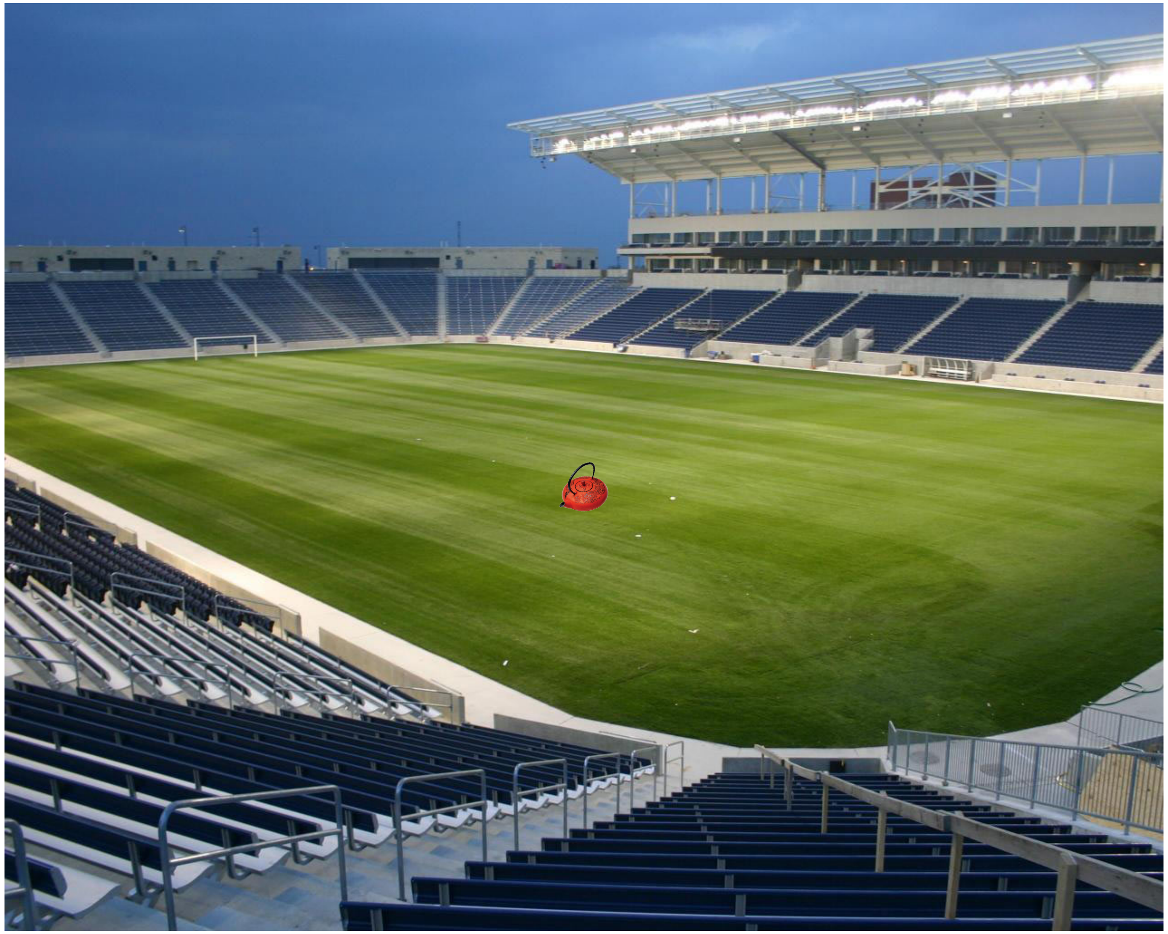
The Optimal Number of Voxels

- Too many cells → slow traversal, heavy memory usage, bad cache utilization
- Too few cells → too many objects/triangles per cell
- Good rule of thumb: choose the size of the cells such that the edge length is about the average size of the objects (e.g., measured by their bbox)
- If you don't know it (or it's too time-consuming to compute), then choose cell edge length = $\sqrt[3]{N}$, N = # objects
- Another good rule of thumb: try to make the cells cuboid-like

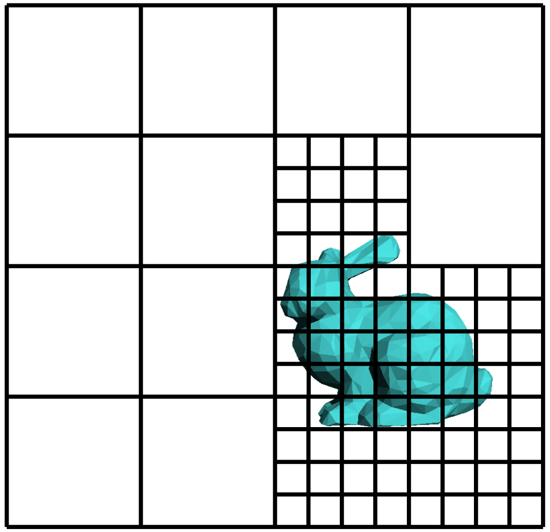
The Teapot in a Stadium Problem

- Problem: regular grids don't adapt well to large variations of local "densities" of the geometry



Recursive Grids

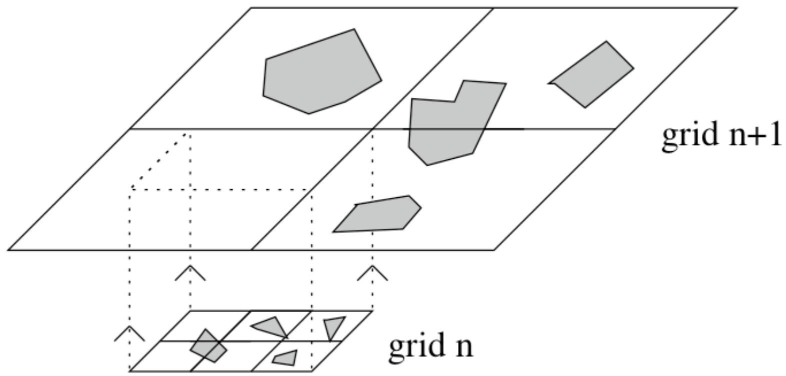
- Idea:
  - First, construct a coarse grid, with cells larger than rule-of-thumb suggests
  - Subdivide "dense" cells again by a finer grid
  - Stopping criterion: less than *n* objects/triangles in the cell, or maximum depth
- Additional Feature: subdivision "on demand", i.e.,
  - In the beginning, create only 1-2 levels
  - If any ray hits a cell that does not fulfill the stopping criteria, then subdivide cell by finer grid



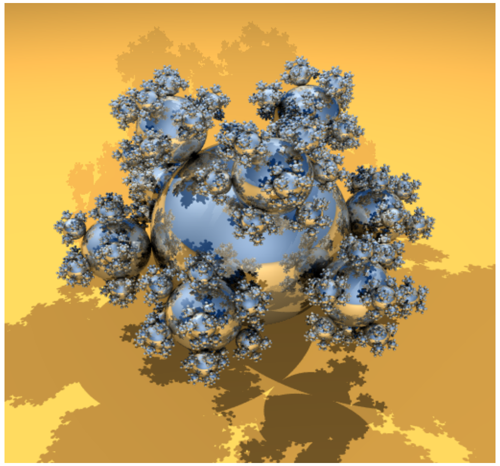
Nested Grids

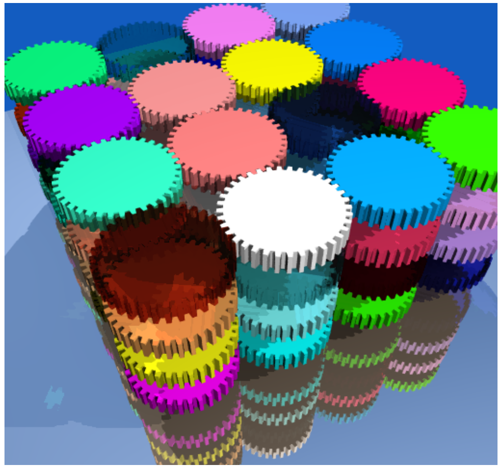
Hierarchical Uniform Grid (HUG) [1994]

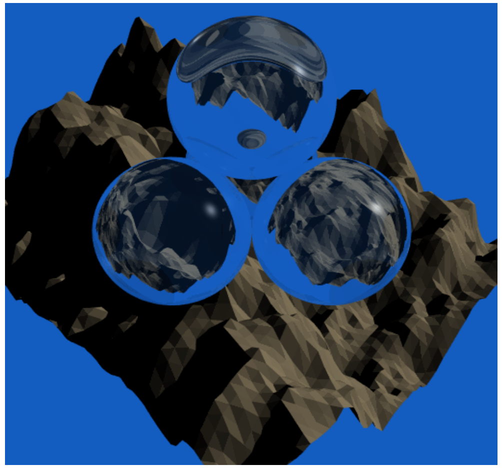
- Problem: if the variance among object sizes is very large, then the average object size is not a good cell size
- Idea:
  - Group objects by size → "size clusters"
  - Group objects within a size cluster by location → local size clusters
  - Construct grid for each local size cluster
  - Construct hierarchy on top of these elementary grids
- Example:



