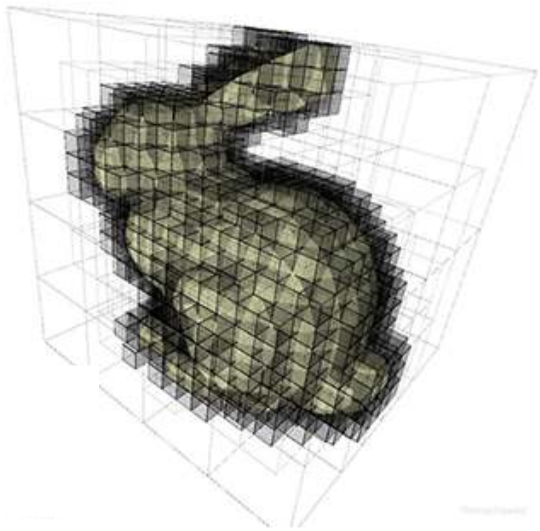
Advanced Computer Graphics

Acceleration Data Structures (a.k.a. Spatial Indexes) with Application to Raytracing et al.

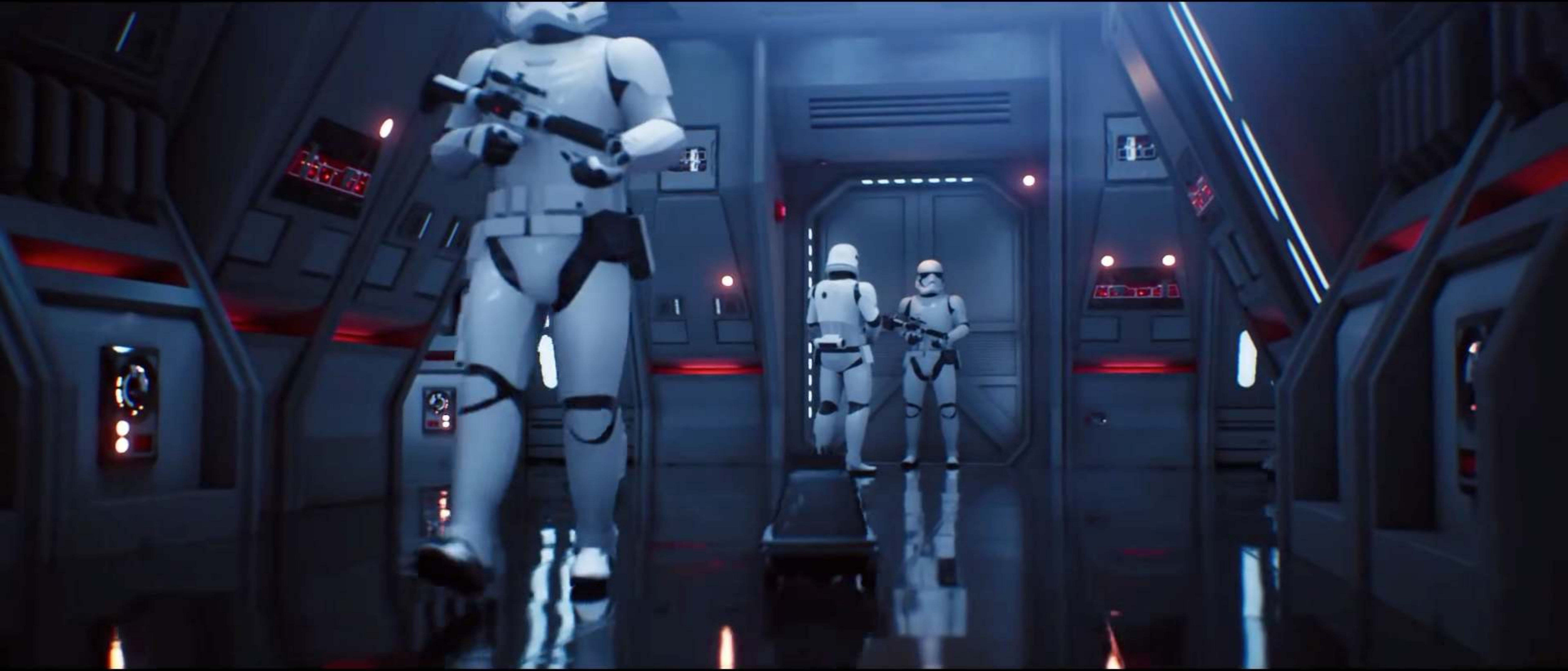


G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

"But is it real-time?"

- Ray Tracing used to be very slow (and still is slower than polygonal)
- "Perhaps, some day, graphics cards will do ray-tracing only …"   [GZ 2006]

Uni Saarbrücken, 2002

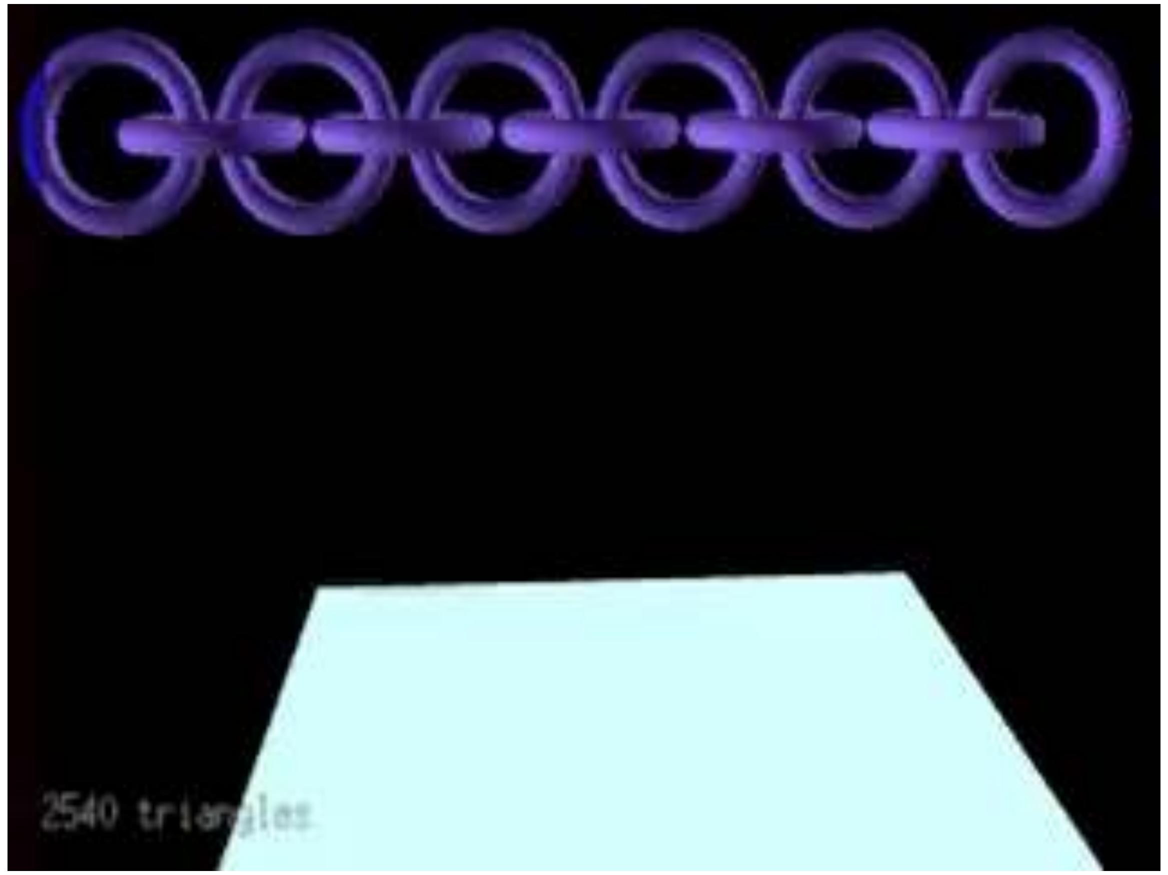


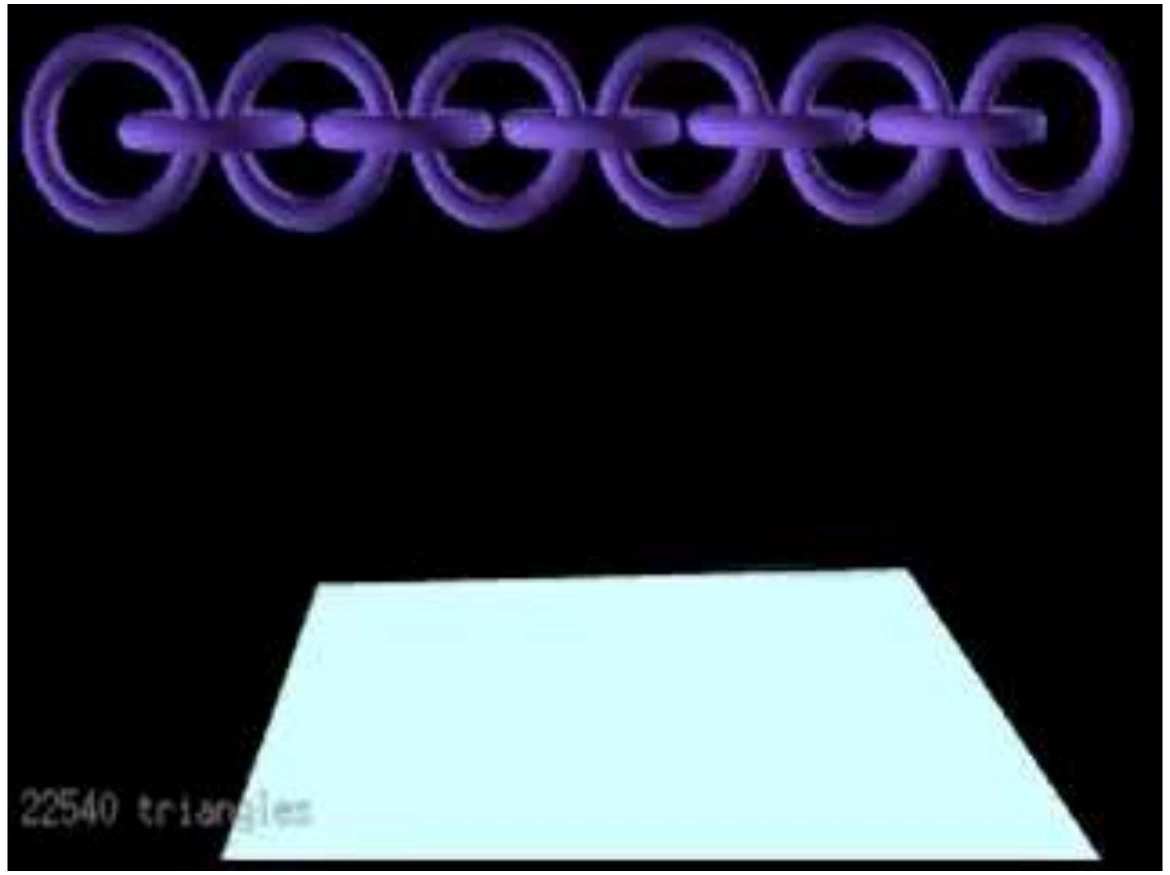
Epic, Nvidia, ILM

Motivation for Acceleration Data Structures

- Rendering animation movies
- Real-time graphics (occlusion culling, point cloud rendering, ...)
- Physics simulation, in particular, collision detection
- Comparison of collision detection with and without acceleration DS:

No acceleration data structure (test all pairs of polygons)





With acceleration data structure (bbox hierarchy)

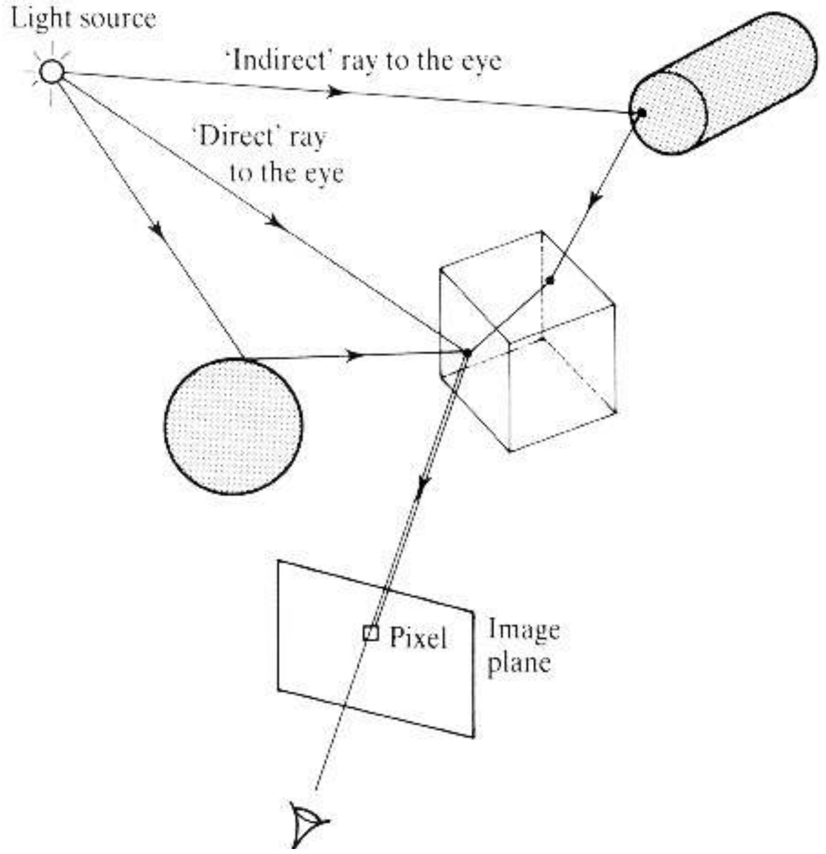
20k triangles

Fun Facts About Animation Movies

Movie	Year	Total render time (on 1 CPU)	Render time per frame	#frames	Size of render farm
Toy Story	1995	800,000 h (91 years)	45 min – 20 hours	110,000	300 CPUs (110 Sun's)
Toy Story 2	1999		10 min –3 days	120,000	1400 proc's
Final Fantasy		900,000 days	90 min		1200 CPUs
Big Hero 6	2014	1,000,000 h			55,000 cores

- Comparison of render times 1995 vs 2010 for Toy Story: on average 4 hours per frame in 1995, 3 minutes in 2010.
  - That is roughly a factor 100. According to Moore's Law, it should be a factor 1000. (All assets were exactly the same, but RenderMan was upgraded)
- Facts about Big Hero 6:
  - Renderer: Hyperion, global-illumination, including sub-surface scattering (BSDF's), created by Disney
  - San Fransokyo: 83,000 buildings, 260,000 trees, 215,000 streetlights and 100,000 vehicles
  - The render farm sucks 1.5 MW power
- About Disney's render farm [as of 2014]: archives are currently 4 Pbytes. The average Disney movie consumes about 4 Tbytes asset data. 1 million render hours per day, about 400 render jobs per day.

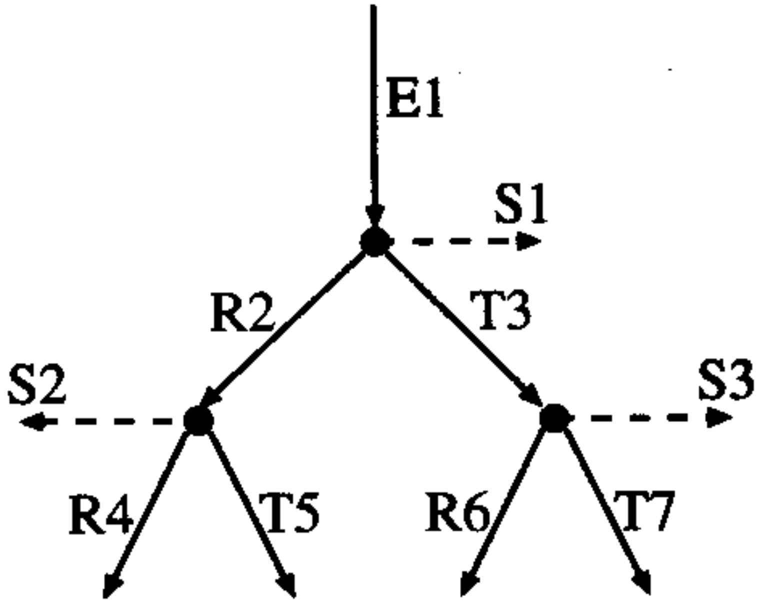
Motivation: the Costs of Ray-Tracing



- #pixels ≈ 2 million (per frame) * 24 FPS * 6000 sec (feature film)
  ≈ 300 billion pixels (x2 for stereo)

- cost per pixel ≈ # primitives tested *
  intersection cost *
  size of recursive ray tree *
  # shadow rays *
  # supersamples *
  # glossy rays *
  # temporal samples *
  # focal samples * . . .

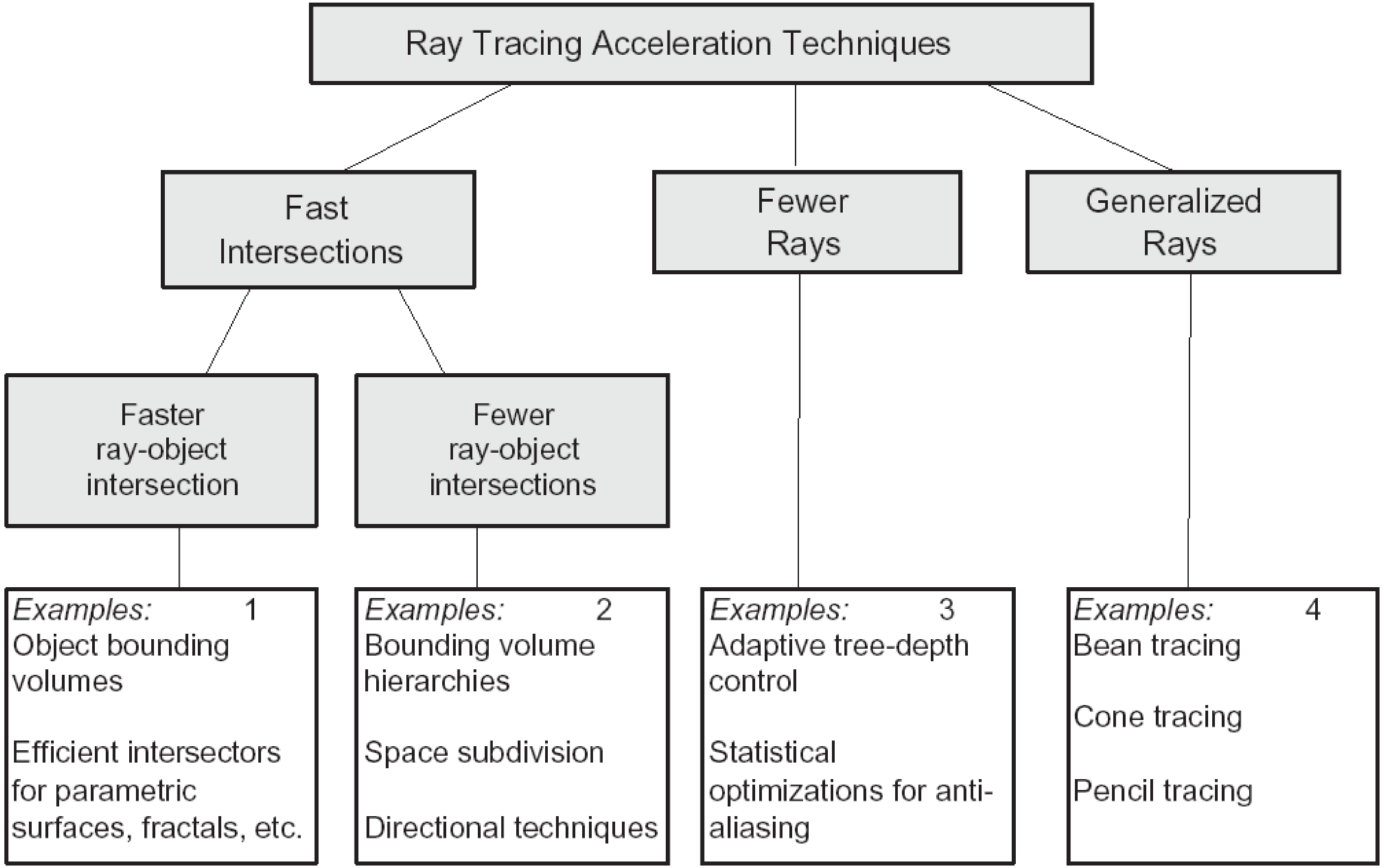
Can we decrease that?



