Bremen

CG
VR

Advanced Computer Graphics
Acceleration Data Structures (for Raytracing et al.)

G. Zachmann
University of Bremen, Germany
cgvr.informatik.uni-bremen.de

The Costs of Ray-Tracing

cost ≈ height * width *

num primitives *

intersection cost *

size of recursive ray tree *

num shadow rays *

num supersamples *

num glossy rays *

num temporal samples *

num focal samples *

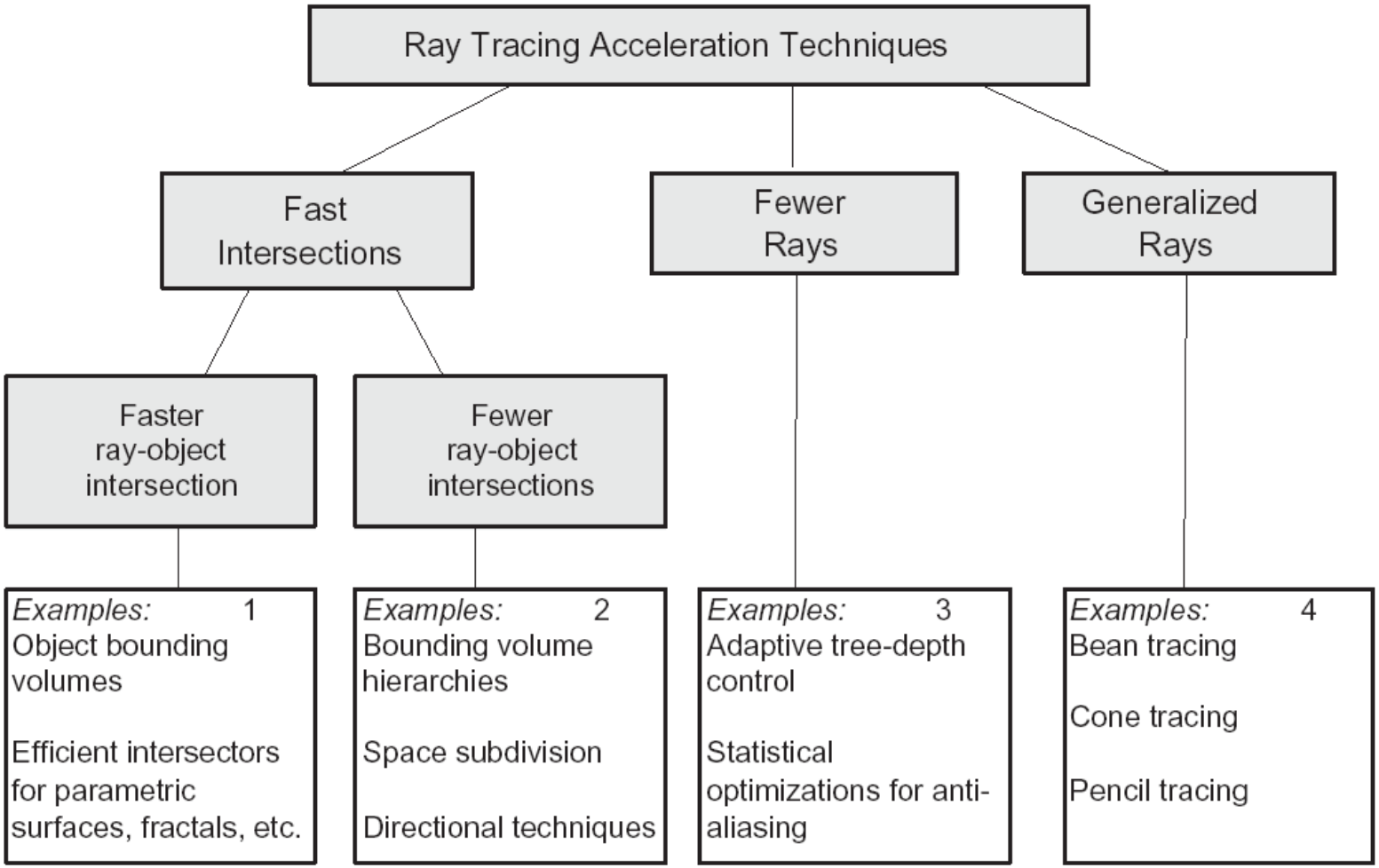
. . .

Can we decrease that?

"Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast."

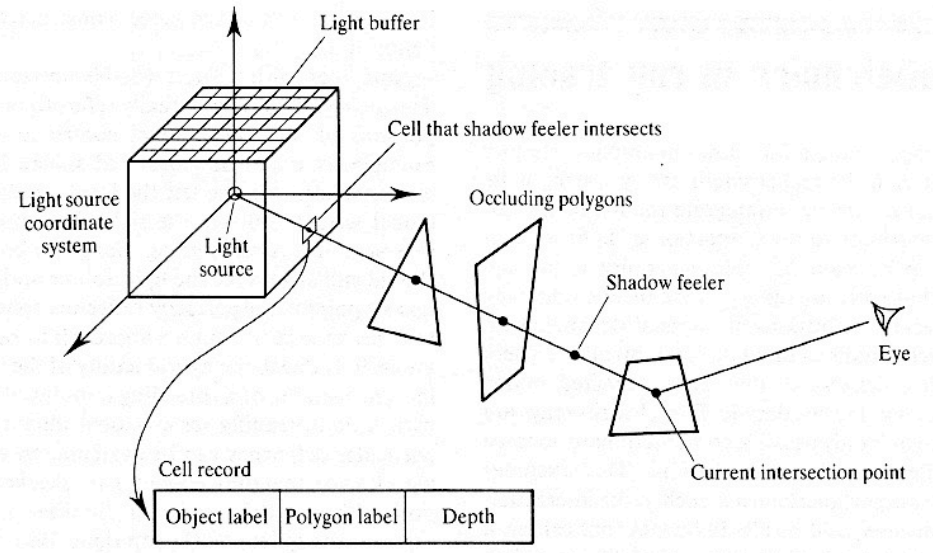
[David Luebke, Nvidia]

A Taxonomy of Acceleration Techniques



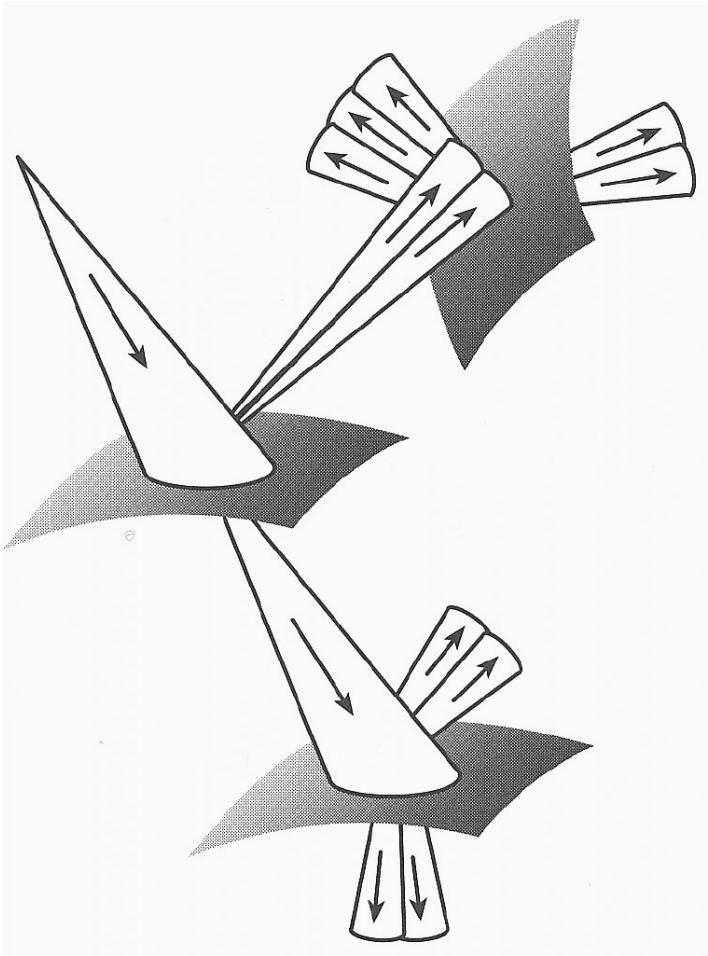
The Light Buffer

- Observation: when tracing shadow rays, it is sufficient to find any intersection with an opaque object
- Idea: for each light source, and for each direction, store a list of polygons lying in that direction when "looking" from the light source
  - The data structure of the light buffer: the "direction cube"
  - Construct either during preprocessing (by scan conversion onto the cube's sides), or construct "on demand" (i.e., insert occluder whenever found one)

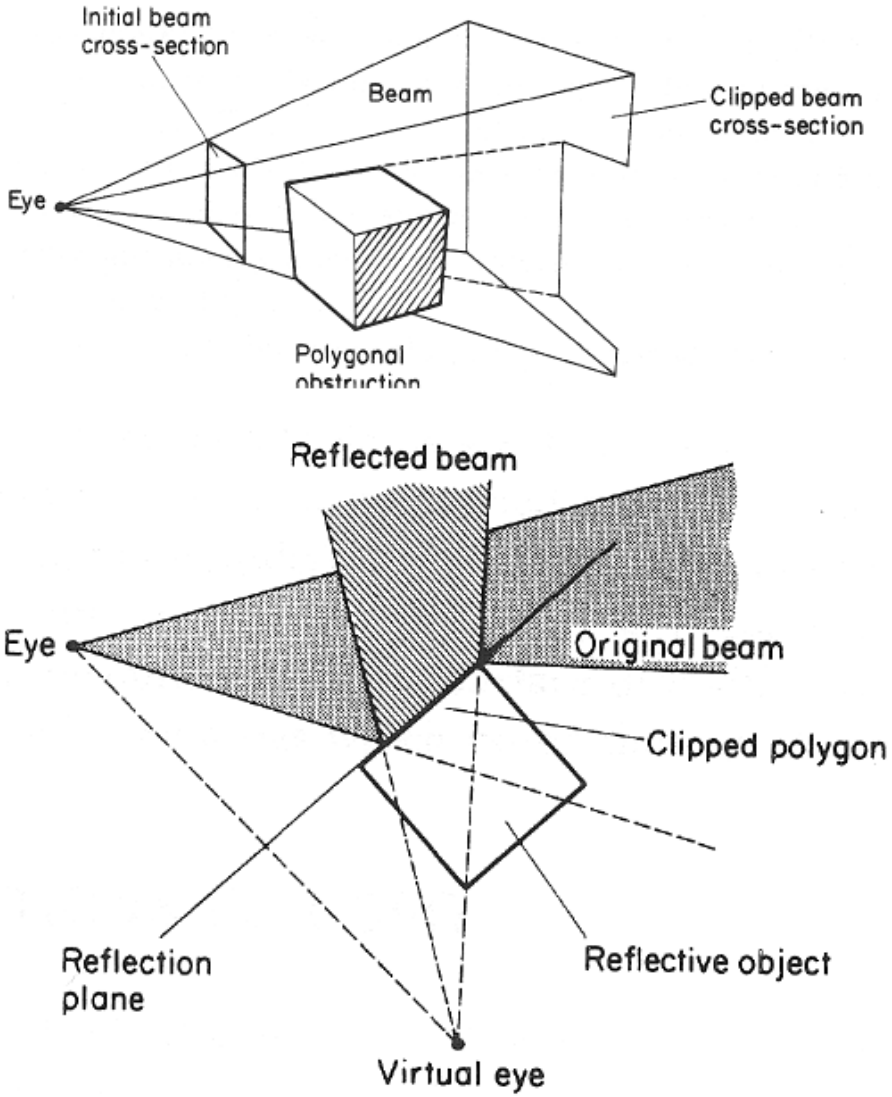


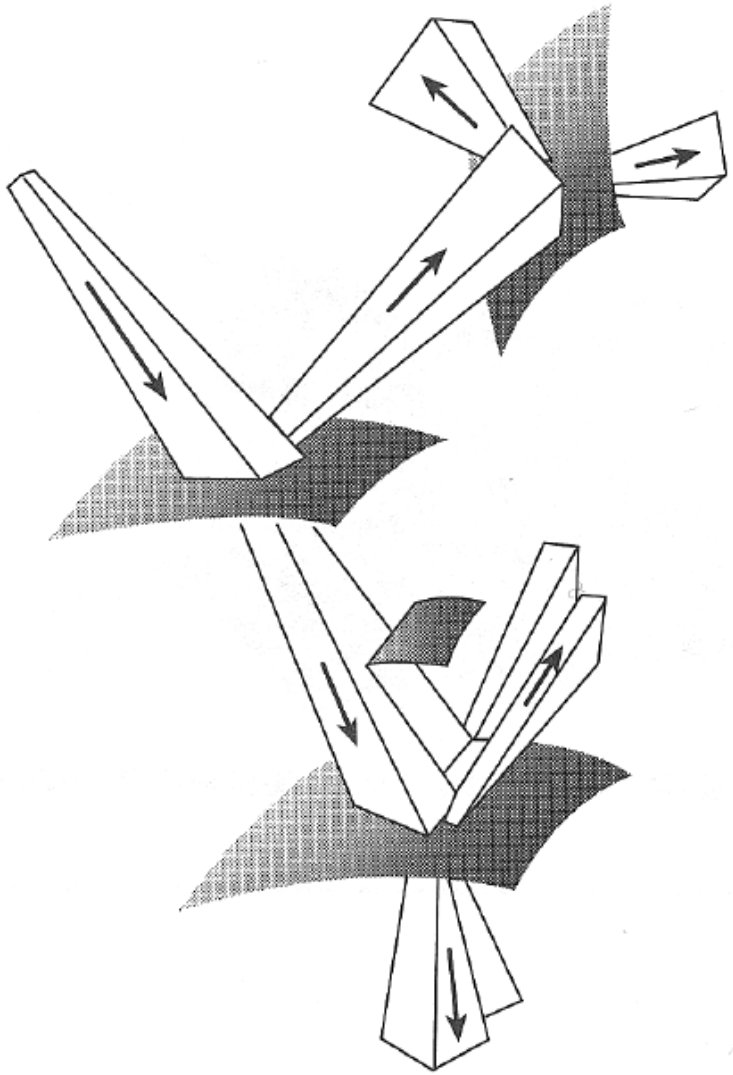
Beam and Cone Tracing

- The general idea: try to accelerate by shooting several or "thick" rays at once
- Beam Tracing:
  - Represent a "thick" ray by a pyramid
  - At the surfaces of polygons, create new beams
- Cone Tracing:
  - Approximate a thick ray by a cone
  - Whenever necessary, split into smaller cones
- Problems:
  - What is a good approximation?
  - How to compute the intersection of beams/cones with polygons?
- Conclusion (at the time): too expensive!



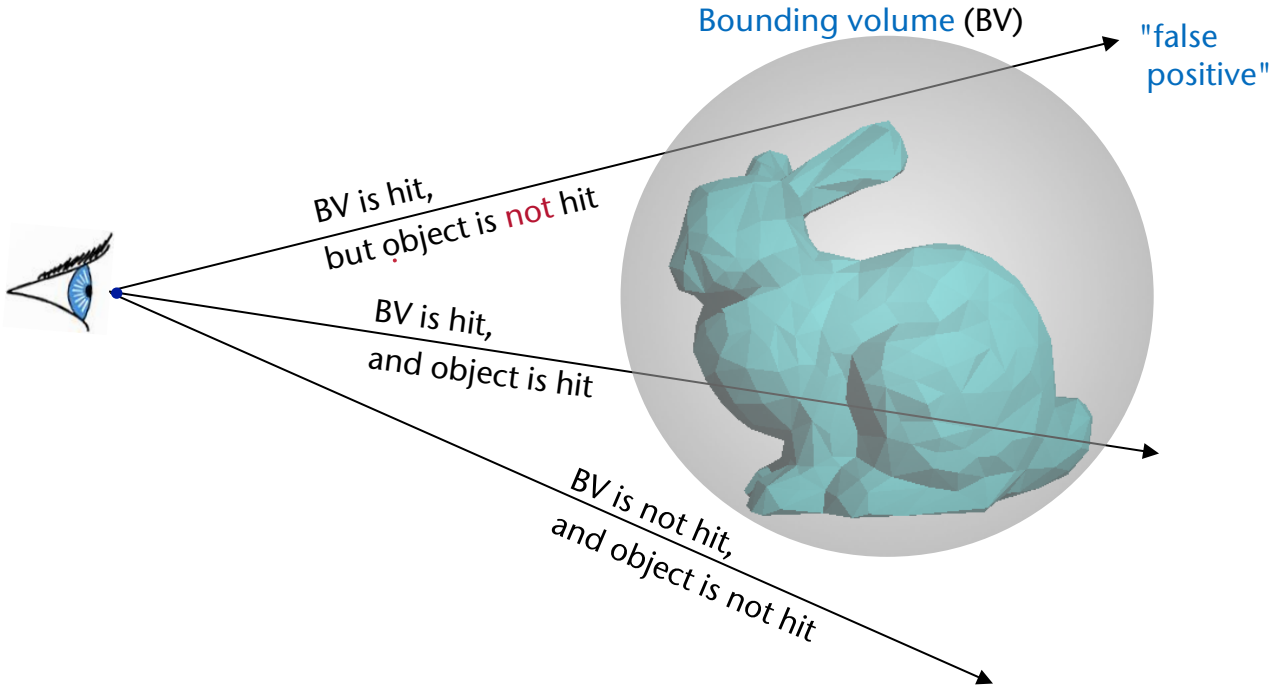
Beam Tracing





Bounding Volumes (BVs)

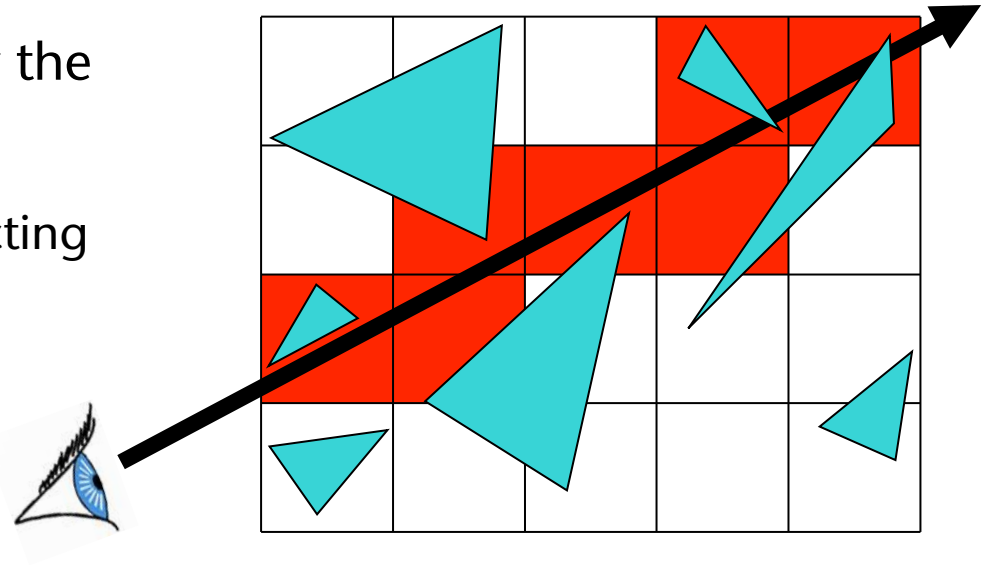
- Basic idea: save costs by precomputations on the scene and filtering of the rays during run-time



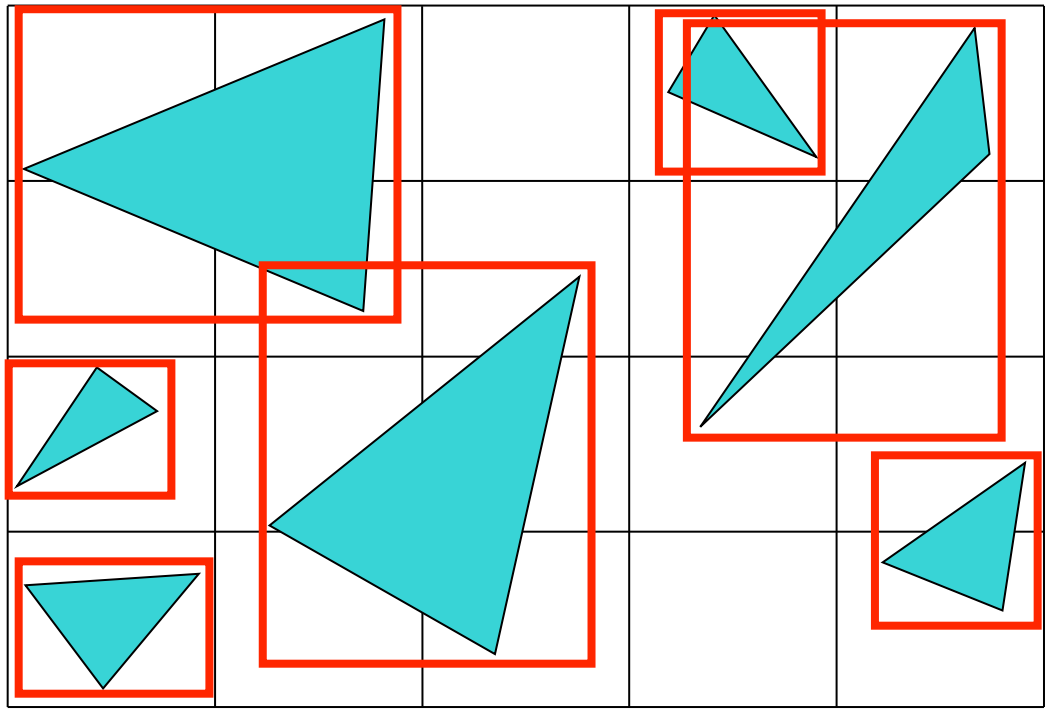
- If the ray misses the BV, then it must also miss the enclosed object

Regular 3D Grids

- Construction of the grid:
  - Calculate BBox of the scene
  - Choose a (suitable) grid resolution (n_x, n_y, n_z)
- For each cell intersected by the ray:
  - Is any of the objects intersecting the cell hit by the ray?
  - Yes: return closest hit
  - No: proceed to next cell



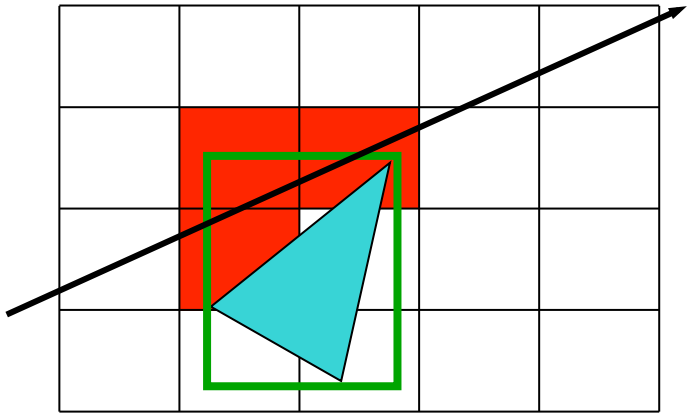
- Precomputation: for each cell store all objects intersecting that cell in a list with that cell⟶ "insert objects in cells"
  - Each cell has a list that contains pointers to objects
- How to insert objects: use bbox of objects
  - Exact intersection tests are not worth the effort
- Note: most objects are inserted in many cells

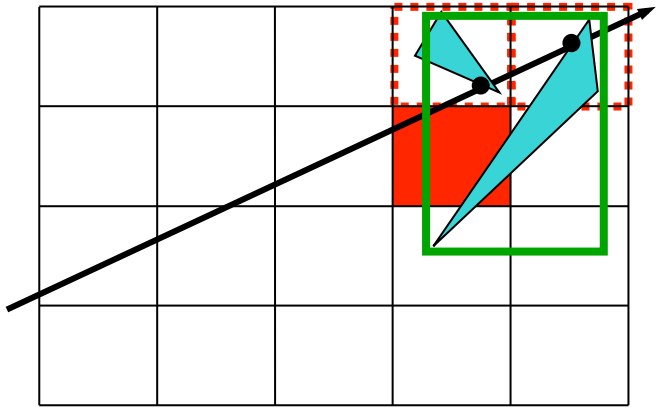


Problems

- Objects could be referenced from many cells

1. Consequence: a ray-object intersection need not be the closest one (see bottom right)
   - Solution: disregard a hit, if the intersection point is outside the current cell
2. Consequence: we need a method to prevent the ray from being intersected with the same object several times (see bottom left)