Construction Time of Different Grids



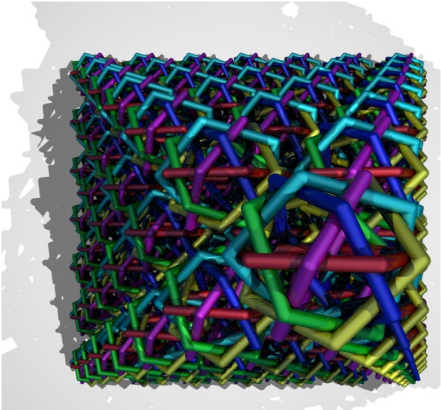
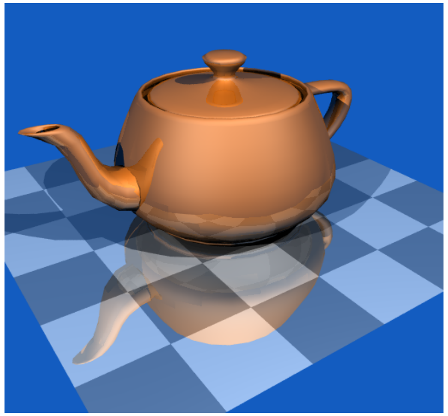
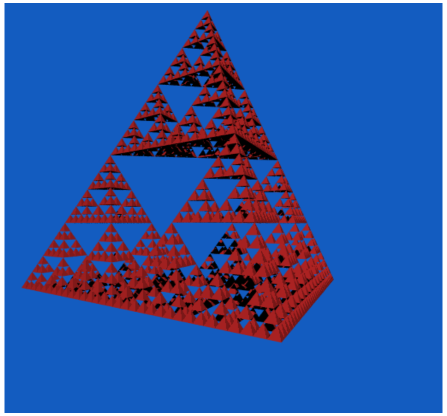
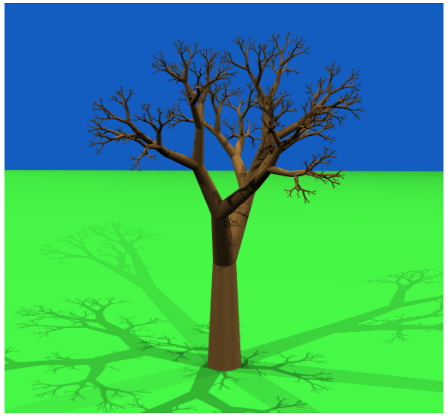




	balls	gears	mount
Uniform, $D = 1.0$	0.19	0.38	0.26
Uniform, $D = 20.0$	0.39	1.13	0.4
Recursive Grid	0.39	5.06	1.98
HUG	0.4	1.04	0.16

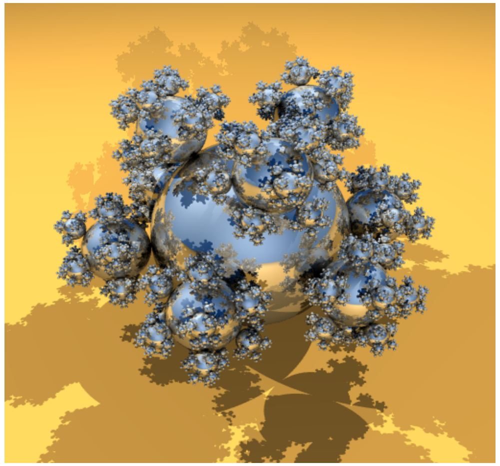
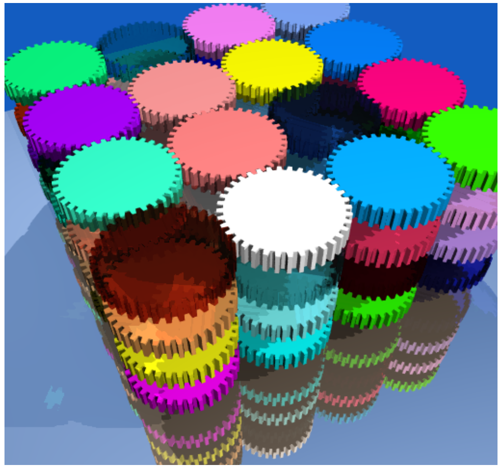
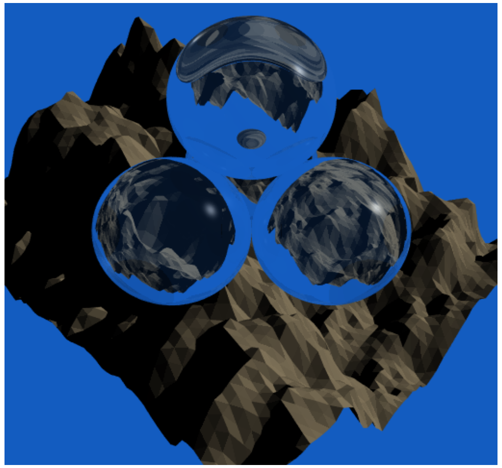
$$D = \frac{\#\text{ voxels}}{\#\text{ objects}}$$

Quelle: Vlastimil Havran, Ray Tracing News vol. 12 no. 1, June 1999, http://www.acm.org/tog/resources/RTNews/html

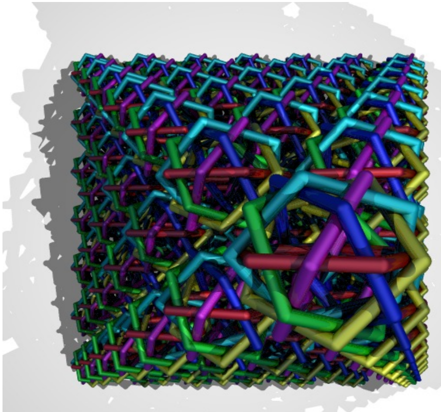
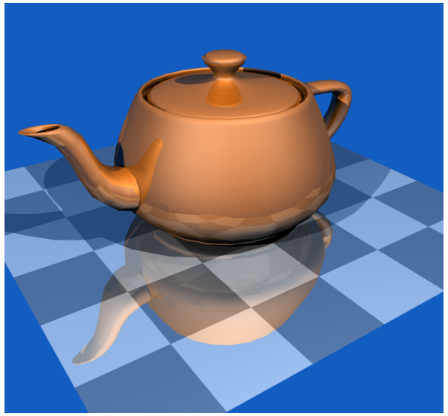
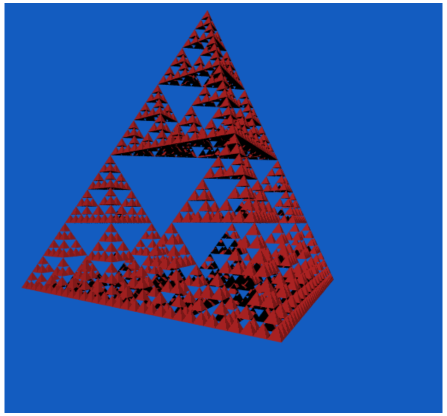
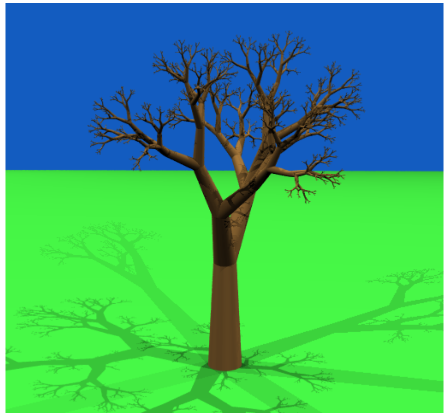
   

	rings	teapot	tetra	tree
Uniform, $D = 1.0$	0.35	0.3	0.13	0.22
Uniform, $D = 20.0$	0.98	0.65	0.34	0.33
Recursive Grid	0.39	1.55	0.47	0.28
HUG	0.45	0.53	0.24	0.48

Running Times of the Ray Tracing (sec)

	Balls	Gears	Mount
Uniform, $D = 1.0$	244.7	201.0	28.99
Uniform, $D = 20.0$	38.52	**192.3**	**25.15**
Recursive Grid	36.73	214.9	30.28
HUG	**34.0**	242.1	62.31

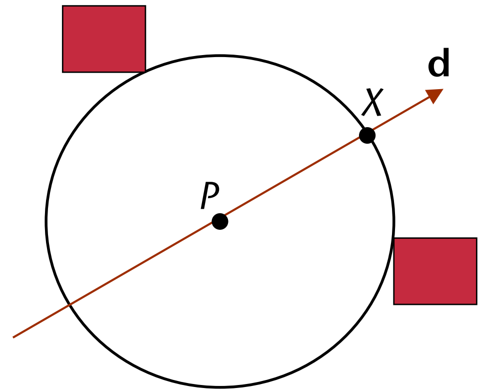
	Rings	Teapot	Tetra	Tree
Uniform, $D = 1.0$	129.8	28.68	5.54	1517.0
Uniform, $D = 20.0$	**83.7**	**18.6**	**3.86**	781.3
Rekursiv	113.9	22.67	7.23	33.91
HUG	116.3	25.61	7.22	33.48
Adaptive	167.7	43.04	8.71	**18.38**

Proximity Clouds

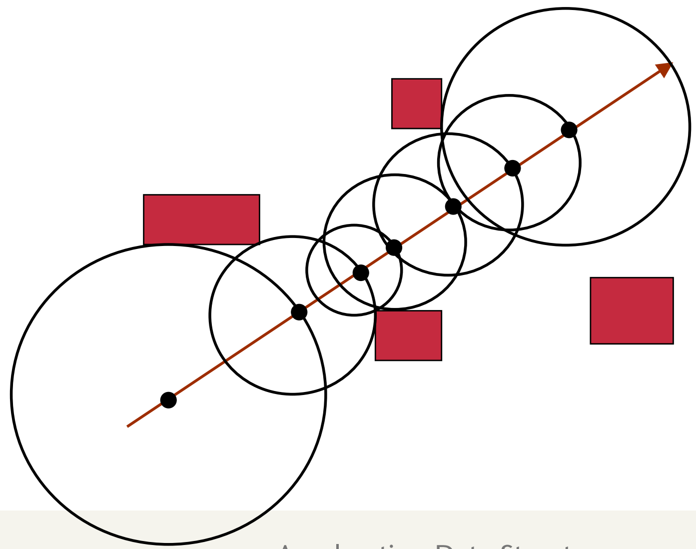
[1994]

- Thought experiment:
  - Assumption: we are sitting on the ray at point P and we know that there is no object within a ball of radius r around P
  - Then, we can jump directly to the point

$$X = P + \frac{r}{\|d\|}\mathbf{d}$$



  - Assumption: we know this "clearance" radius for each point in space
  - Then, we can jump through space from one point to its "clearance horizon" and so on …