E = primary ray
R = reflected
S = shadow
T = transmitted

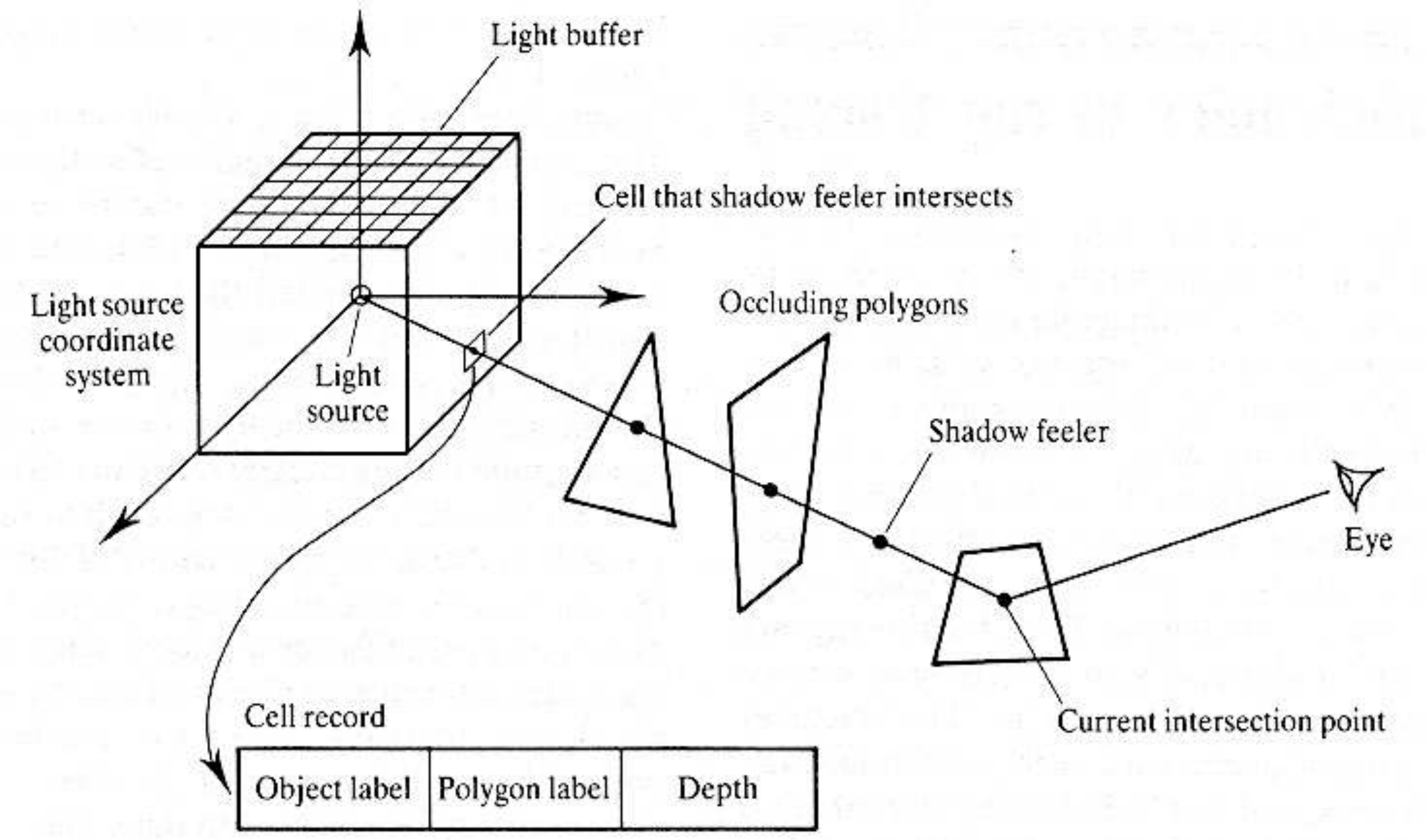
"Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast."
[David Luebke, Nvidia]

A Taxonomy of Acceleration Techniques



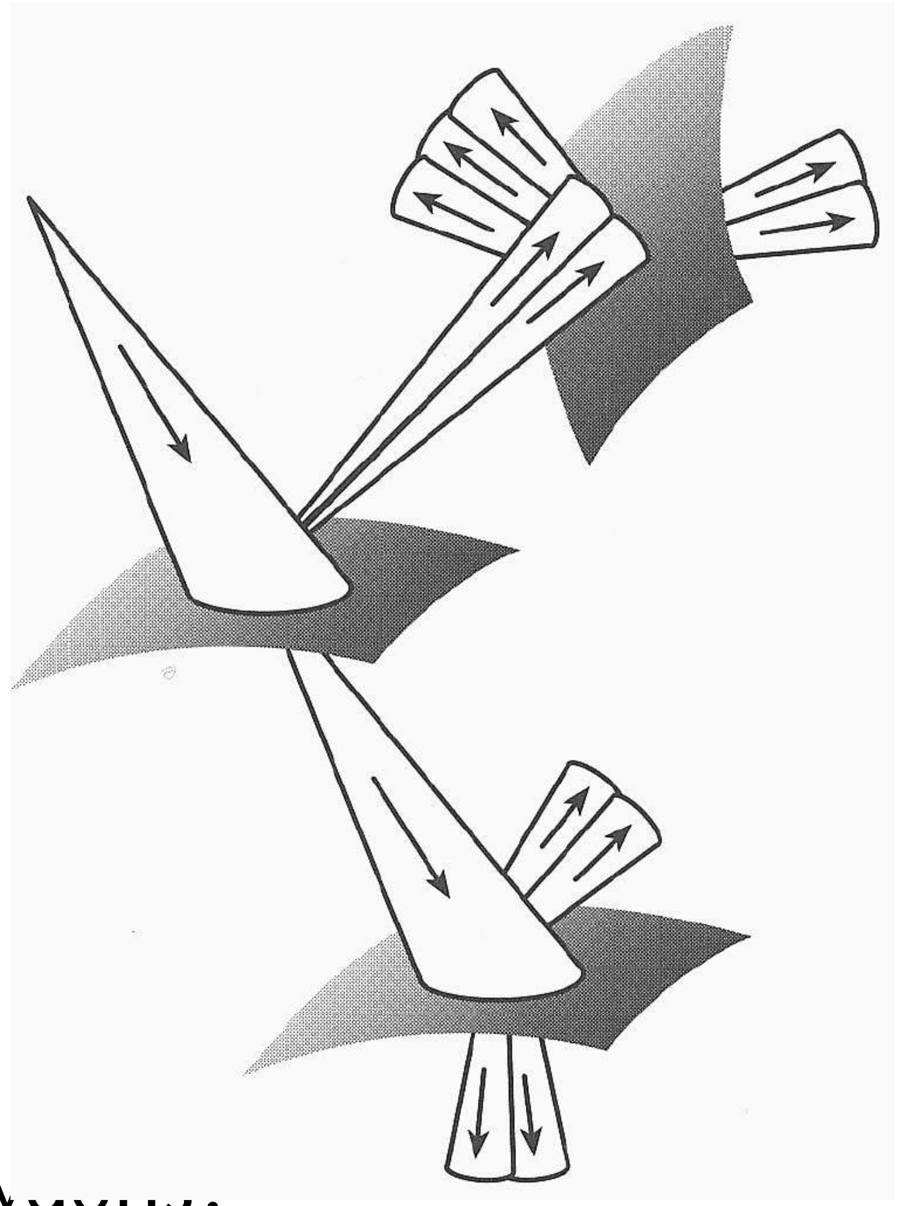
The Light Buffer

- 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 one is found)

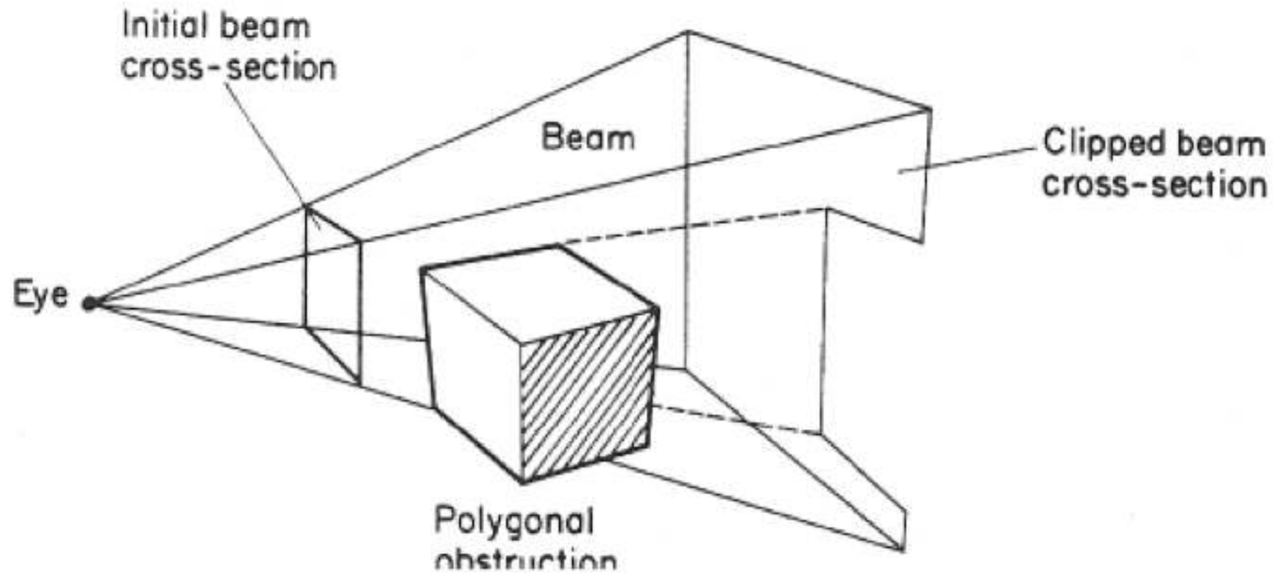


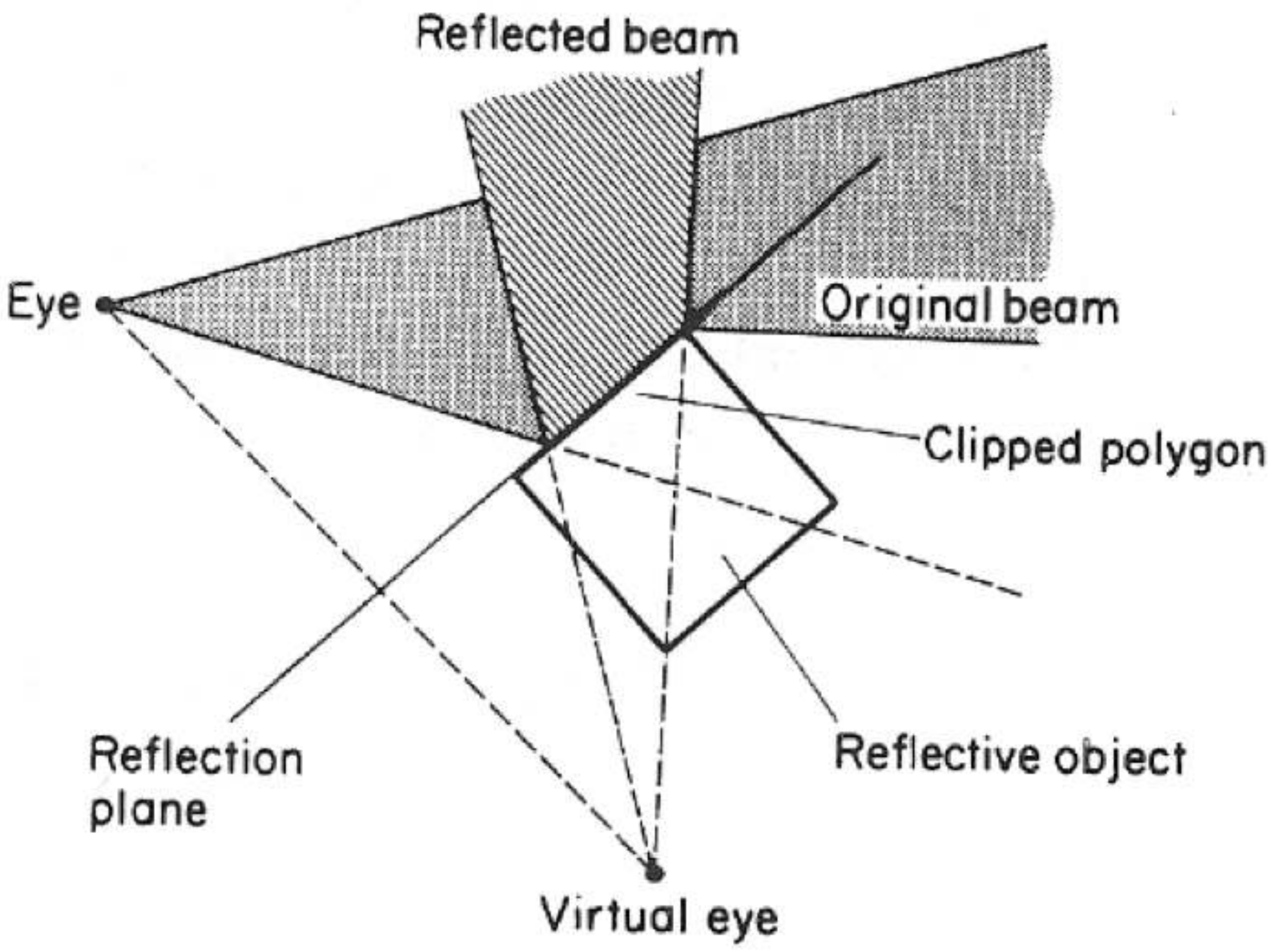
Beam and Cone Tracing

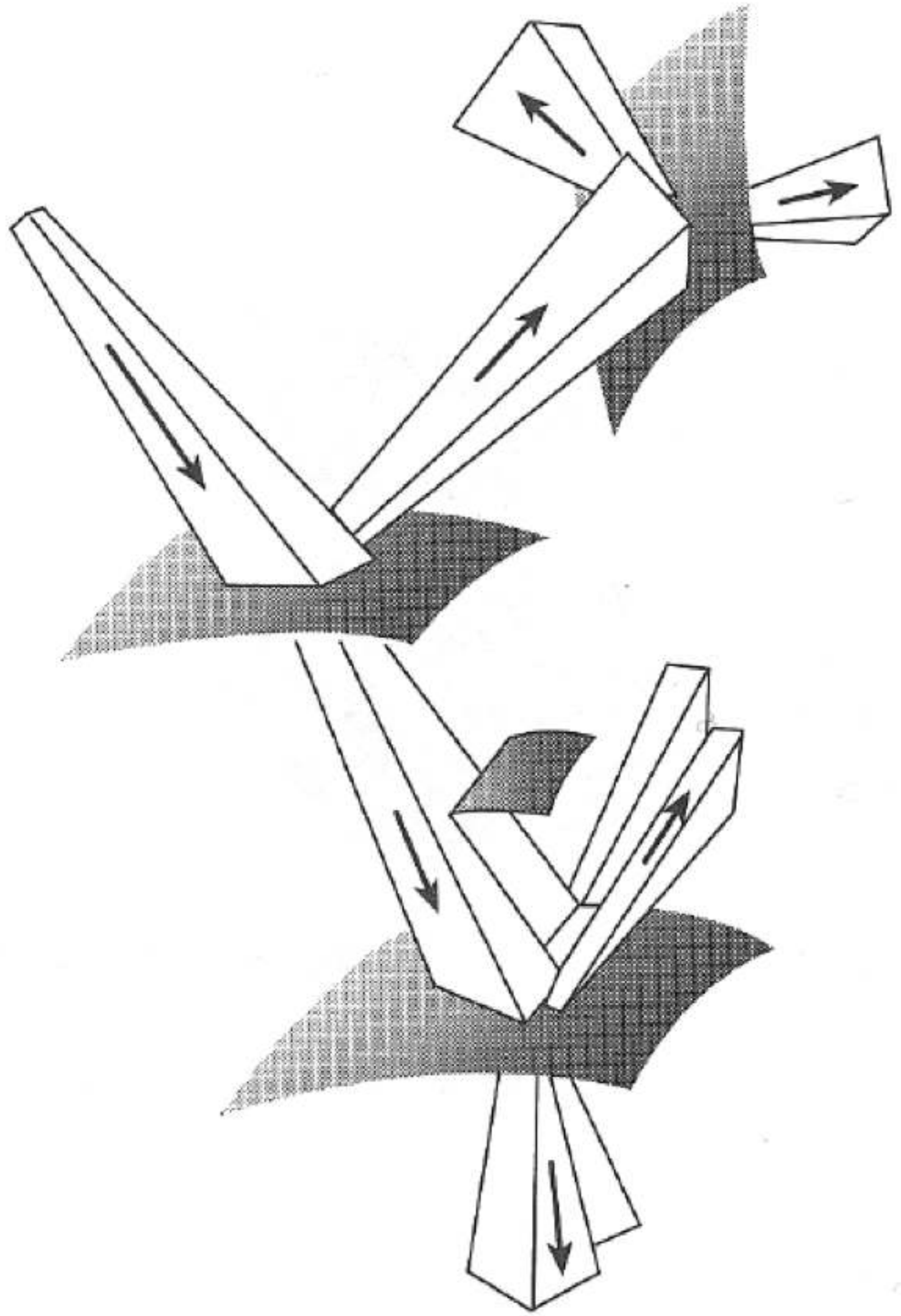
- The general idea: try to accelerate by shooting fewer, but "thick" rays
- Beam Tracing:
  - Represent a "thick" ray by a pyramid
  - At the surfaces of polygons, create new beams
- Cone Tracing:
  - Approximate a thick ray by a cone
  - Whenever necessary, split into smaller cones
- Problems:
  - What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



Beam Tracing

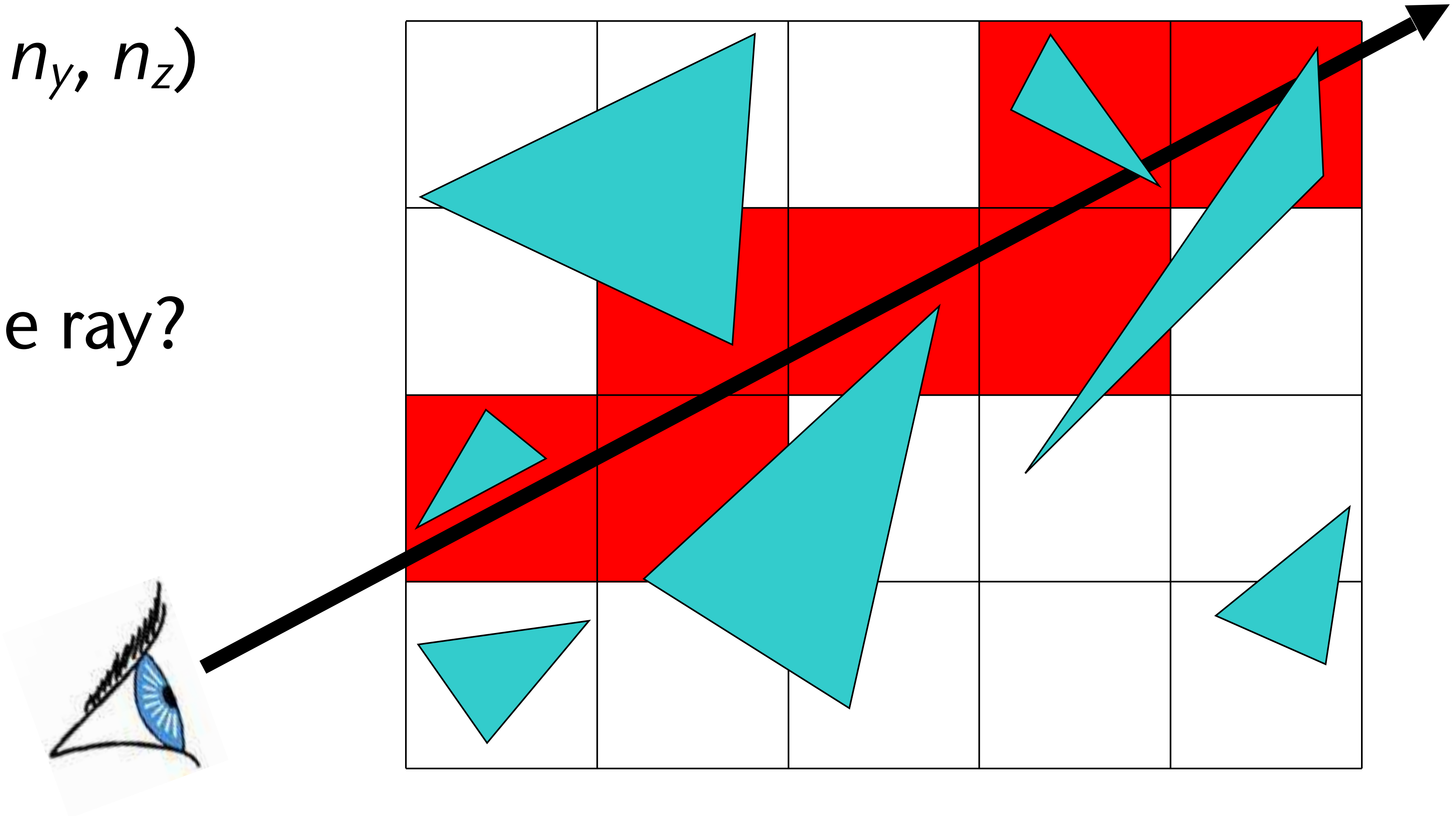






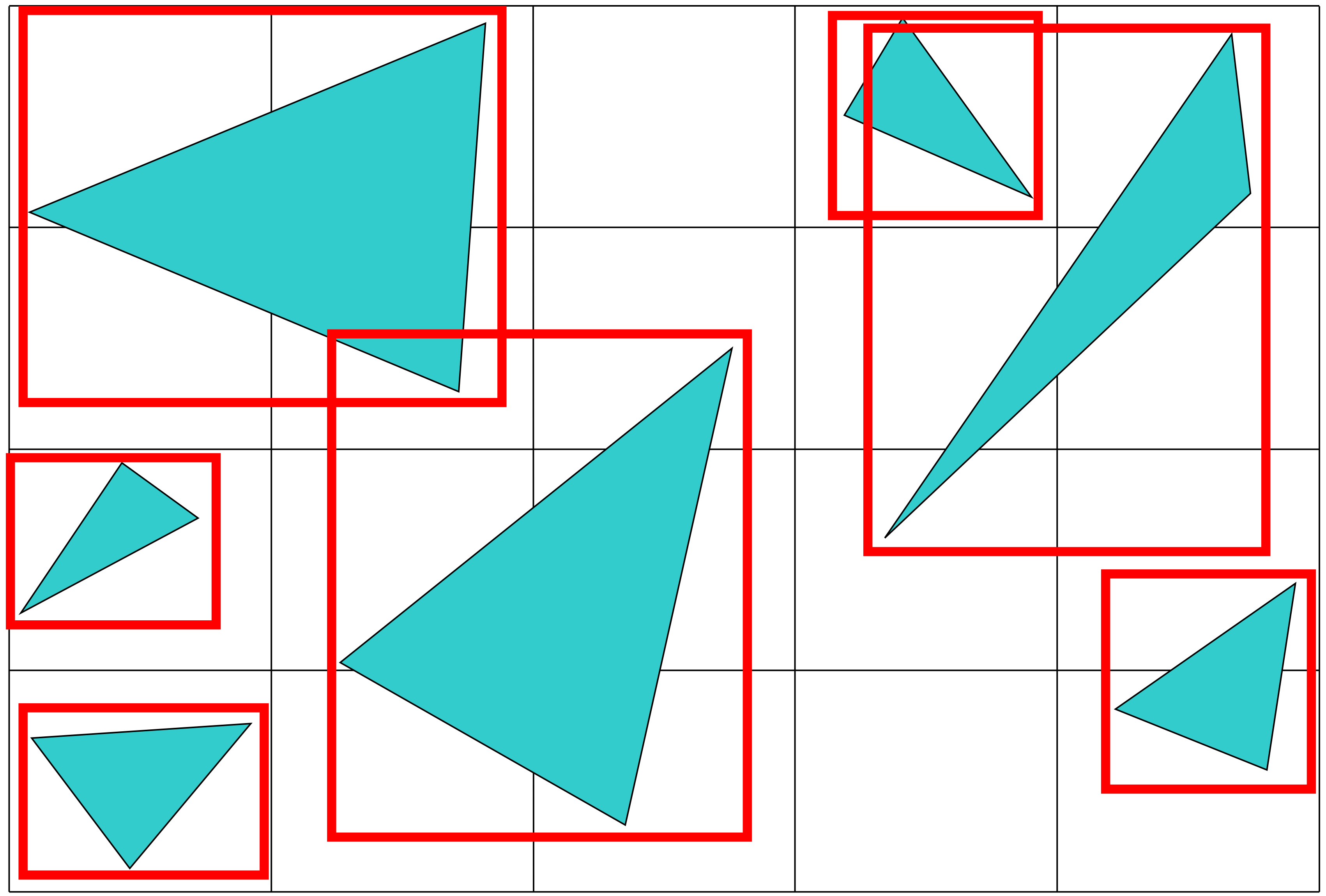
Regular 3D Grids

- Approach: partition scene into 3D grid; insert objects in cells; visit all cells along the ray; intersect ray with objects stored in cell
- 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 in the cell hit by the ray?
  - Yes: return closest hit
  - No: proceed to next cell