The Mailbox Technique

- Solution: assign a mailbox with each object (e.g., just an integer instance variable), and generate a unique ray ID for each new ray
  - For the ray ID: just increment a counter in the constructor of the ray class
- After each intersection test with an object, store the *ray ID* in the object's *mailbox*
- Before an intersection test, compare the ray ID with the ID stored in the object's mailbox:
  - Both IDs are equal ⟶ the intersection point can be read out from the mailbox;
  - IDs are not equal ⟶ perform new ray-object intersection test, and save the result in the mailbox (together with the ray ID)

Optimization of the Mailbox Technique

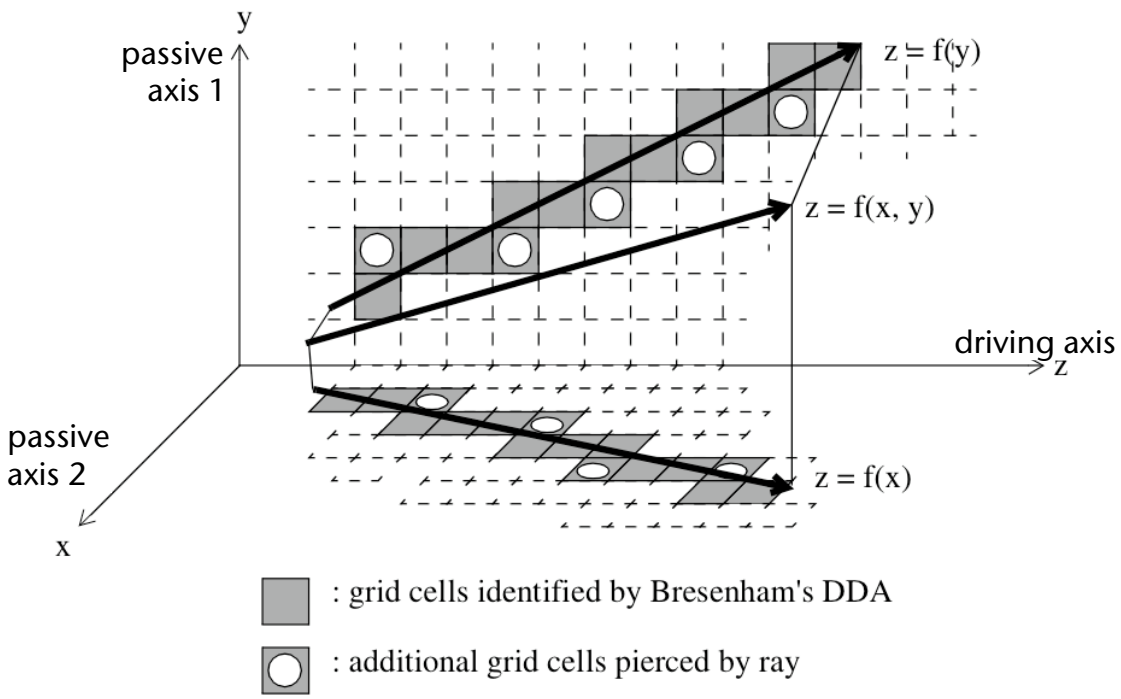
- Problems with the naive method:
  - Writing the mailbox invalidates the cache
  - You cannot test several rays in parallel
- Solution: store mailboxes seperately from geometry
  - Maintain a small hash-table with each ray that stores object IDs
    - Works, because only few objects are hit by a ray
    - So, the hashtable can reside mostly in level 1 cache
  - A simple hash function is sufficient
  - Now, checking several rays in parallel is trivial
- Remark: this is another example of the old question, whether one should implement it using an

  "Array of Structs" (AoS) or a "*Struct of Arrays*" (SoA)

  ?

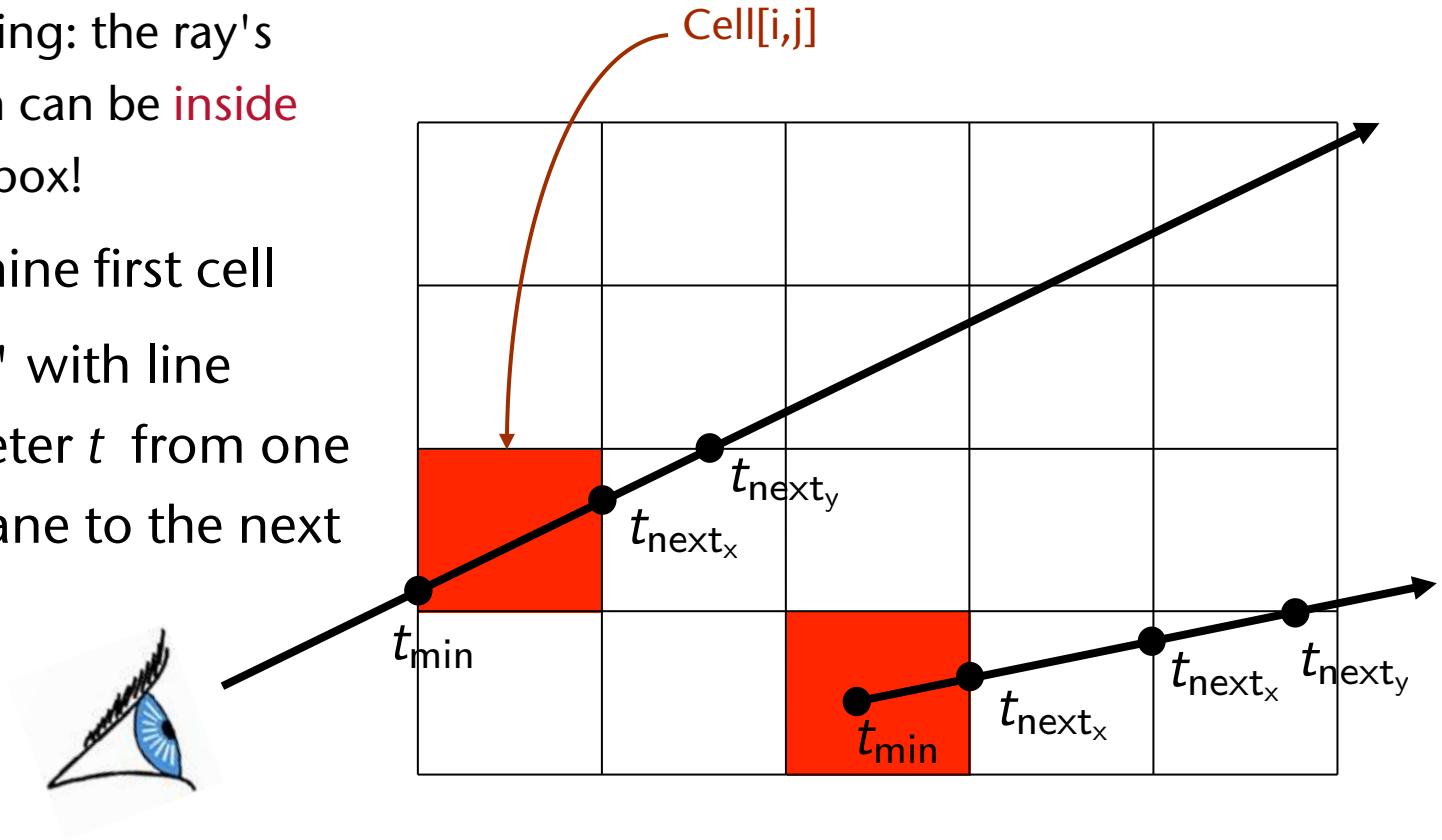
Traversal of a 3D Grid

- Simple idea: utilize 2 synchonized DDA's → 3D-DDA
  - Just like in 2D, there is a "driving axis"
  - In 3D, there are now two "passive axes"

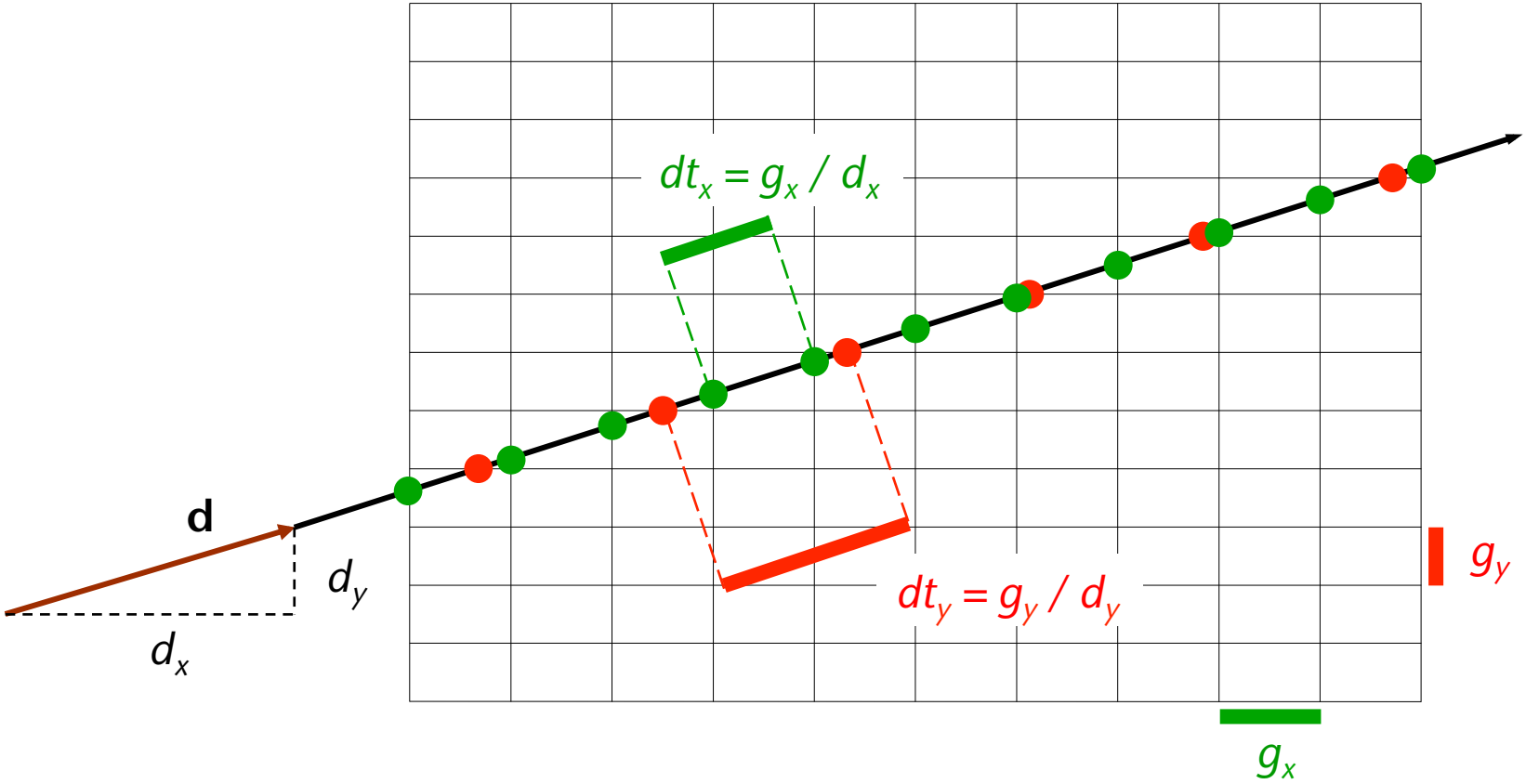


Better Grid Traversal Algorithm

- Intersect ray with Bbox of the whole scene
  - Warning: the ray's origin can be inside the Bbox!
- Determine first cell
- "Jump" with line parameter t from one grid plane to the next

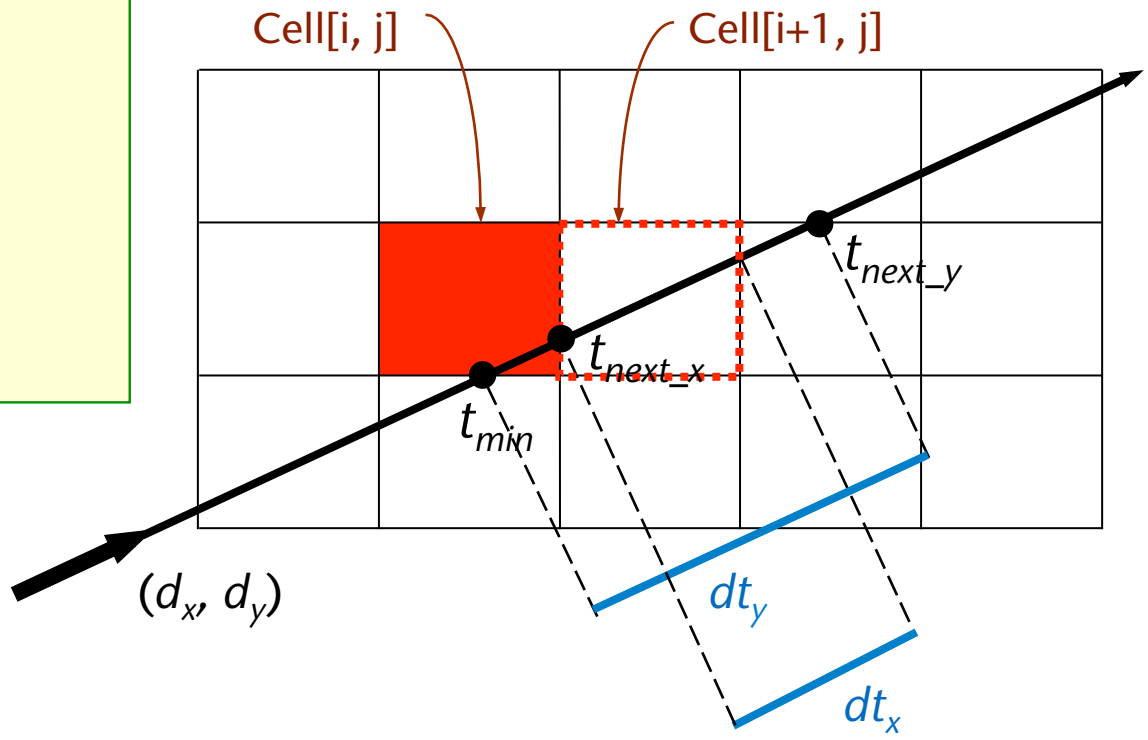


- Is there a pattern in the cell transitions?
- Yes, all horizontal and all vertical transitions have the same distance (among themselves)



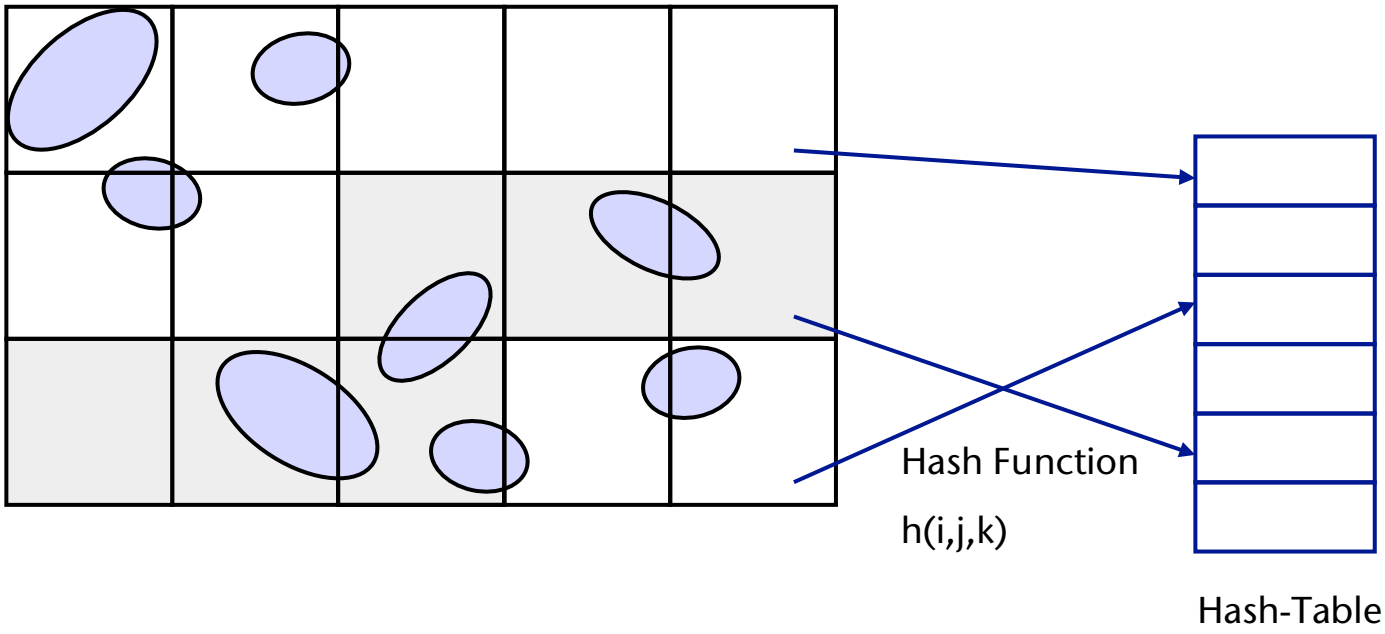
The Algorithm

```
if t_next_x < t_next_y :
  i += s_x
  t_min = t_next_x
  t_next_x += dt_x
else:
  j += s_y
  t_min = t_next_y
  t_next_y += dt_y
```

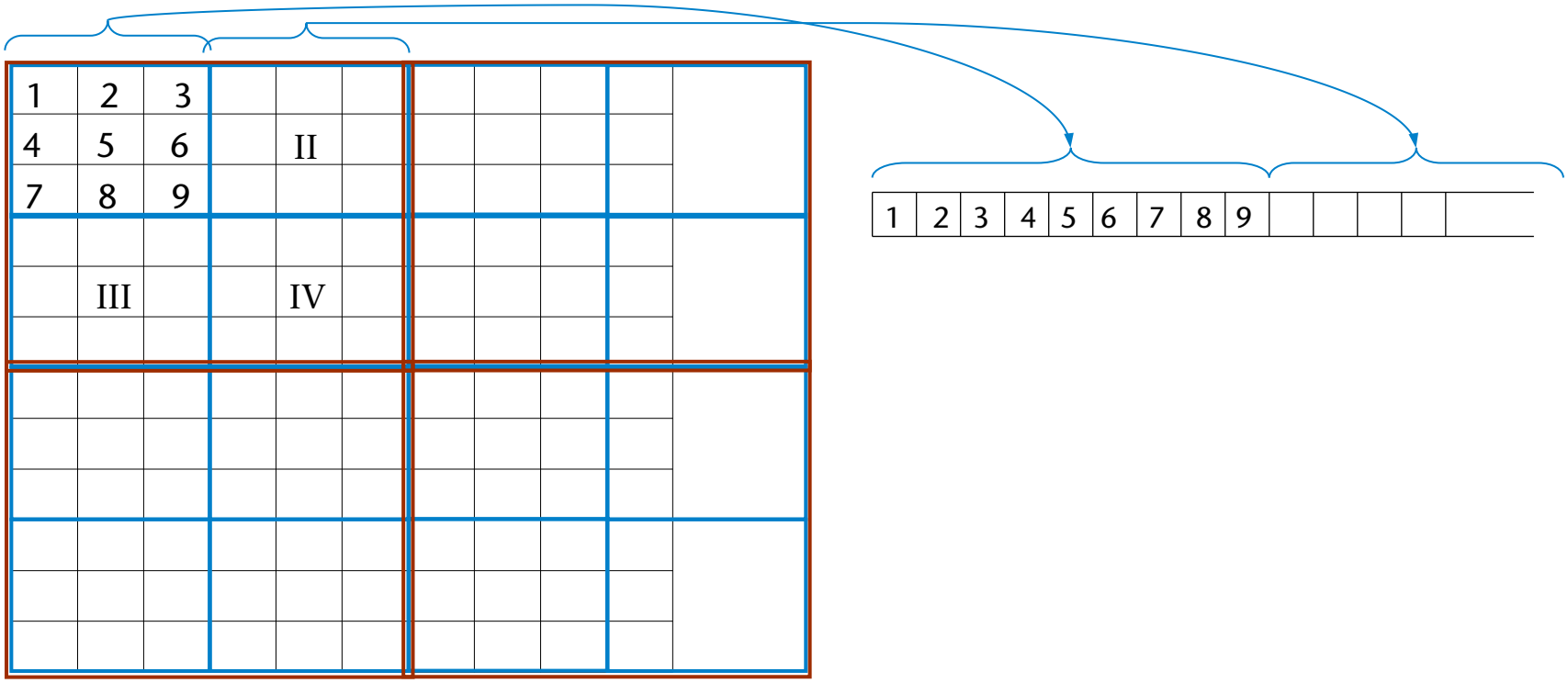


Storage

- Lots of empty cells → represent grid by hash table



- Dense grid → use blocking (aka memory bricking)
  - Partition grid into blocks, store each block in a contiguous memory region, such that 1 block = 1 L1 cache line
  - Aggregate blocks to "macro blocks", so that one macro block fits entirely into L2 cache

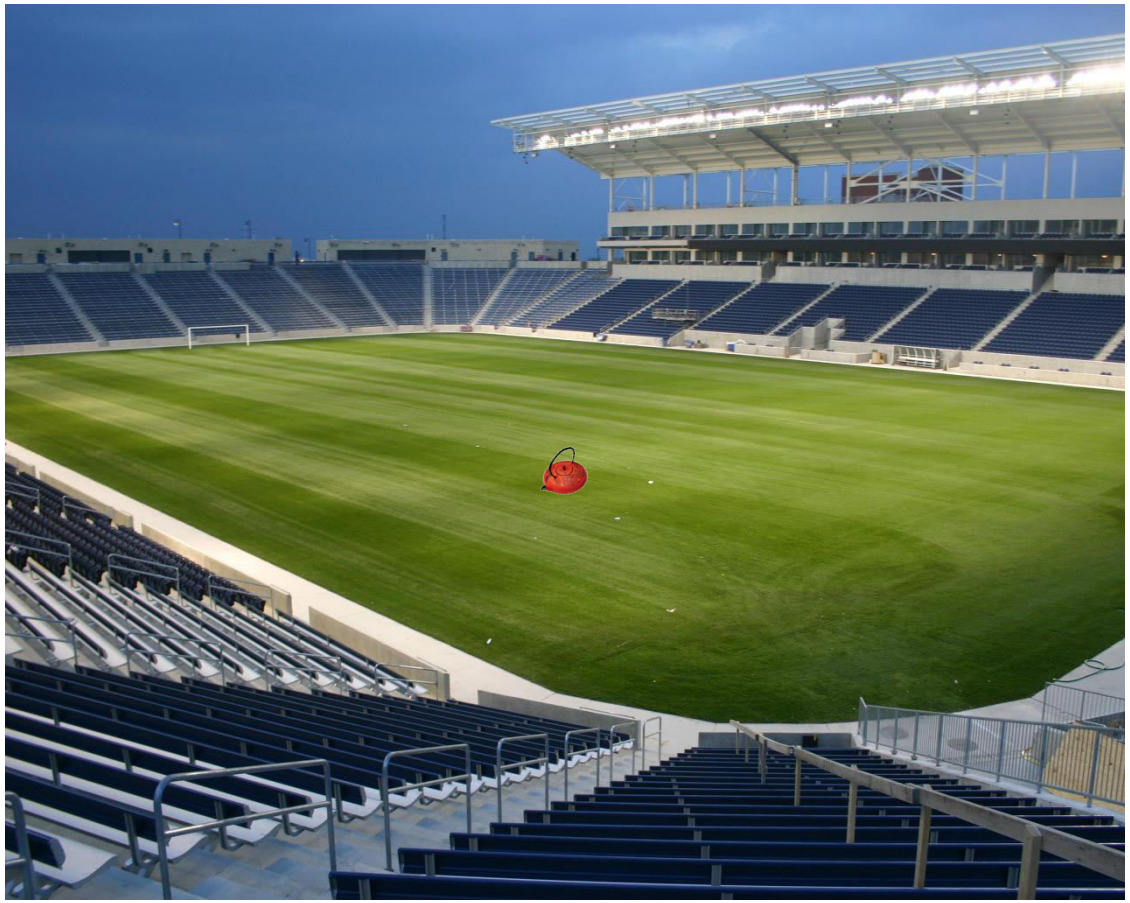


The Optimal Number of Voxels

- Too many cells → slow traversal, heavy memory usage, bad cache utilization
- Too few cells → too many objects 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 = $\sqrt[3]{N}$
- Another good rule of thumb: try to make the cells cuboid-like