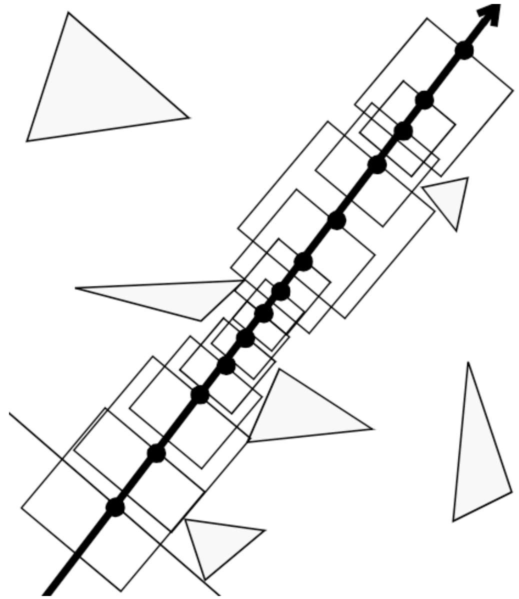


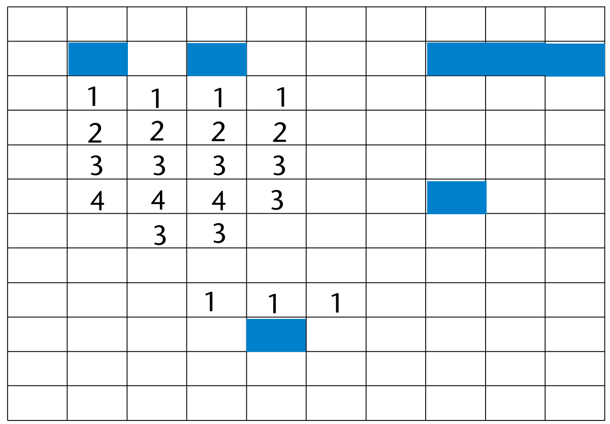
- The general idea is called empty space skipping
  - Comes in many different guises

- The idea works with any other metric, too
- Problem: we cannot store the clearance radius in *every* point in space
- Idea: discretize space by grid
  - For each grid cell, store the minimum clearance radius, i.e., the clearance radius that works in any direction (from any point within that cell)

➢ Such a data structure is called a **distance field**

- Example:





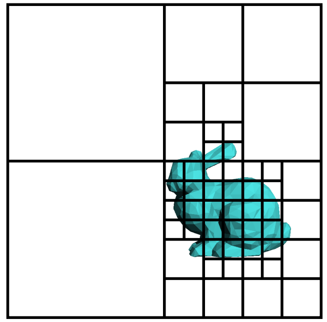
General Rules for Optimization

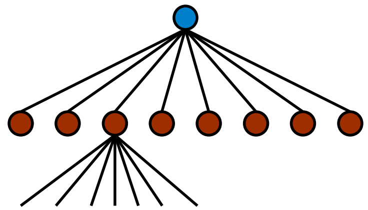
Optional

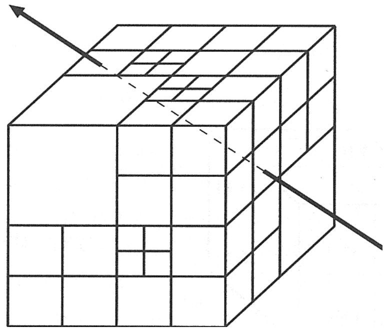
- "Premature Optimization is the Root of All Evil"   [Knuth]
  - *First*, implement your algorithm naïve and slow, *then* optimize!
  - After each optimization, do a before-after benchmark!
    - Sometimes/often, optimization turn out to perform worse
  - Only make small optimizations at a time!
  - Do a profiling before you optimize!
    - Often, your algorithm will spend 80% of the time in quite different places than you thought it does!
  - *First*, try to find a smarter algorithm,
    then do the "bit twiddling" optimizations!

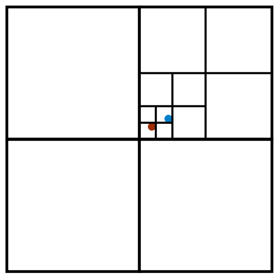
The Octree / Quadtree

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

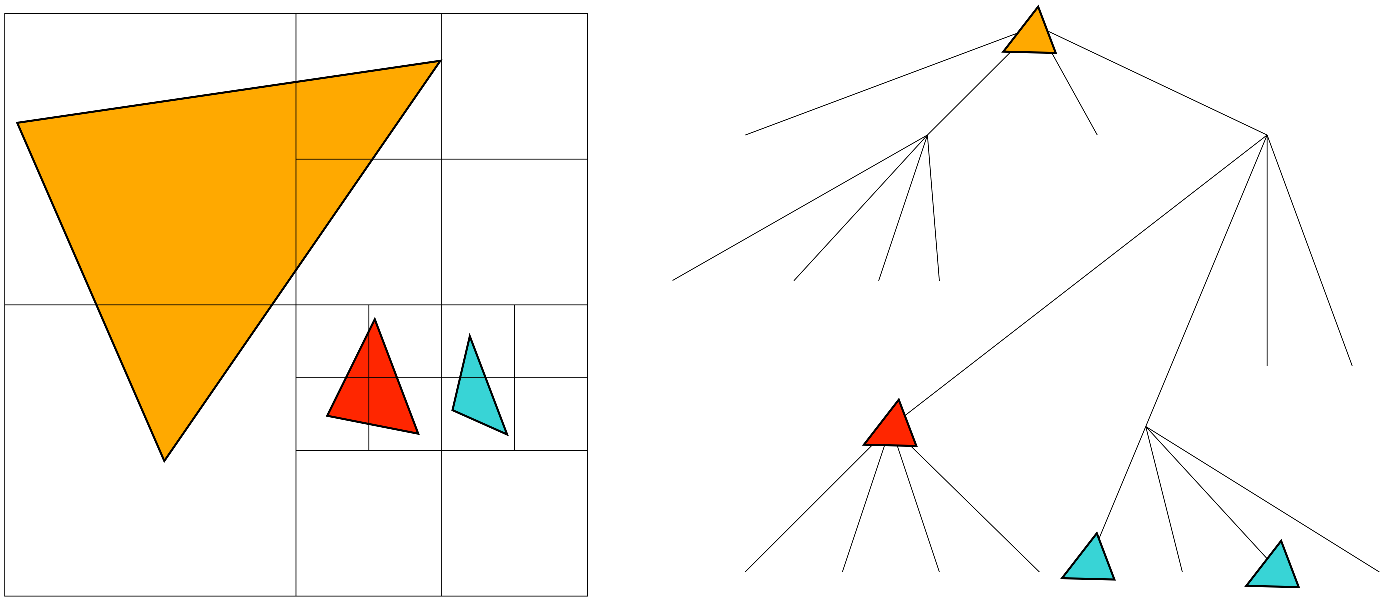








- What about large objects in octrees?
- Must be stored with inner nodes, or …
- In leaves only, but then they need to be stored in many nodes



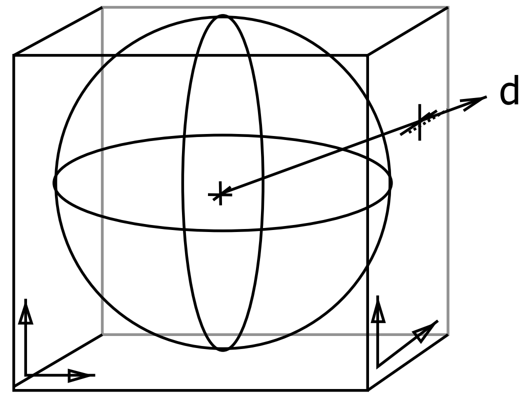
Octree/(Quadtree)

The 5D Octree for Rays

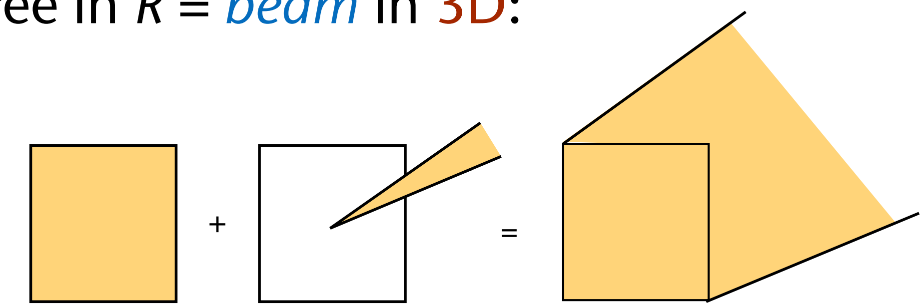
[Arvo u. Kirk 1987]

- What is a ray?
  - Point + direction = 5-dim. object
- Octree over a set of rays:
  - Construct bijective mapping between directions and the direction cube:

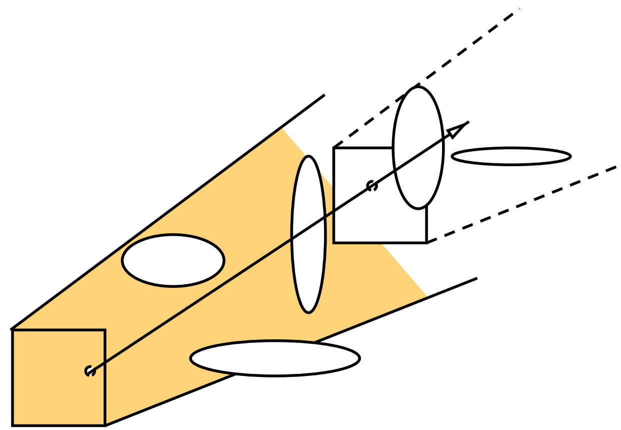
$$S^2 \leftrightarrow D := [-1, +1]^2 \times \{\pm x, \pm y, \pm z\}$$



  - All rays in the universe $U = [0, 1]^3$ are given by the set: $R = U \times D$
- A node in the 5D octree in R = *beam* in 3D:



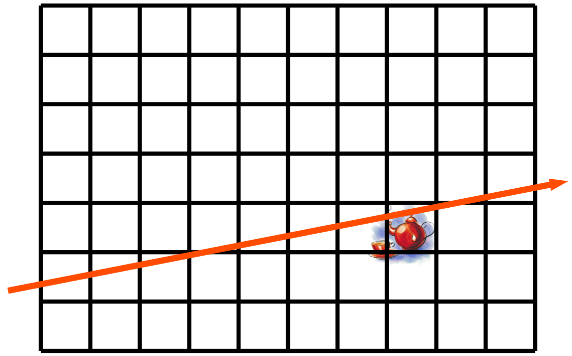
- Construction (6x):
  - Associate object with an octree node $\leftrightarrow$ object intersects the beam
  - Start with root = $U \times [-1, +1]^2$ and the set of all objects
  - Subdivide node (32 children), if
    - too many objects are associated with the current node, *and*
    - the cell is too large.
    - Associate all objects with one or more children
- The ray intersection test:
  - Map ray to 5D point
  - Find the leaf in the 5D octree
  - Intersect ray with its associated objects
- Optimizations …



Remarks

- The method basically pre-computes a complete, discretized visibility for the entire scene
  - I.e., what is visible from each point in space in each direction?
- Very expensive pre-computation, very inexpensive ray traversal
  - The effort is probably not balanced between pre-computation and run-time
- Very memory intensive, even with *lazy evaluation*
- Is used rarely in practice ...