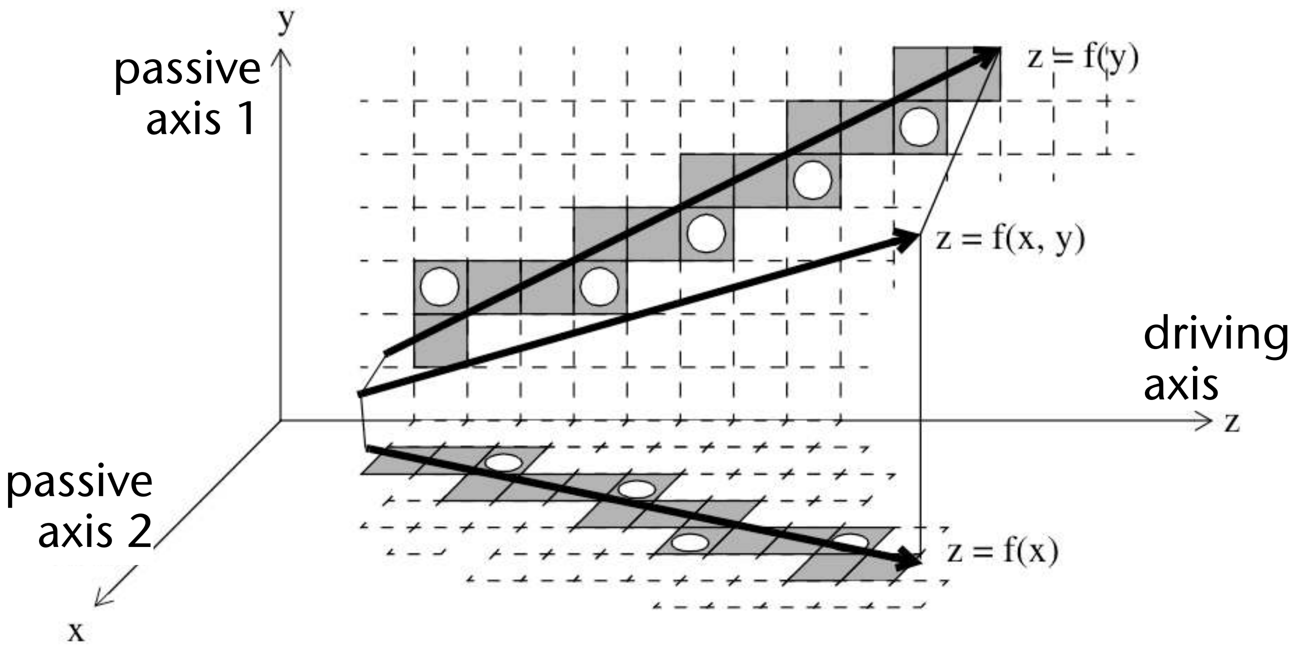
Precomputation

- 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 cells (not just one)

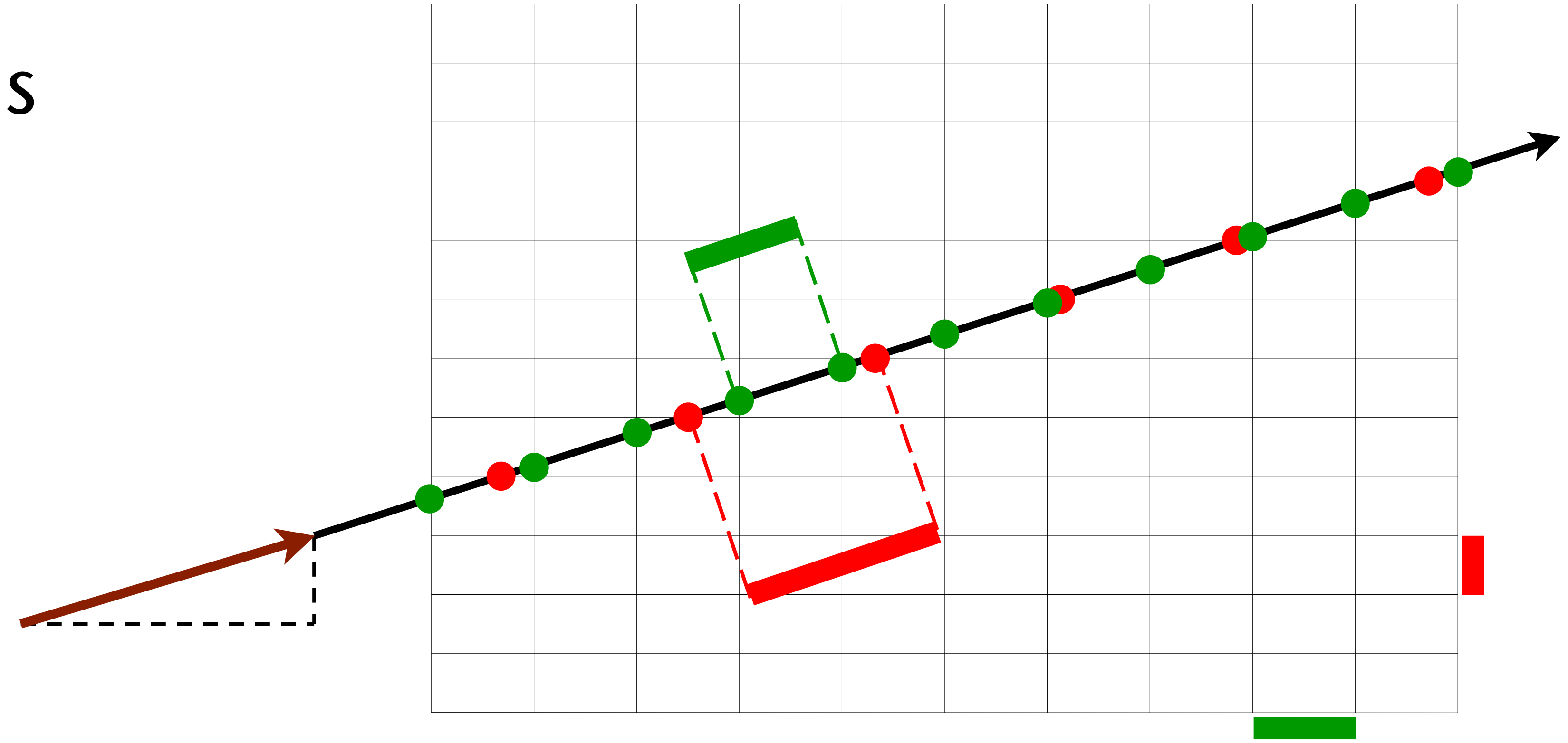


Traversal of a 3D Grid

1. Approach: utilize 2 synchronized DDA's (integer arithmetic) ⟶ 3D-DDA
   - One "driving axis", two "passive axes"



2. Approach: use line parameter
   - Increment all 3 *t*-values for intersections with xy-, xz-, and yz-planes
   - Pick the closest one



- Please review CG1 material

Complexity of a Grid Traversal

- Assumption: grid has N cells in total (at least conceptually, even if stored as a hash table)
- A complete ray query: could mean marching along the whole ray, cell by cell
- Worst-case time complexity: $O\left(\sqrt[3]{N}\right)$

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 polygons/objs (e.g., measured by their bbox)
- If you don't know it (or it's too time-consuming to compute), then choose $n_x, n_y, n_z = \sqrt[3]{N}$, N = # objects
  - More precisely: resolution = $\lambda\sqrt[3]{N}$, where λ depends on time for intersection & time for step in grid (tune at the end)
  - Consequence: #cells = space complexity $\in O(N)$ [good]
- Another good rule of thumb: try to make the cells cuboid-like

Are You Familiar with Hash Tables?

https://www.menti.com/v3qk8zeeby

Practical Storage: Background Grid and Spatial Hashing

- Don't use a 3D array for storage!
  - Most cells would be empty (unless you make the grid very coarse ...)
- Grid = background grid: only for generating hash values $\longrightarrow$ spatial hashing
  - Given point $\mathbf{p} = (p_x, p_y, p_z)$, e.g., lower left corner of bbox
  - Convert to integers:

$$\bar{p}_x = n_x \cdot \left\lfloor \frac{p_x}{U_x} \right\rfloor$$

where (U_x, U_y, U_z) = size of "universe"

  - Convert to hash value: concatenate $(\bar{p}_x, \bar{p}_y, \bar{p}_z)$ into a byte-string, then compute

$$h(\bar{p}_x \bar{p}_y \bar{p}_z) = h \in [0, N] \quad , \quad N = 2^k$$

    - Probably better: concatenate only the lower 16 bits of each of $(\bar{p}_x, \bar{p}_y, \bar{p}_z)$, if $n_x, n_y, n_z < 2^{16}$
  - Store obj ID / enumerate all obj's in hash table slot(s)
    - Use any of the standard collision resolution techniques (linear, quadratic, cuckoo, ...)

Marginal Note: FNV-1 is a Good Hash Function

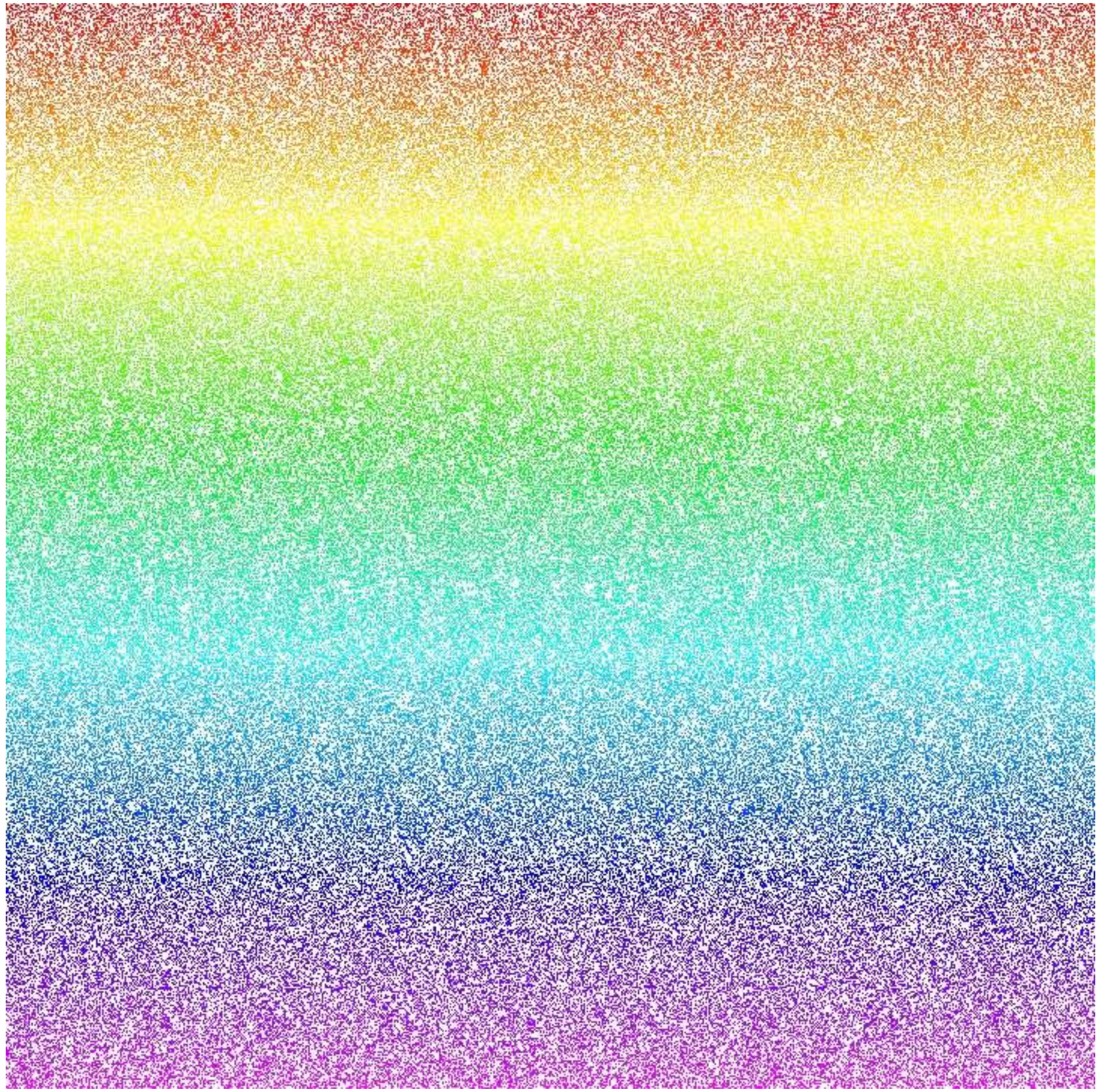
- The procedure:

```
h = fnv_offset              // "magic number", check literature
for i = 0 .. len(input str)-1:
  h = h * fnv_prime         // resembles Linear Congruential Generator
  h = h xor str[i]          // str[i] ∈ [0,255]
mask = ( (1 << k) - 1)      // in case k=16, mask = 0xffff
h = (h >> k) ^ (h & mask)  // "xor-fold" to range of N = 2**k
```

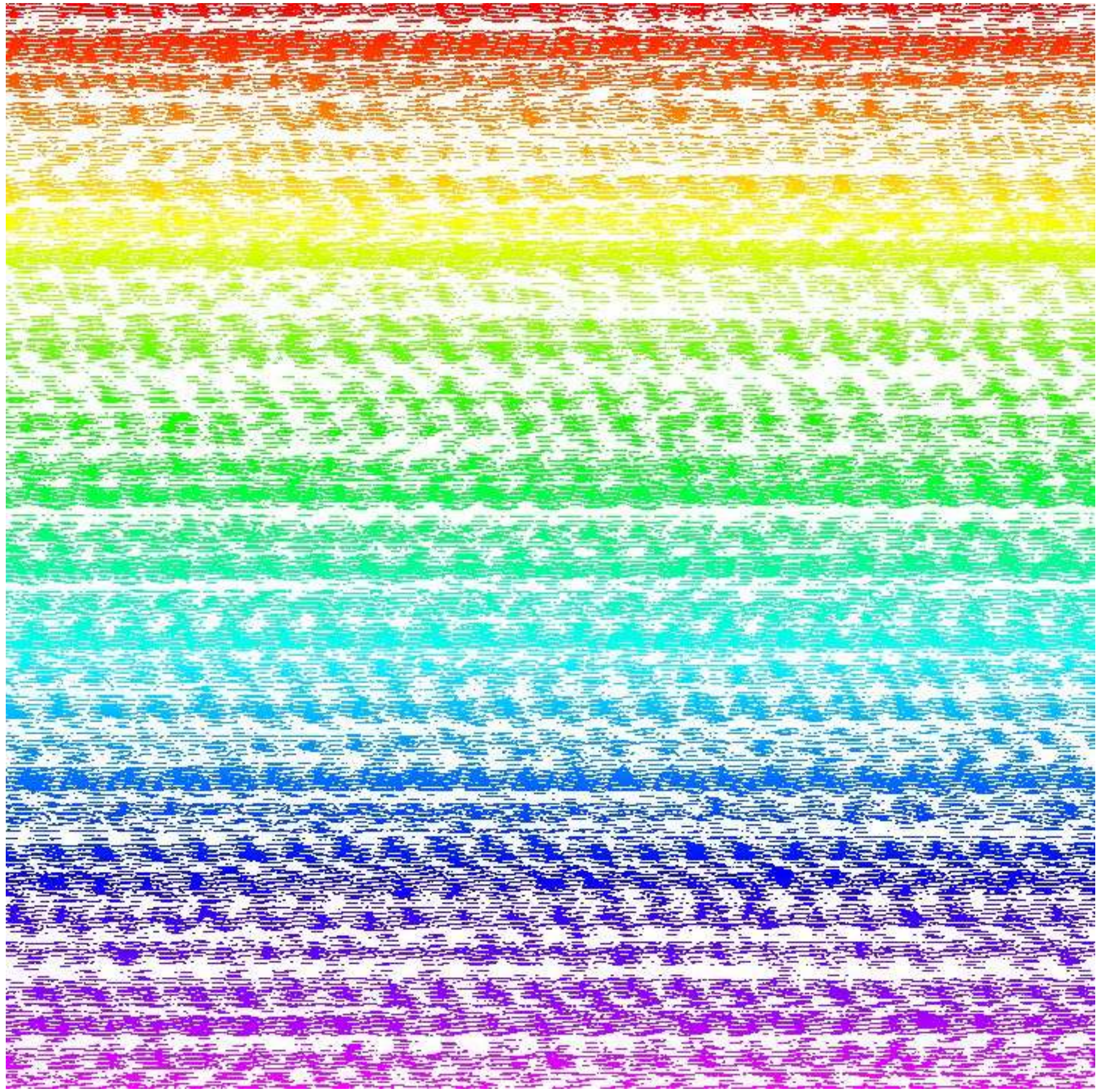
- All variables must be **`unsigned int`**; **`str[i]`** must be **`unsigned byte`**
- N (= size of hash table) must be a power of 2, i.e., $N = 2^k$
- Values for offset and prime depend on bit size of the data types:
  - If **`unsigned int`** = 64 bits, then prime = 1099511628211, offset = 14695981039346656037
  - If **`unsigned int`** = 32 bits, then prime = 16777619, offset = 2166136261