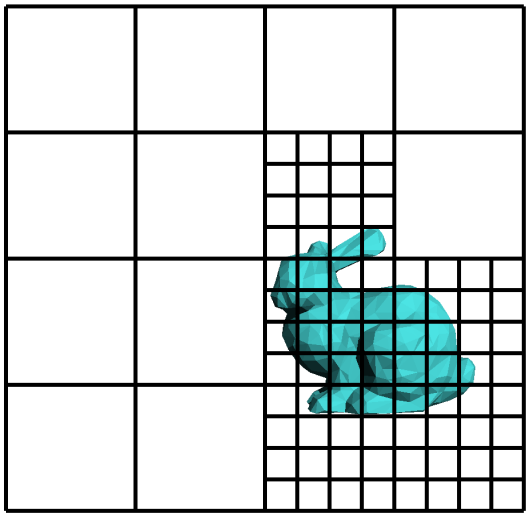
The Teapot in a Stadium Problem

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



Recursive Grids

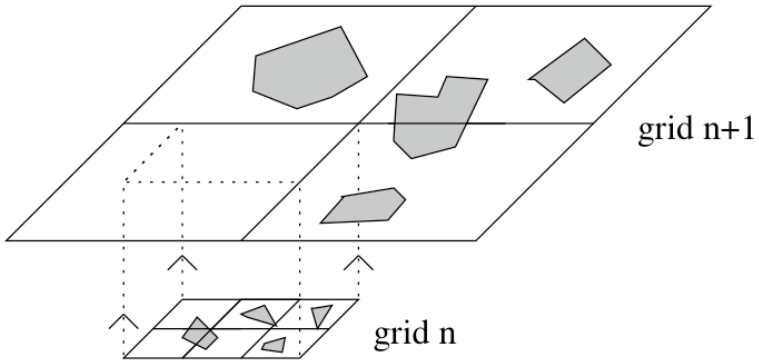
- Idea:
  - First, construct a coarse grid
  - Subdivide "dense" cells again by a finer grid
  - Stopping criterion: less than n objects in the cell, or maximum depth
- Results in a k^3-ary tree
  - How do you store nodes that have k^3 child pointers?
- Additional Feature:
  subdivision "on demand", 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



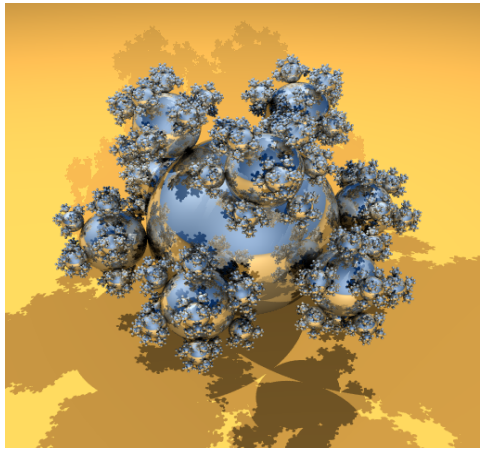
Nested Grids

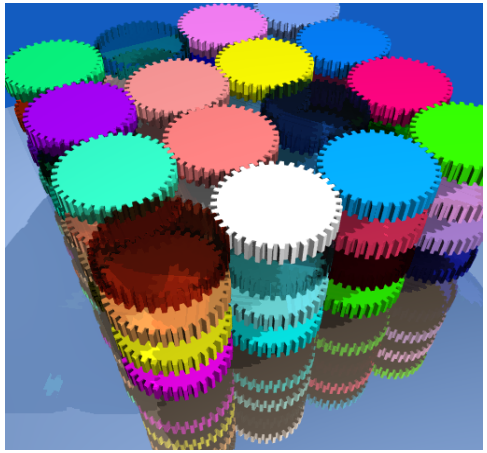
Hierarchical Uniform Grid (HUG) [1994]

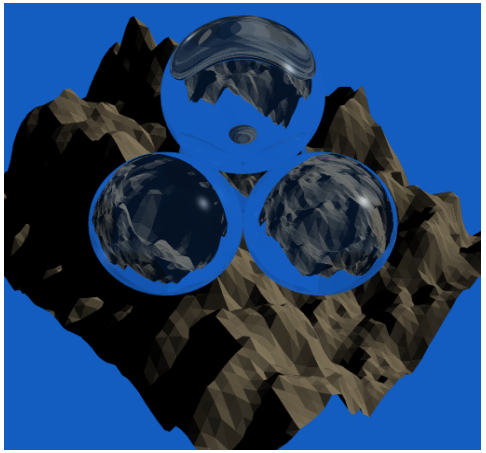
- Problem: if the variance among object sizes is very large, then the average object size is not a good cell size
- Idea:
  - Group objects by size → "size clusters"
  - Group objects within a size cluster by location → local size clusters
  - Construct grid for each local size cluster
  - Construct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



Construction Time of Different Grids



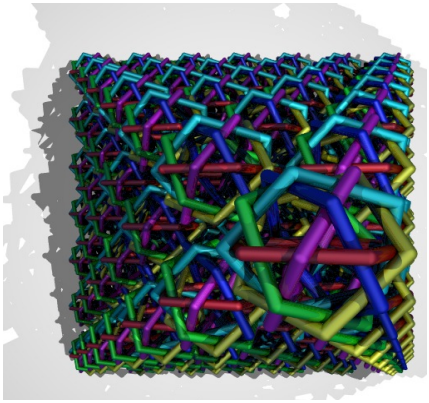
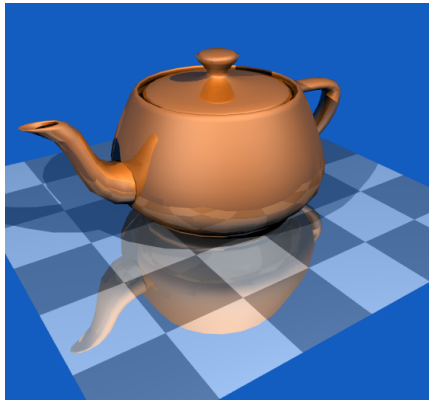
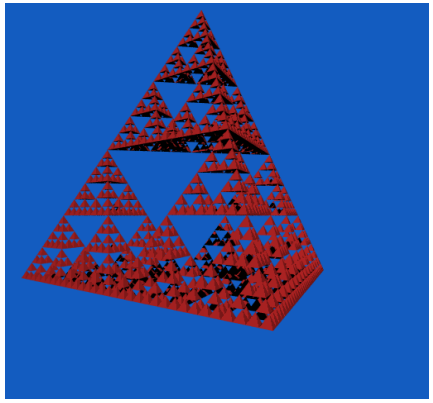
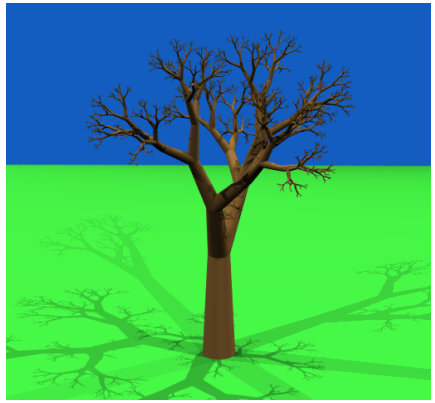




	balls	gears	mount
Uniform, $D = 1.0$	0.19	0.38	0.26
Uniform, $D = 20.0$	0.39	1.13	0.4
Recursive Grid	0.39	5.06	1.98
HUG	0.4	1.04	0.16

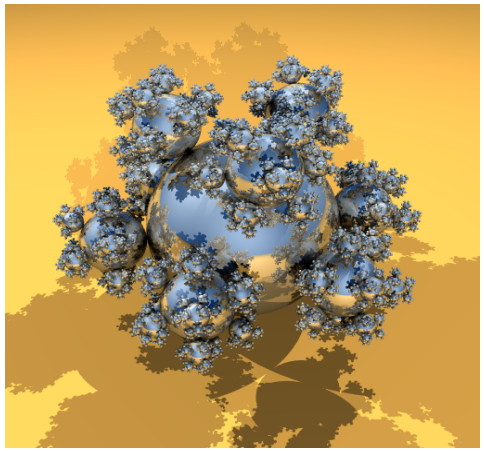
$$D = \frac{\#\text{ voxels}}{\#\text{ objects}}$$

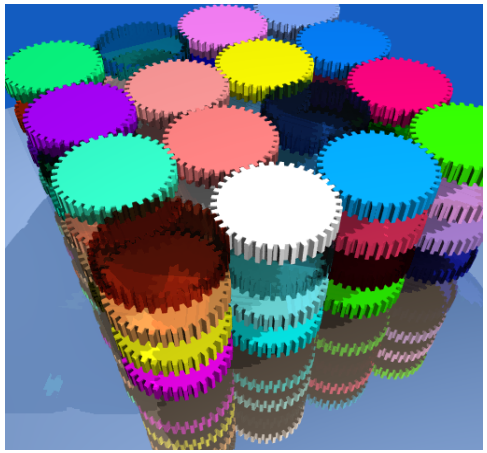
Quelle: Vlastimil Havran, Ray Tracing News vol. 12 no. 1, June 1999, http://www.acm.org/tog/resources/RTNews/html

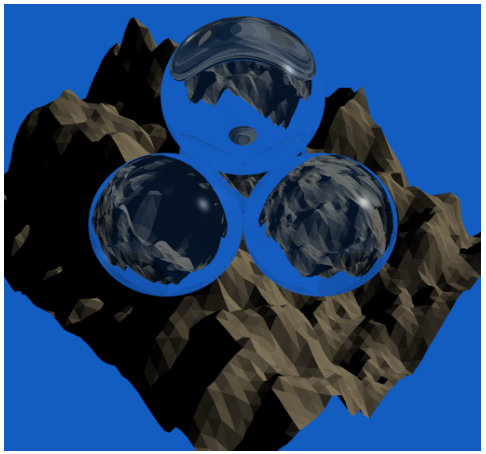





	rings	teapot	tetra	tree
Uniform, $D = 1.0$	0.35	0.3	0.13	0.22
Uniform, $D = 20.0$	0.98	0.65	0.34	0.33
Recursive Grid	0.39	1.55	0.47	0.28
HUG	0.45	0.53	0.24	0.48

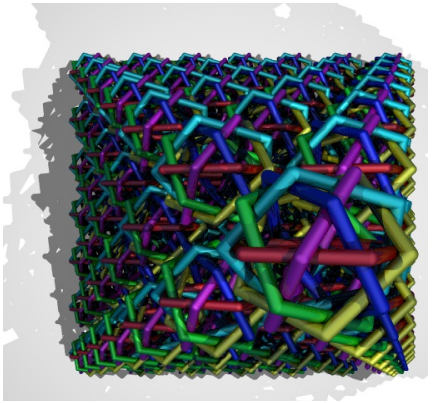
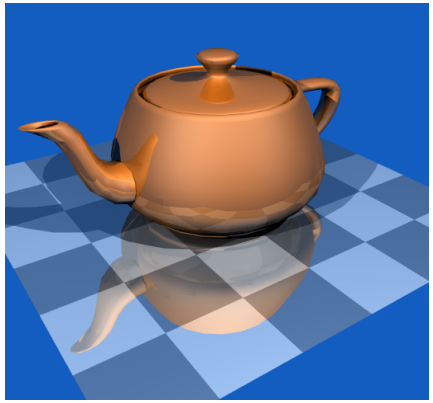
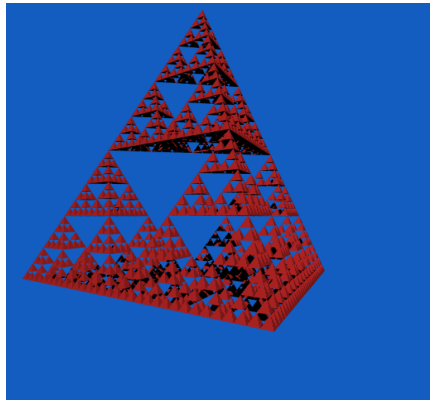
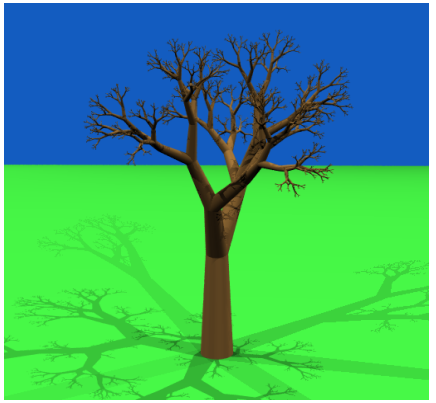
Running Times of the Ray-Tracing (sec)







	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

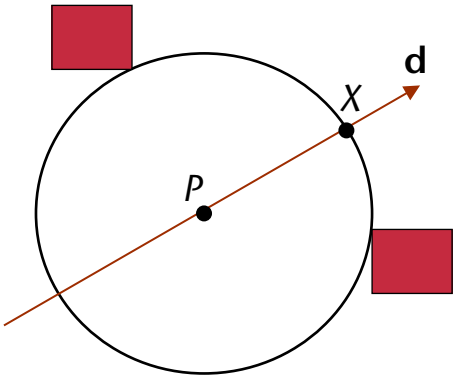
	Rings	Teapot	Tetra	Tree
Uniform, $D = 1.0$	129.8	28.68	5.54	1517.0
Uniform, $D = 20.0$	**83.7**	**18.6**	**3.86**	781.3
Rekursiv	113.9	22.67	7.23	33.91
HUG	116.3	25.61	7.22	33.48
Adaptive	167.7	43.04	8.71	**18.38**

Proximity Clouds

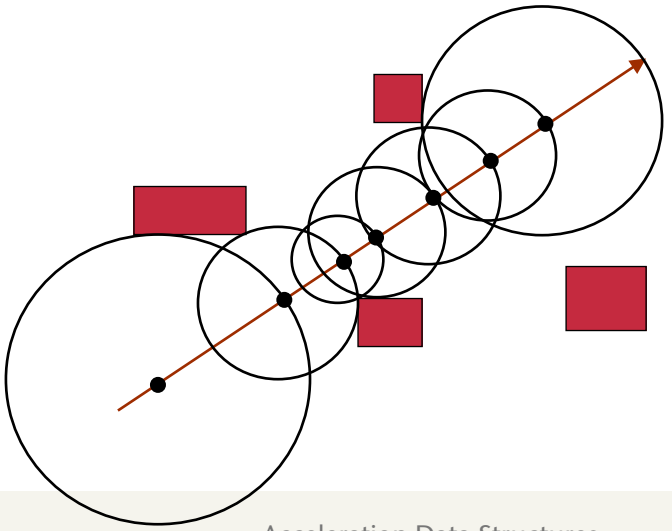
[1994]

- 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 ...

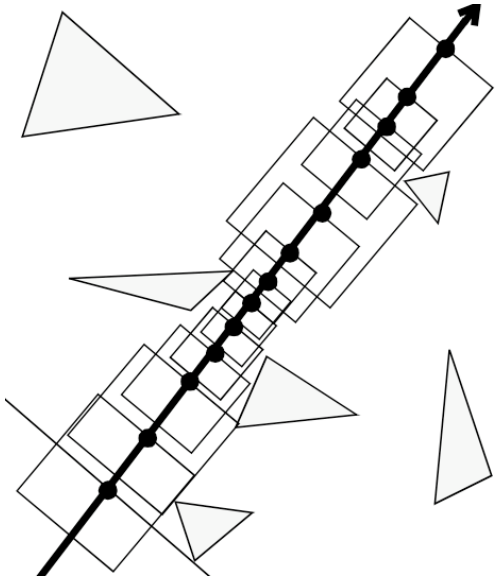


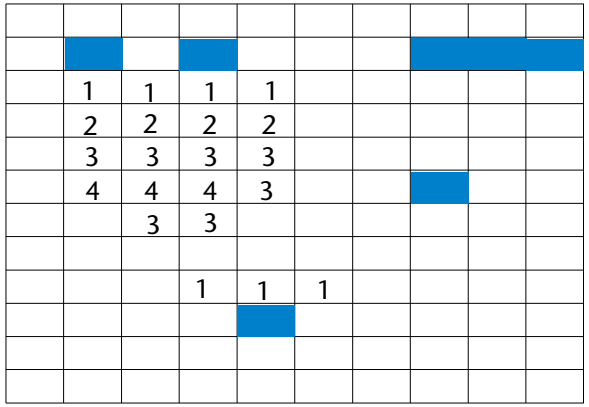
- The general idea is called empty space skipping
  - Comes in many different guises

- 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)

➢ This data structure is called a distance field

- Example:



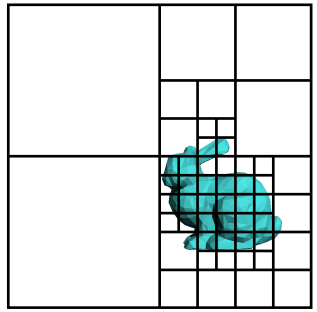


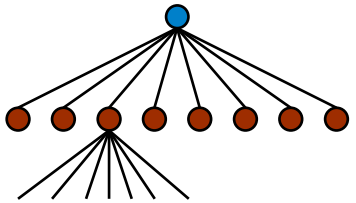
General Rules for Optimization

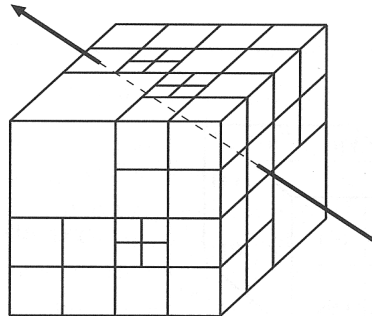
- "Premature Optimization is the Root of All Evil" [Knuth]
  - *First*, implement your algorithm naïve and slow, *then* optimize!
  - After each optimization, do a before-after benchmark!
    - Sometimes/often, optimization turn out to perform worse
  - Only make small optimizations at a time!
  - Do 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!

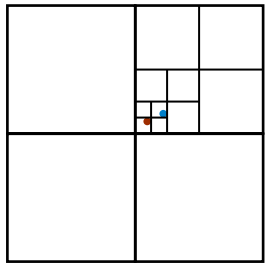
The Octree / Quadtree

- 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

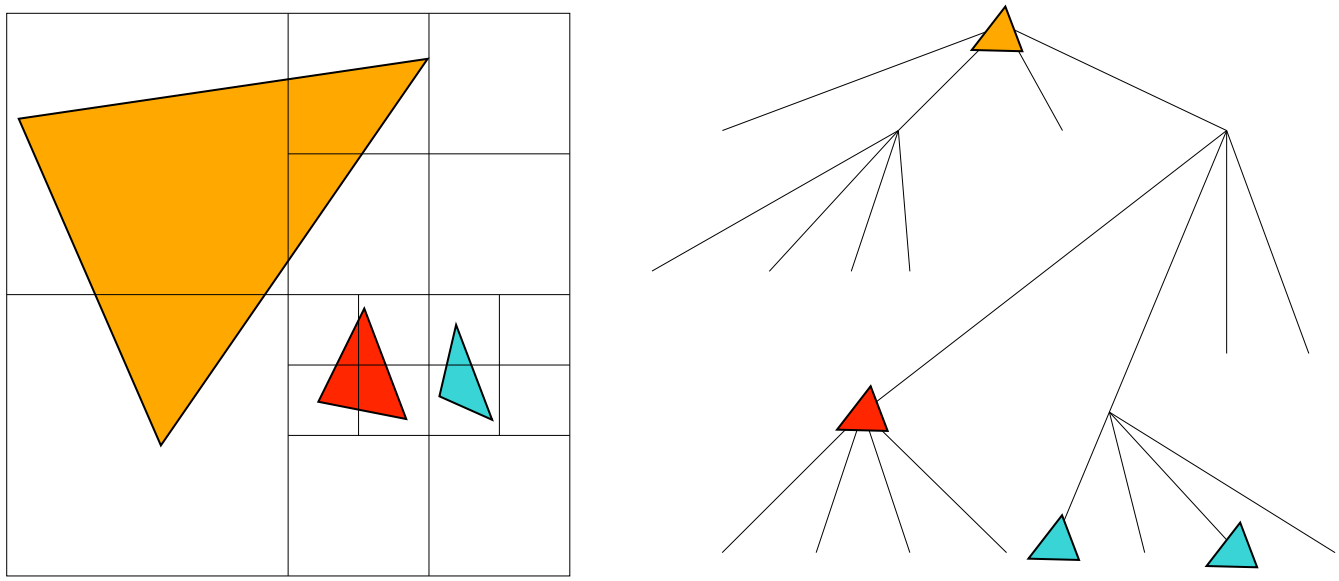








- What about large objects in octrees?
- Must be stored with inner nodes, or ...
- In leaves only, but then they need to be stored in many nodes



Octree/(Quadtree)

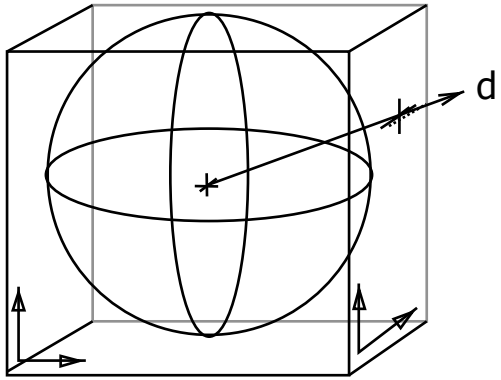
The 5D Octree for Rays

[Arvo u. Kirk 1987]

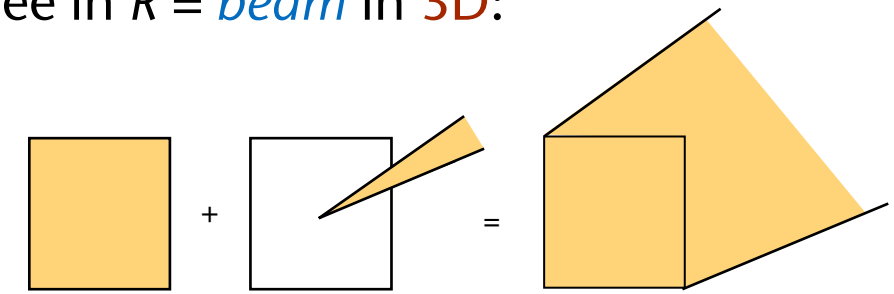
- What is a ray?
  - Point + direction = 5-dim. object
- Octree over a set of rays:
  - Construct bijective mapping between directions and the direction cube:

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

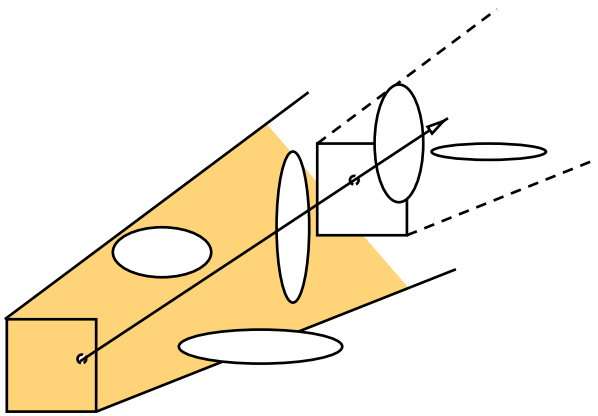
  - All rays in the universe $U = [0,1]^3$ are given by the set: $R = U \times D$



- A node in the 5D octree in R = *beam* in 3D:



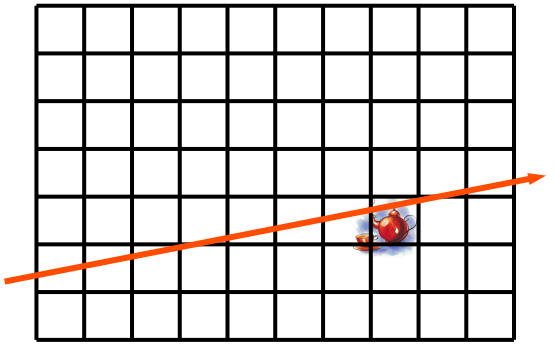
- Construction (6x):
  - Associate object with an octree node ⟷ object intersects the beam
  - Start with root = $U \times [-1, +1]^2$ and the set of all objects
  - Subdivide node (32 children), if
    - too many objects are associated with the current node, *and*
    - the cell is too large.
    - 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 ...