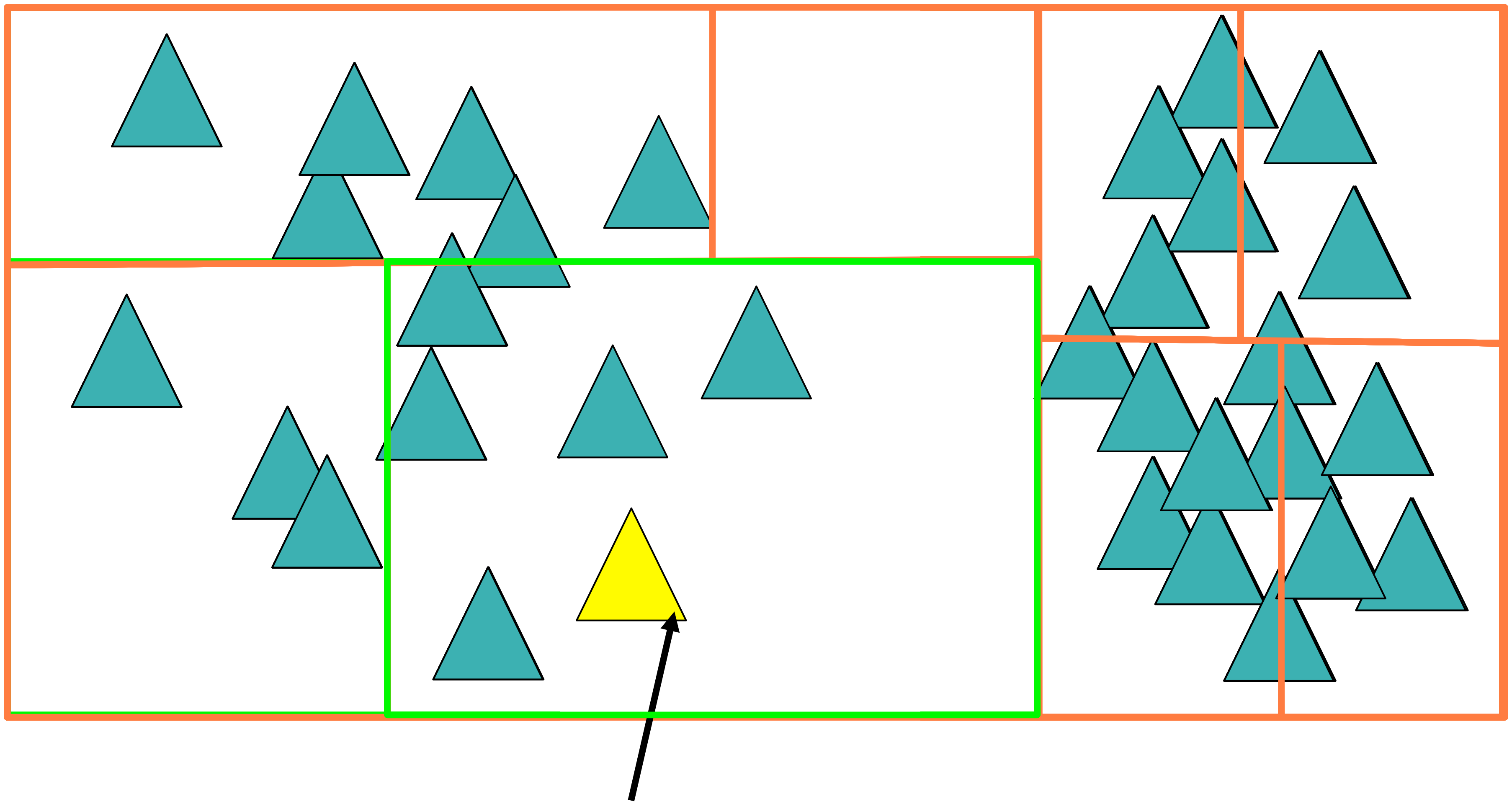
kD-Trees



- Problem with grid: "teapot in a stadium"
- Problem with octrees:
  - Very inflexible subdivision scheme
    (always at the center of the father cell)
  - But subdivision in all directions is not always necessary
- Solution: hierarchical subdivision that can adapt more flexibly to the distribution of the geometry
- Idea: subdivide space recursively by just one plane:
  - Subdivide given cell with a plane
  - Choose plane perpendicular to one coordinate axis
  - Free choices: the axis (x, y, z) & place along that axis
- "Best known method" [Siggraph Course 2006]
  - ... at least for static scenes

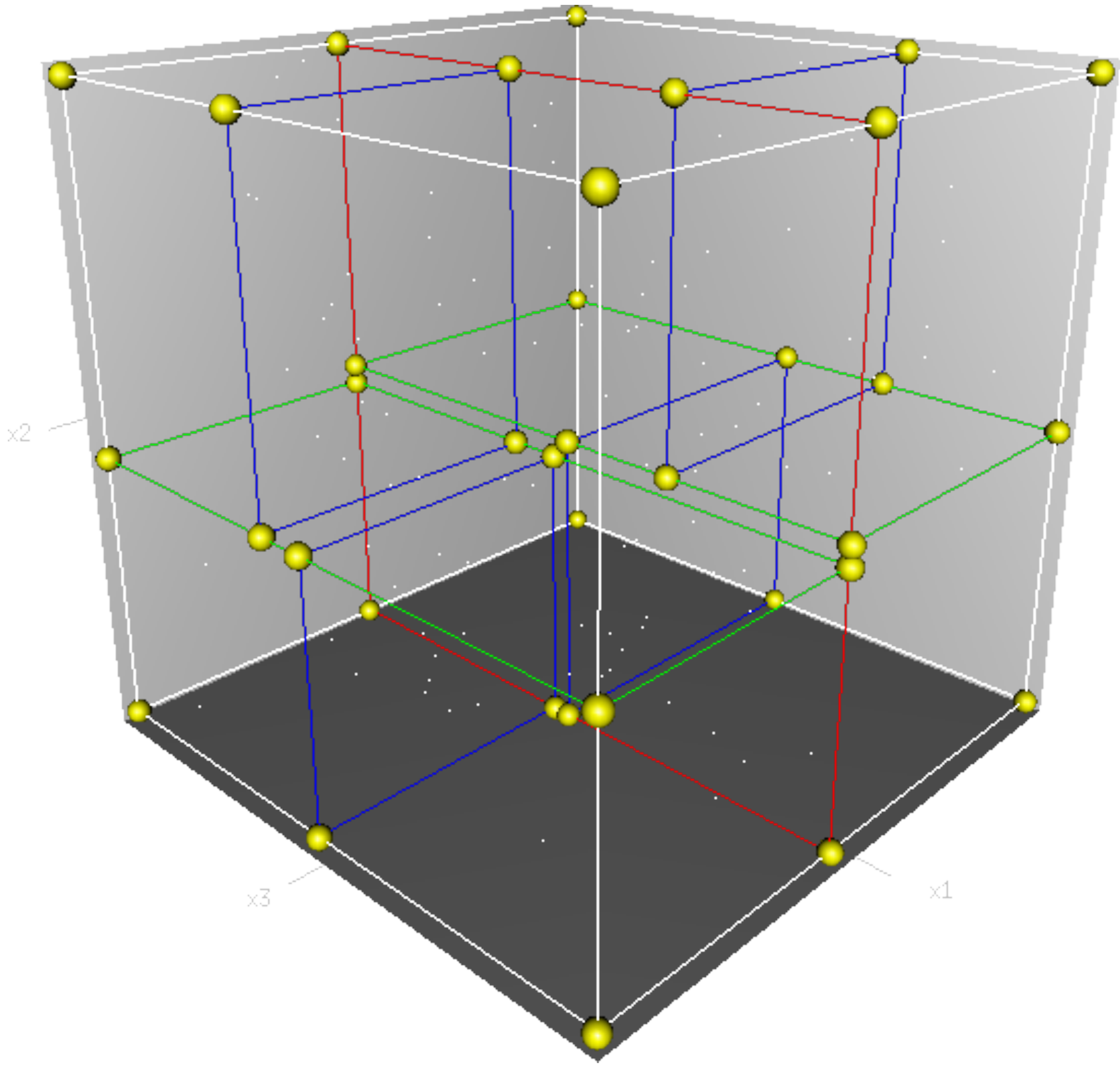
- Informal definition:
  - A kd-tree is a binary tree, where
    - Leaves contain single objects (polygons) or a list of objects;
    - Inner nodes store a splitting plane (perpendicular to an axis) and child pointer(s)
  - Stopping criterion:
    - Maximal depth, number of objects, some cost function, …
- Advantages:
  - Adaptive
  - Compact nodes (just 8 bytes per node)
  - Simple and very fast ray traversal
- Small disadvantage:
  - Polygons must be stored several times in the kd-tree