Comparison with Other Hash Functions

Visualization of "spread" / "randomness" over hash table



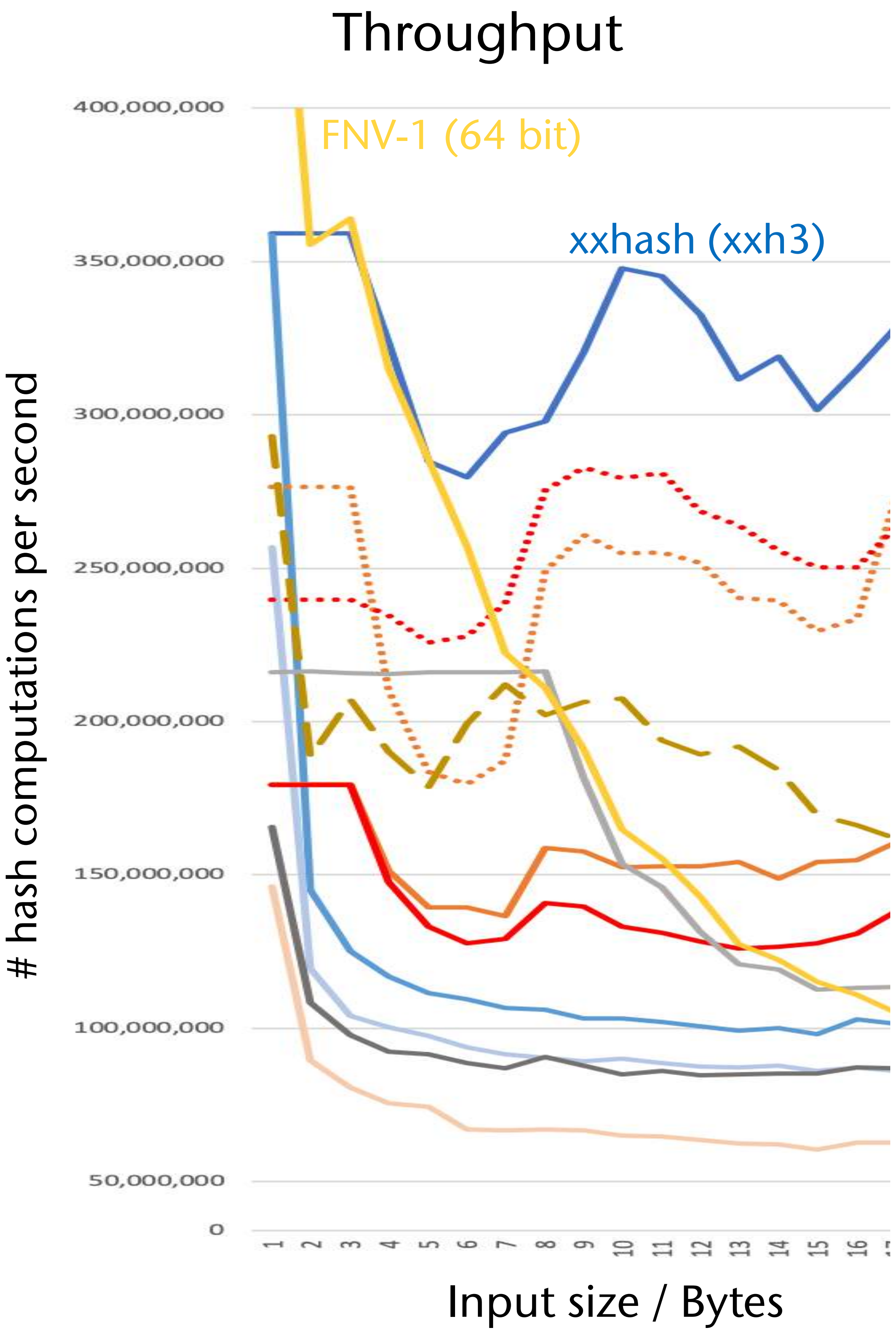
FNV-1a

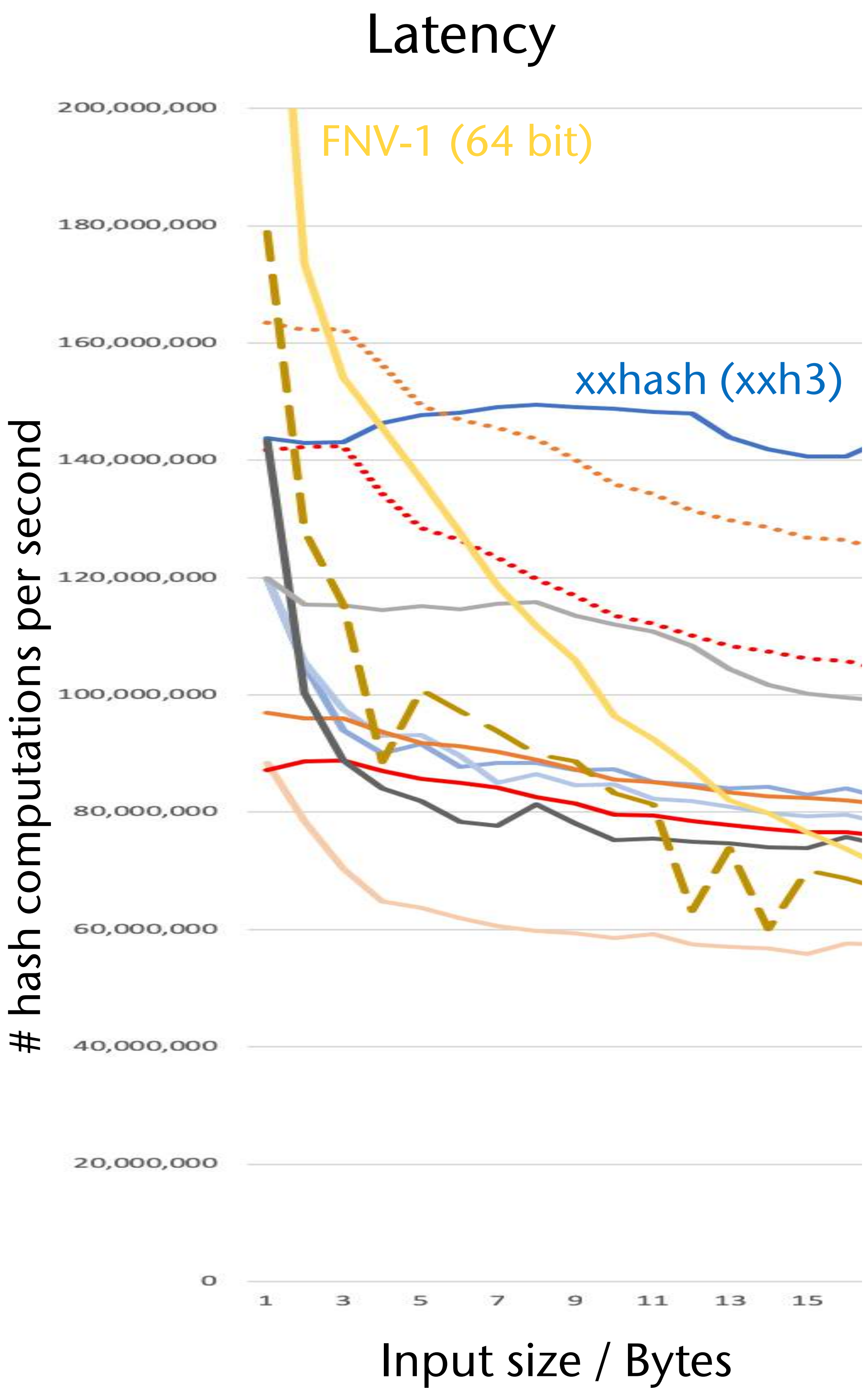


DJB2

Performance

FYI

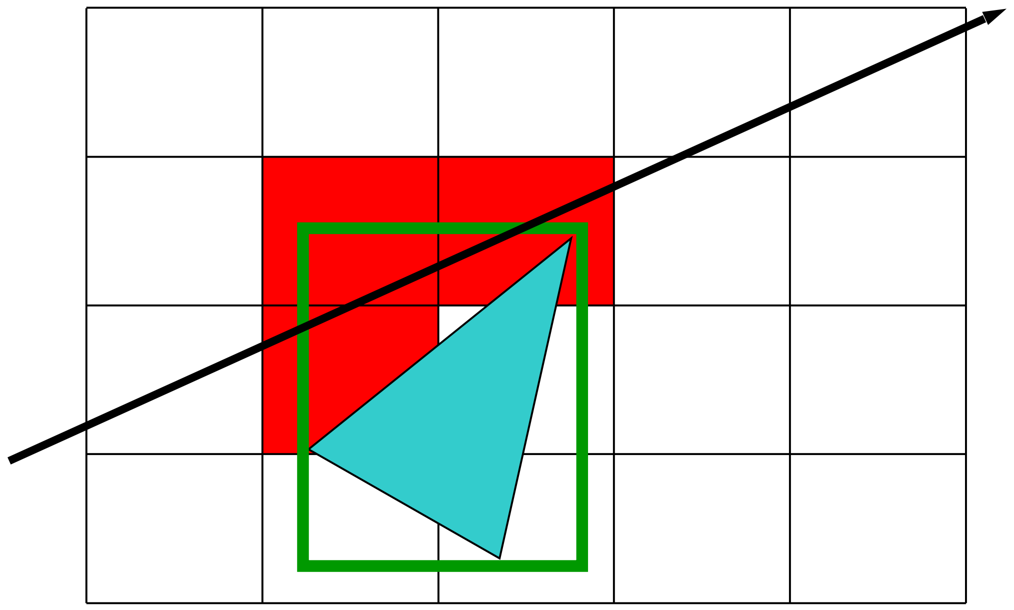


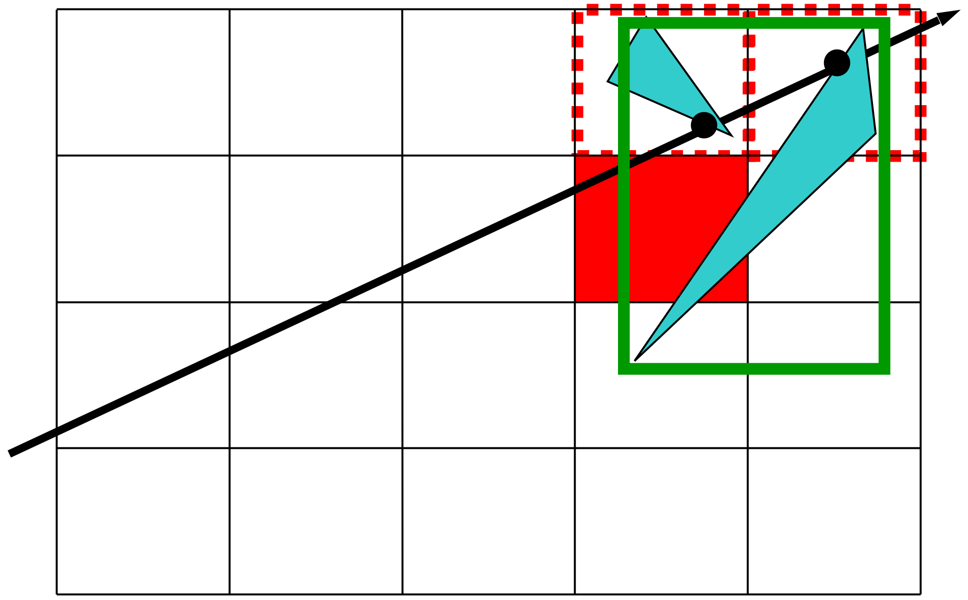


Problems

- 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 checked for intersection with the same object several times (see bottom left)





The Mailbox Technique

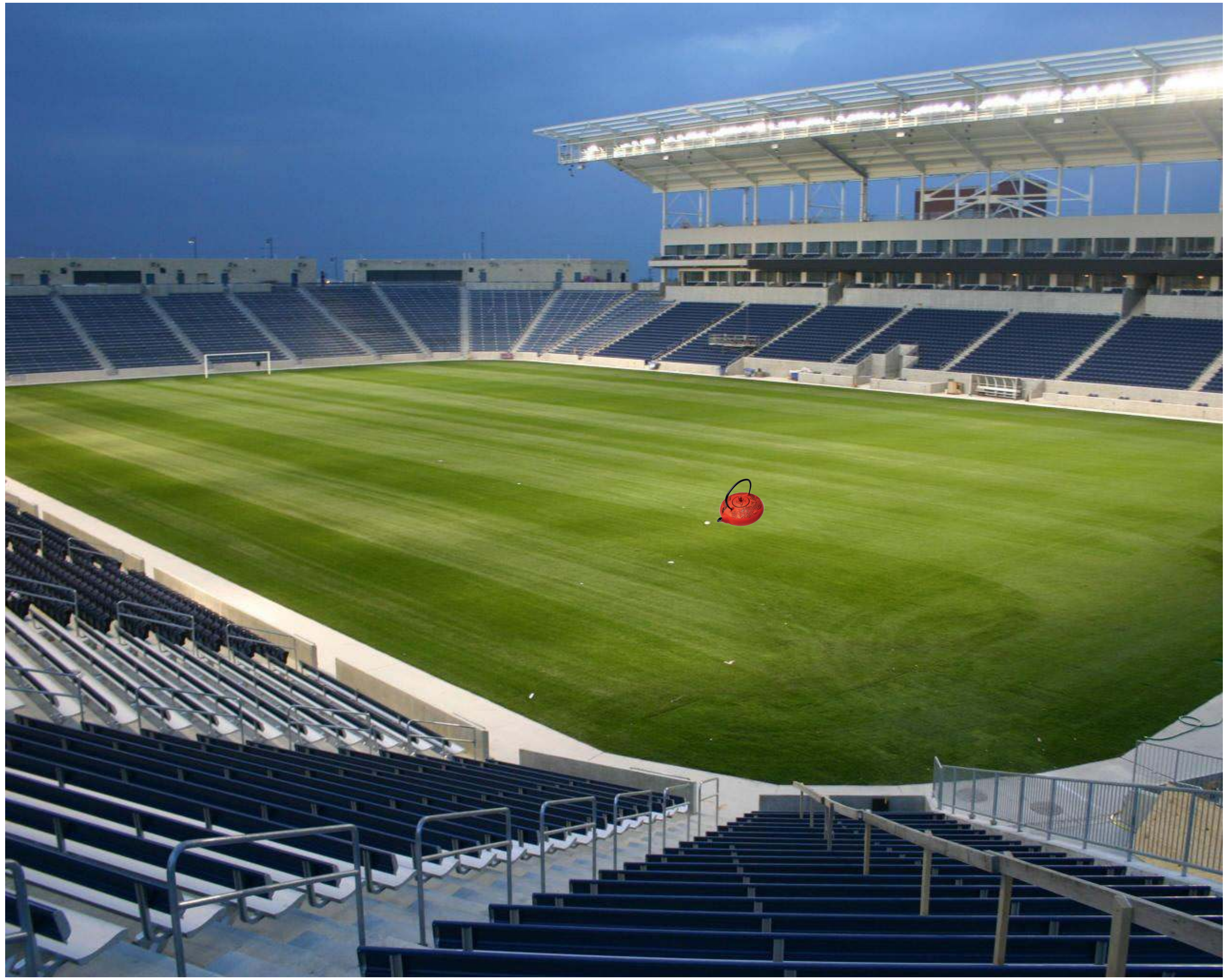
- Solution: assign a mailbox with each object (e.g., just an integer instance variable), and generate a unique ray ID for each new ray
  - For the ray ID: just increment a counter in the constructor of the ray class
- After each intersection test with an object, store the *ray ID* in the object's *mailbox*
- Before an intersection test, compare the ray ID with the ID stored in the object's mailbox:
  - IDs are equal $\rightarrow$ the intersection point can be read out from the mailbox;
  - IDs are not equal $\rightarrow$ 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
  - Mailbox could cause congestion when testing many rays in parallel
- Solution: store mailboxes separately from geometry
  - Maintain a small hash-table *with each ray,* which 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 use "Array of Structs" (AoS) or a "Struct of Arrays" (SoA)

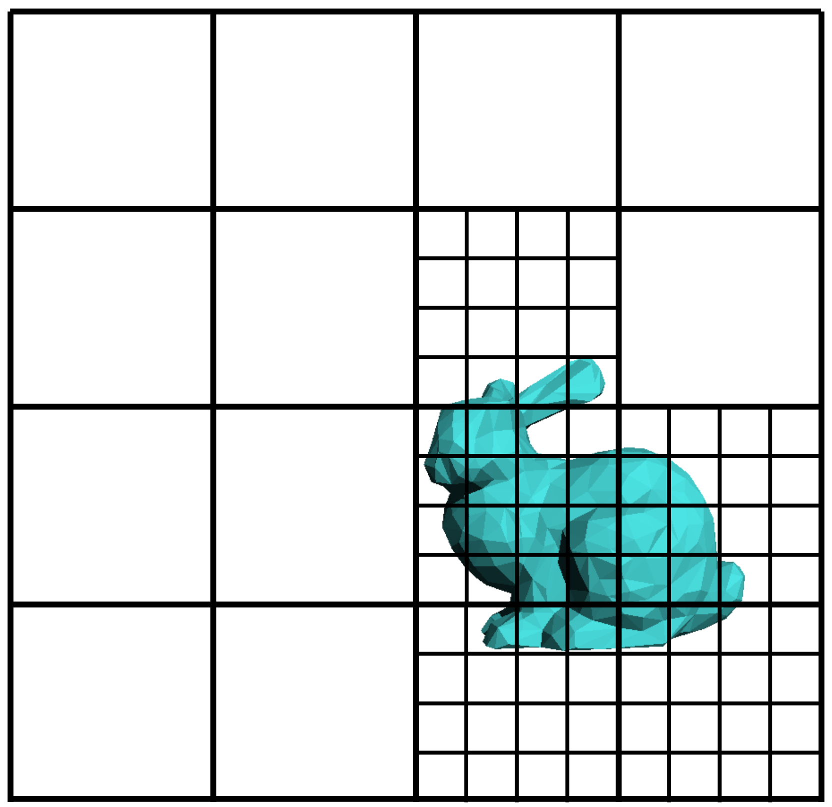
The Teapot in a Stadium Problem

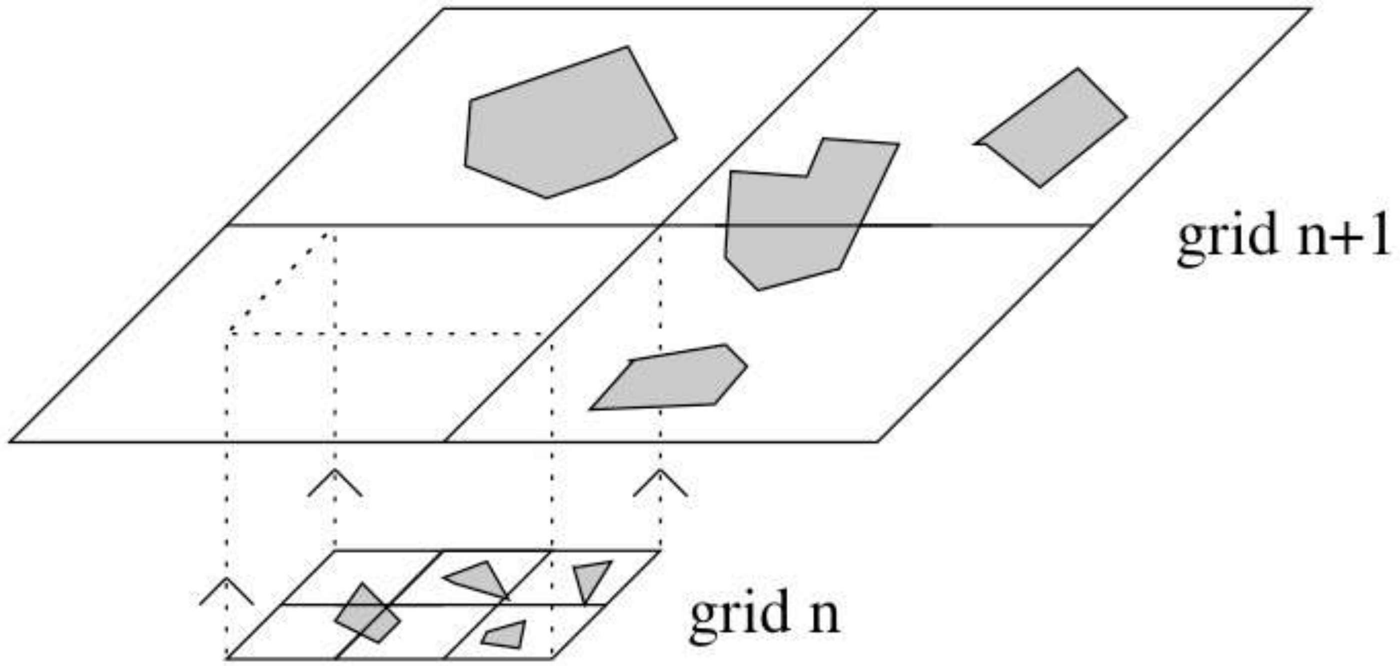
- Problem: regular grids don't adapt well to large variations of local "densities" of the geometry
- Average object size is a bad estimator for good cell size



Possible Solutions: Hierarchical Grid or Recursive Grid

- Similar ideas
- Recursive grid: start with coarse grid, partition "crowded" cells with further grids inside
- Hierarchical grid: Group objects by size (e.g. "big", "medium", "small"), construct grid for each group (think "layers of grids")

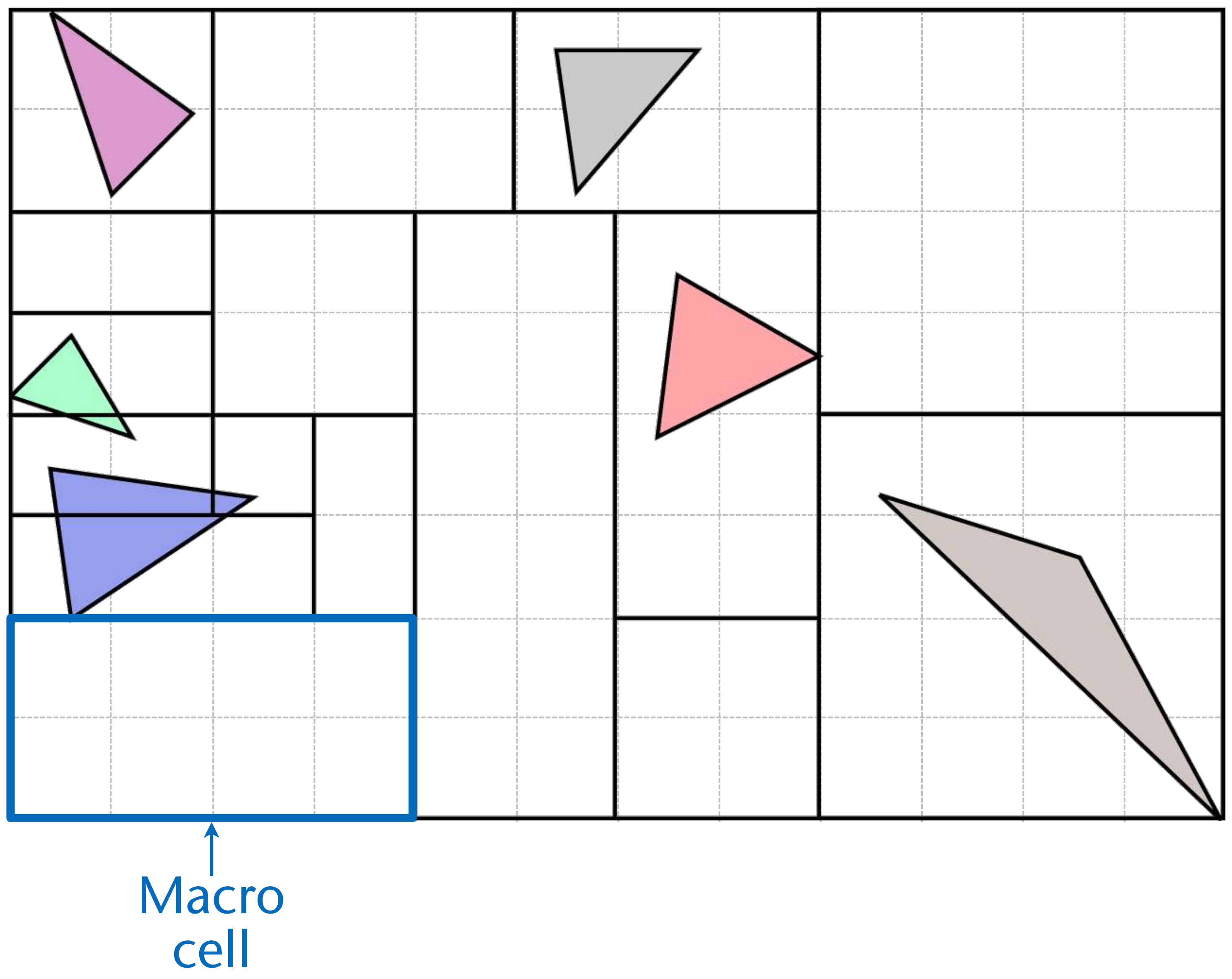




Irregular Grids

[Pérard-Gayot et al, 2017]

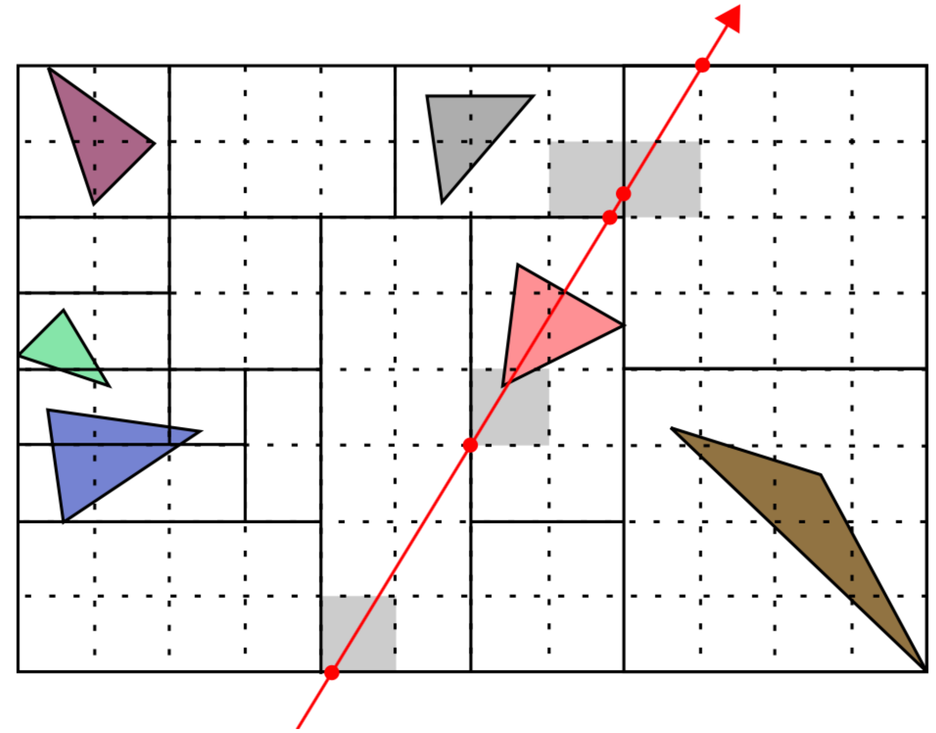
- Overall idea:
  - Discretize universe with a background grid (which is never explicitly constructed)
  - Create partitioning of the universe by boxes aligned to the background grid
- Similar idea: "macro regions" [Devillers 1998]
- Limitations:
  - Probably only suitable for ray-tracing (what about coll.det.?)
- Advantages:
  - Allows construction and ray-tracing on the GPU (see course "Massively Parallel Algorithms")
  - Suitable for static and dynamic scenes (b/c of fast construction)



Traversal of an Irregular Grid

- The overall algorithm:

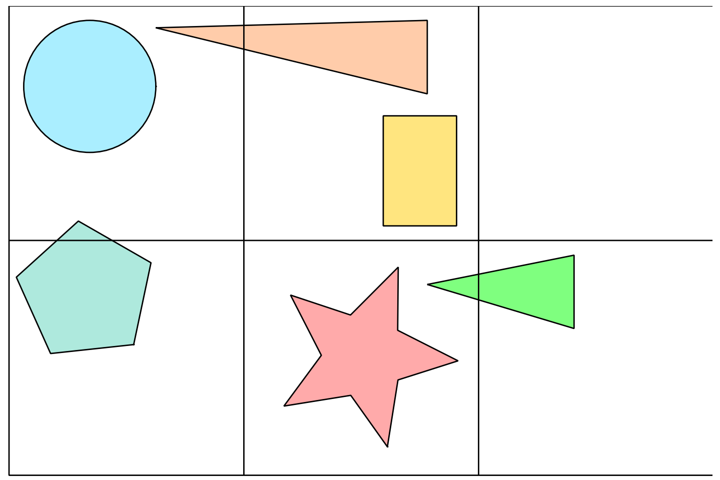
```
Init p ⟵ origin of ray
repeat:
  determine next virtual grid cell containing p          // (1)
  determine macro cell containing virtual grid cell      // (2)
  check ray for intersection with objs in macro cell
  calculate exit point of ray wrt. current macro cell ⟶ p
until hit is found, or ray leaves universe
```



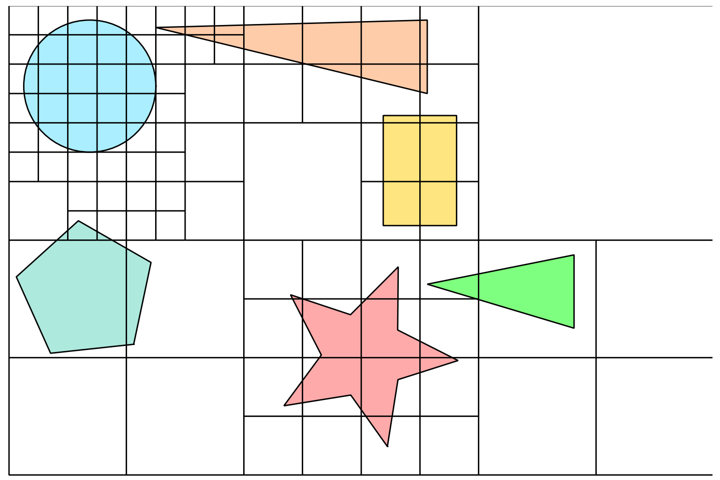
- Remarks:
  1) Point p is always exactly on the border of a cell ⟶ make sure "correct" virtual cell is identified
  2) Technical details omitted here
  3) Exit point: need to calc only 3 ray-plane intersections (axis-aligned planes)

Construction of Irregular Grids (Without the Details, Without GPU)