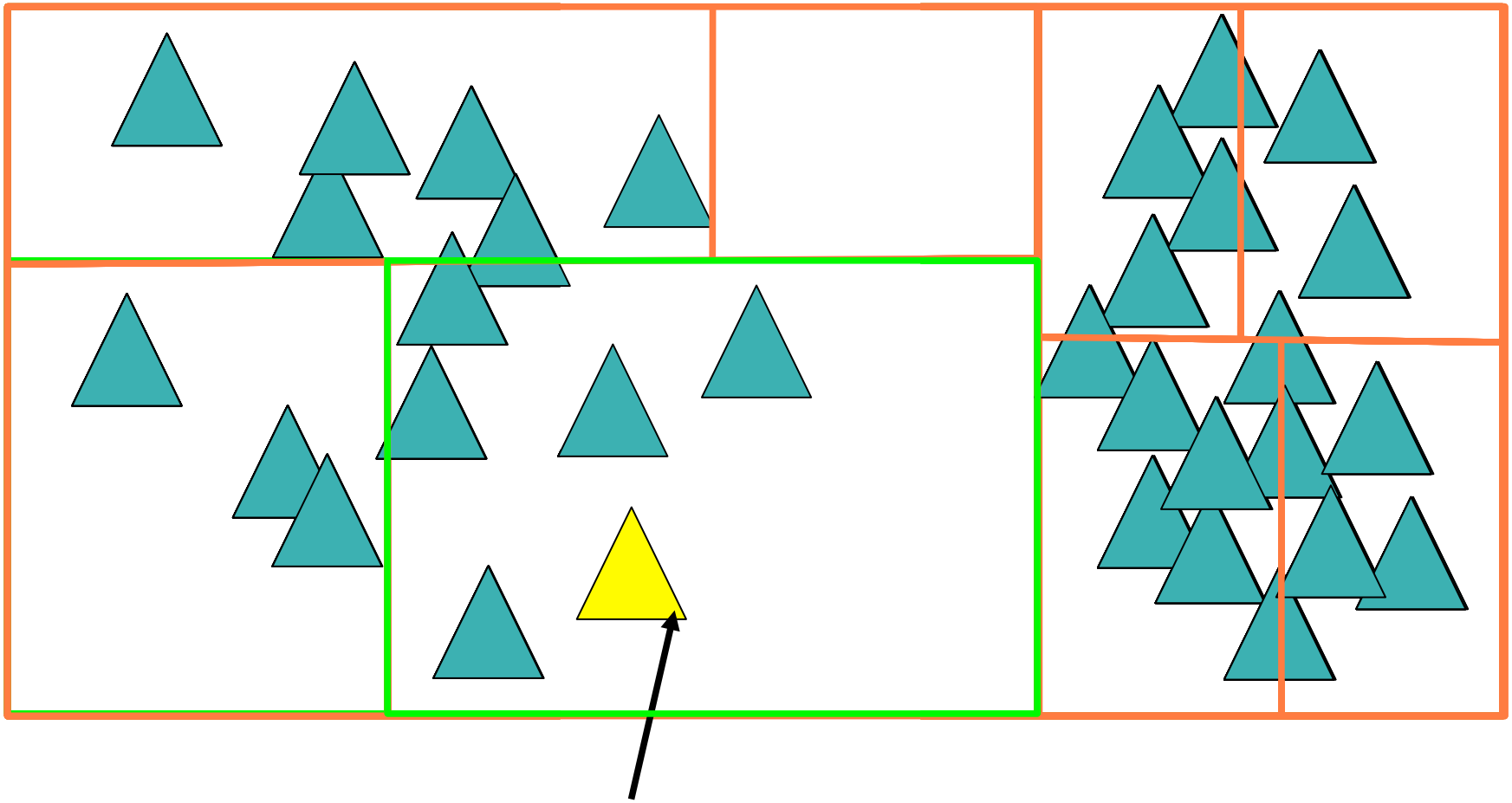
kD-Trees



- Problem with grid: "teapot in a stadium"
- Probem with octrees:
  - Very unflexible 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, otherwise arbitrary
- "Best known method" [Siggraph Course 2006]
- ... at least for static scenes

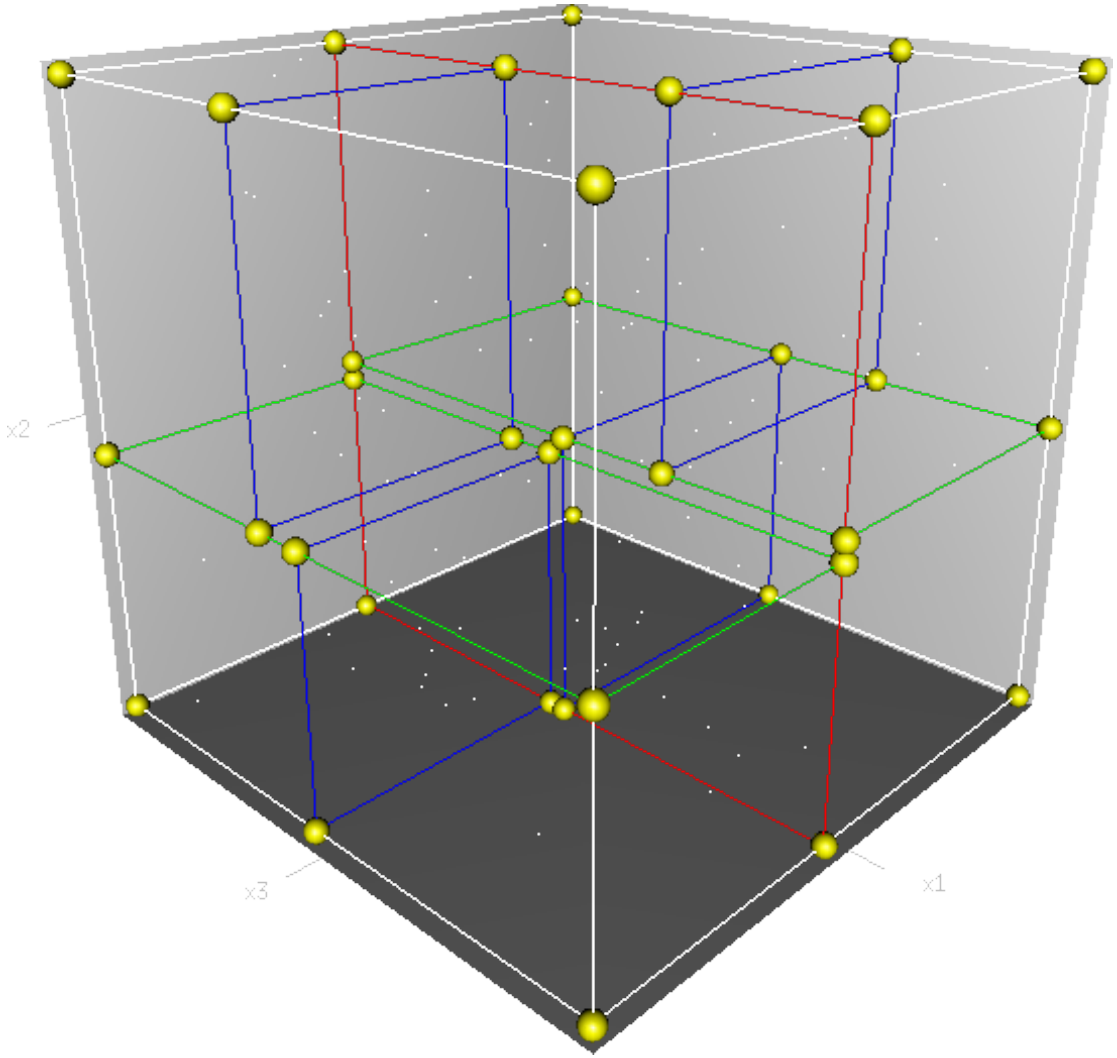
- Informal definition:
  - A kd-tree is a binary tree, where
    - Leaves contain single objects (polygons) or a list of objects;
    - Inner nodes store a splitting plane (perpendicular to an axis) and child pointer(s)
  - Stopping criterion:
    - Maximal depth, number of objects, some cost function, ...
- 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



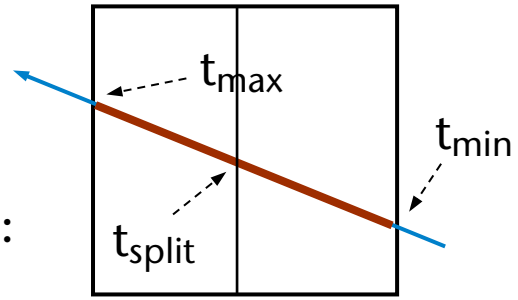
[Slide courtesy Martin Eisemann]

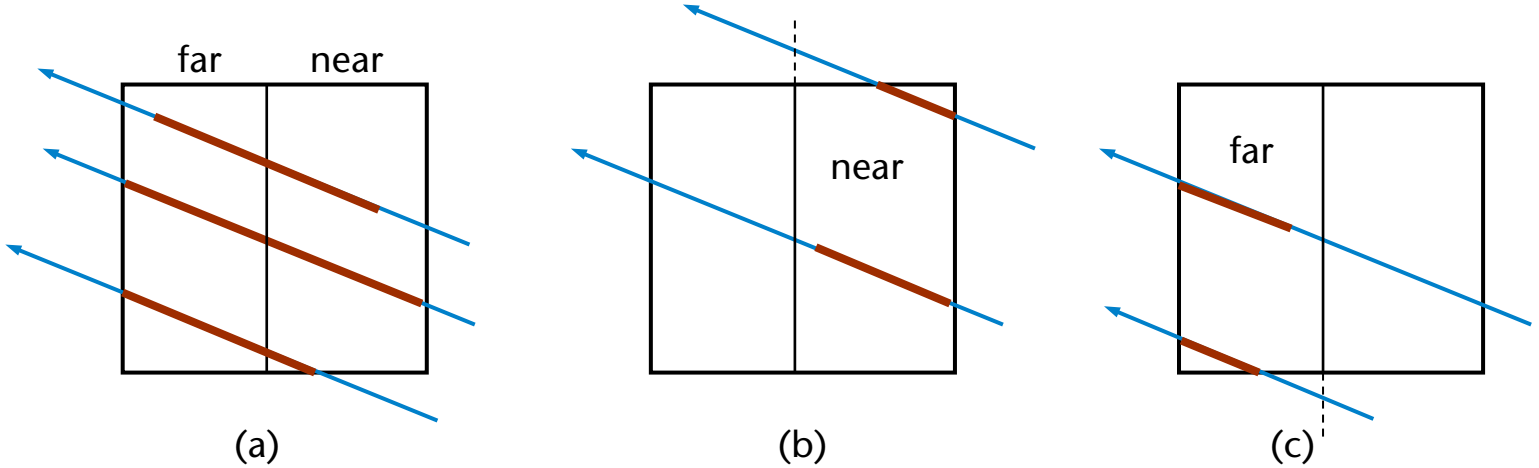
3D-Beispiel



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





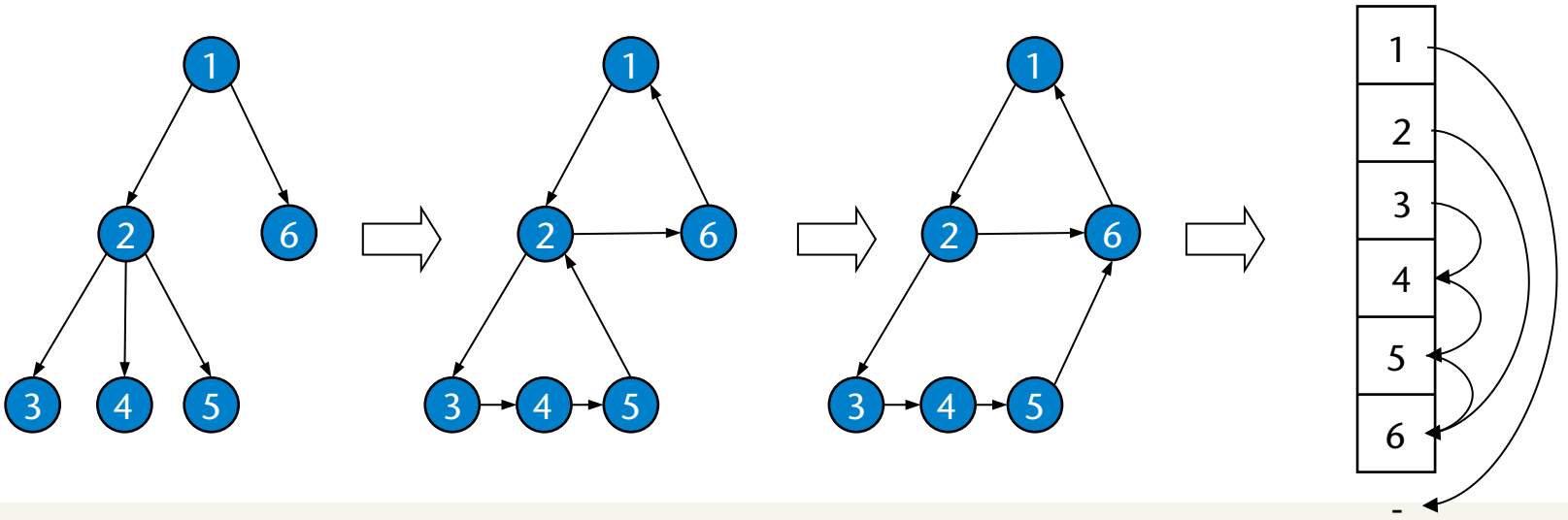
Pseudo-Code für die Traversierung

```
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 [1999]

- 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:



- 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"
```

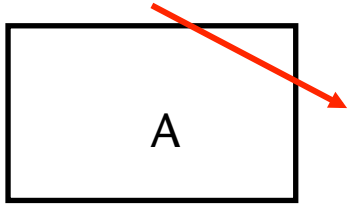
Diplomarbeit ...

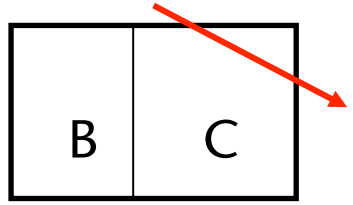
Construction of a kD-Tree

- **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)

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 = $t_i N$
  - Split → costs = $t_t + t_i(p_B N_B + p_C N_C)$





where t_i = time for 1 ray-primitive test

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

p_B =probability, that the ray hits cell B

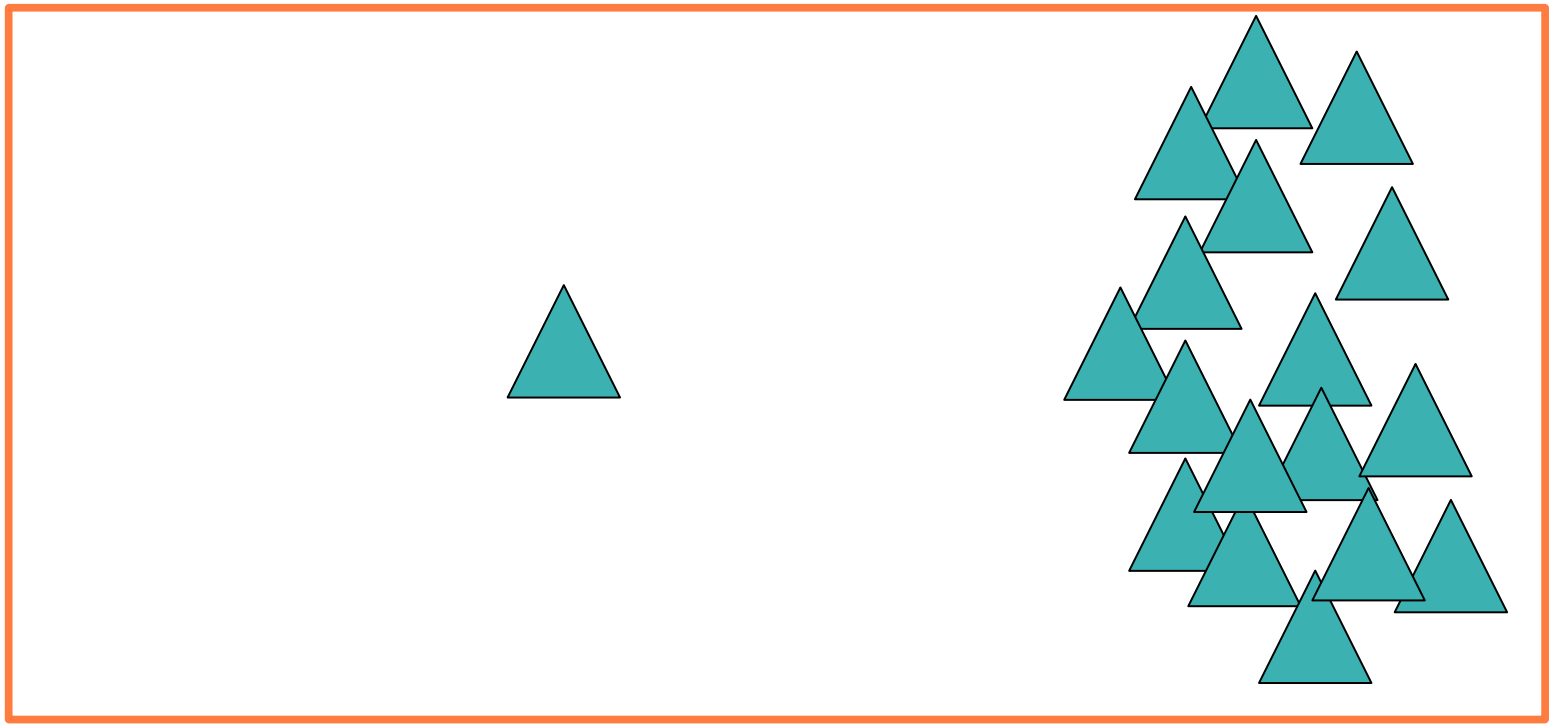
N = number of primitives

- We make the following simplifying assumptions: $p_B \propto \frac{S_B}{S_A}$
  - t_i = const for all primitives
  - $t_i : t_t = 80 : 1$ (determined by experiment)
- We will determine p_B in a minute

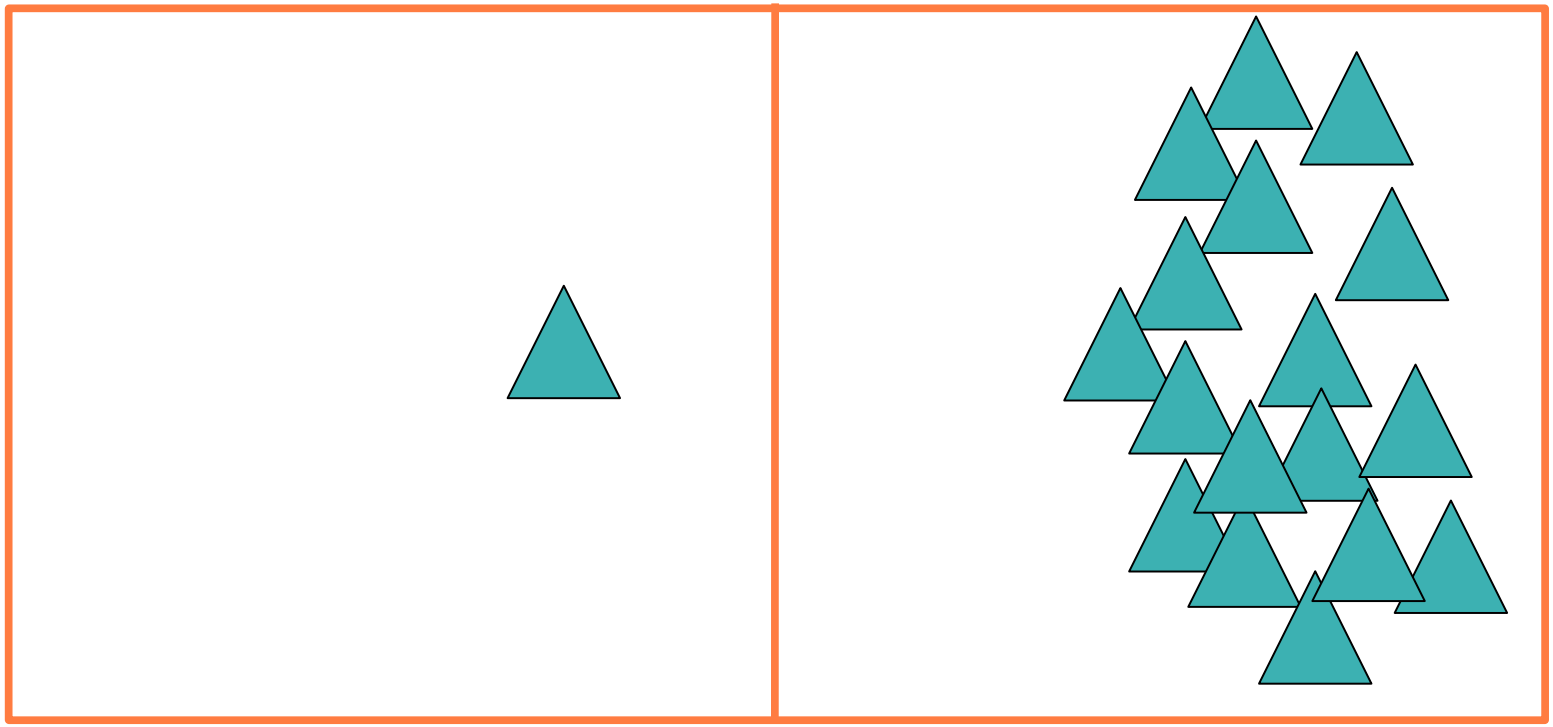
On Selecting a Splitting-Plane

- **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

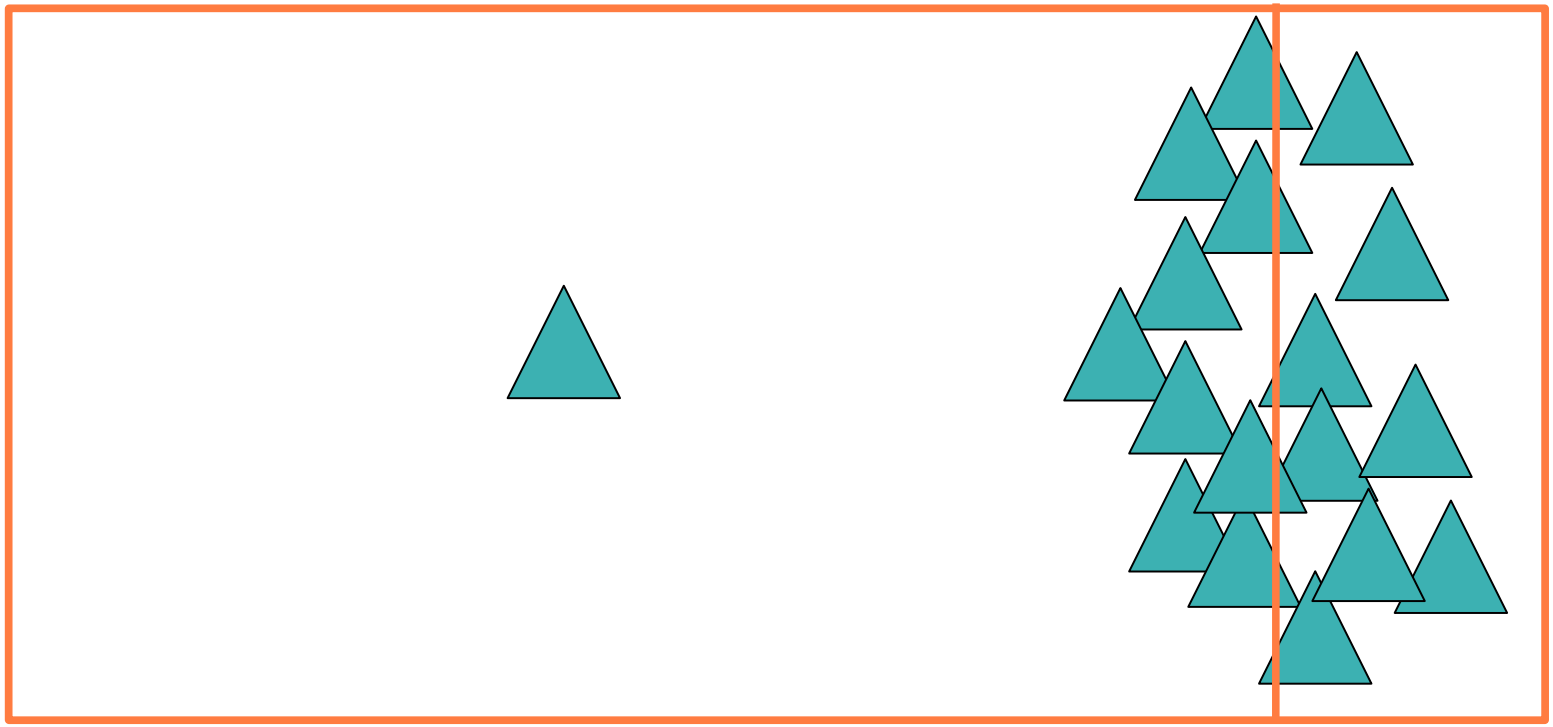


- Split in the middle:



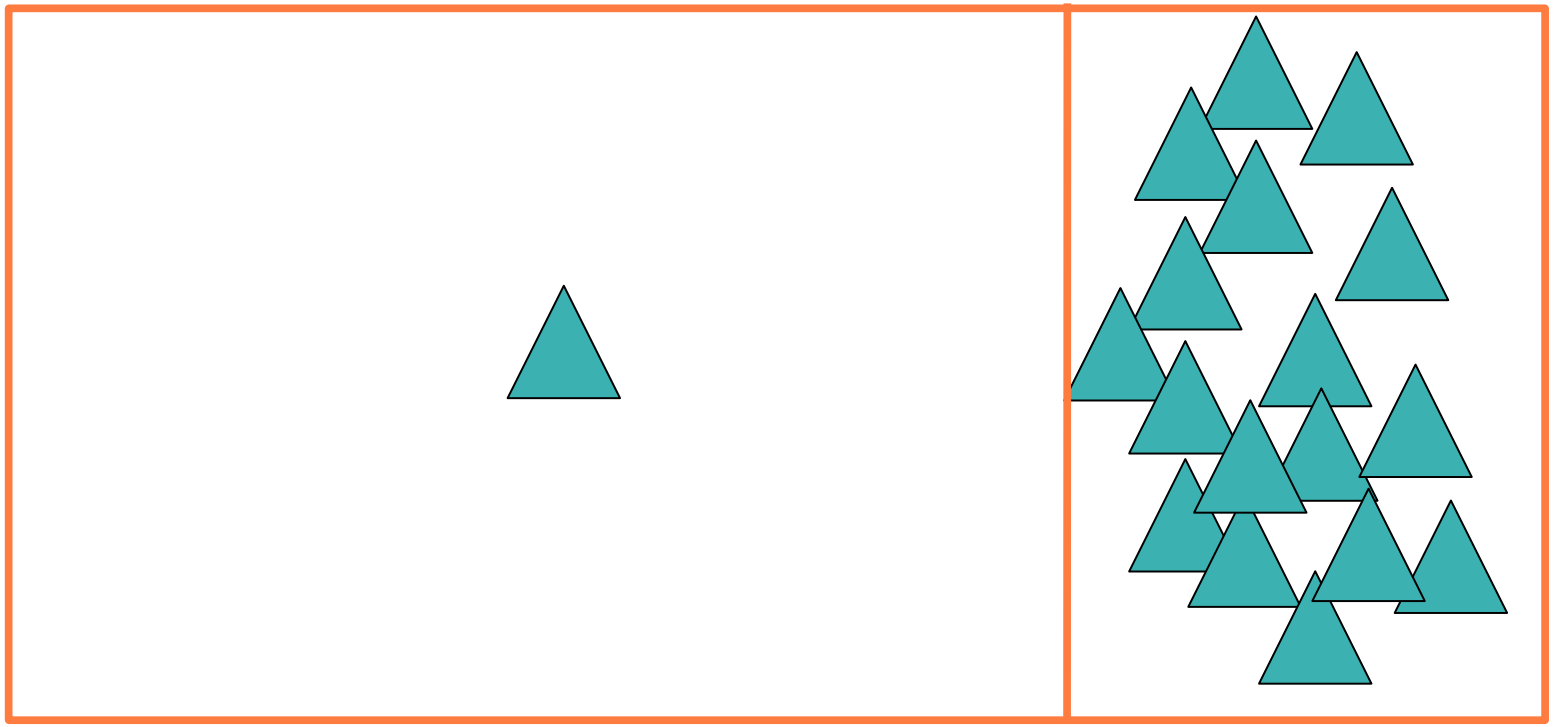
- The probability of a ray hitting the left or the right child is equal
- But, he expected costs for handling the left or the right child are very different!

- Split along the geometry median:



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

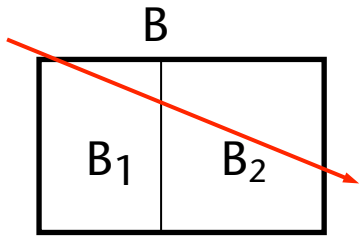
- Cost-optimized heuristic:



- 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 (~ time):



$$C(B) = P[\text{Schnitt mit } B_1] \cdot C(B_1) + P[\text{Schnitt mit } B_2] \cdot C(B_2)$$

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

- The probability is:

$$P[\text{Schnitt mit } B_1 \mid \text{Schnitt mit } B] = \frac{\theta_1}{\theta} \approx \frac{\text{Area}(B_1)}{\text{Area}(B)}$$

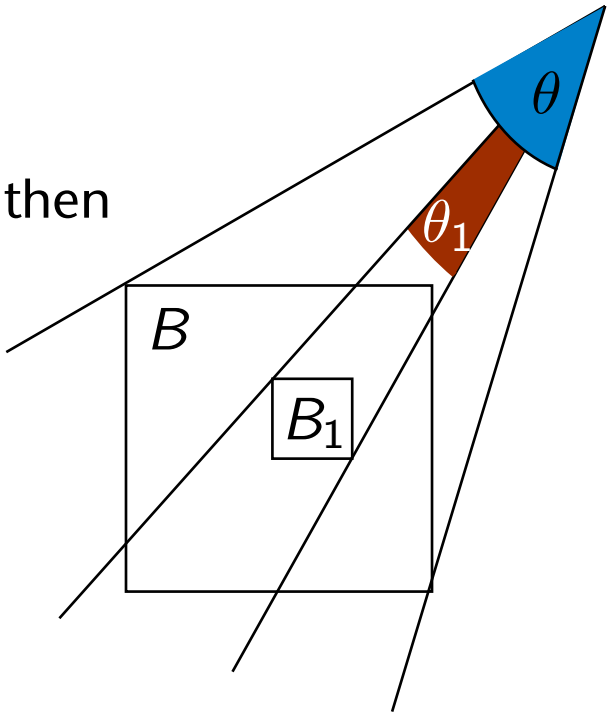
where $\frac{\theta}{\theta_1}$ is the spherical angle spanned by B and. B_1, resp.

- Explanation: The surface of a sphere is

$$A = 4\pi r^2$$

and if the origin of the rays is far away, then

$$r \sim \sin(\theta) \approx \theta$$



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

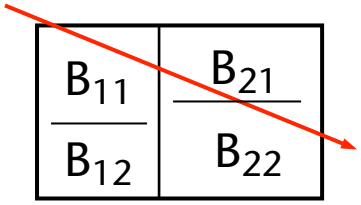
$$C(B_i) \approx \text{Anzahl Dreiecke in } B_i$$

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

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

- **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“

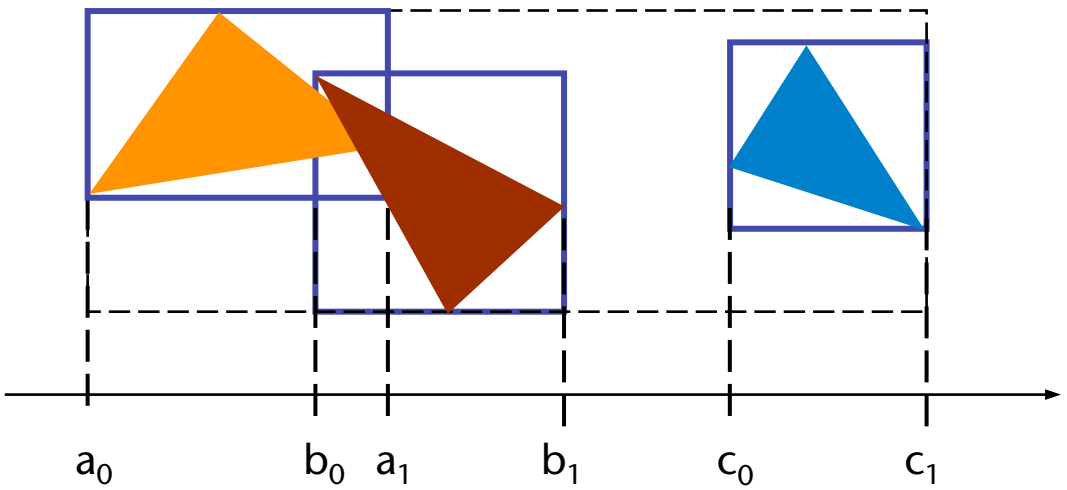


$$
\begin{aligned}
C(B) = & P[\text{Schnitt mit } B_1] \cdot C(B_1) \\
& + P[\text{Schnitt mit } B_2] \cdot C(B_2) \\
& = P[B_1] \cdot (P[B_{11}] C(B_{11}) + P[B_{12}] C(B_{12})) \\
& + P[B_2] \cdot (P[B_{21}] C(B_{21}) + P[B_{22}] C(B_{22})) \\
& \ldots
\end{aligned}
$$

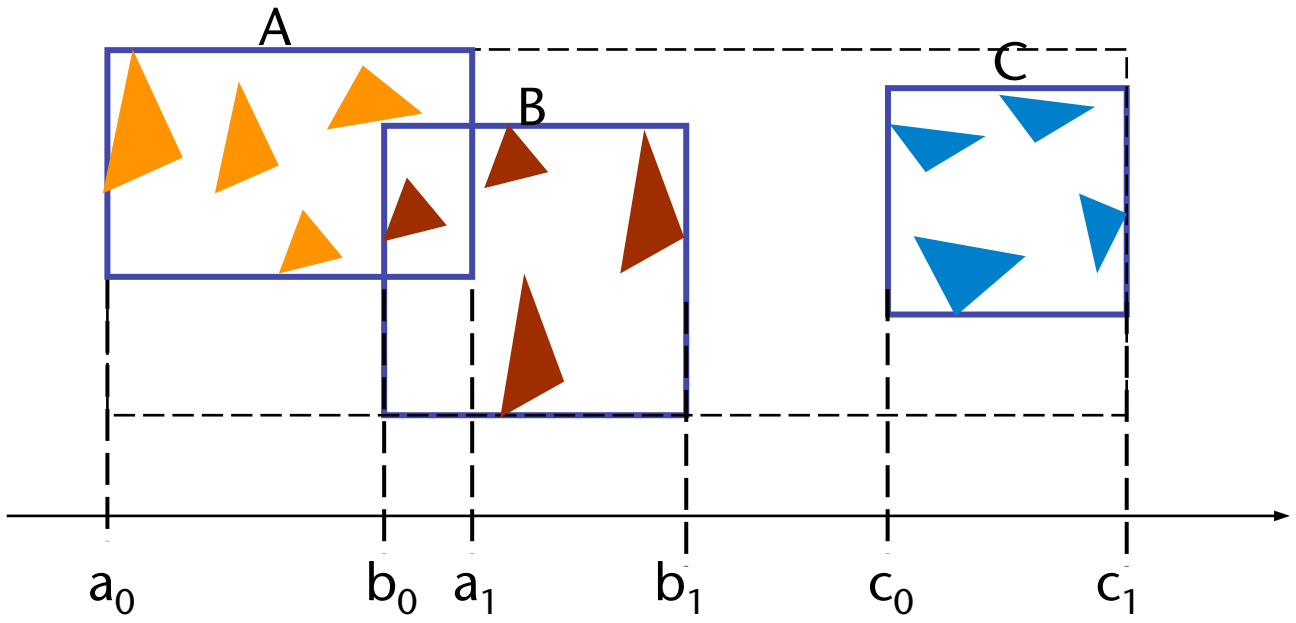
Diplomarbeit ...

Remarks

- 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 interval bisection and, thus, a faster evaluation



- 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



Zusätzliche Kriterien

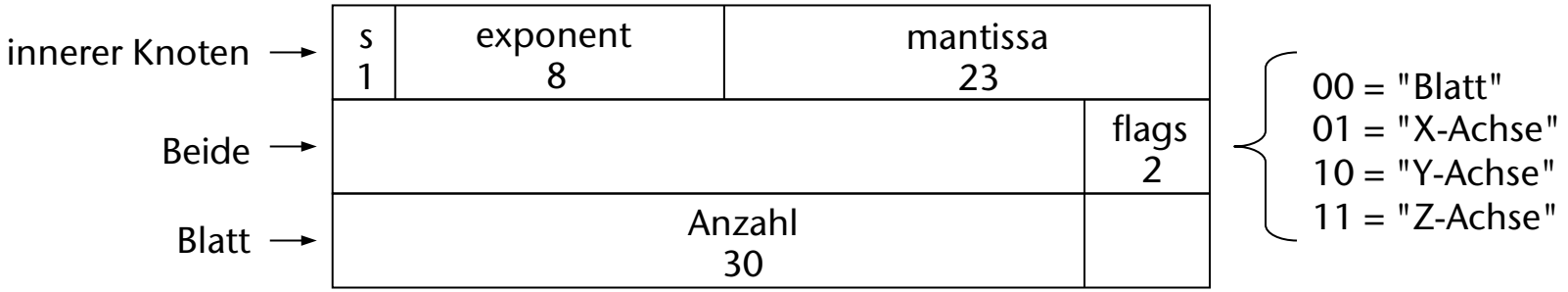
[2005]

- Teste vor der SAH folgende Regel:
  - Falls eine leere Kind-Zelle abgespalten werden kann, dann erzeuge diese (überspringe SAH)
- Teste folgendes zusätzliches Abbruchkriterium:
  - Falls das Volumen der aktuellen Zelle zu klein ist, dann keine Aufteilung
  - Kriterium für "zu klein" (z.B.): Vol(Zelle) < 0.1·Vol(Root)
  - Sinn: solche Zellen werden wahrscheinlich sowieso nicht getroffen
  - Spart Speicherplatz, ohne Laufzeit zu kosten
- Für Architekturmodelle:
  - Falls es eine Splitting-Plane gibt, die komplett von Polygonen bedeckt wird, dann verwende diese; schlage diese Polygone der kleineren Zelle zu
  - Sinn: dadurch passen sich die Zellen eher den "Räumen" an (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
};
```

- 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
```

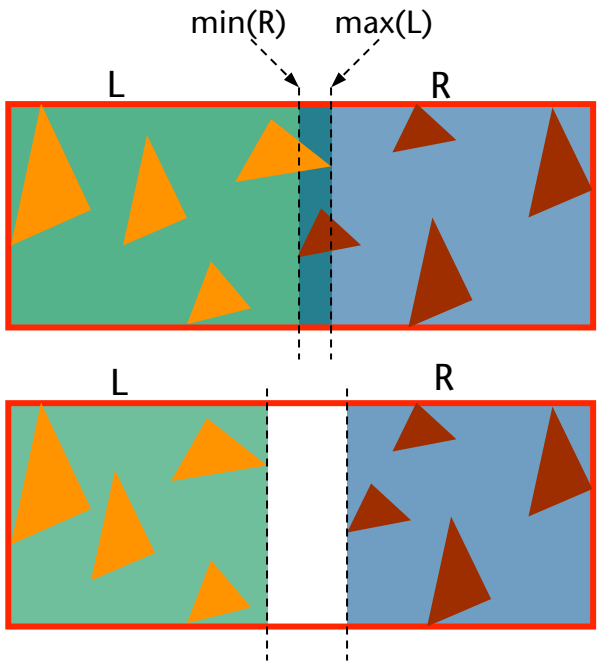
Falls m_nPrims == 1

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]

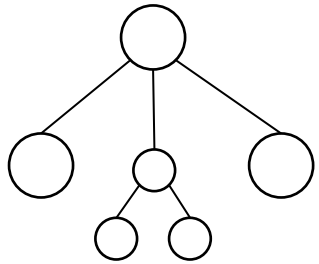
- A variant of the kD-Tree
- Other names: BoxTree, "bounding interval hierarchy" (BIH)
- Difference to the regular kd-tree:
  - 2 parallel splitting planes per node
  - Alternative: the 2 splitting planes can be oriented differently



- 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 trianles,
    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

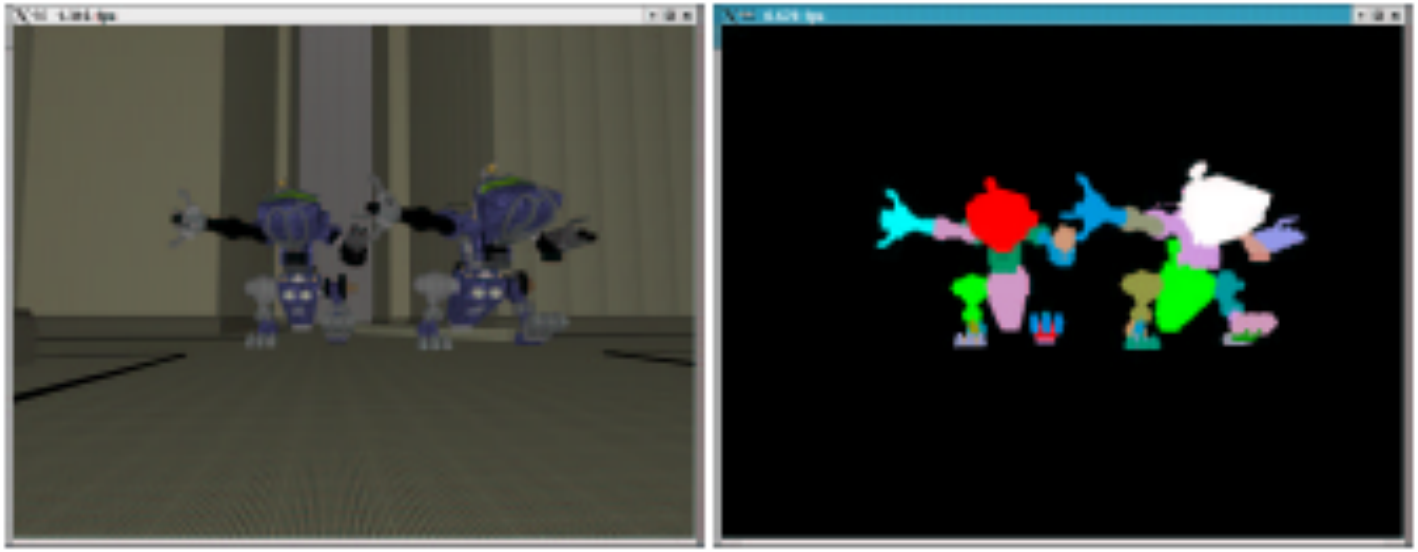
Oversized Objects

- Problem:
  - manchmal sind die Größen der Dreiecke sehr verschieden (z.B. Architektur-Modelle)
  - Diese erschweren das Finden von guten Splitting-Planes
- Lösung: ternärer Baum
- Aufbau:
  - Vor jedem Splitting: filtere "oversized objects" heraus
  - Falls viele "oversized objects": baue eigenen kd-Tree
  - Sonst: einfache Liste

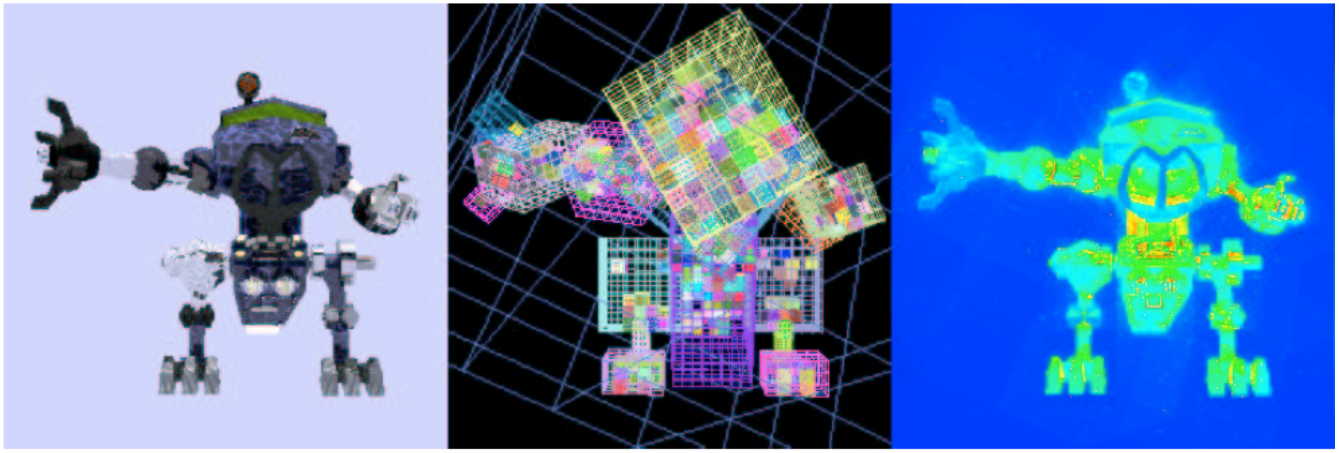


Zwei-stufige Datenstrukturen

- Beobachtung:
  - Oft ist nur ein Teil der Szene dynamisch
  - Die dynamischen Teile sind oft sog. *"articulated bodies"*, d.h., sie bestehen aus starren, miteinander beweglich verbundenen Teilen (z.B. Roboter)



- Idee:
  - Verwende für jedes in sich starre Teil ein eigenes Gitter (oder eine andere DS)
  - Verwende ein globales Gitter, in dem die einzelnen Teile als elememtare Objekte einsortiert werden
  - Bei Bewegung der Figur muß nur dieses globale Gitter aktualisiert werden



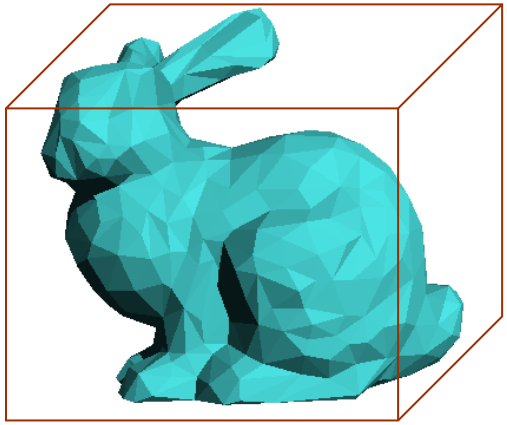
Articulated Body | Gitter für jedes Teil | Ray-Tracing-Zeit pro Pixel

Spatial Partitioning vs. Object Partitioning

- So far: acceleration data structure subdivided space, objects (=triangles) are associated afterwards to the cells
- Now: partition the set of objects, associated a bounding volume (= subset of space) with each
- In reality, the borders between the two categories are not clear-cut!

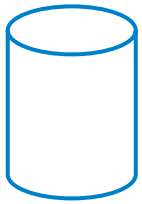
Bounding Volumes (BVs)

- General idea: approximate complex, geometric objects, or sets of objects, by some outer "hull"
- Requirements:
  - The objects must be completely inside the BV
  - More to come ...