Example



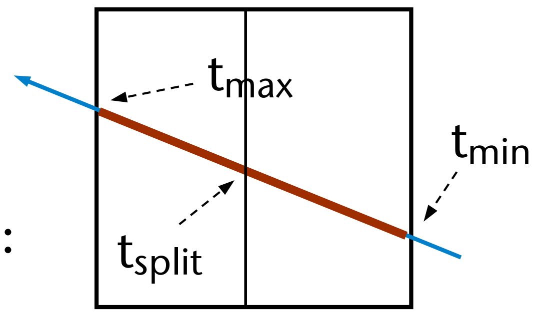
[Slide courtesy Martin Eisemann]

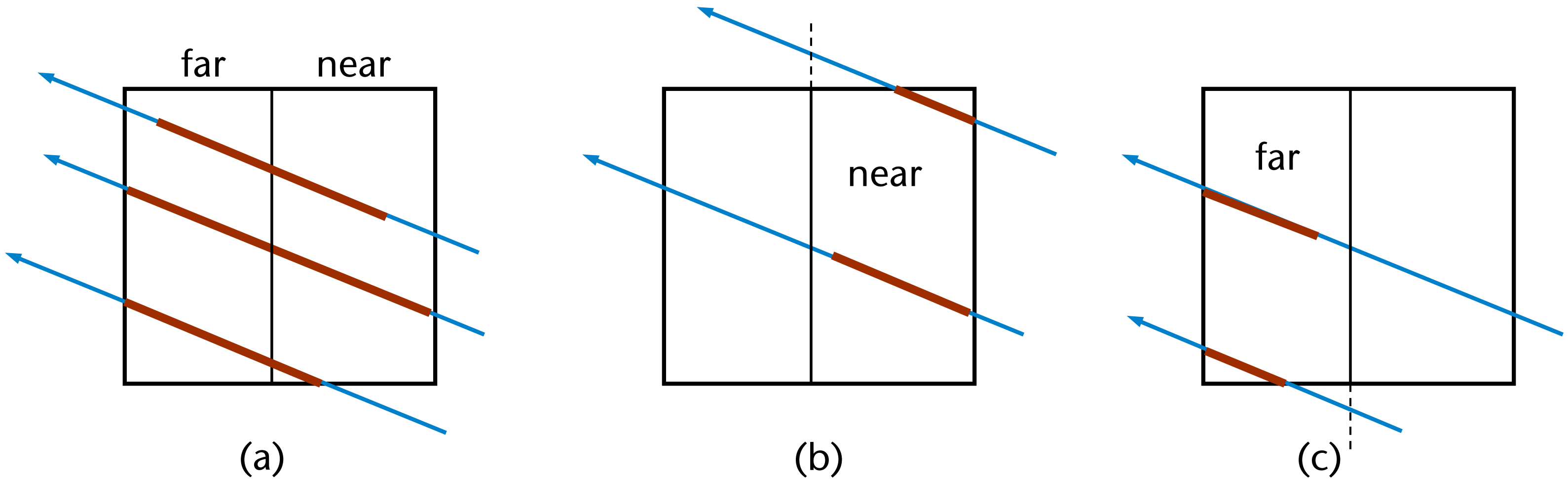
3D Visualization



Ray-Traversal through a Kd-Tree

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





Pseudo-Code für die Traversierung

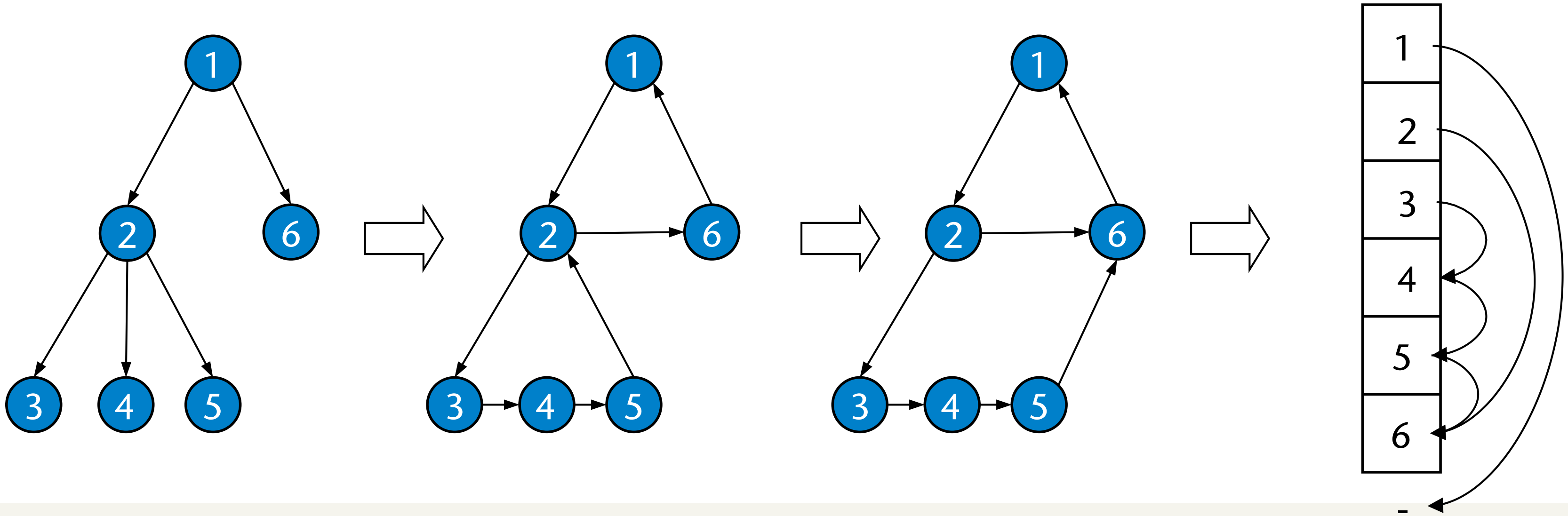
```
traverse( Ray r, Node n, float t_min, float t_max ):
  if n is leaf:
    intersect r with each primitive in object list,
        discarding those farther away than t_max
    return object with closest intersection point (if any)

  t_split = signed distance along r to splitting plane of n
  near = child of n containing origin of r     // test signs in r.d
  far  = the "other" child of n
  if t_split > t_max:
    return traverse( r, near, t_min, t_max )   // (b)
  else if t_split < t_min:
    return traverse( r, far, t_min, t_max )    // (c)
  else:                                        // (a)
    t_hit = traverse( r, near, t_min, t_split )
    if t_hit < t_split:
      return t_hit                             // early ray terminat'n
    return traverse( r, far, t_split, t_max )
```

Optimized Traversal

Optional

- Observation:
  - 90% of all rays are shadow rays
  - Any hit is sufficient
- Consequence:
  - The order the children of the kD-tree are visited does not matter (in the case of shadow rays) → perform pure DFS
- Idea: replace the recursion by an iteration
- Transform the tree to achieve that:



Optional

- Algorithm:

```
traverse( Ray ray, Node root ):
  stopNode = root.skipNode
  node = root
  while node < stopNode:
    if intersection between ray and node:
      if node has primitives:
        if intersection between primitive and ray:
          return intersection
      node ++
    else:
      node = node.skipNode
  return "no intersection"
```