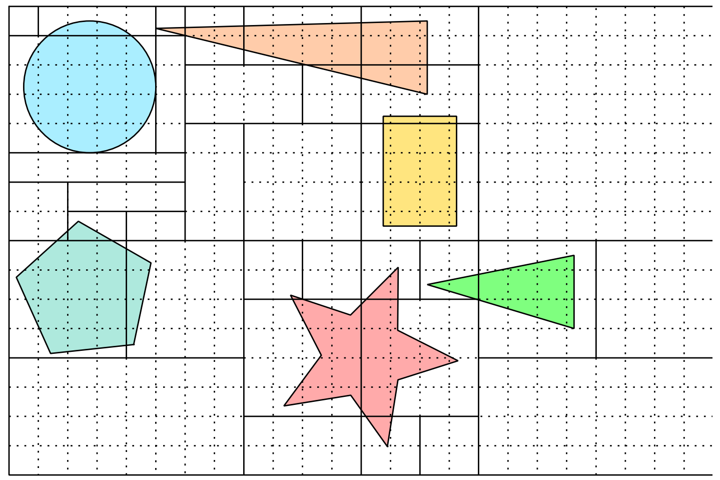
1. Construct coarse, uniform 3D grid



2. In each coarse cell: construct individual octree
   - With the usual stopping criteria
   - Call leaves "second-level cells"
   - Maximum octree depth over all coarse cells ⟶ resolution of virtual grid



3. Greedily merge adjacent cells of the virtual grid:
   - Merge only, if raytracing costs are reduced
   - Stop, when cost reduction is < threshold



Cell Merging, Based on Surface Area Heuristic (SAH)

- Costs of a macro cell c:

$$C(c) = \left(C_i \cdot N + C_t\right) \cdot \text{Area}(c)$$

where C_i = cost for ray-triangle intersection,
N = number of polygons in c,
C_t = cost for step to next macro cell

- Perform merge between cells c_1 and c_2, iff costs after < cost before:

$$C(c_1 \cup c_2) < C(c_1) + C(c_2)$$

i.e.

$$\left(C_i(N_1+N_2)+C_t\right) \text{Area}(c_1 \cup c_2) < \left(C_i N_1+C_t\right) \text{Area}(c_1)+\left(C_i N_2+C_t\right) \text{Area}(c_2)$$

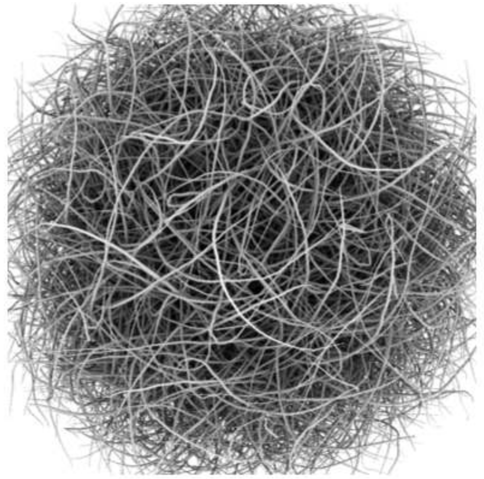
- Constraint: merged cell must be a regular AABB again

Some Results

- Test scenes:


Sponza


Conference


Hairball


Crown


San Miguel

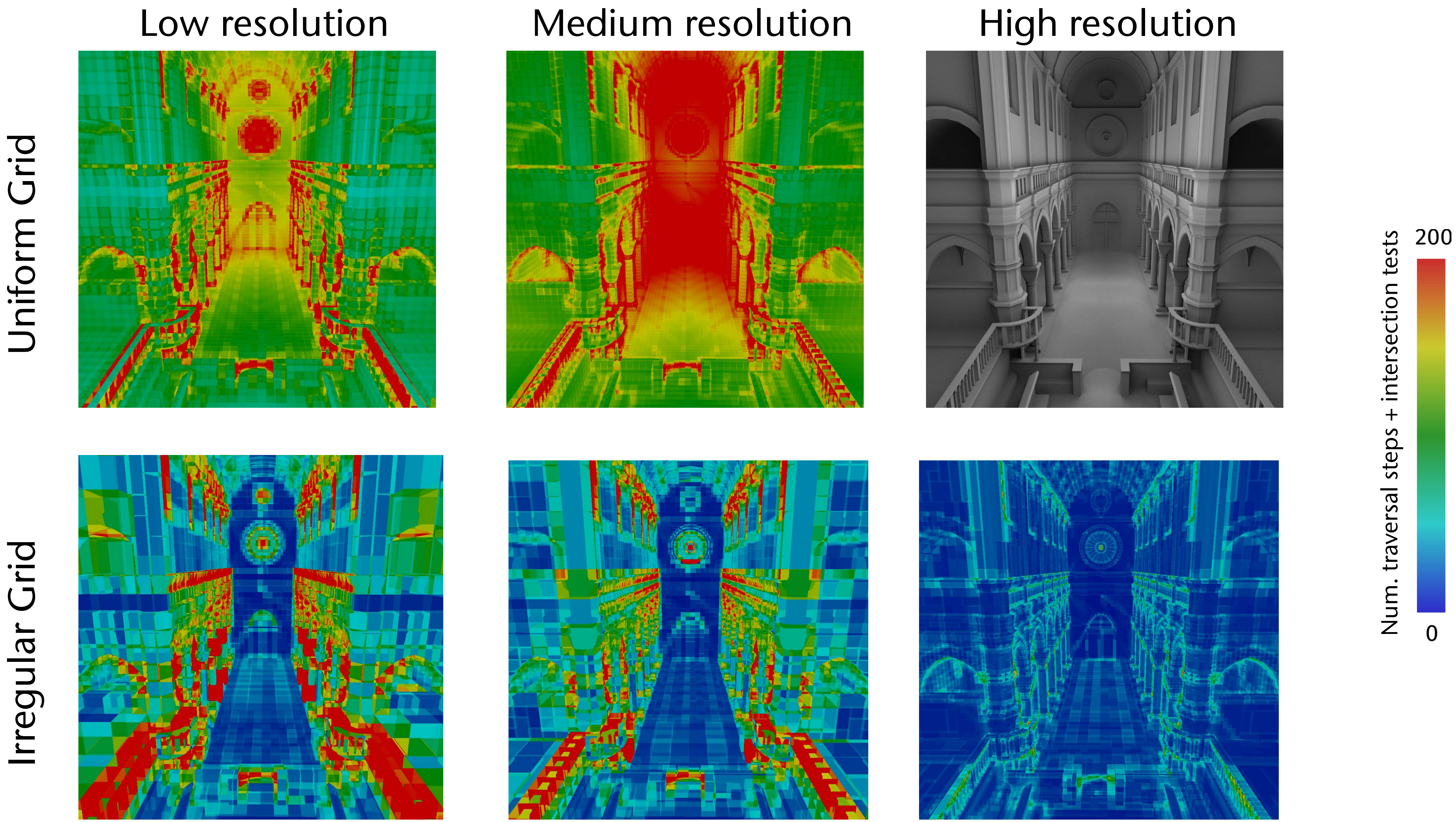
- Performance (on the GPU!):

		Build times (s)		Traversal (MRay/s)		Memory (MB)	
Scene	#Tris	Ours	2L Grid	Ours	2L Grid	Ours	2L Grid
Sponza	262K	[0.012, 0.026]	0.007	[201, 653]	145	[4, 23]	24
Conference	283K	[0.016, 0.022]	0.007	[182, 597]	77	[4, 12]	27
Hairball	2.9M	[0.349, 0.893]	0.177	[79, 148]	37	[138, 779]	668
Crown	3.5M	[0.066, 0.203]	0.039	[115, 296]	74	[53, 278]	182
San Miguel	7.9M	[0.162, 0.492]	0.071	[97, 291]	63	[107, 565]	323

"Ours" = irregular grid, "2L Grid" = two-level grid

For "ours", a range is given, because there is a quality parameter (resolution of the background grid) that allows to balance speed against memory usage

Ray Casting Effort per Pixel

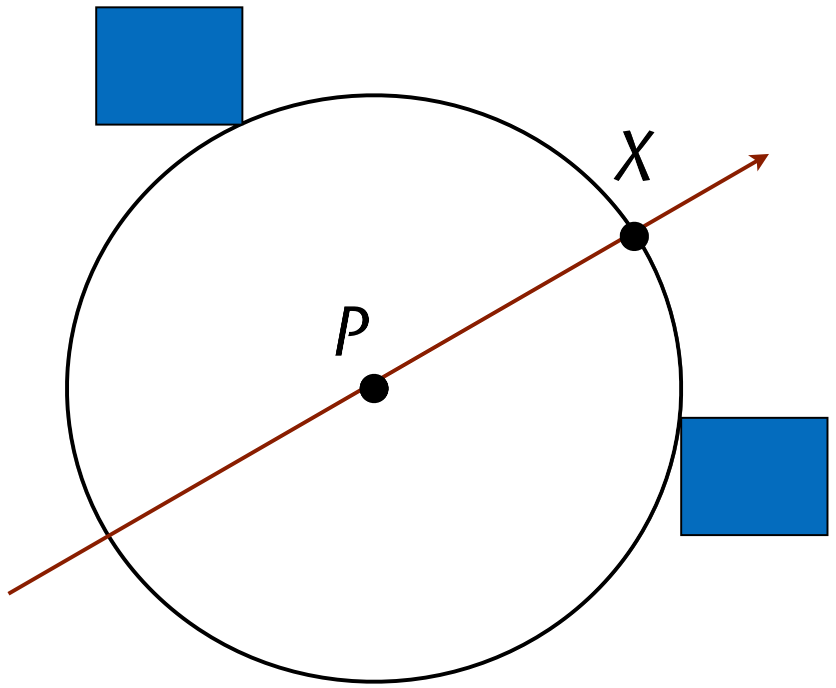


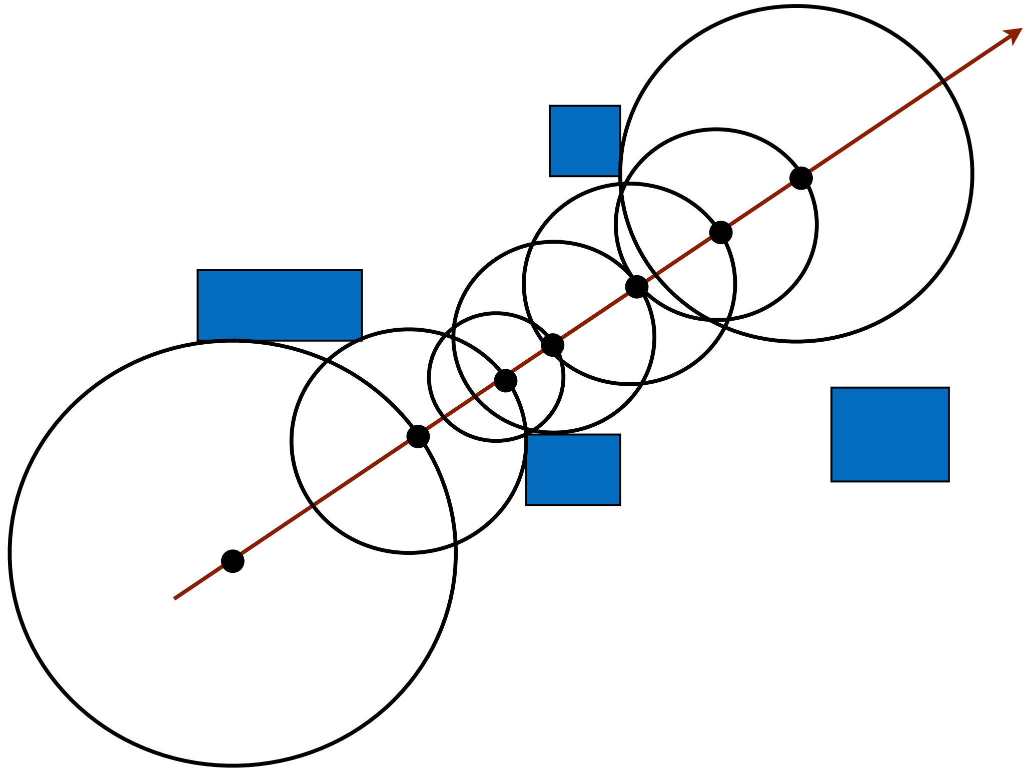
two-level grid
2 rays / pixel

irregular grid
9 rays / pixel

Proximity Clouds, a.k.a. Sphere Tracing

- 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 $X = P + \frac{r}{\|d\|}\mathbf{d}$
  - What if we knew this "clearance" radius *r* for each point in space
  - Then, we could jump through space from one point to its "clearance horizon", and so on ...
- This general idea is called empty space skipping
  - Comes in many different guises

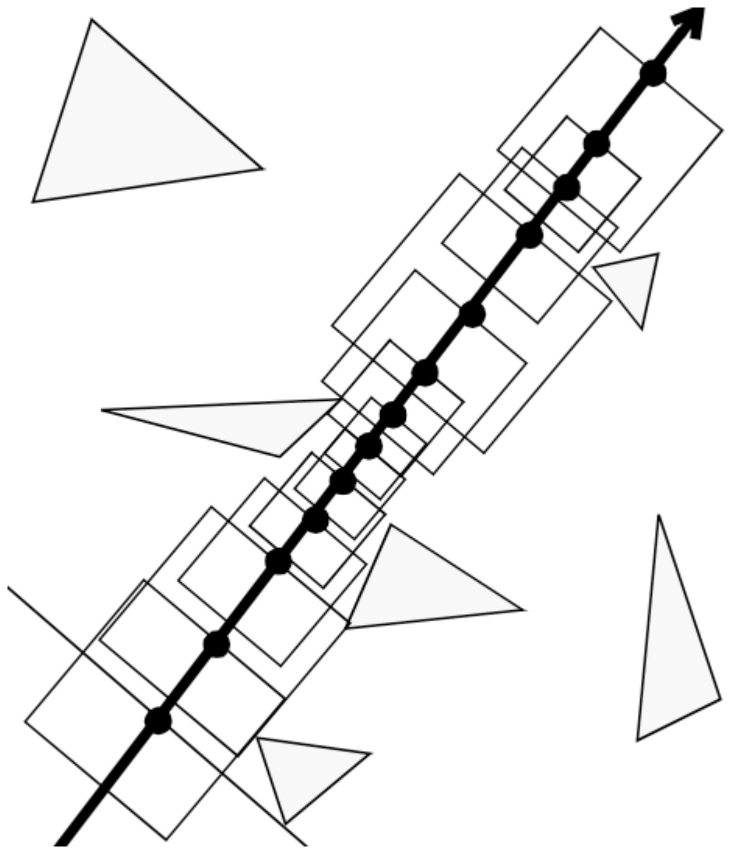


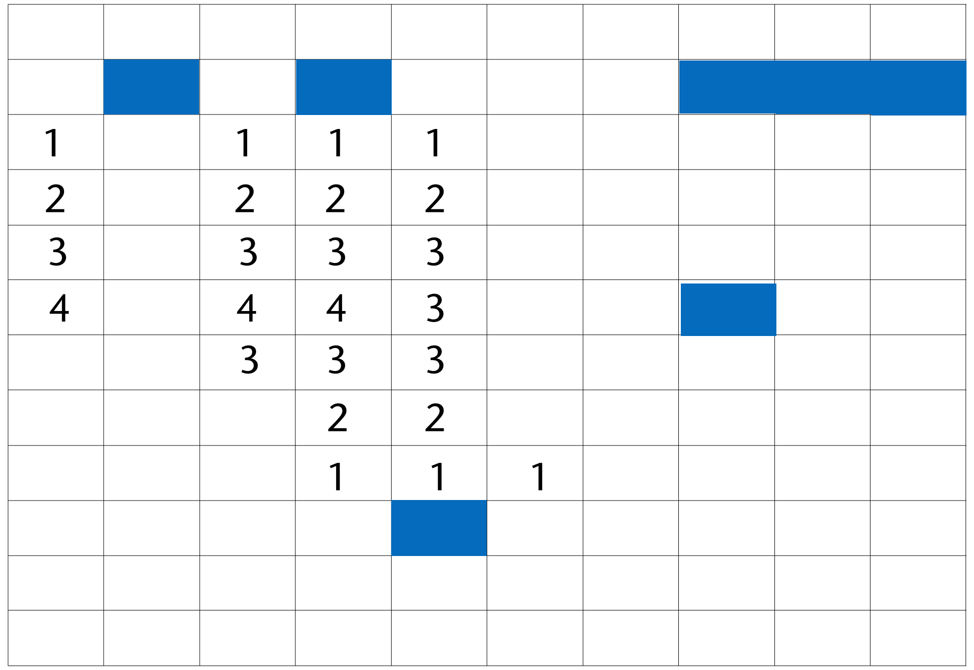


- The idea works with any other metric, too
- Problem: we cannot store the clearance radius in *every* point in space
- Idea: discretize space by grid
  - For each grid cell, store the minimum clearance radius, i.e., the clearance radius that works in any direction (from any point within that cell)

➢ Such a data structure is called a distance field

- Example:

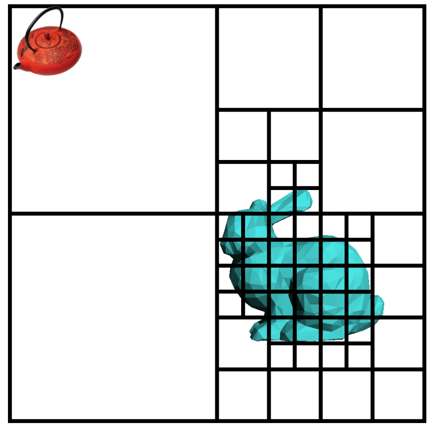


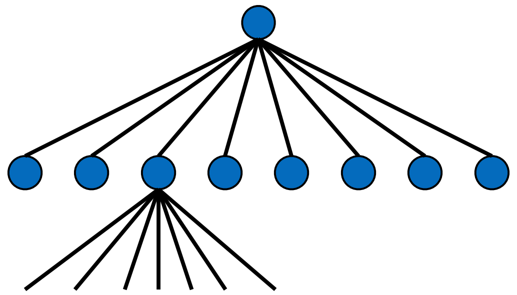


General Rules for Optimization Just FYI

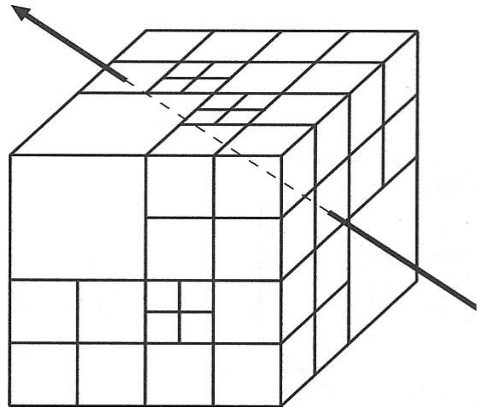
- "Premature Optimization is the Root of All Evil" [Knuth]
- *First*, implement your algorithm naïve and slow, *then* optimize!
- After each optimization, do a before-after benchmark!
  - Sometimes, optimizations turn out to perform *worse!*
- Only make small optimizations, one at a time!
- Do a profiling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

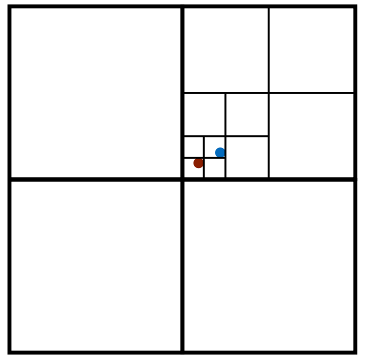
The Octree / Quadtree





- 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 levels are needed to discriminate objects

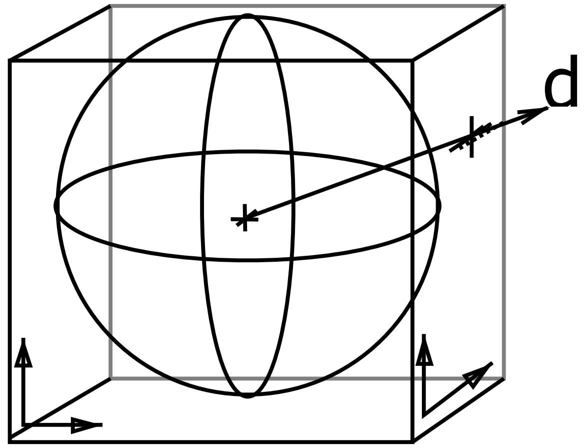




The 5D Octree for Rays

Optional

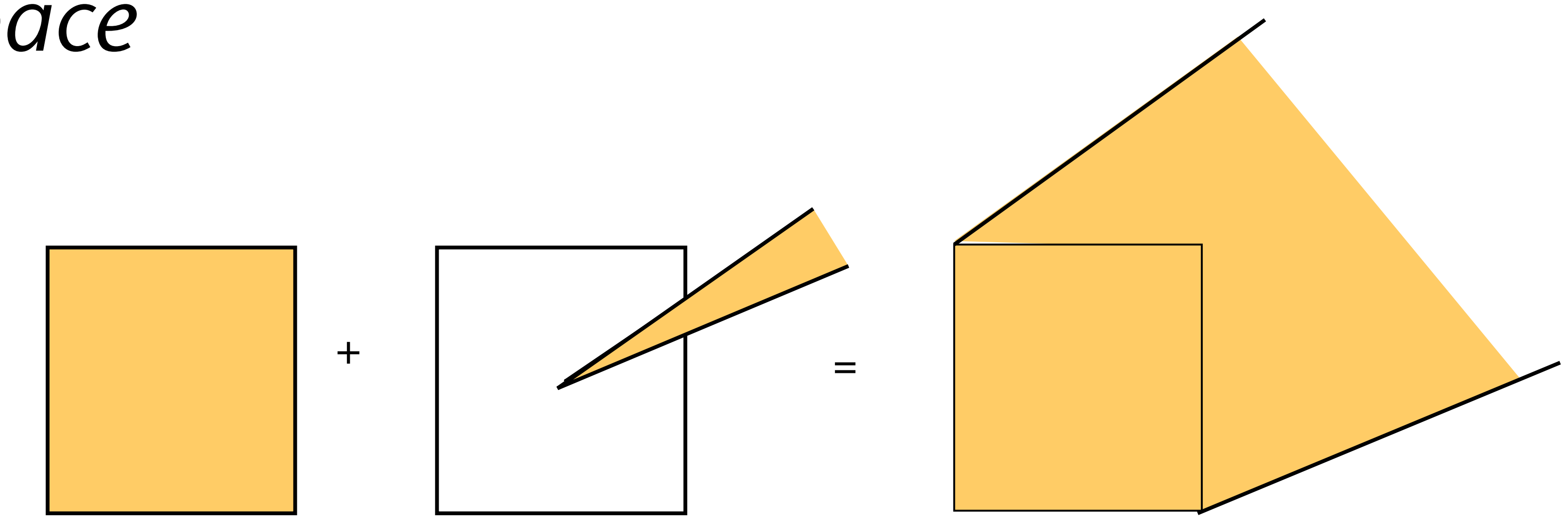
[Arvo & Kirk, 1987]