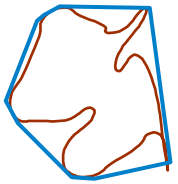
Examples of Bounding Volumes

Diplomarbeit ...



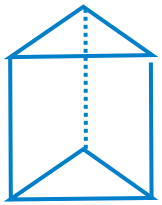
Cylinder
[Weghorst et al., 1985]

Box, AABB (R*-trees)
[Beckmann, Kriegel, et al., 1990]

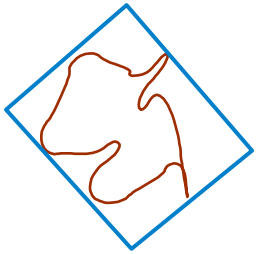


Convex hull
[Lin et. al., 2001]

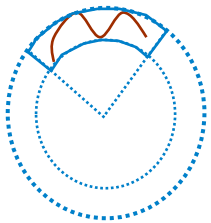
Sphere
[Hubbard, 1996]



Prism
[Barequet, et al., 1996]

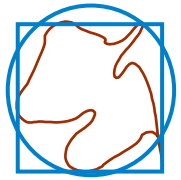


OBB (oriented bounding box)
[Gottschalk, et al., 1996]



Spherical shell
[...]

k-DOPs / Slabs
[Zachmann, 1998]

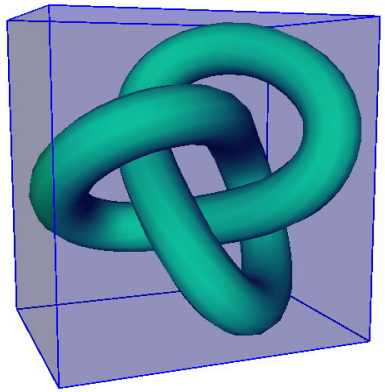


Schnitt mehrerer
anderer BVs

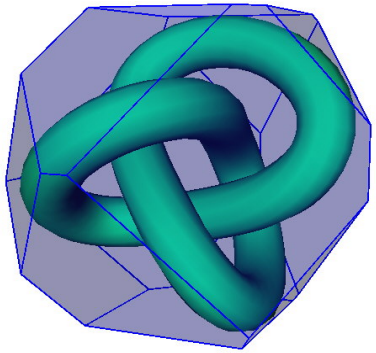
k-DOPs

- Examples:

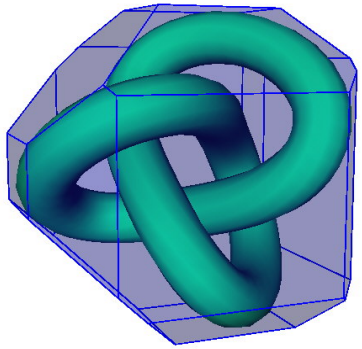
6-DOP
(AABB)



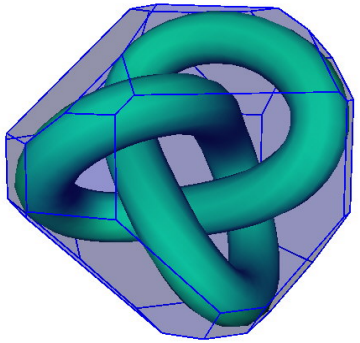
14-DOP



18-DOP



26-DOP



The Costs of BVs

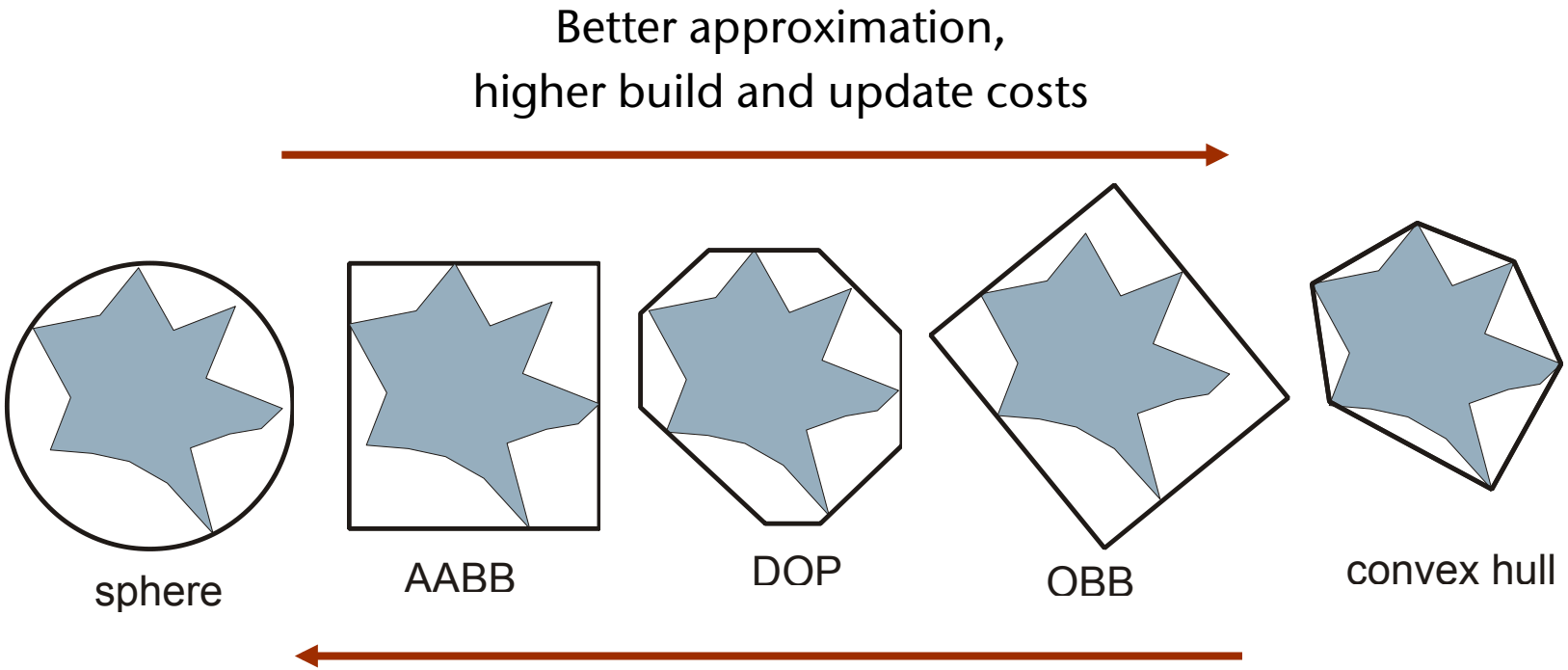
- Costs of a ray intersection with a subset of the scene, enclosed in a BV:

$$T = n \cdot B + m \cdot I$$

T = total costs
n = number of rays tested against the BV
B = costs for one ray-BV intersection test
m = number of rays that actually intersect the BV
I = costs for testing the objects in the BV

- Goal: minimze T
- Consequence: 2 incompatible requirements on BVs:
  - BVs should be simple (e.g., sphere or box) = small costs for ray tests, B; downside: number of ray hits, m, is usually large
  - BVs should be compact (e.g., exact, convex hull) = small m; downside: intersection costs, B, are high

- Qualitative comparison:

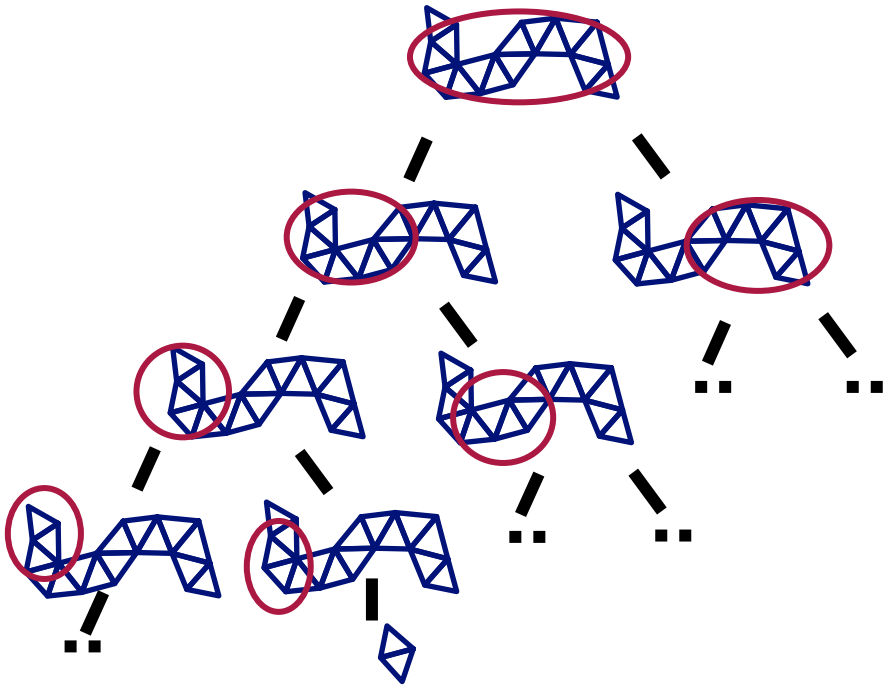


The Bounding Volume Hierarchy (BVH)

- Definition:
  A BVH over a set of primitives, $\mathcal{P}$, is a tree where each node are associated
  - a subset of $\mathcal{P}$; and
  - a BV $\mathcal{B}$, that encloses all primitives in the subset.
- Remark:
  - Often, we use the BV as a synonym for the node in the BVH
  - Primitives are usually stored only at child nodes
    - Feel free to experiment; exceptions can make sense
  - Most of the time, primitives are partitioned, i.e., if $\mathcal{P}$ is the set of primitives associated with a node, and $\mathcal{P}_i$ are the subsets of primitives associated with the children, then

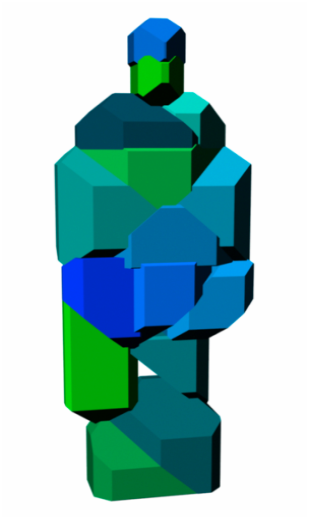
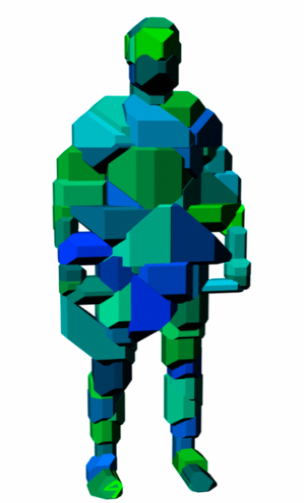
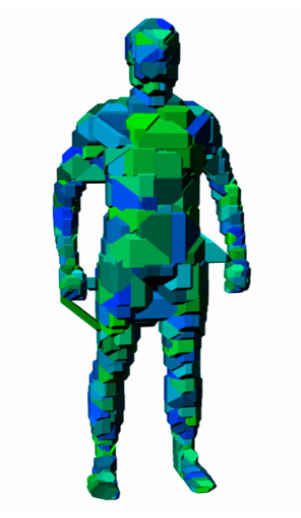
$$\mathcal{P} = \mathcal{P}_1 \dot{\cup} \ldots \dot{\cup} \mathcal{P}_n$$

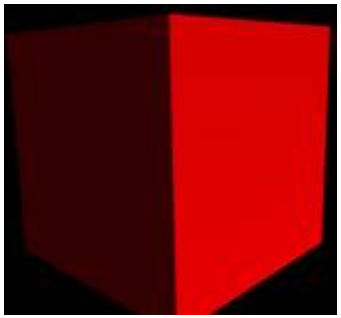
- Schematic example:

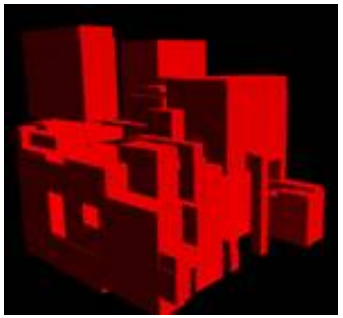
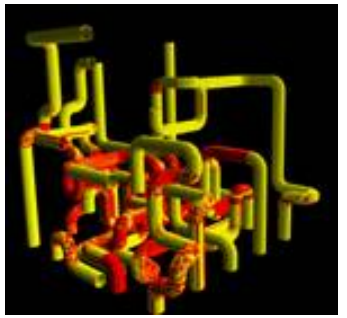


- Parameters:
  - The kind of BV used
  - *"Arity"* (degree of the nodes)
  - Stopping criterion (inparticulr, number of triangles per leaf)
  - Criterion for partitioning the primitives (guiding the construction)

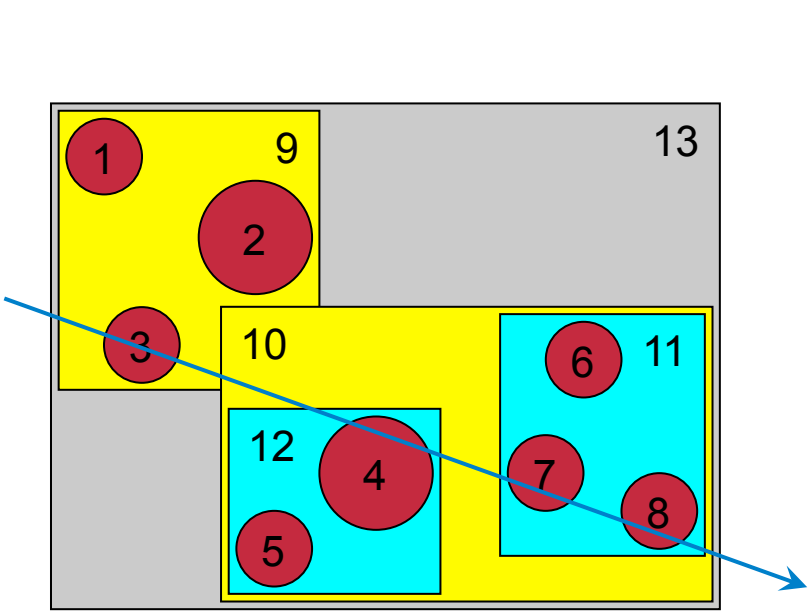
Examples

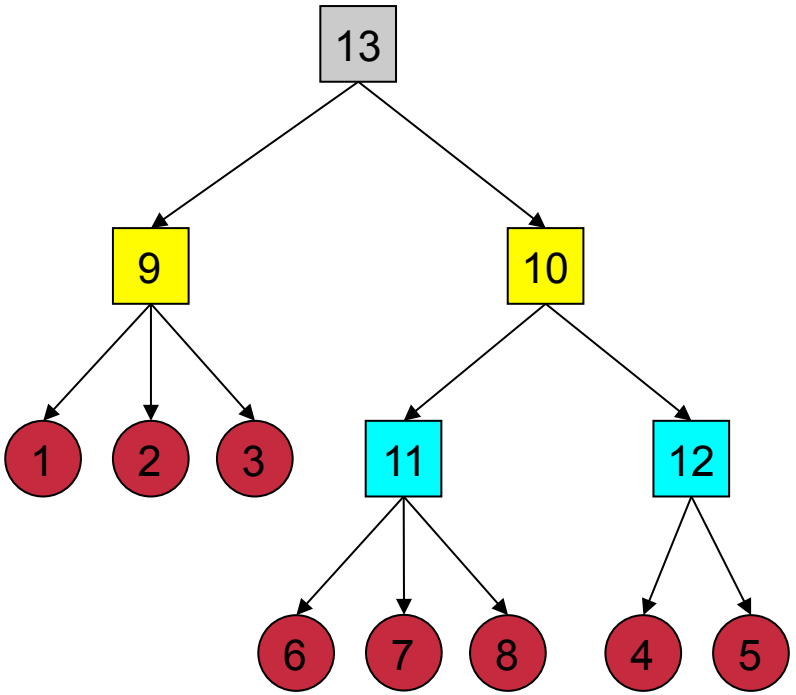




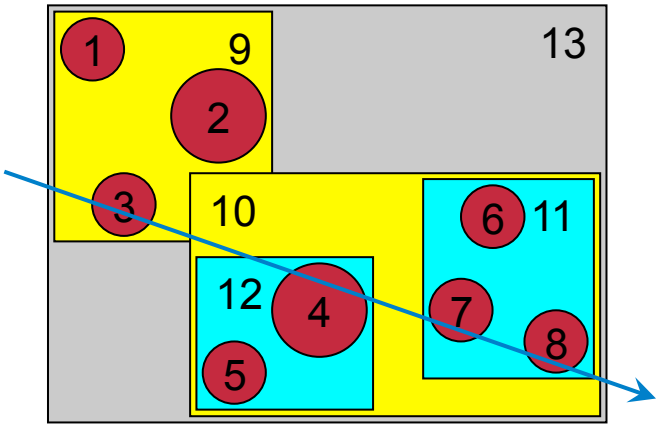


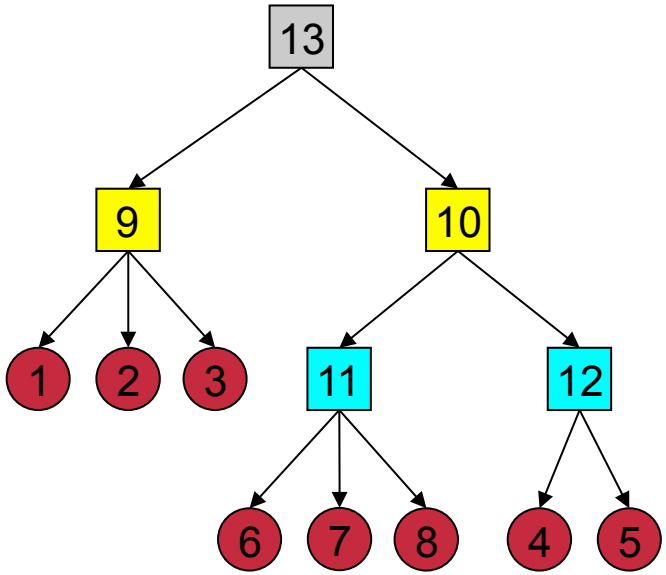



Example for the Traversal of a BVH with a Ray









- Test 13 ⟶ yes
  - Test 9 ⟶ yes
    - Test 1 ⟶ no
    - Test 2 ⟶ no
    - Test 3 ⟶ yes
  - Test 10 ⟶ yes, but intersection point is farther away

➢ Result: only 3 instead of 8 tests with objects, plus 3 tests with BVs

- Question: why did we start with BV 9?

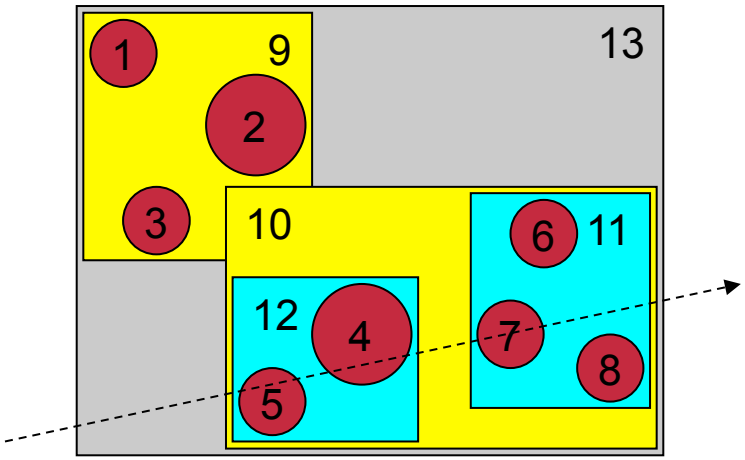
Better Hierarchy Traversal

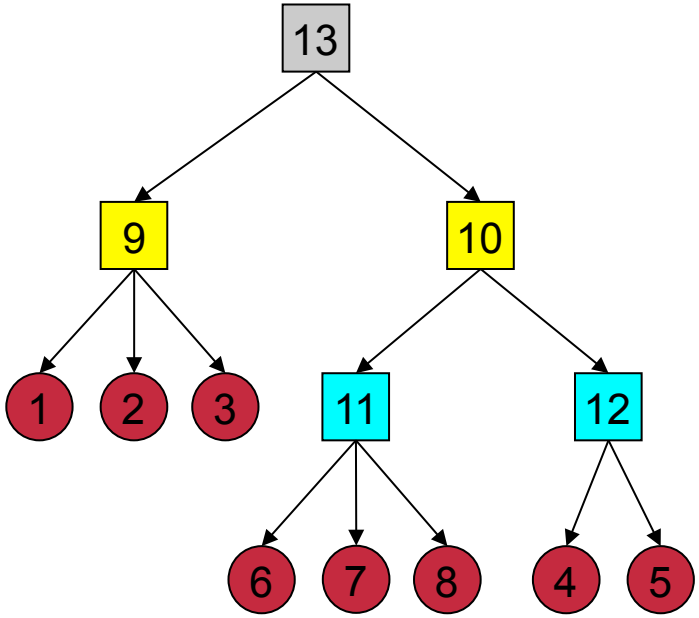
- Problem: the order by which nodes are visited with pure depth-first search (DFS) depends *only* on the topology of the tree
- Better: consider the spatial layout of the BV's, too
- Criterion: distance between origin of ray and intersection with BV (*estimated distance*)
- Consequence: can't use simple recursion / stack any more
- Use **priority queue**

Algorithm

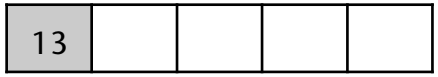
- Berechne die Distanz zwischen dem Strahlursprung und dem Schnittpunkt eines Strahls mit dem aktuell besuchten BV
- Ist die Distanz größer als die Distanz zu einem bereits gefundenen Schnittpunkt mit einem Obj, so kann dieses BV und dessen Teilbaum ignoriert werden
- Sonst: Rekursion
- Sortiere alle noch zu testenden BVs gemäß ihrer Distanz zum Strahlursprung in einem Heap
  - Einfügen eines Elementes und Extrahieren des minimalen Elements haben Aufwand von O(log n)
- Als nächster Kandidat wird immer dasjenige BV gewählt, das dem Strahlursprung am nächsten ist

Example

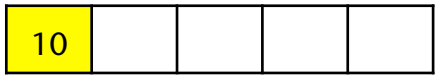




- Test with 13 ⟶ yes, insert

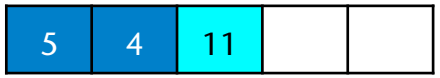


- Pop front of queue ⟶ 13
  - Test with 9 ⟶ no
  - Test with 10 ⟶ yes, insert

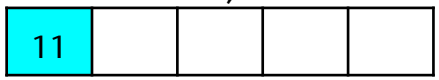


- Pop front of queue ⟶ 10
  - Test with 11 ⟶ yes
  - Test with 12 ⟶ yes

- 12 herausnehmen
  - Schnitt mit 4 → Ja
  - Schnitt mit 5 → Ja



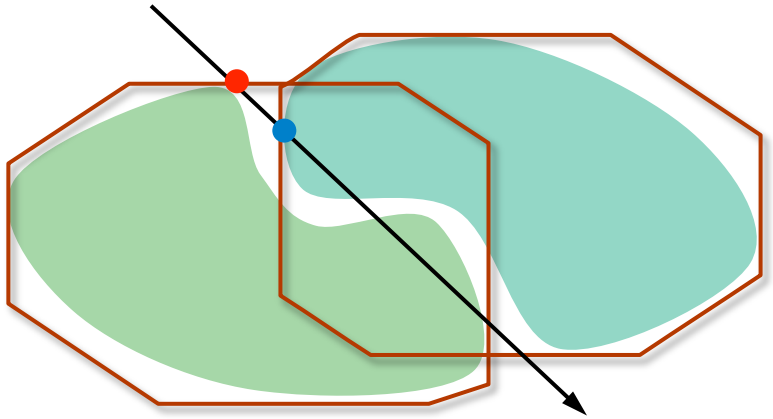
- 5 herausnehmen, Test mit Primitiv
- 6 herausnehmen, Test mit Primitiv



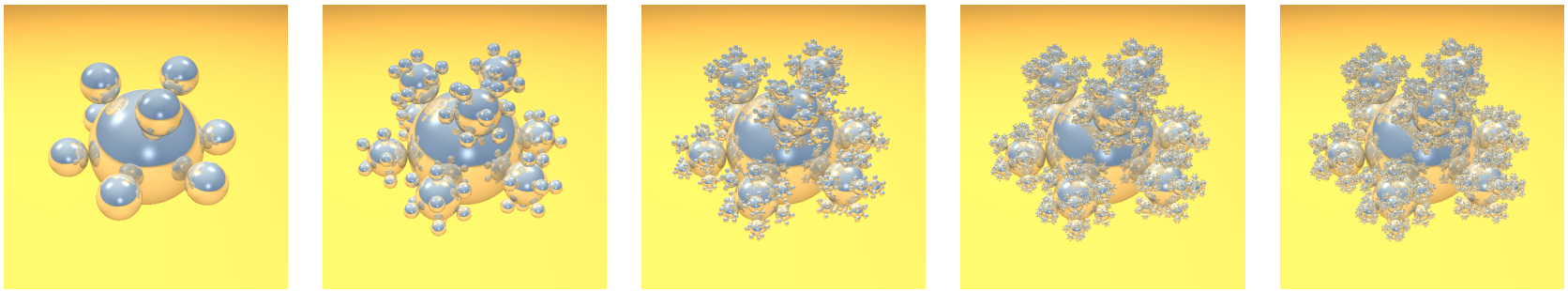
- 11 herausnehmen ...