Diplomarbeit …

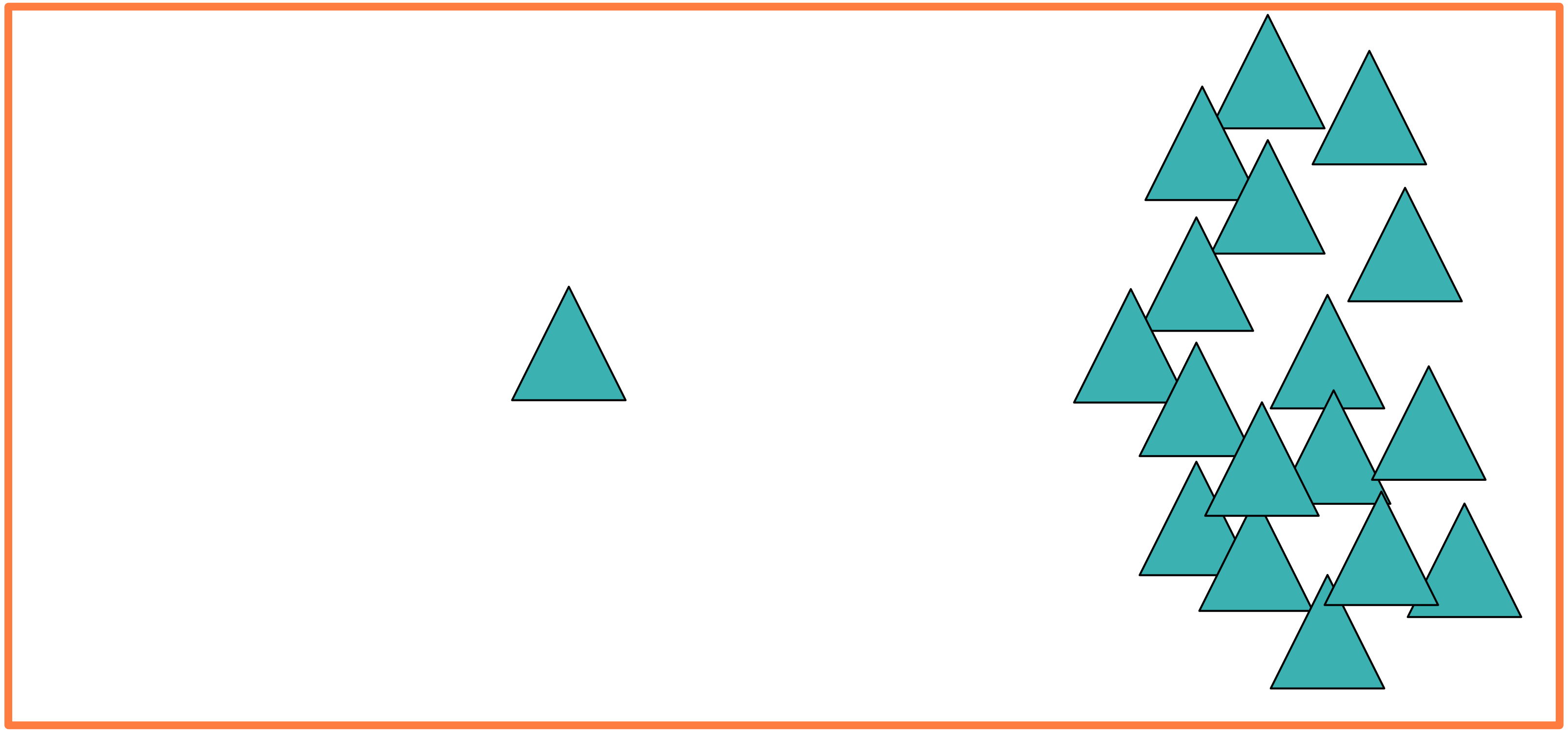
Construction of a kD-Tree

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

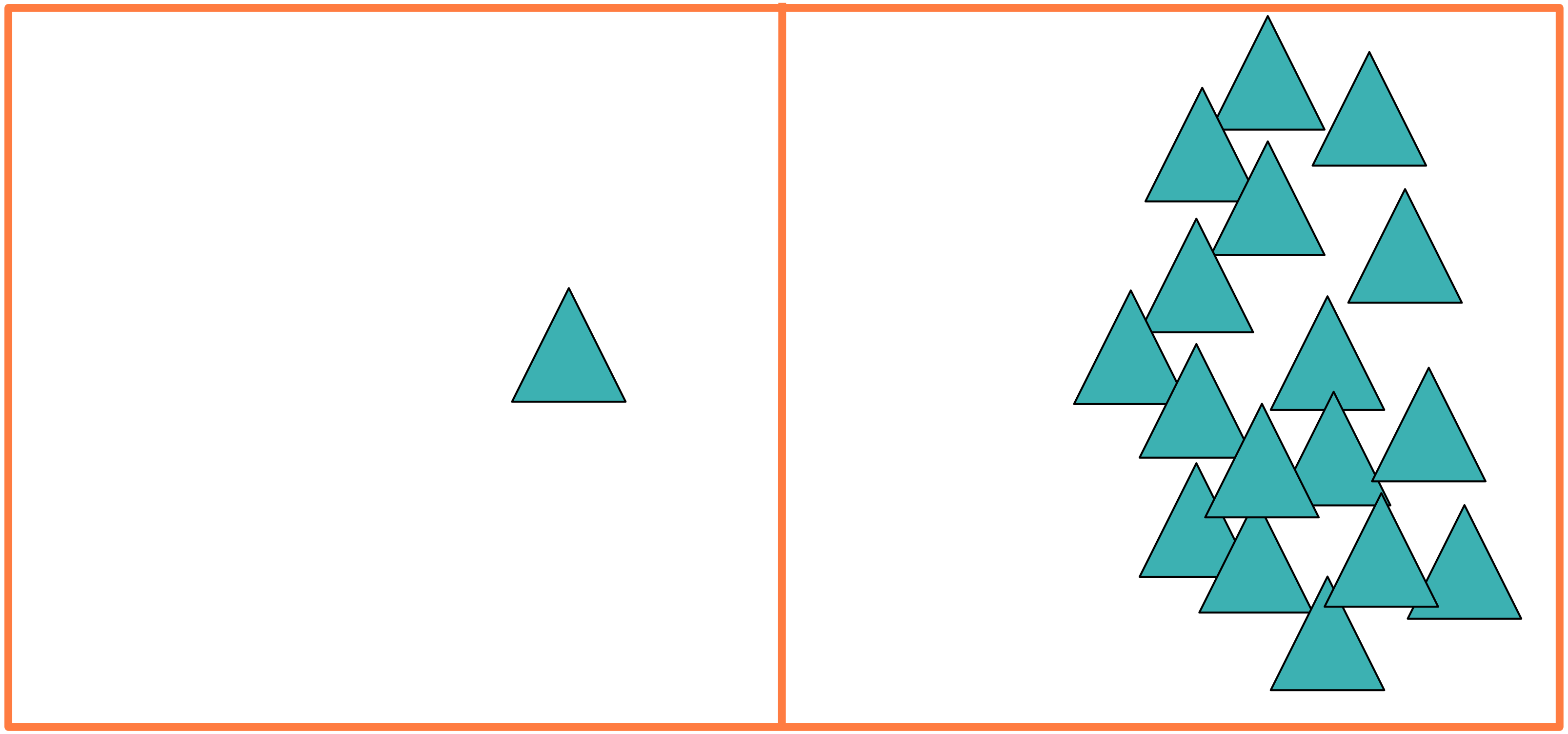
On Selecting a Splitting-Plane

- **Naïve Selection of the Splitting-Plane:**
  - Splitting-Axis:
    - Round Robin (x, y, z, x, …)
    - Split along the longest axis
  - Split-Position:
    - Middle of the cell
    - Median of the geometry
- **Better: Utilize a Cost Function**
  - We should choose a splitting plane such that the expected costs of a ray test are distributed equally among both subtrees
  - Try all 3 axes
  - Search for the minimum along each axis
  - Choose the axis and split-position with the smallest minimum

Motivation der Kostenfunktion

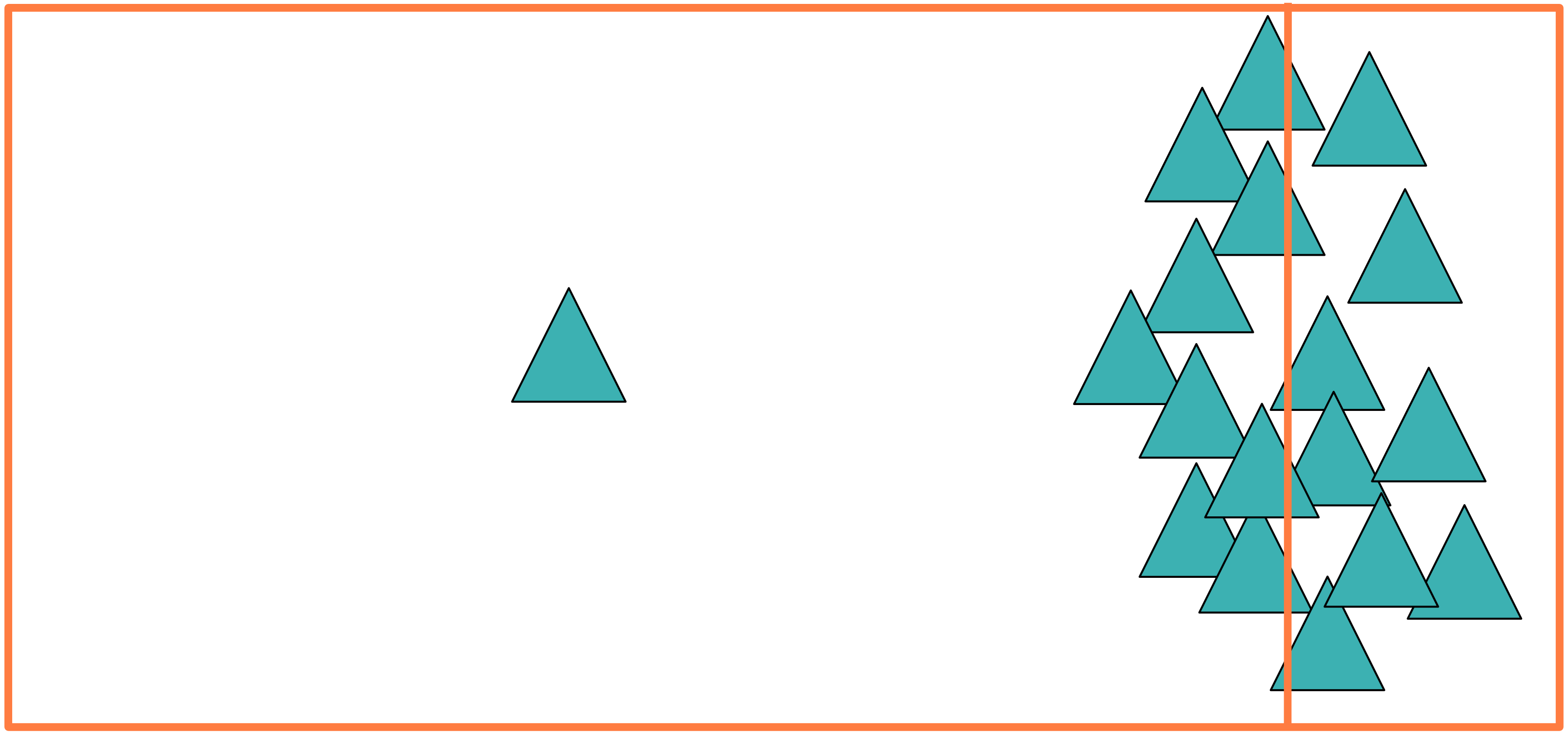


- Split in the middle:



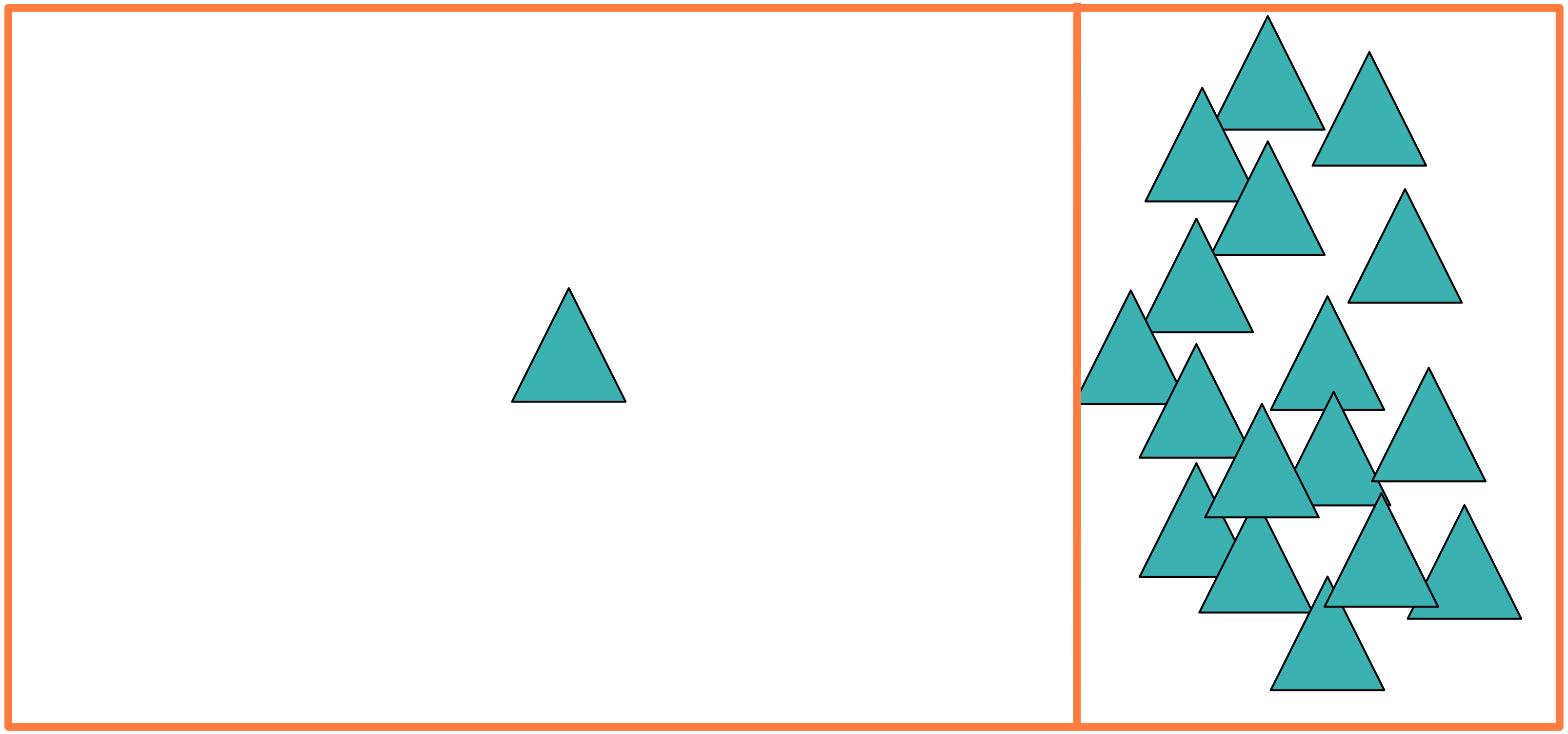
- The probability of a ray hitting the left or the right child is equal
- But, he expected costs for handling the left or the right child are very different!

- Split along the geometry median:



  - The computational efforts for left or right child are equal
  - But not the probability of a hit

- Cost-optimized heuristic:

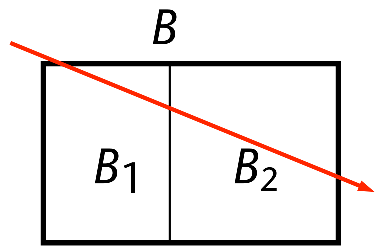


  - The total expected costs are approximately similar
    - Probability for a left hit is higher, but on the other hand there are less polygons in the left child

The Surface Area Heuristic (SAH)

[1990]

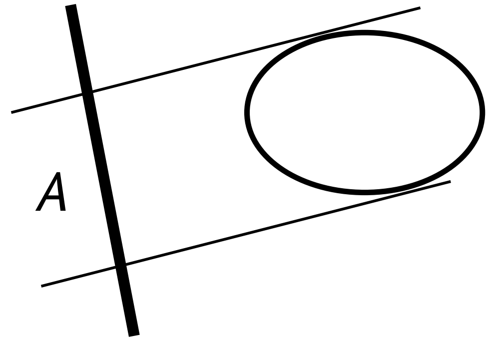
- Question: How to measure the costs of a given kD-Tree?
- Expected costs of a ray test:
  - Assume, we have reached cell B during the ray traversal
  - Cell B has children B_1, B_2
  - Expected costs = expected traversal time =

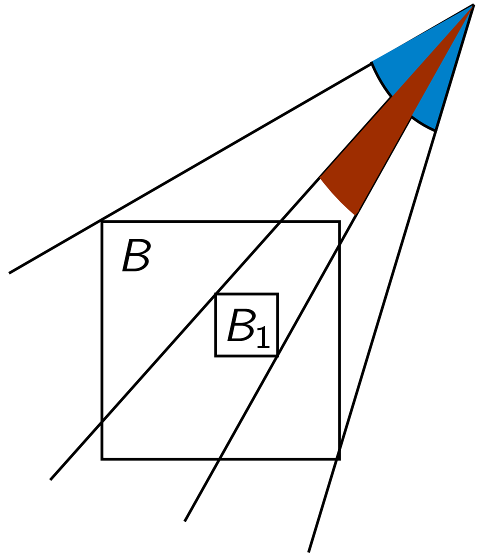


$$
\begin{aligned}
C(B) = &\ \text{Prob}[\text{intersection with } B_1] \cdot C(B_1) \\
&+ \text{Prob}[\text{intersection with } B_2] \cdot C(B_2)
\end{aligned}
$$

- Assumptions in the following:
  - All rays have the same, far away origin
  - All rays hit the root-BV of the kD-tree

- Number of rays in a given direction that hit an object is proportional to its projected area
- Total "number" of rays, summed over all possible directions $= 4\pi \bar{A}$ where $\bar{A}$ = sum of all projected areas, again summed over all possible directions
- Crofton's theorem (integral geometry): For convex objects, $\bar{A} = \frac{1}{4} S$, where S = area of surface of object
- Therefore, the probability is





$$\text{Prob}[\text{ intersection with } B_1 \mid \text{intersection with } B\ ] = \frac{\text{Area}(B_1)}{\text{Area}(B)}$$

- Solution of the "recursive" equation:
  - How to compute $C(B_1)$ and $C(B_2)$ respectively?
  - A simple heuristic: set

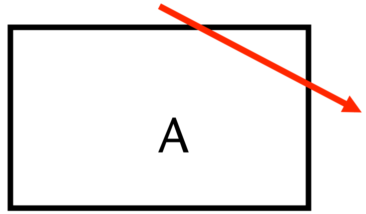
$$C(B_i) \approx \# \text{ triangles in } B_i$$

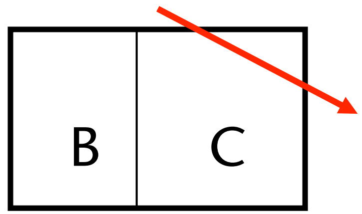
- The complete Surface Area Heuristic :
  minimize the following function when distributing the set of polygons

$$C(B) = \text{Area}(B_1) \cdot N(B_1) + \text{Area}(B_2) \cdot N(B_2)$$

A Stopping Criterion

- How to decide whether or not a split is worth-while?
- Consider the costs of a ray intersection test in both cases:
  - No split → costs = $t_p N$
  - Split → costs = $t_s + t_p(p_B N_B + p_C N_C)$





where t_p = time for 1 ray-primitive test

t_s = time for 1 intersection test of ray with splitting plane of the kD-tree node

p_B =probability, that the ray hits cell B

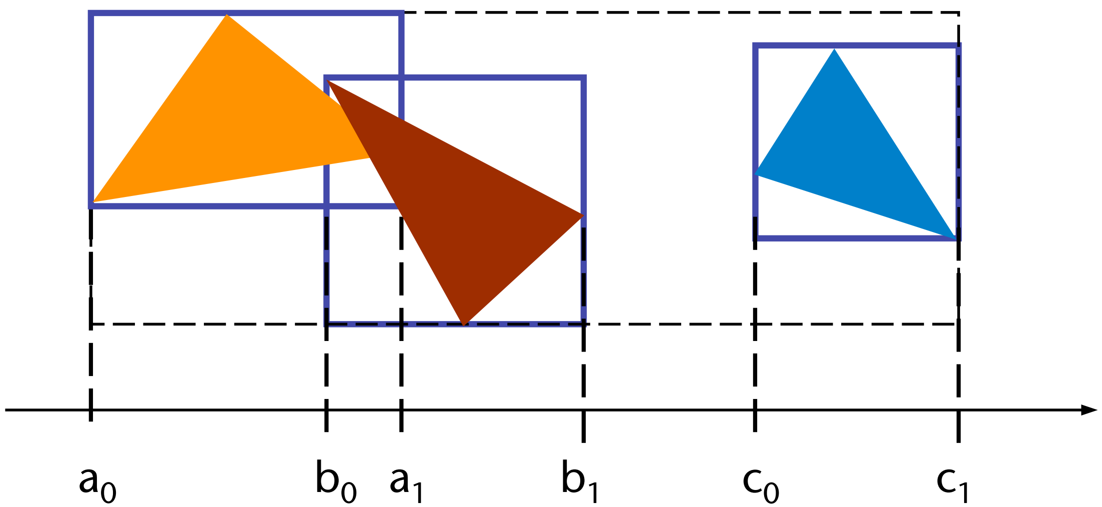
N = number of primitives

- In practice, we can make the following simplifying assumptions :
  - t_p = const for all primitives
  - $t_p : t_s = 80 : 1$ (determined by experiment)

Remarks

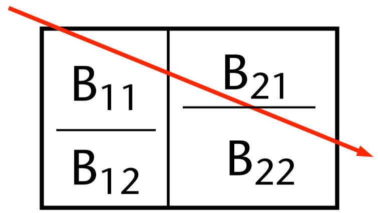
Optional

- It suffices to evaluate the cost function (SAH) only at a finite set of points
  - The points are the borders of the bounding boxes of the triangles
  - In-between, the value of the SAH must be worse
- Sort all the Bboxes by their boundary coordinates, evaluate the SAH at all these points (*plane sweep*)
- Sorting allows *golden section search* and, thus, a faster evaluation



- **Warning**: for other queries (e.g. points, boxes,…) the surface area is not necessarily a good measure for the probability!