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

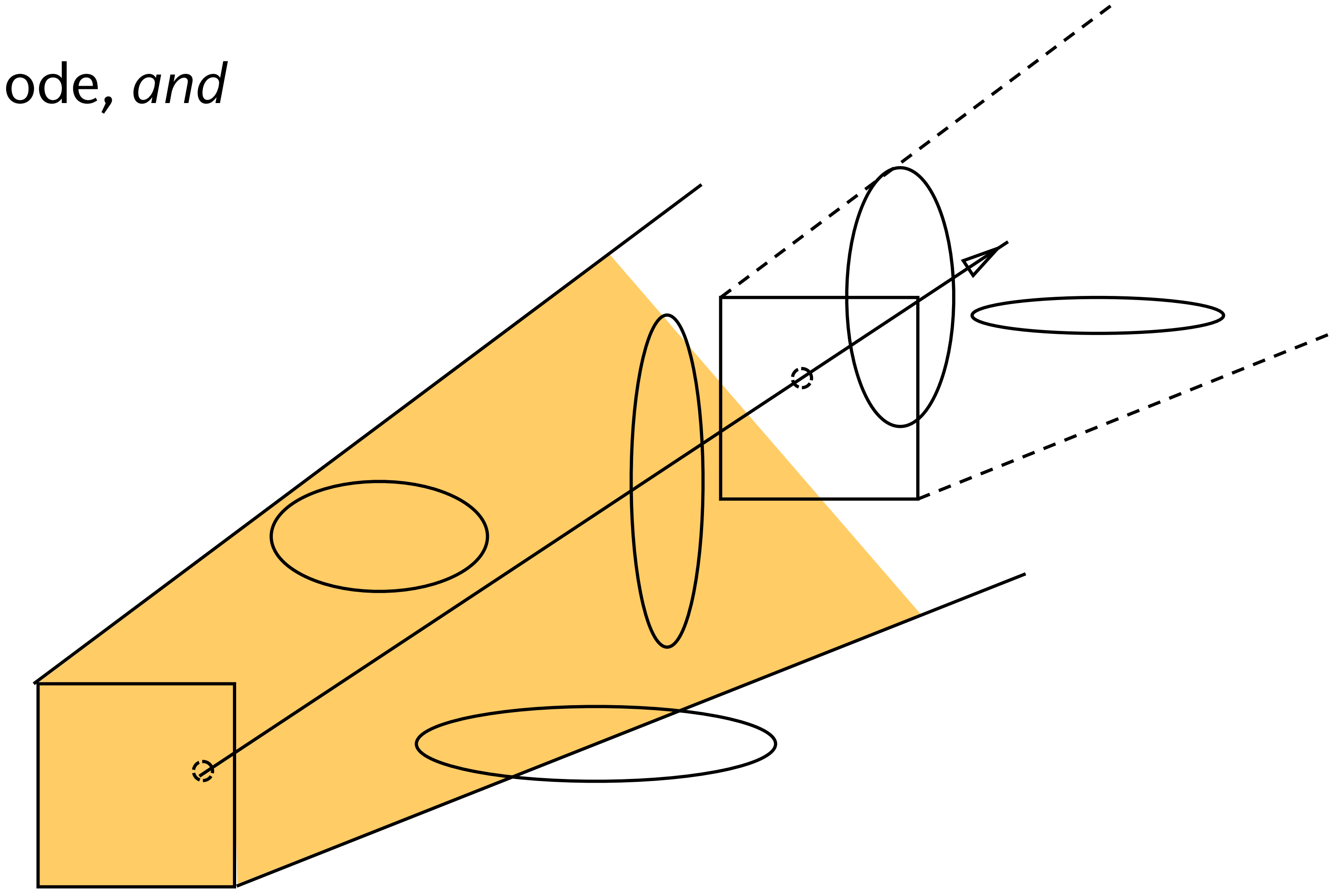
$$S^2 \leftrightarrow D := [-1, +1]^2 \times \{\pm x, \pm y, \pm z\}$$

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



Optional

- 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 …



Optional

Remarks

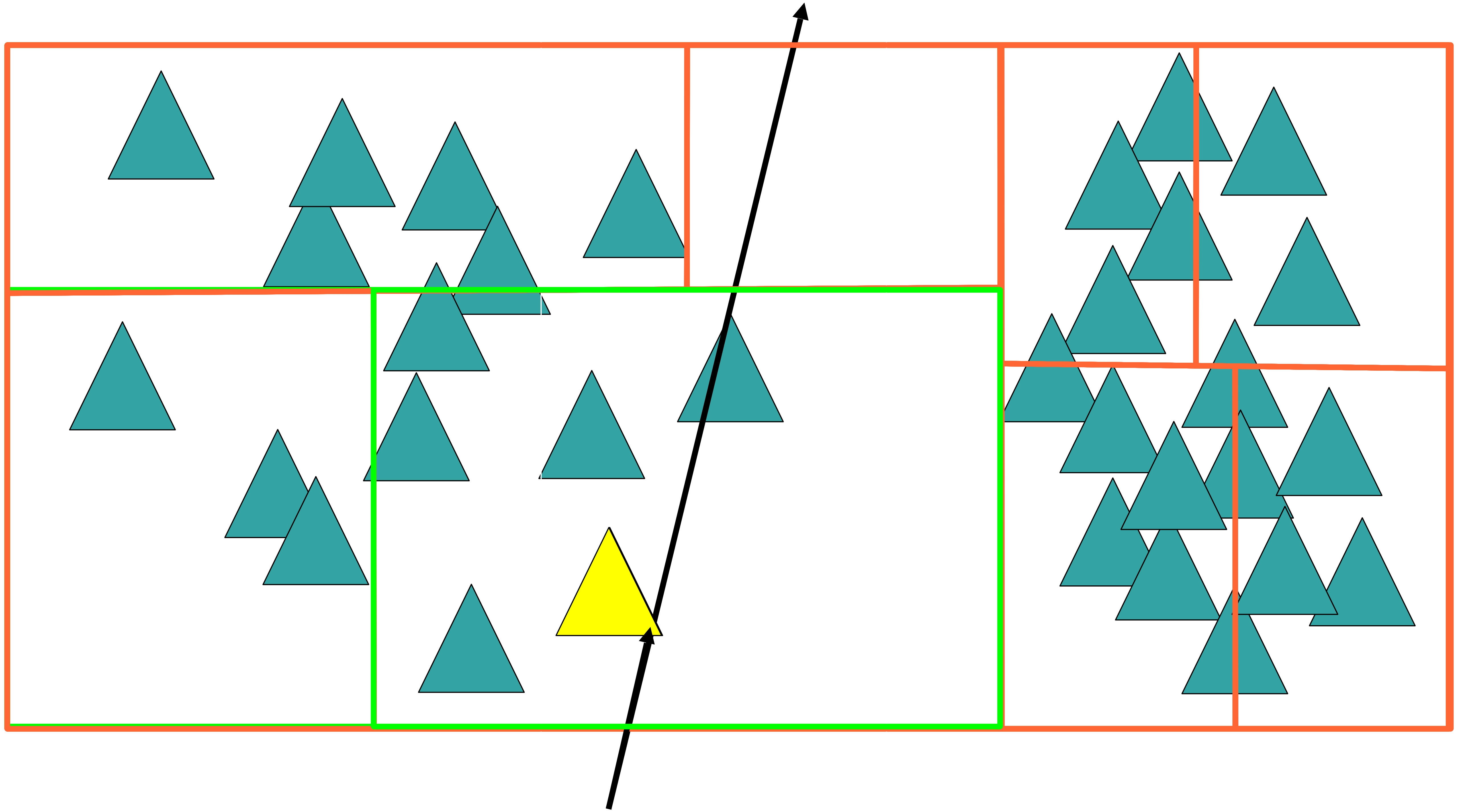
- The method basically pre-computes a complete, discretized visibility for the entire scene
  - I.e., what is visible from each point in space in each direction?
- Very expensive pre-computation, very inexpensive ray traversal
  - The effort is probably not balanced between pre-computation and run-time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

*k*D-Trees

- Problem with octrees:
  - Very inflexible subdivision scheme (always at the center of the parent 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/cells recursively by just *one* plane:
  - Start with root = bbox of our universe
  - Choose a plane perpendicular to one coordinate axis
  - Free choices: the axis (x, y, z) & place along that axis
- "Best known method for ray-tracing" (... at least for static scenes) [Siggraph Course 2006]

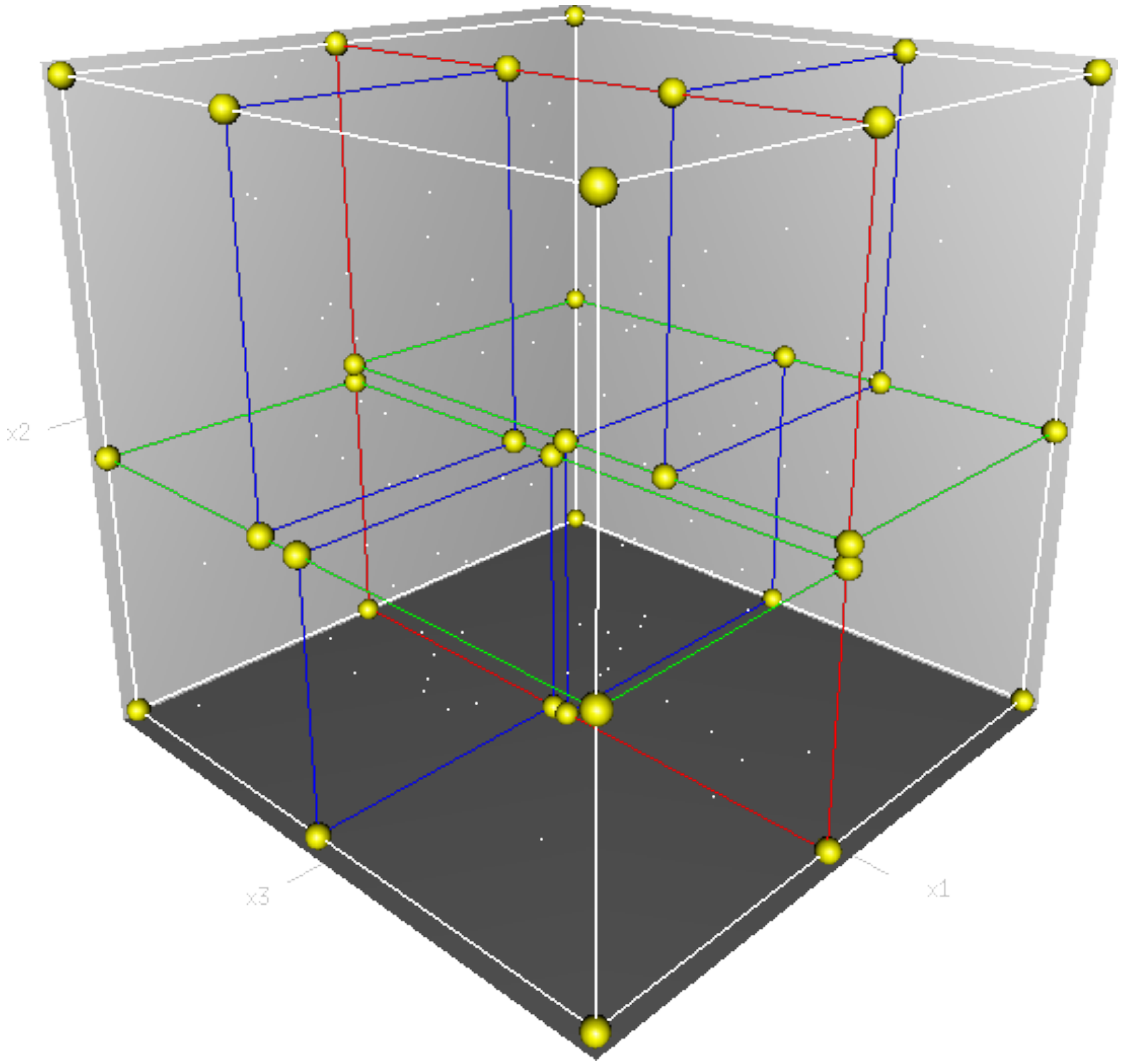
- Informal definition: a *k*d-tree is a binary tree, where
  - Leaves contain single objects (polygons) or a list of at most *b* objects (binning)
  - Inner nodes store a splitting plane (perpendicular to an axis) and child pointer(s)
  - Stopping criterion:
    - Maximal depth, number of objects, a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

Example



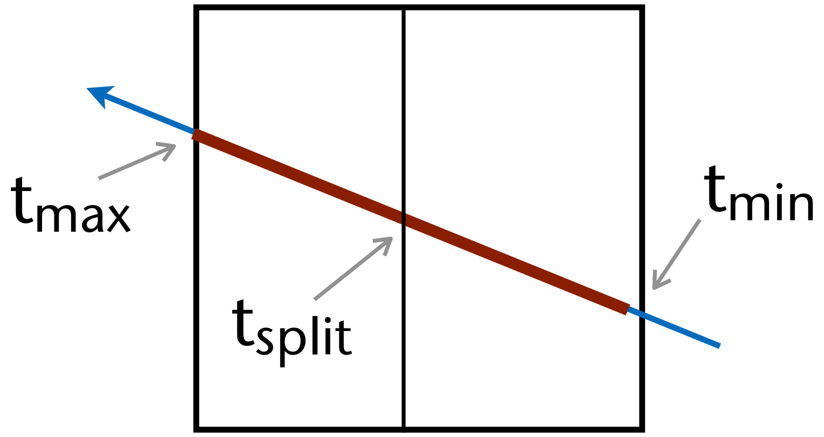
[Slide courtesy Martin Eisemann]

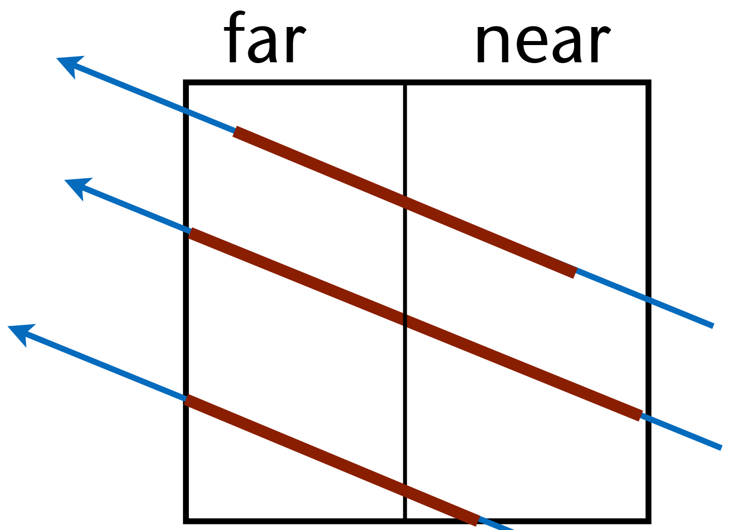
3D Visualization



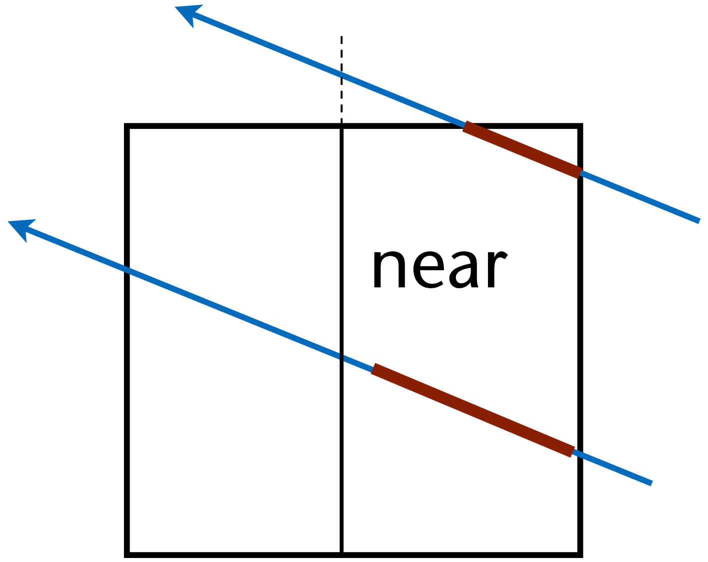
Ray-Traversal Through a KD-Tree

- Intersect ray with root-box → t_{min}, t_{max}
- Recursion:
  - Update [t_{min}, t_{max}] throughout tree traversal
  - 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

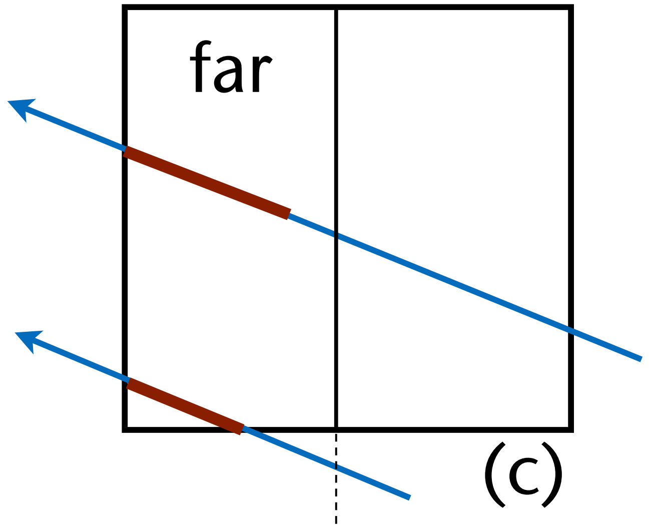




(a)



(b)



(c)

Pseudo-Code for the Traversal of a KD-Tree Along a Ray

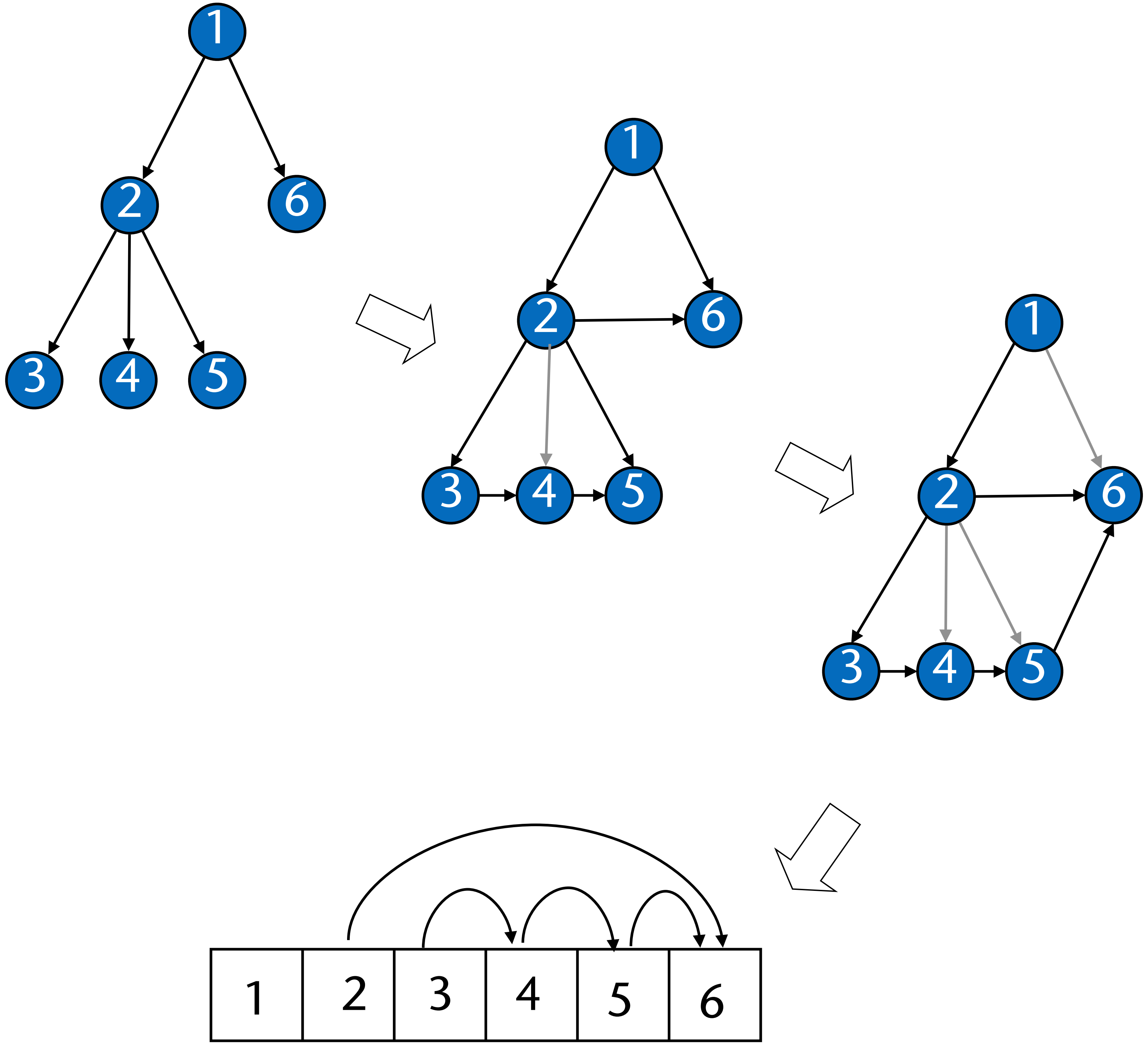
```
traverse( Ray r, Node n, float t_min, float t_max ):

if n is leaf:
  intersect r with each primitive in object list,
    discard 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
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 exit
  return traverse( r, far, t_split, t_max )
```

Optimized Traversal for Shadow Rays

Just FYI

- Observation:
  - 90% of all rays are shadow rays
  - Any hit is sufficient
- Consequence (in case of shadow ray):
  - The order the children in the kd-tree are visited does not matter
  - So, perform "stupid" DFS
- Idea: replace the recursion by an iteration
- Augment the kd-tree by more pointers to achieve that



The Algorithm

Just FYI

```
straverse( 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"
```

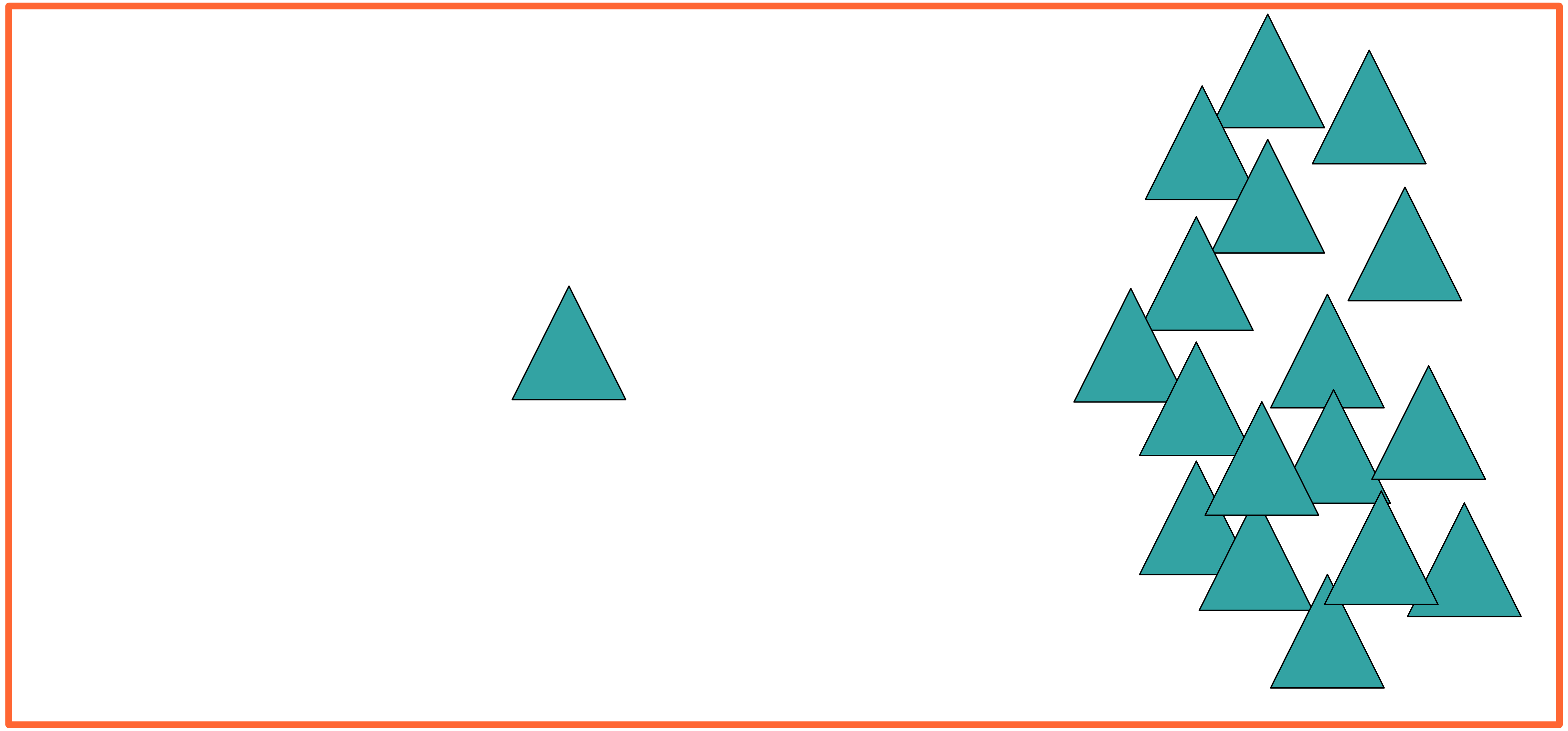
Construction of a kd-Tree

- **Given:**
  - An axis-aligned bbox enclosing part of the scene (cell / node of the kd-tree)
    - At the root, the box encloses the whole universe
  - List of the geometry primitives contained in this cell
- **The procedure (top down):**
  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)