Remarks

- We don't need a complete ordering among the BV's in the priority queue, because in each step, we only need to extract the BV that has the closest intersection (among all others in the queue)
- This can be implemented efficiently with a **heap**
- **Warning:** the closest ray-BV intersection and the closest ray-primitive intersection can happen in different BV's!



How Much Do We Gain?

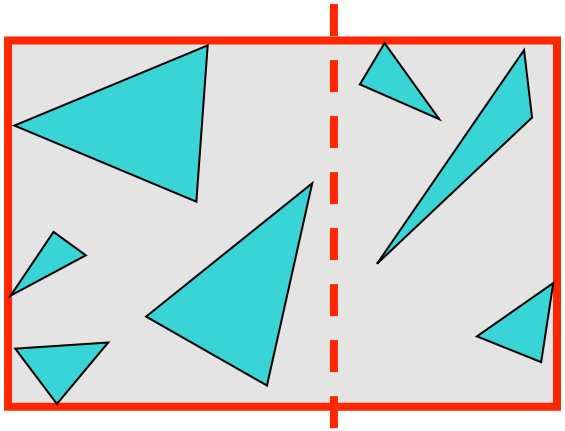


Number of spheres	10	91	820	7381	66430
Brute-force	2.5	11.4	115.0	2677.0	24891.0
Goldsmith/ Salmon BVH	2.3	2.8	4.1	5.5	7.4

Rendering times in seconds, Athlon XP 1900+
(Markus Geimer)

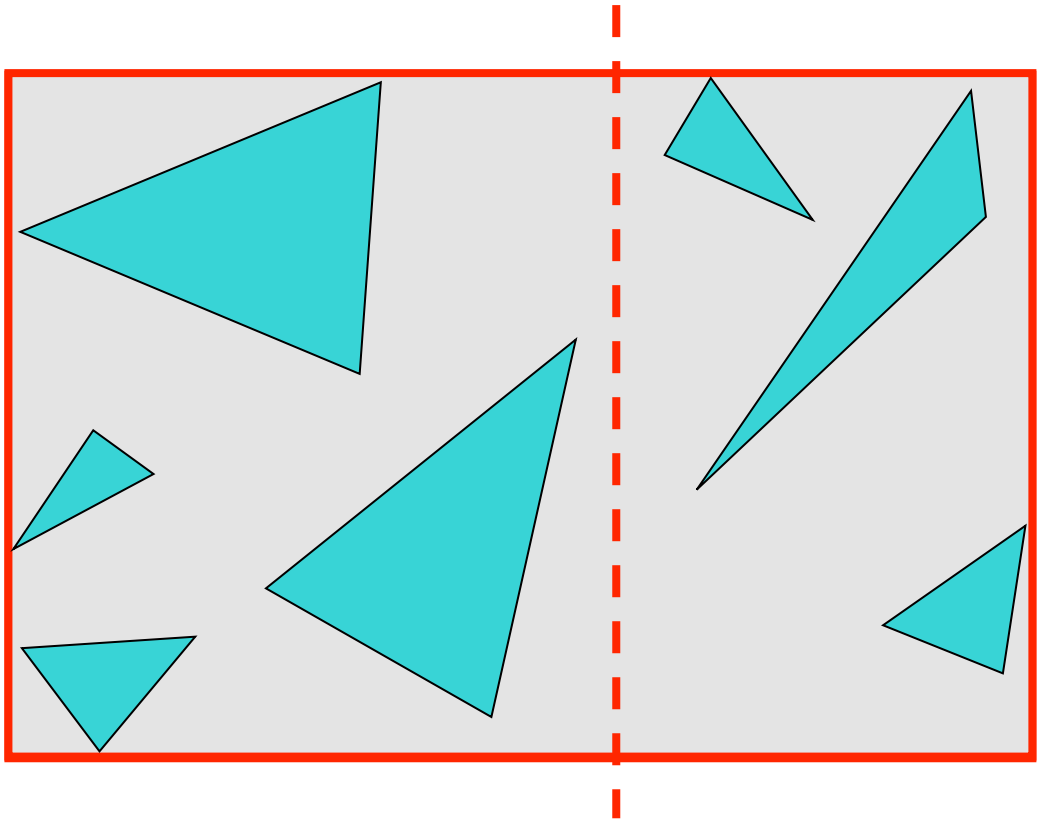
The Construction of BV Hierarchies

1. Given by modeling process (e.g., in form of scene graph)
2. Bottom-up:
   - Recursively combine objects/BV's and enclose in (larger) BV
   - Problem: how to choose the objects/BV's to be combined?
3. Top-down:
   - Partition the set of primitives recursively
   - Problem: how to partition the set?
4. Iterative Insert:
   - Heuristic developed by Goldsmith/Salmon



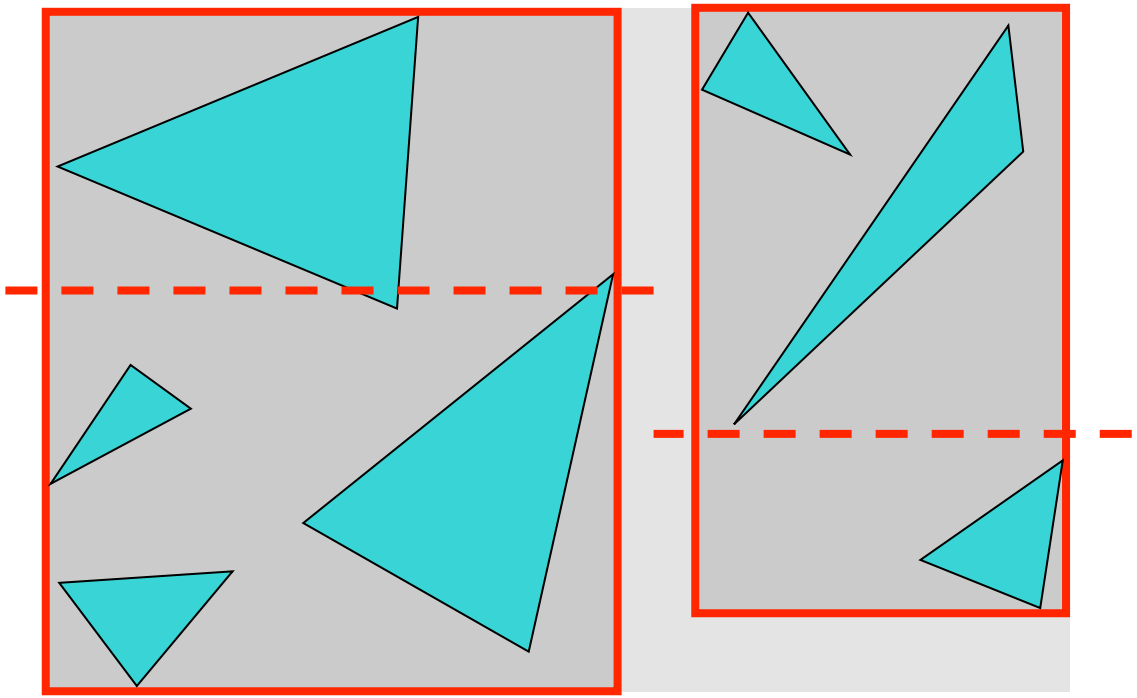
Example for the Construction of a BVH

- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse



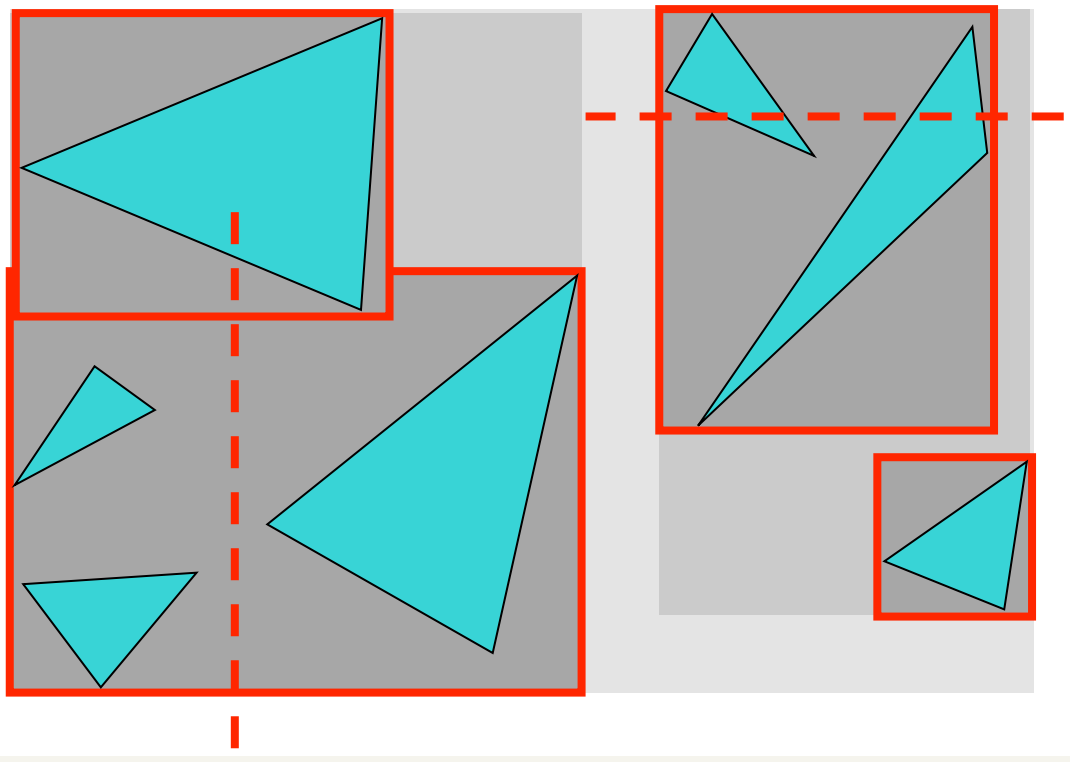
Example for the Construction of a BVH

- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse



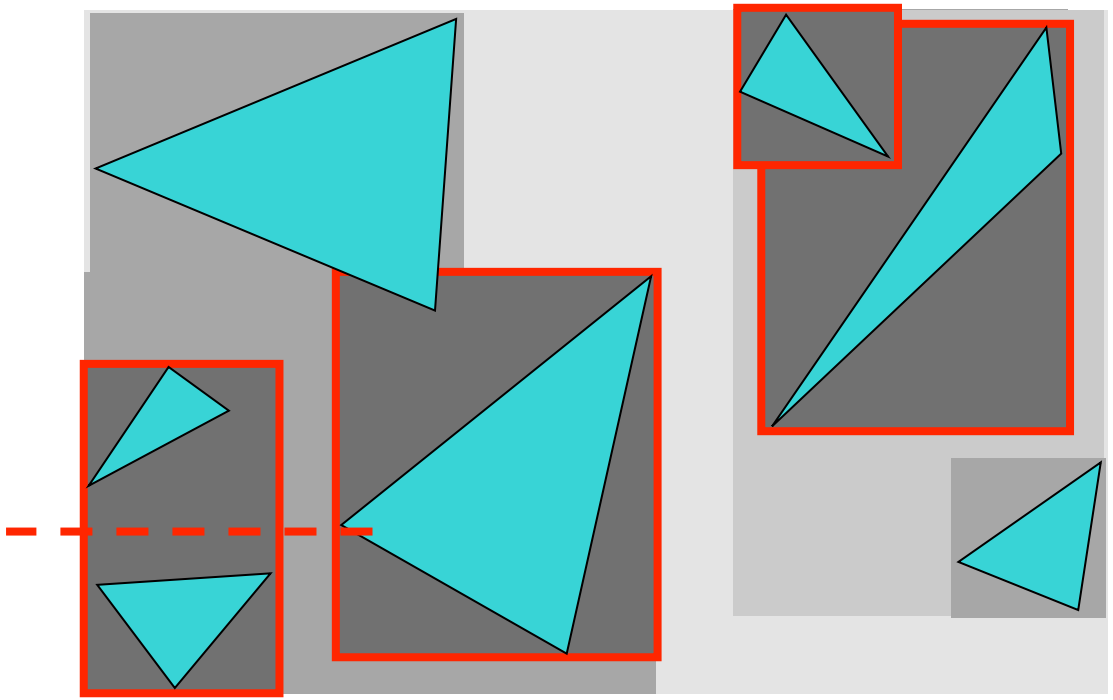
Example for the Construction of a BVH

- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse



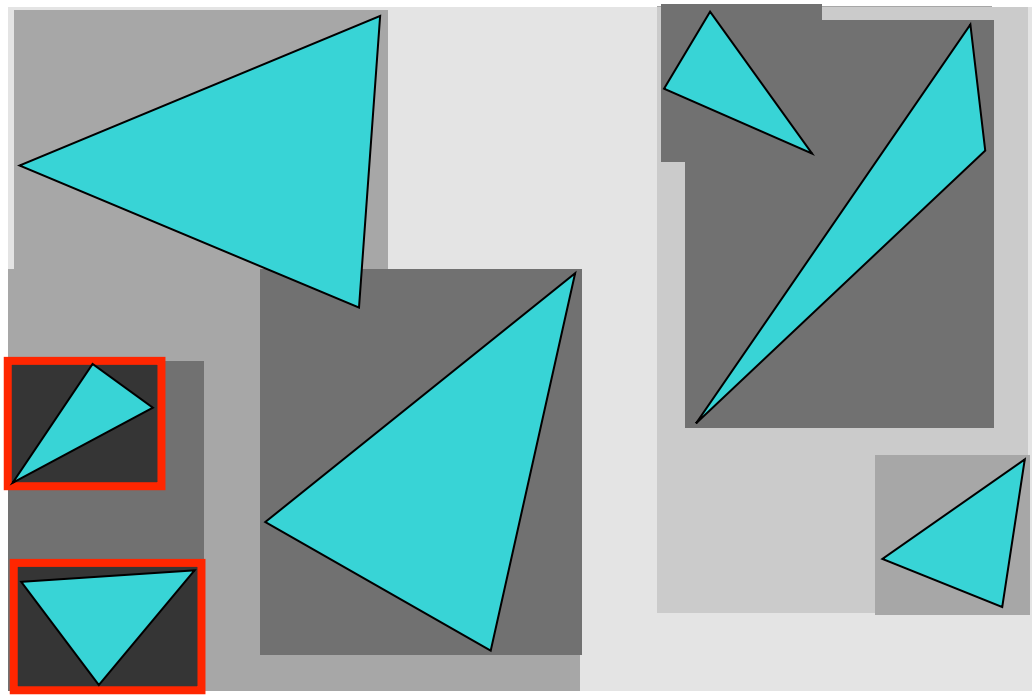
Example for the Construction of a BVH

- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse



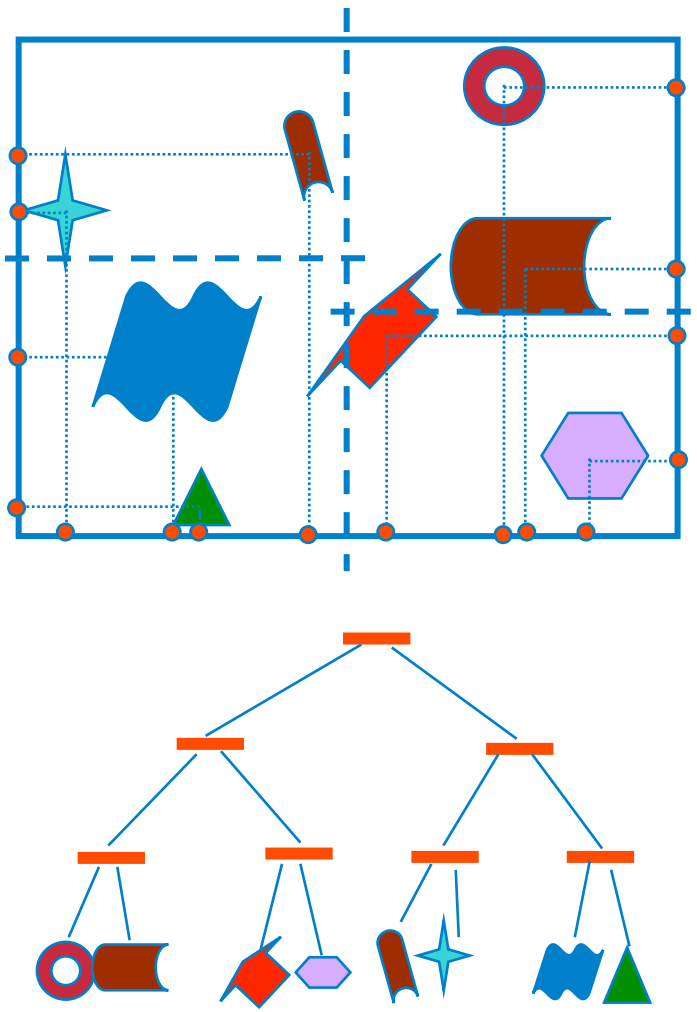
Example for the Construction of a BVH

- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse



Simplest Heuristic for Top-Down Construction: Median Cut

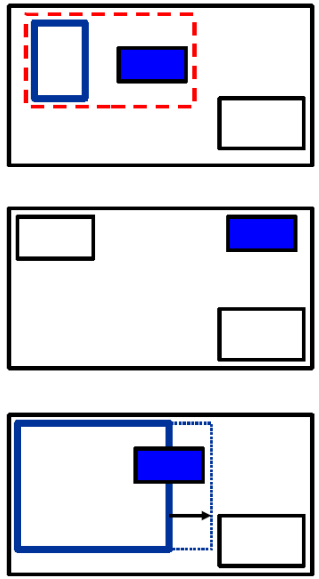
1. Construct BV around all objects
2. Sort all objects according to their "center" along the x-axis
3. Partition the scene along the median on the x-axis; assign half of the objects to the left and the right sub-tree, resp.
   1. Variant: cyclically choose a different axis on each level
   2. Variant: choose the axis with the longest extent
4. Repeat 1-3 recursively
   - Terminate, when a node contains less than *n* objects



Iterative Insert

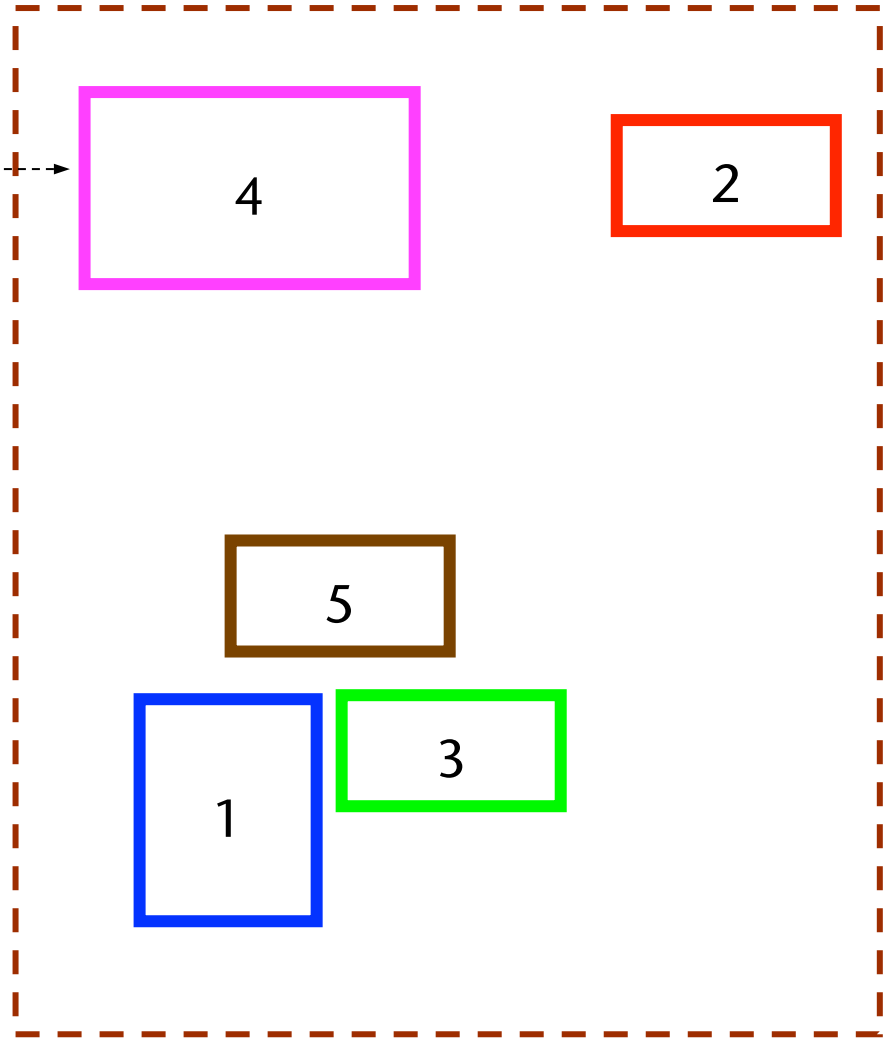
[Goldsmith & Salmon, 1987]

- Start with an empty root node
- Iteratively insert one triangle after another into the BVH, possibly thereby extending the BVH:
  - Let the triangle "sift" to the bottom of the BVH
    - Vergrößere dabei ggf. das BV der Knoten
  - Ist das Dreieck an einem Blatt angekommen →
    - Ersetze das Blatt durch einen inneren Knoten
    - füge das neue und das alte Dreieck als dessen Kinder an
  - Steht man an einem inneren Knoten → treffe eine der folgenden Entscheidungen:
    - füge das Dreieck am aktuellen (inneren) Knoten an
    - lasse das Dreieck in den linken / rechten Teilbaum sickern



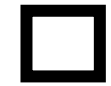
Beispiel für Goldsmith und Salmon

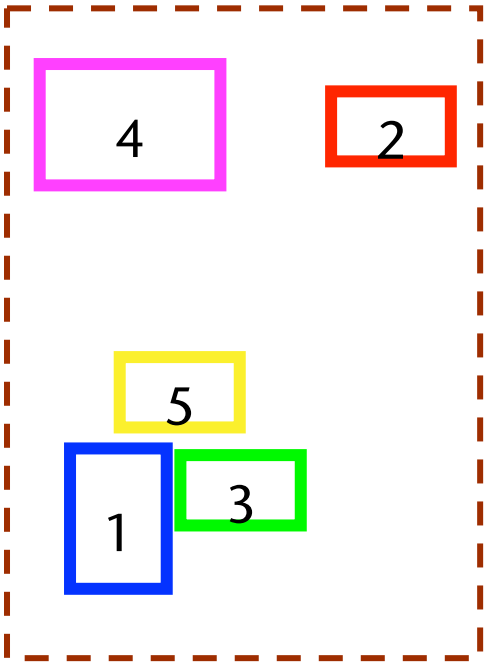
- Szene vor der Erzeugung der Hierarchie
- Jedes Objekt wird durch sein Bounding Volume umgeben
- Das gestrichelte Viereck ist die gesamte Szene