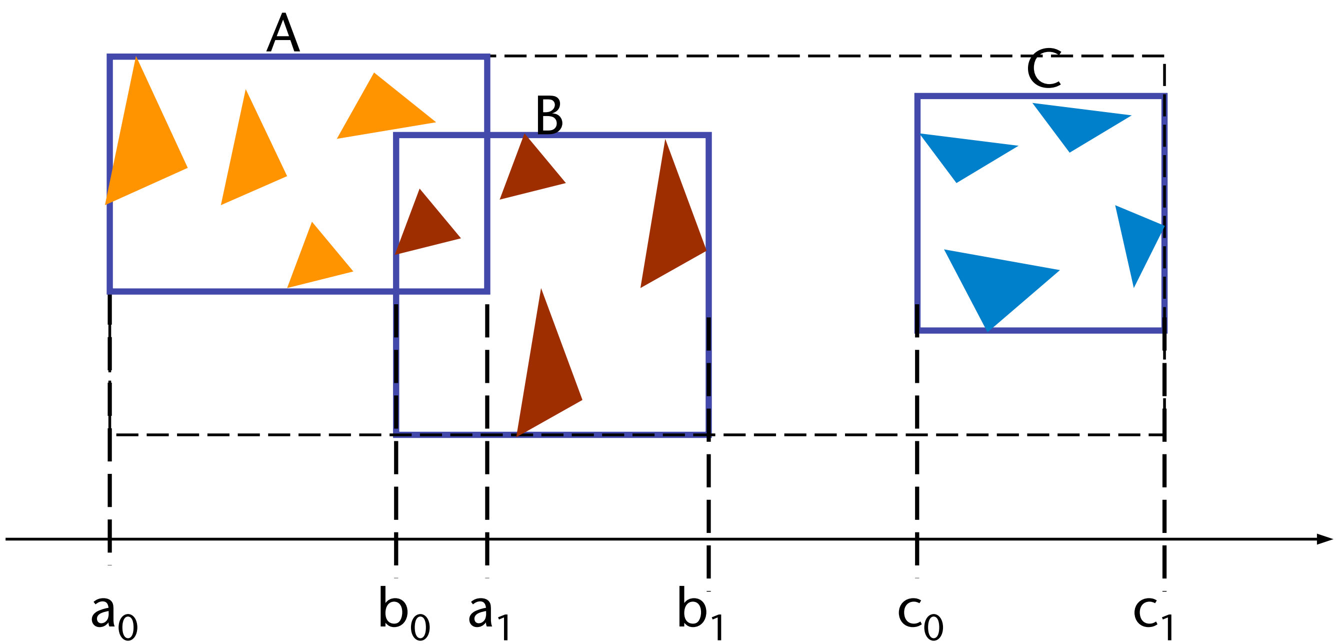
- A straight-forward, better (?) heuristic: make a „look-ahead"



$$
\begin{aligned}
C(B) = & P[\text{Schnitt mit } B_1] \cdot C(B_1) \\
& + P[\text{Schnitt mit } B_2] \cdot C(B_2) \\
& = P[B_1] \cdot (P[B_{11}] C(B_{11}) + P[B_{12}] C(B_{12})) \\
& + P[B_2] \cdot (P[B_{21}] C(B_{21}) + P[B_{22}] C(B_{22})) \\
& \quad \ldots
\end{aligned}
$$

Diplomarbeit …

- If the number of polygons is very large (> 500,000, say) → only try to find the approximate minimum [Havran et al., 2006]:
  - Sort polygons into "buckets"
  - Evaluate SAH only at the bucket borders



Better kd-Trees for Raytracing

Optional

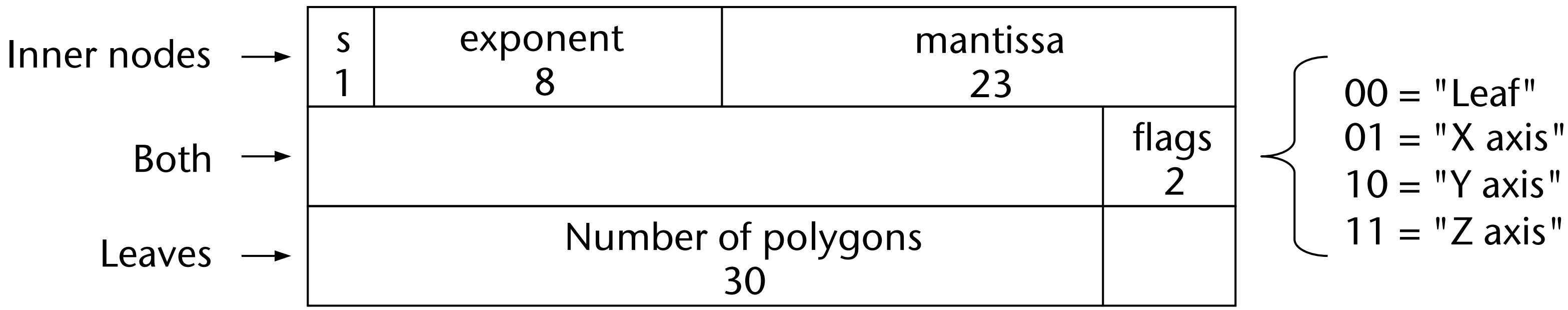
[2005]

- Before applying SAH, test whether an empty cell can be split off that is "large enough" ; if yes, do that, no SAH-based splitting
- Additional stopping criterion:
  - If volume of cell is too small, then no further splitting
  - Criterion for "too small" (e.g.): $\text{Vol(cell)} < \varepsilon \cdot \text{Vol(root)}$
  - Reason: such cells probably won't get hit anyway
  - Saves memory (lots) without sacrificing performance
- For architectural scenes:
  - If there is a splitting plane that is covered completely by polygons, then use it and put all those polygons in the smaller of the two children cells
  - Reason: that way, cells adapt to the rooms of the buildings (s.a. *portal culling*)

Storage of a kD-Tree

- The data needed per node:
  - One flag, whether the node is an inner node or a leaf
  - If inner node:
    - Split-Axis (uint),
    - Split-position (float),
    - 2 pointers to children
  - If leaf:
    - Number of primitives (uint)
    - The list of primitives (pointer)
- Naïve implementation: 16 Bytes + 3 Bits — very cache-inefficient
- Optimized implementation:
  - 8 Bytes per node (!)
  - Yields a speedup of 20% (some have reported even a factor of 10!)

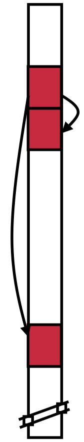
- Idea of optimized storage: Overlay the data
- Assemble all flags in 2 bits
- Overlay flags, split-position, and number of primitives




```
union
{
  unsigned int m_flags;     // both
  float m_split;            // inner node
  unsigned int m_nPrims;    // leaf
};
```

Optional

- Für innere Knoten: nur 1 Zeiger auf Kinder
  - Verwalte eigenes Array von kd-Knoten (nicht **`malloc()`** oder **`new`**)
  - Speichere beide Kinder in aufeinanderfolgende Array-Zellen; oder
  - speichere eines der Kinder direkt hinter dem Vater.
- Überlagere Zeiger auf Kinder mit Zeiger auf Primitive



- Zusammen:

```
class KdNode
{
private:
   union {
      unsigned int m_flags;       // both
      float m_split;              // inner node
      unsigned int m_nPrims;      // leaf
   };
   union {
      unsigned int m_rightChild; // inner node
      Primitive * m_onePrim;      // leaf
      Primitive ** m_primitives; // leaf
```

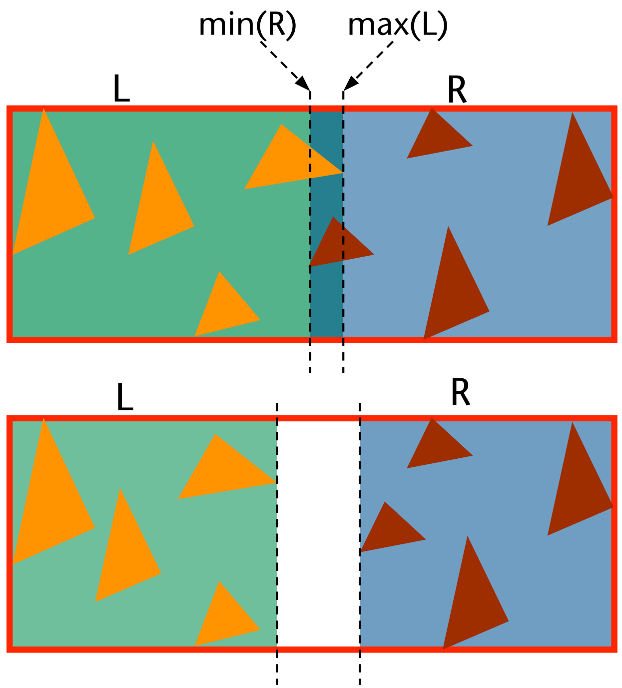
Falls m_nPrims == 1

Falls m_nPrims > 1

- Achtung: Zugriff auf Instanzvariablen natürlich nur noch über Kd-Node-Methoden!
  - Z.B.: beim Schreiben von m_split muß man darauf achten, daß danach (nochmals) m_flags geschrieben wird (ggf. mit dem ursprünglichen Wert)!
  - Beim Schreiben/Lesen von m_nPrims muß ein Shift durchgeführt werden!

Spatial KD-Trees (SKD-Tree) [1987/2002/2006]

- A variant of the kD-Tree
- Other names: BoxTree, "bounding interval hierarchy" (BIH)
- Difference to the regular kd-tree:
  - 2 parallel splitting planes per node
  - Alternative: the 2 splitting planes can be oriented differently
- Advantage: "*straddling*" polygons need not be stored in both subtrees
  - With regular kD-trees, there are $2\text{-}3 \cdot N$ more pointers to triangles than there are triangles (N), N = number of triangles in the scene



- Disadvantage: Overlapping child boxes → the traversal can not stop as soon as a hit in the "near" subtree has been found