if the scene is fat (ratio of longest and shortest side is small)
- Possible to implement on GPU: suitable for games



Ray Cache in Final Gathering

- Store the rays into cache according to direction
- When a bucket is filled in, shoot all of them at once
- Improves cache access pattern for incoherent queries
- Sorting brings the speedup up to 30%



Surveys on Ray Shooting and Ray Tracing

- G. Simiakakis: *Accelerating Ray Tracing with Directional Subdivision and Parallel Processing*, PhD Thesis, 1995
- V. Havran: *Heuristic Ray Shooting Algorithms*, PhD Thesis, 2001
- I. Wald: *Real Time Ray Tracing and Global Illumination*, PhD Thesis, 2004
- A. Y-H. Chang: *Theoretical and Experimental Aspects of Ray Shooting*, PhD Thesis, 2005
- Wald et al.: *State of the Art in Ray Tracing Animated Scenes,* Computer Graphics Forum, 2009


Thank You for
Your Attention