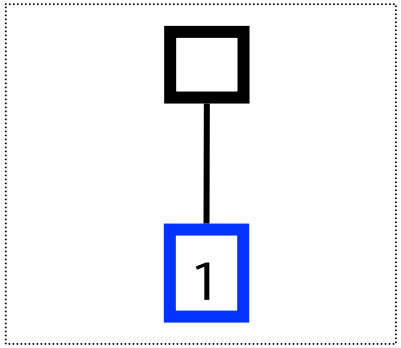
1. Iteration

Gegenwärtiger Baum



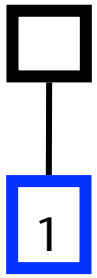


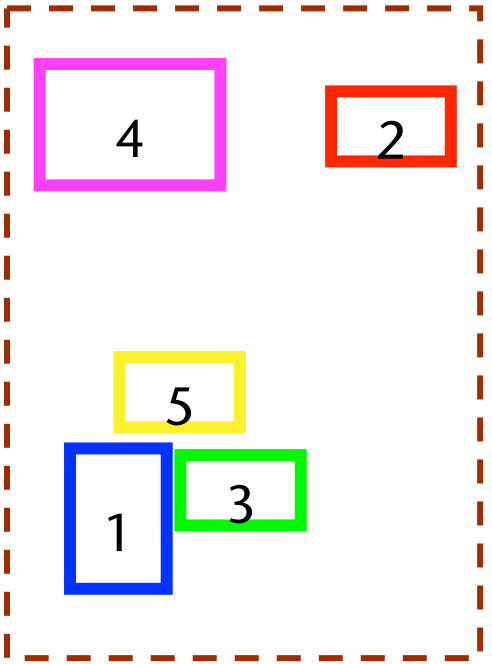
Möglichkeiten



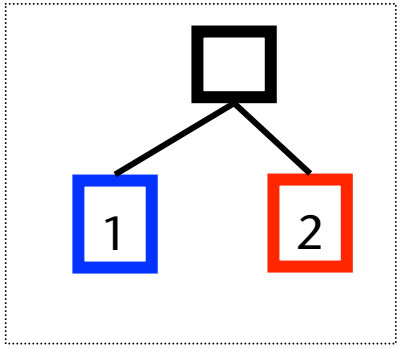
2. Iteration

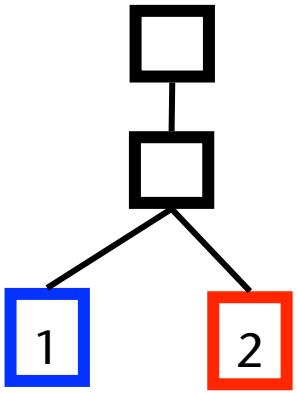
Gegenwärtiger Baum





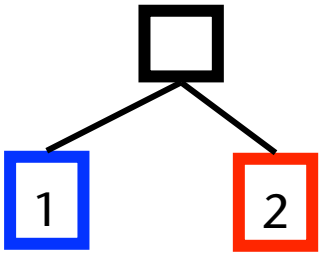
Möglichkeiten

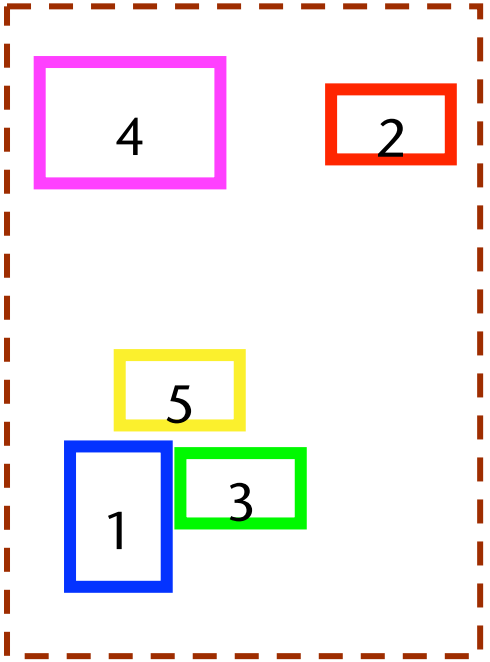




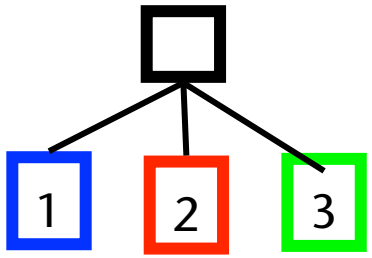
3. Iteration

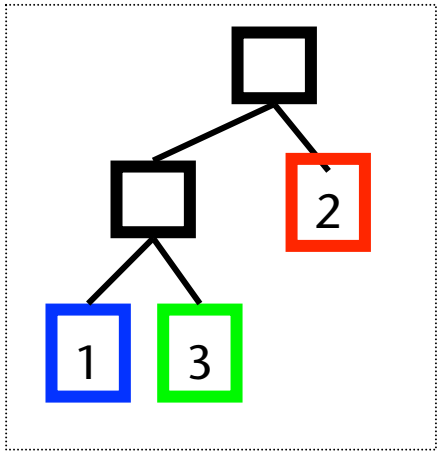
Gegenwärtiger Baum

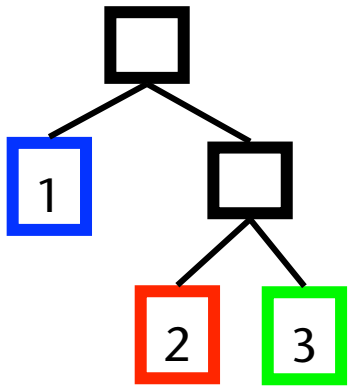




Möglichkeiten

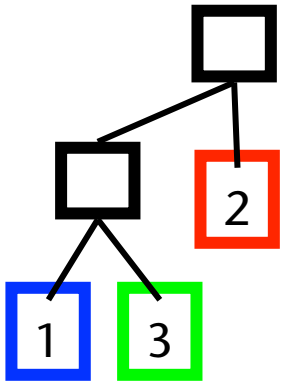


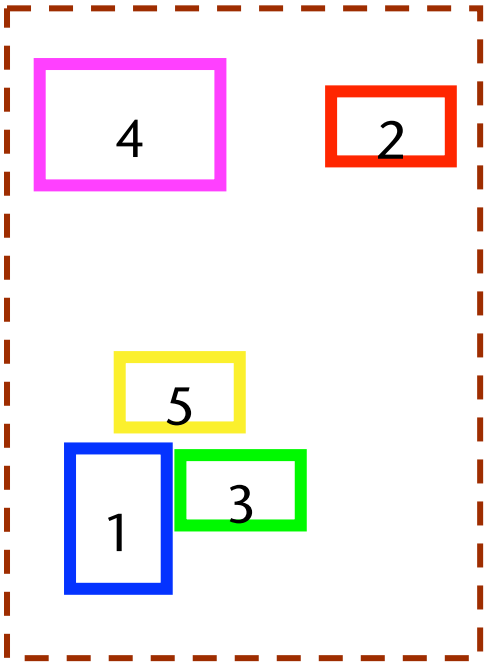




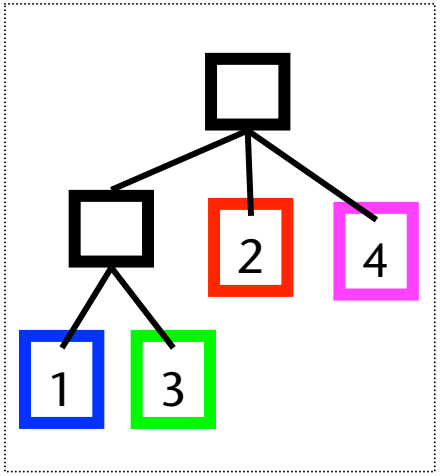
4. Iteration

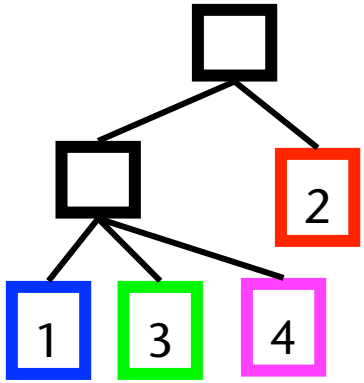
Gegenwärtiger Baum

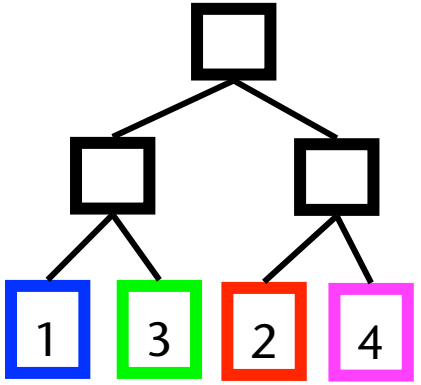




Möglichkeiten

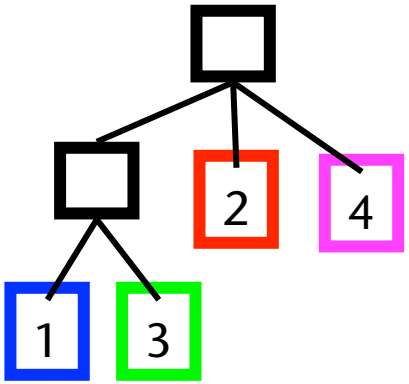


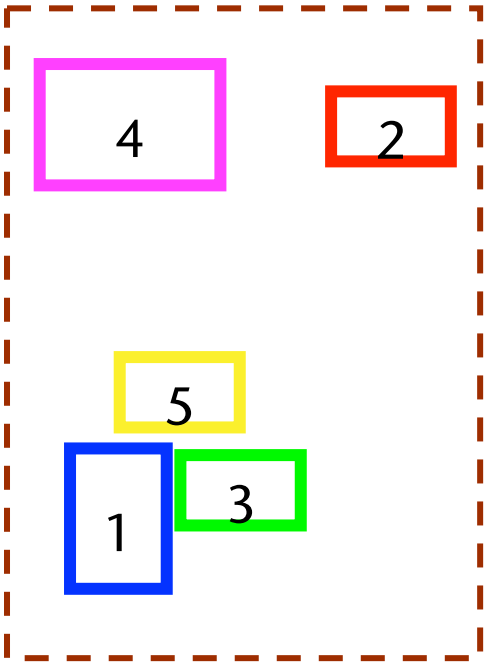




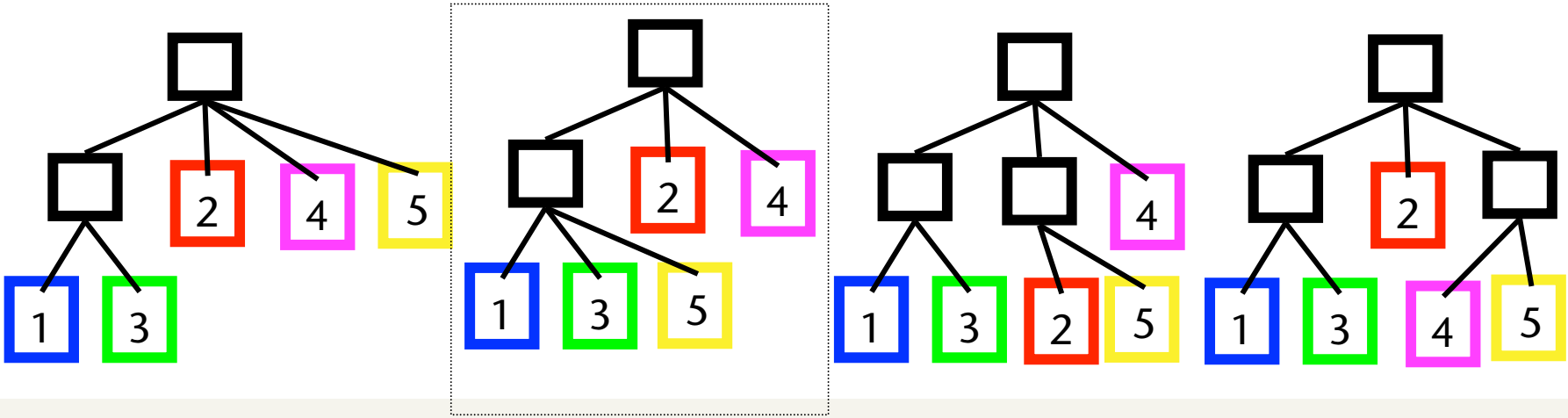
5. Iteration

Gegenwärtiger Baum



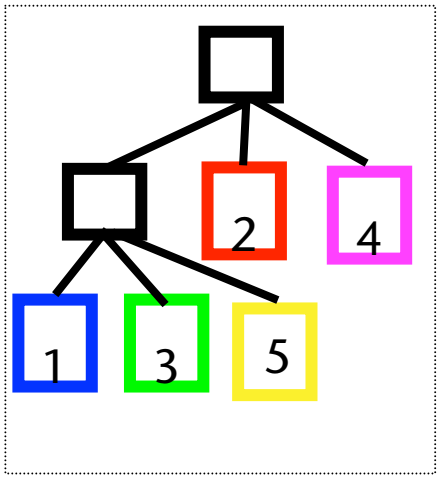


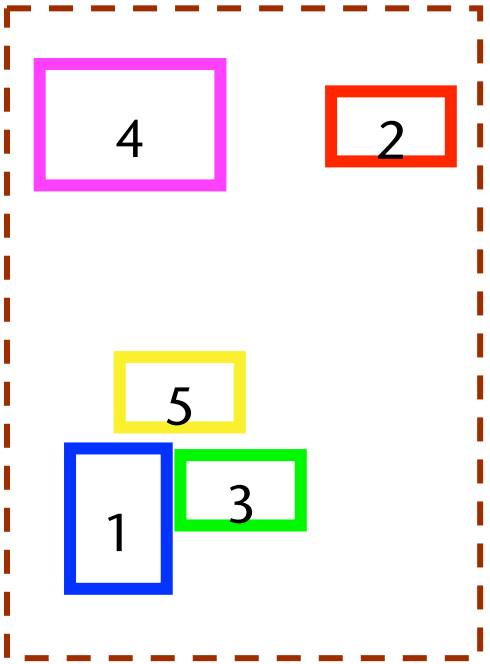
Möglichkeiten



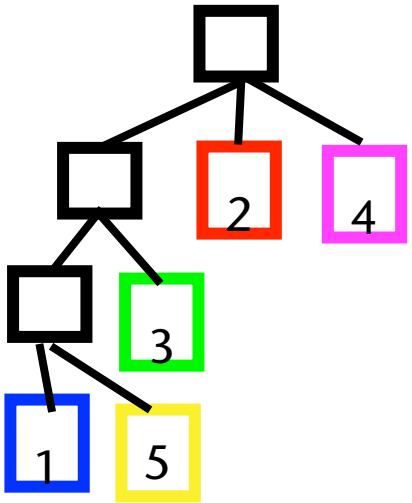
5. Iteration

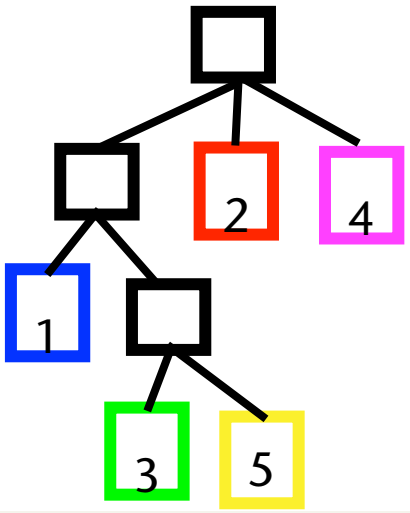
Gegenwärtiger Baum





Möglichkeiten





Bemerkungen

- Die Reihenfolge, in der die Objekte eingefügt werden, hat einen sehr großen Einfluss darauf, wie gut der Baum wird
- Goldsmith/Salmon experimentierten mit:
  - Reihenfolge wie im geladenen Modell
  - zufällig (shuffled)
  - Sortiert entlang einer Koordinatenachse

Zahl der Schnitt-Berechnungen pro Strahl bei verschiedenen Testszenen

User Supplied	5.94	19.9	12.9	10.1	32.0	63.2
Sorted	6.53	20.0	15.9	13.3	32.0	55.2
Average Shuffled	6.21	19.9	14.3	9.4	40.5	44.8
Best Shuffled	5.94	19.9	12.4	8.7	36.7	42.4
Worst Shuffled	6.32	19.9	17.4	18.3	48.2	47.2

Die entscheidende Frage

- Bei Salmon/Goldsmith (inkrementell):
  Zu welchem Teilbaum soll ein Dreieck hinzugefügt werden?
- Bei top-down Aufbau:
  Welches ist, zu einer geg. Menge von Dreiecken, die optimale Aufteilung in zwei Teilmengen? (wie bei kd-Tree)

- Verwende die Surface-Area-Heuristic (SAH):
  teile B so auf, daß

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

  minimal wird

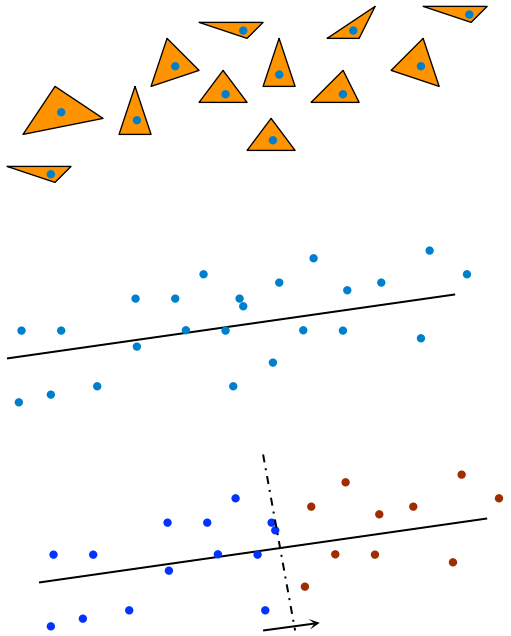
- Anwendung auf Salmon/Goldsmith:
  - Propagiere das Objekt in denjenigen Unterbaum, der dadurch die geringste Kostenerhöhung für das Ray-Tracing verursacht
  - Falls beide die gleichen Kosten verursachen (z.B. 0), verwende eine andere Heuristik, z.B. Anzahl Dreiecke im Teilbaum
  - Falls alle Unterbäume zu hohe Kosten verursachen (z.B. Flächen-zunahme auf 90% der Fläche von Vater), hänge Objekt als direktes Kind an den aktuellen Knoten (BVH ist also nicht notwendig binär)

- Anwendung auf rekursive top-down BVH-Konstruktion:
  - Berechne BV zu gegebener Menge von Objekten (= elem. BVs)
  - Partitioniere Menge der Objekte in 2 Teilmengen (oder mehr)
  - Konstruiere BVH für jede der Teilmengen
- Gesucht: optimale Aufteilung

$$C(B) = \min_{B' \in \mathcal{P}(B)} C(B', B \setminus B')$$

  wobei B = Menge der Polygone im Vater-BV
- Ist natürlich nicht praktikabel

- Heuristischer Aufbau einer BVH:
  - Repräsentiere Objekte (Dreiecke) durch deren Mittelpunkte
  - Bestimme die Achse der größten Ausdehnung
  - Sortiere die Punkte entlang dieser Achse
  - Suche entlang dieser Achse das Minimum gemäß Kosten-Heuristik mittels Plane-Sweep:



$$k = \arg\min_{j=1\ldots n} \left\{ \frac{\text{Area}(b_1 \ldots b_j)}{\text{Area}(B)} \cdot j + \frac{\text{Area}(b_{j+1} \ldots b_n)}{\text{Area}(B)} \cdot (n-j) \right\}$$

wobei die $b_j \in B$ die elementaren BVs sind und j bzw. (n-j) die Anzahl der Objekte in B_1 bzw. B_2).

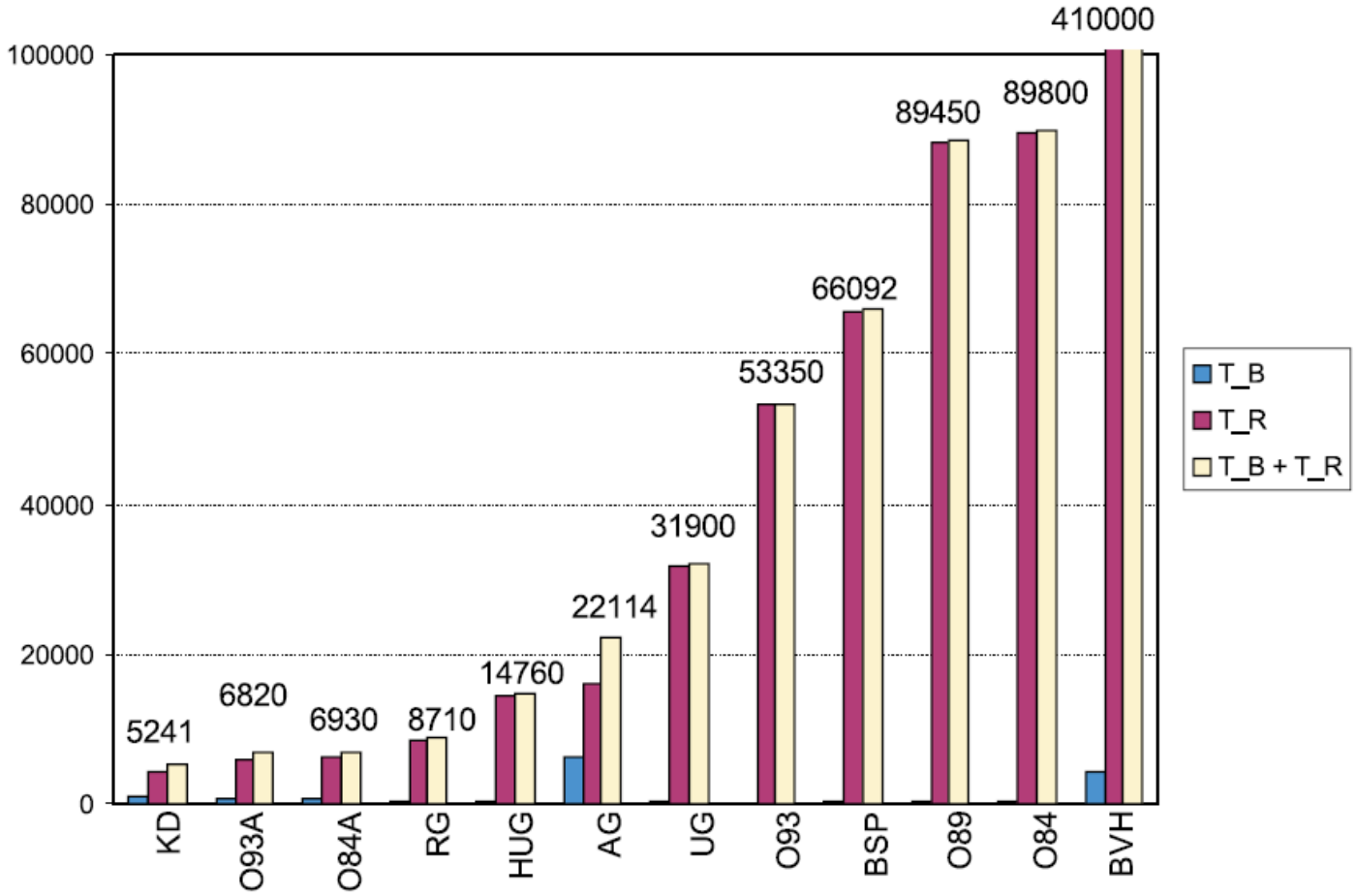
- Laufzeit:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + O(n \log n)$$

$$\in O(n \log^2 n)$$

- Bemerkungen:
  - Abruchkriterium bei top-down Verfahren: analog zum kd-Tree
  - Top-down-Verfahren liefert i.A. bessere BVHs als iteratives Verfahren

Vergleich verschiedener Datenstrukturen [Havran, 2001(?)]



- Achtung: mit Vorsicht genießen!


CG
VR