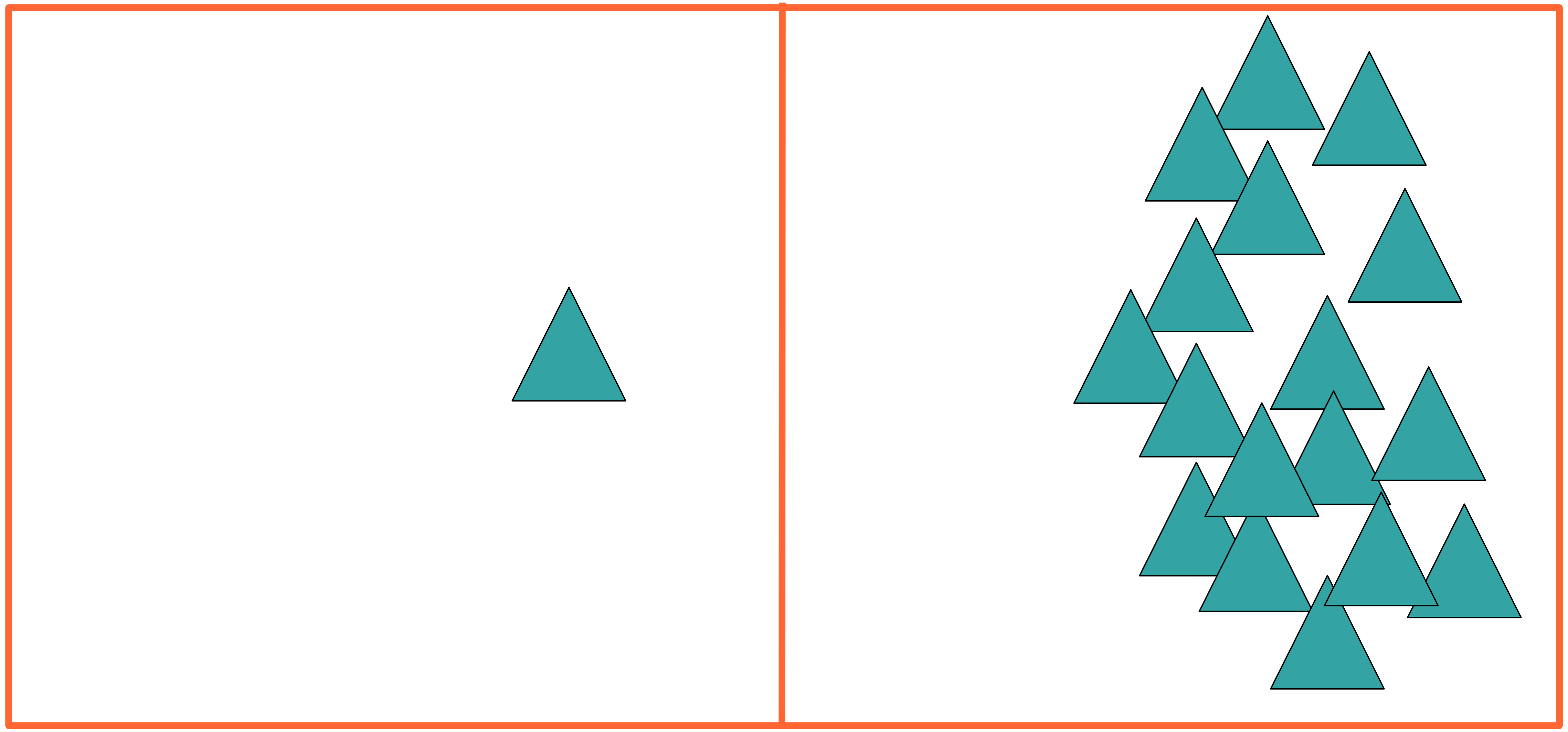
On Selecting a Splitting-Plane

- **Naïve selection of the splitting plane:**
  - Splitting axis:
    - Round Robin (x, y, z, x, y, z, ...)
    - Best: split along the longest axis of the node's region (not bbox of its contents!)
  - Split position (along the splitting axis):
    - Middle of the cell
    - Median of the geometry
- **In case the intended application is known: use a cost function!**
  - Choose a splitting plane such that the *expected* costs of a ray test are minimal
  - Try all 3 axes: search for the minimum along each axis
  - Choose axis / split position with the smallest minimum

Motivation of the Cost Function

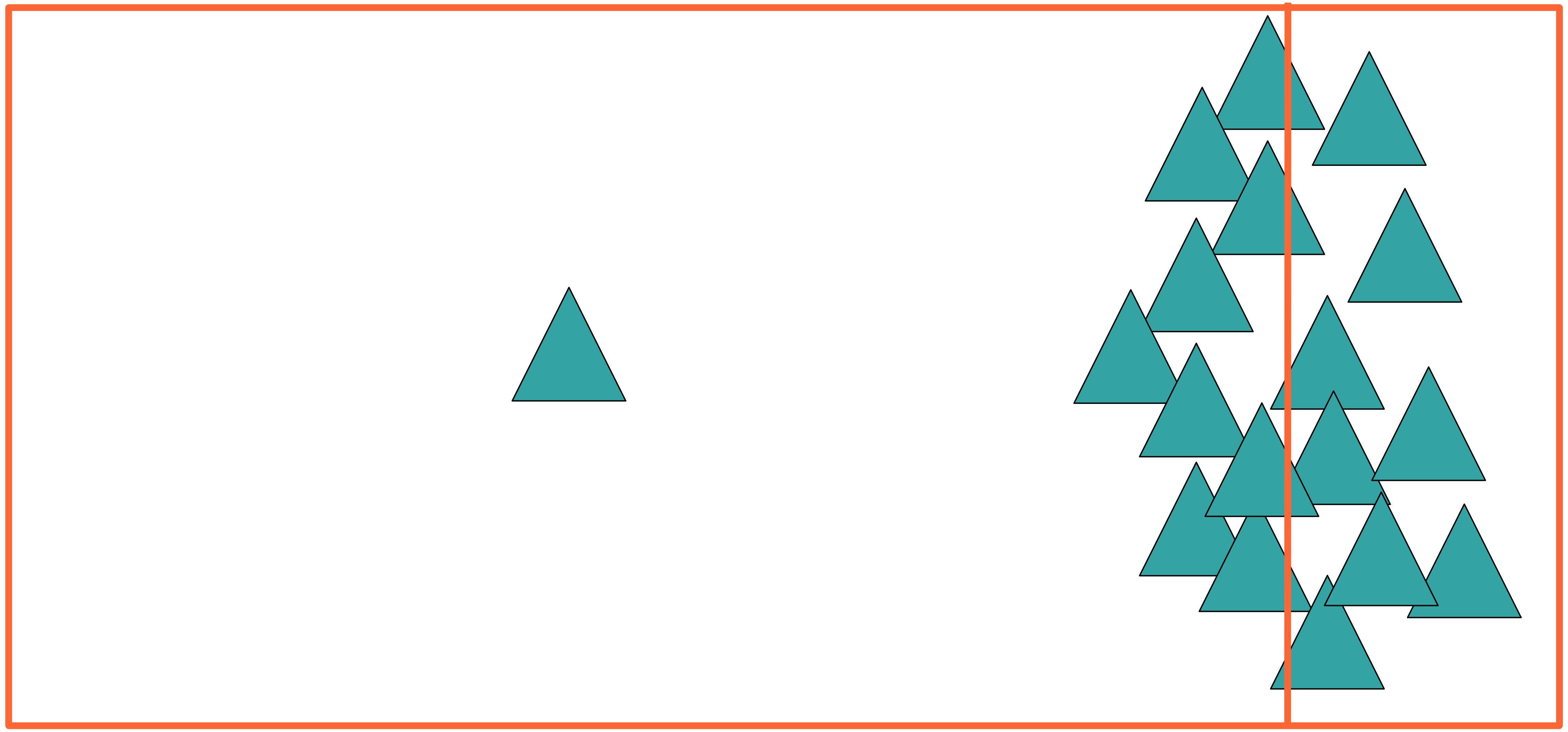


- Split in the middle:



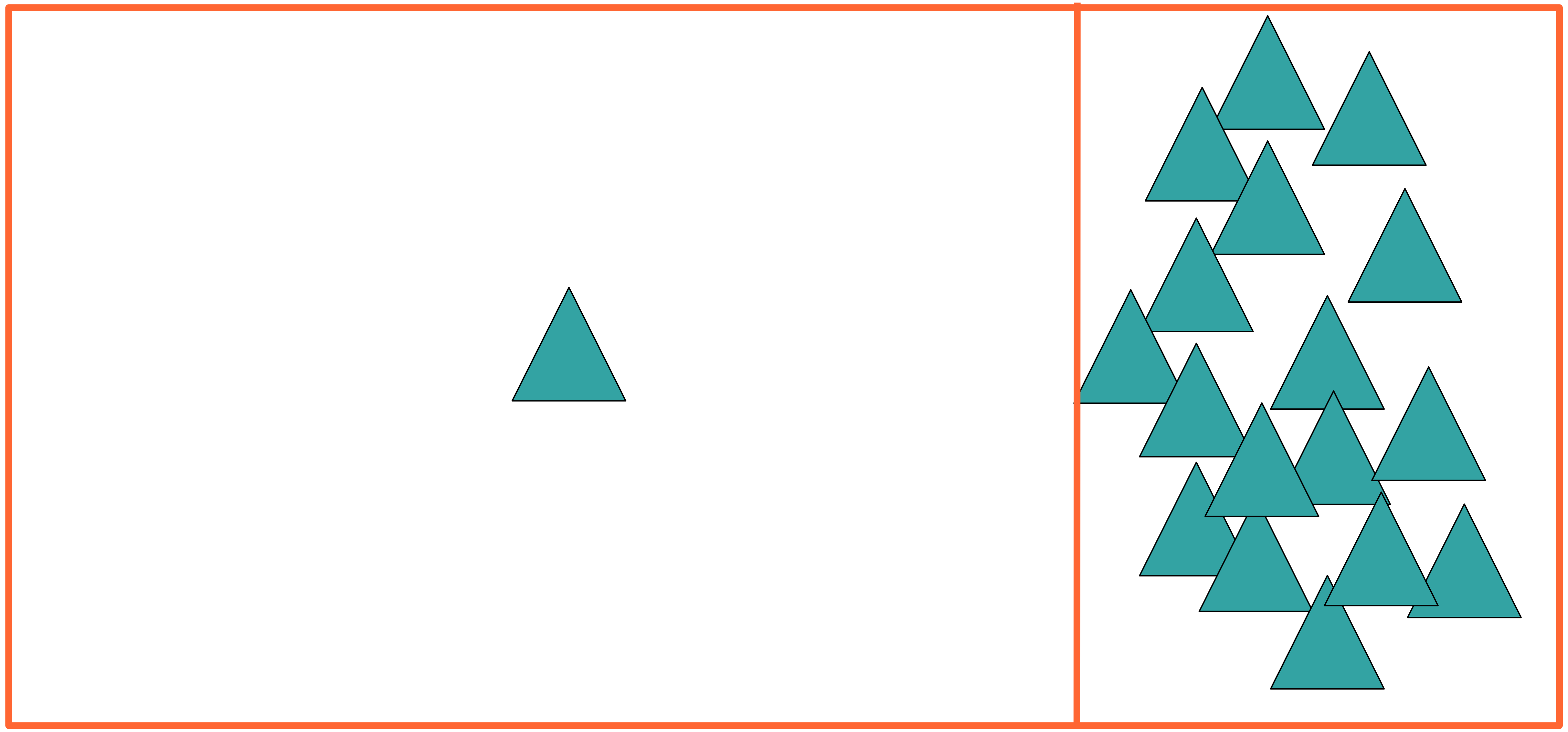
- The probability of a ray hitting either child is equal
- But the expected costs for handling are very different!

- Split along the geometry median:



  - The computational efforts for either child are equal
  - But the probability of a hit are very different!

- Cost-optimized heuristic:

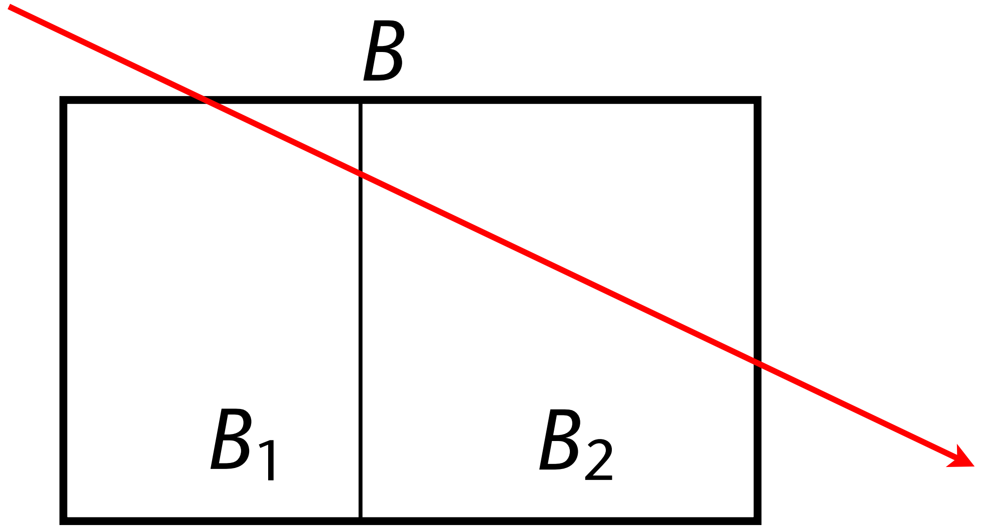


- The total expected costs are approximately similar

The Surface Area Heuristic (SAH)

- Question: How to measure the costs of a given kd-tree?
- Expected costs of a ray test:
  - Assume, we have reached node *B* during the ray traversal
  - Node *B* has children B_1, B_2
  - Expected costs = expected traversal time =

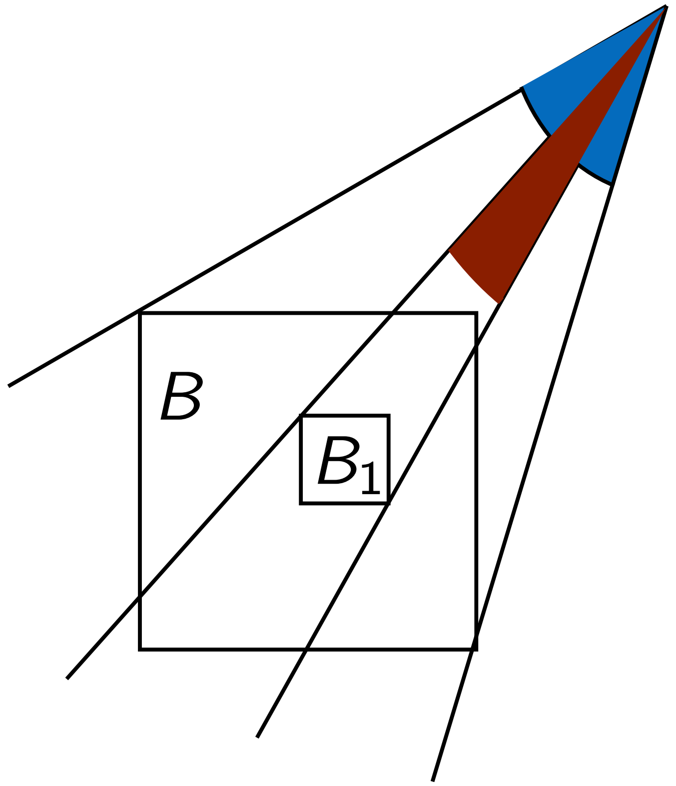
$$
\begin{aligned}
C(B) = &\ \mathrm{Prob}[\text{intersection with } B_1] \cdot C(B_1) \\
+ &\ \mathrm{Prob}[\text{intersection with } B_2] \cdot C(B_2)
\end{aligned}
$$

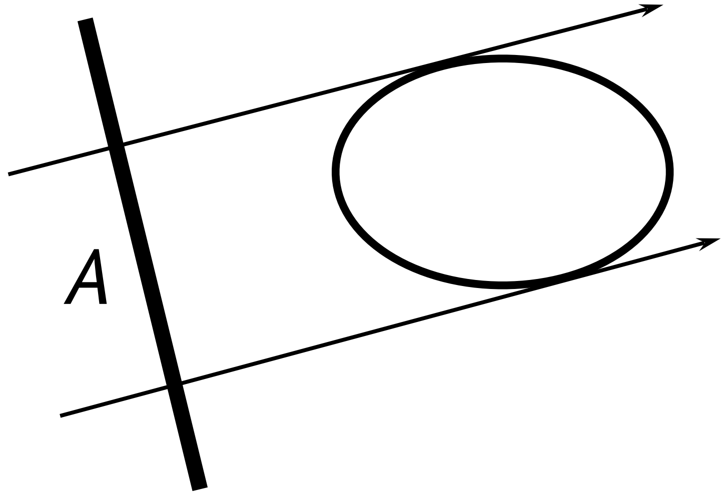


A "Handwavy" Derivation of the Probability

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

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





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

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

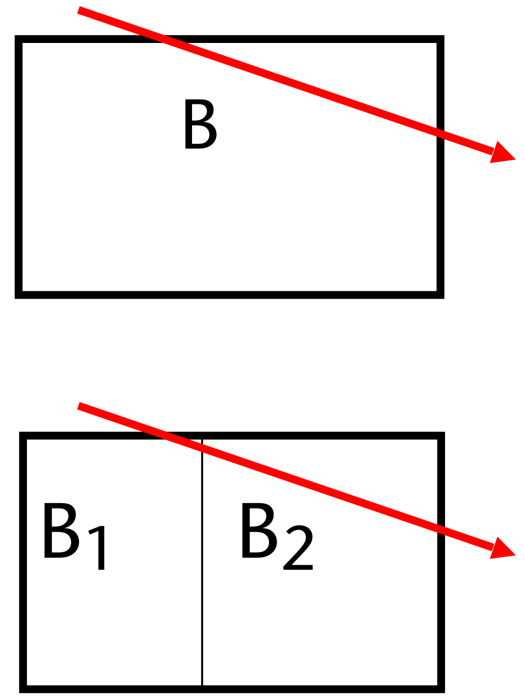
- The complete Surface Area Heuristic :
  minimize the following function when determining the splitting plane (thus, distributing the set of polygons):

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

A Stopping Criterion During KD-Tree Construction

- 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 = $t_p N$
  - Optimal split → costs = $t_s + t_p \left(\frac{\text{Area}(B_1)}{\text{Area}(B)} N_1 + \frac{\text{Area}(B_2)}{\text{Area}(B)} N_2 \right)$

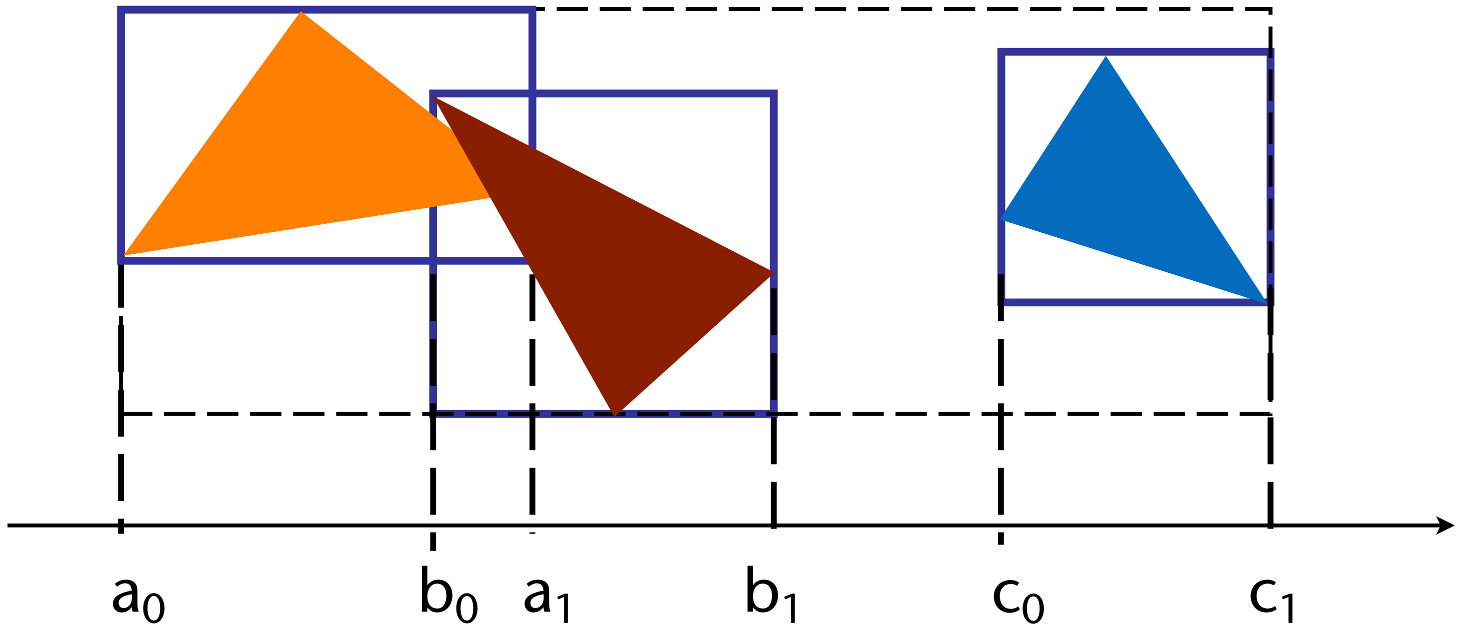
  where t_p = time for one ray-primitive test
  t_s = time for one intersection test of a ray with the splitting plane of the kd-tree node
  N = number of primitives



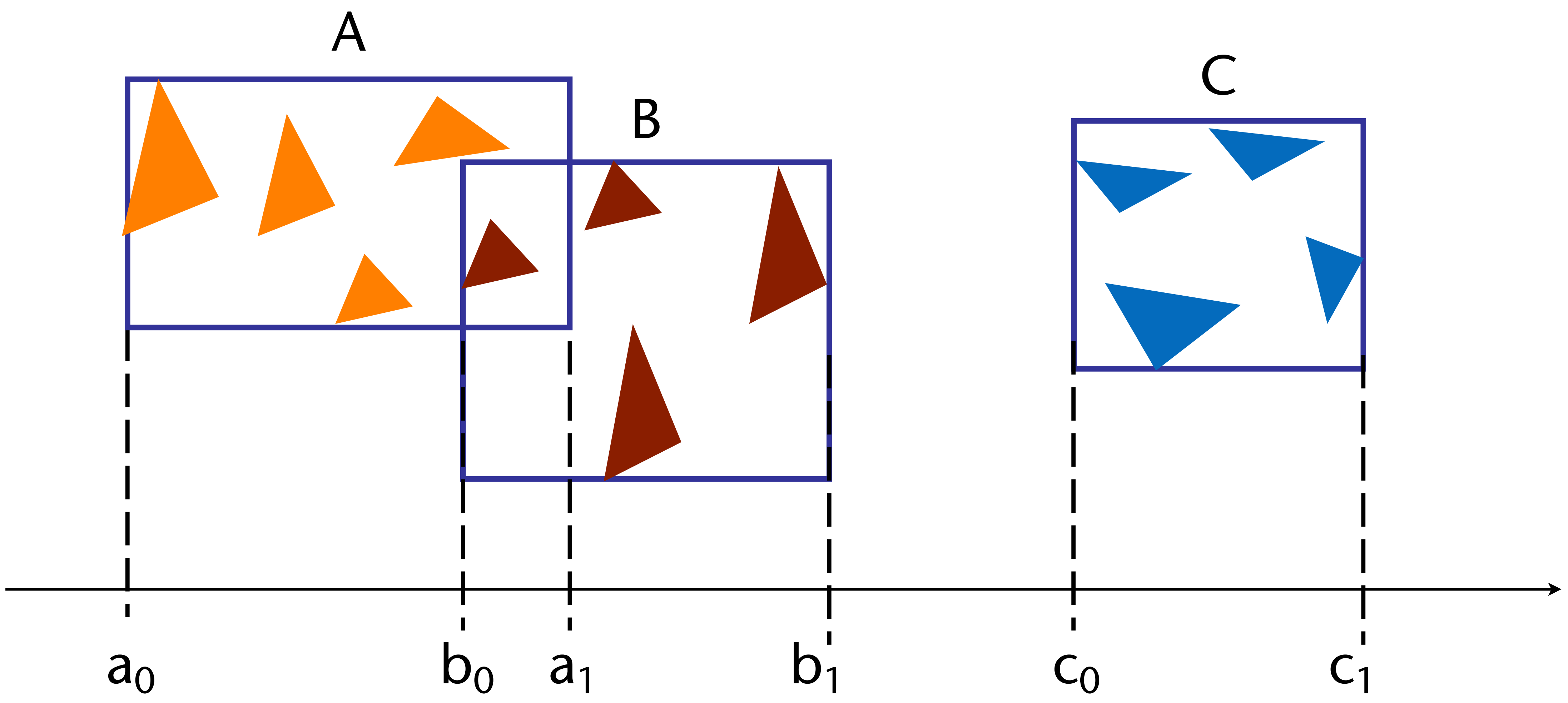
- Do the split iff costs of case 2 < costs of case 1
- In practice, we can make the following simplifying assumptions:
  - t_p = const for all primitives
  - $t_p : t_s = 80 : 1$ (determined by experiment, YMMV)

Approximation of the Optimal Splitting Plane

- It suffices to evaluate the cost function (SAH) only at a *finite* set of points along the splitting axis
  - The points are the borders of the bounding boxes of the triangles
  - In-between, the value of the SAH cost function must be (slightly) higher (because split polygons contribute to both sides)
- Sort all the end-points of all bboxes along the splitting axis, evaluate the SAH only at these points (*plane sweep*)
- Use *golden section search* to find global minimum*:*
  - Similar to bisection, but for bitonic fct's



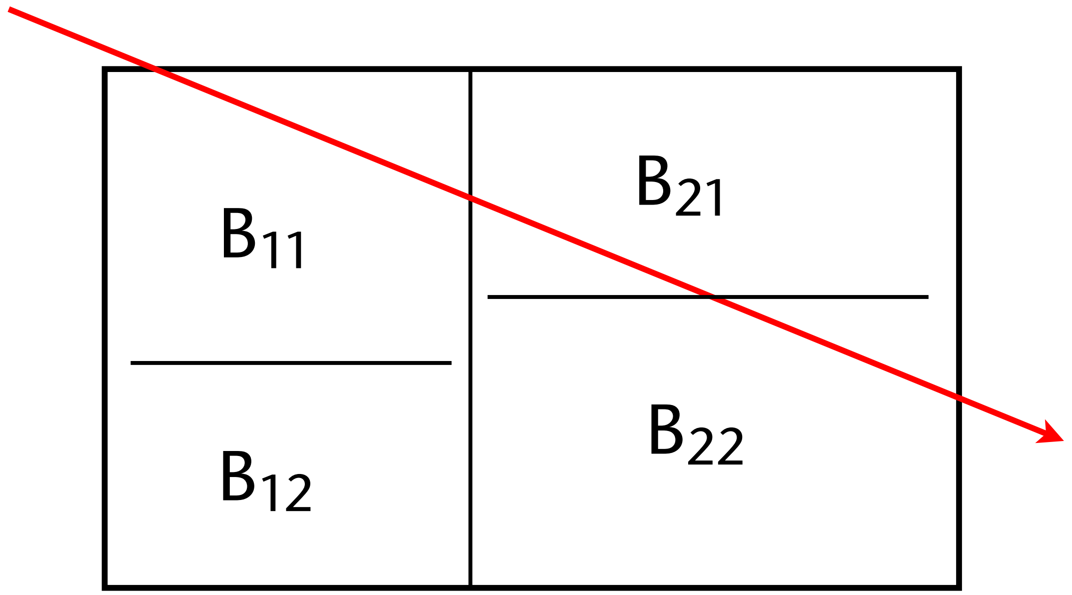
- If the number of polygons is very large (> 500,000, say) → only try to find the approximate minimum:
  - Sort polygons into "buckets", e.g., by simple clustering
  - Evaluate SAH only at the bucket borders



Remarks

- **Warning**: for other queries (e.g. range queries, collision detection, …) 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 …

Better KD-Trees for Raytracing

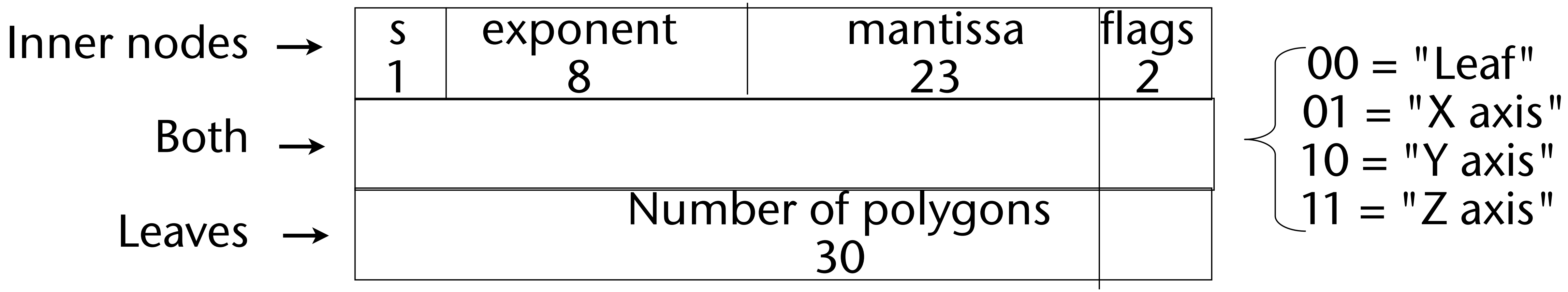
- Before applying SAH, test whether an empty cell can be split off that is "large enough" ; if yes, do that, no SAH-based splitting
- Additional stopping criterion:
  - If the volume of the cell is too small, then no further splitting
  - Criterion for "too small" (e.g.): Vol(cell) < $\varepsilon \cdot$ 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 *contains* many polygons, then use that 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 (one pointer)
- Naïve implementation: 16 Bytes + 1 Bit = 17 Bytes $\longrightarrow$ very cache-inefficient
- Optimized implementation:
  - 8 Bytes per node (!)
  - Yields a speedup of 20% (some have reported even a factor of 10!)

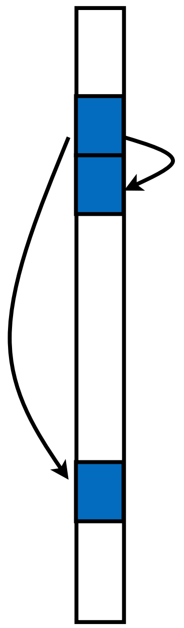
Concrete Implementation in C

- Idea of optimized storage: overlay the data
- Store all flags in just 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
};
```

- For inner nodes: just 1 pointer to the children
  - Maintain array of kd-tree nodes yourself (no **`malloc()`** nor **`new`**)
  - Store both children in contiguous array elements; or
  - store one child always directly after the parent.



- Overlay pointer to children with pointer to primitives
- Together:

```
class KdNode
{
 sprivate:
 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
 ...
```

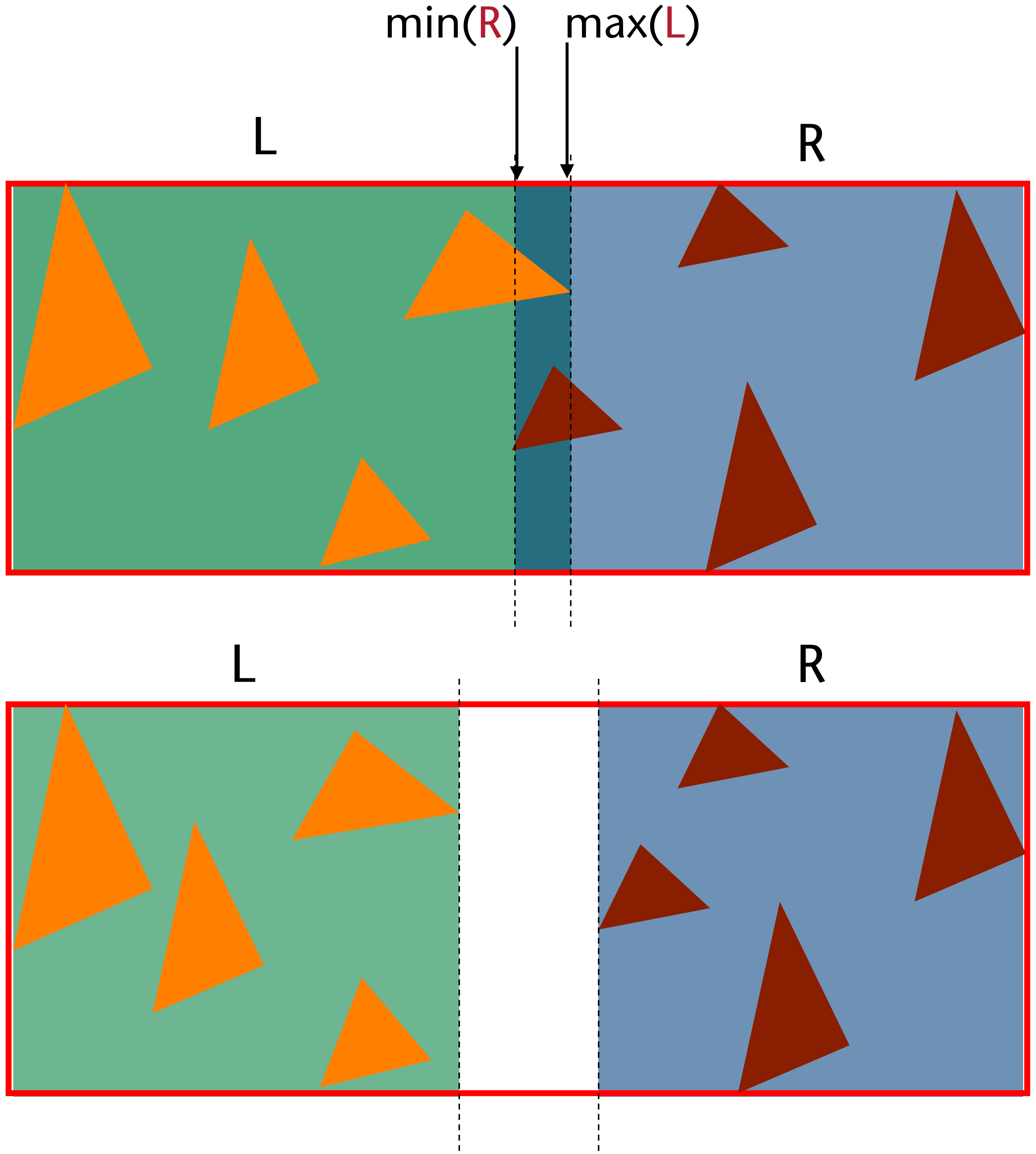
If m_nPrims == 1

If m_nPrims > 1

- Note: this showcases very well why access to instance variables ("member variables" in C++ lingo) has to be done strictly via methods! (no direct access)
  - When writing **m_split** , make sure that **m_flags** is maintained (e.g., by overwriting the lower two bits with the original value again)!
  - When reading/writing **m_nPrims** , don't forget to shift the value!

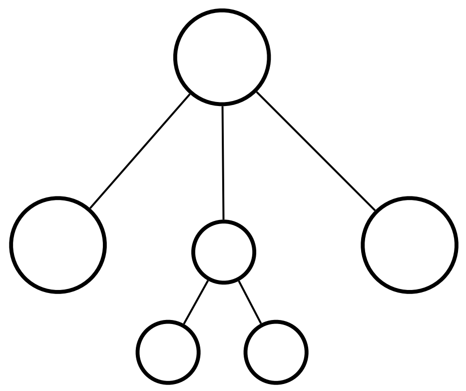
BoxTrees / Spatial KD-Trees (SKD-Tree) [1987/2002/2006]

- A variant of the kd-tree with potentially *overlapping* child boxes
- 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 usually 2*N*-3*N* pointers to triangles, *N* = number of unique triangles in the kd-tree
- Disadvantage: 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

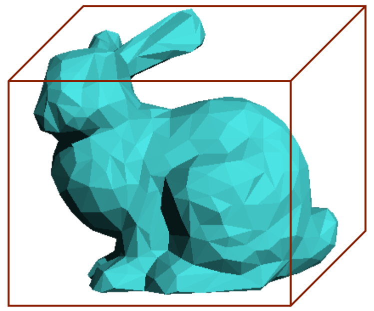


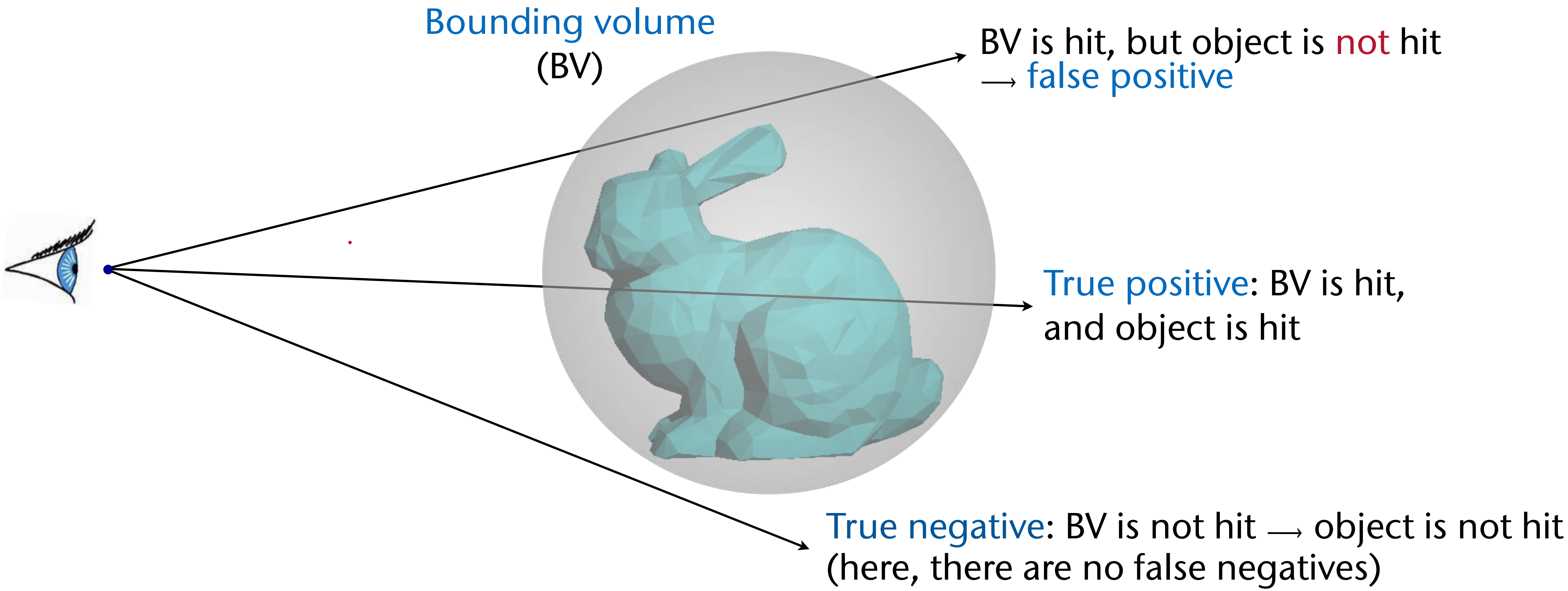
Spatial Partitioning vs. Object Partitioning

- Spatial partitioning: acceleration data structure subdivides space, objects (e.g., triangles) are associated afterwards to the cells
- Object partitioning: partition the set of objects, associate a bounding volume (= subset of space) with each
- In reality, the borders between the two categories are not clear-cut!

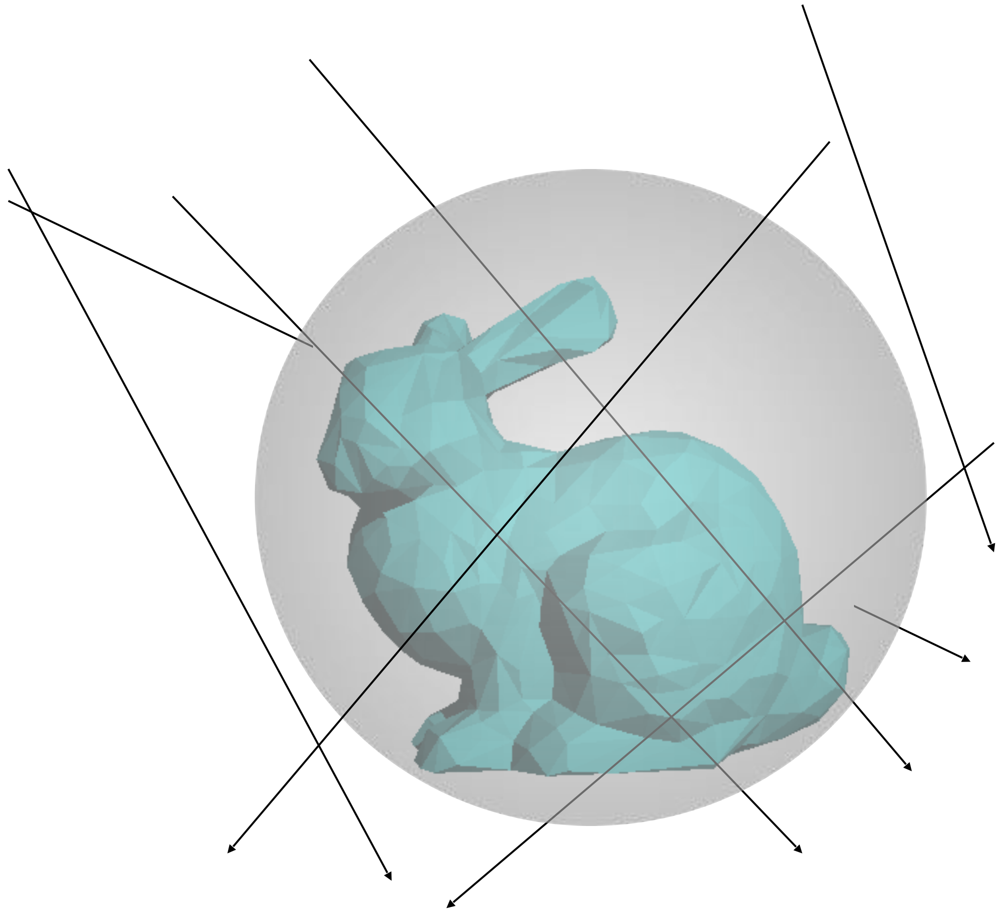
Bounding Volumes (BVs)

- Basic idea: save costs by doing precomputations on the scene allowing for fast filtering of the rays during run-time
- Here: approximate complex, geometric objects, or sets of objects, by some outer "hull"





- Is it worthwhile to use BVs?
- Consider a large number of rays, coming in from all different directions
- Then, the method does improve performance, iff



$$\overbrace{T_{BV} + \frac{\#\text{ rays hitting BV}}{\text{total }\#\text{ rays}} \cdot T_{Obj}}^{\text{Average cost per ray } \textit{with} \text{ BV}} < \overbrace{T_{Obj}}^{\text{Average cost per ray } \textit{without} \text{ BV}} \qquad \Leftrightarrow$$

$$T_{BV} < \left(1 - \frac{\#\text{ rays hitting BV}}{\text{total }\#\text{ rays}}\right) \cdot T_{Obj} = \frac{\#\text{ rays missing BV}}{\text{total }\#\text{ rays}} \cdot T_{Obj}$$

where T_{BV} = cost for intersection with BV,
T_{Obj} = cost for intersection with object (e.g., n polygons)

Bremen

The Dichotomie of BVs

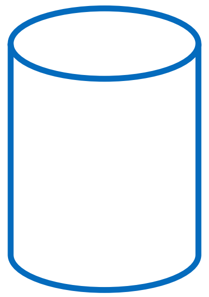
- Either, we try to make T_{BV} *small*,

  i.e., we try to make the BV "simple" (with respect to ray intersection)

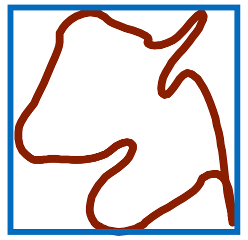
- Or, we try to make $\frac{\#\text{ rays missing BV}}{\text{total }\#\text{ rays}}$ *large*,

  i.e., we try to make the BV tight

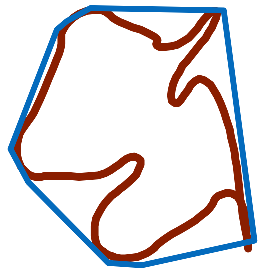
Examples of Bounding Volumes



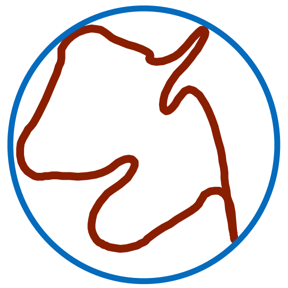
Cylinder
[Weghorst et al., 1985]



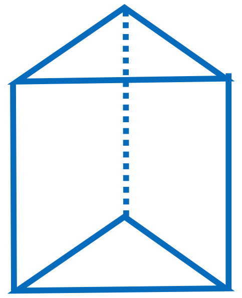
Box, AABB (R*-trees)
[Beckmann, Kriegel, et al., 1990]



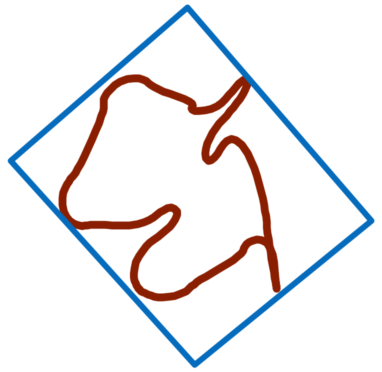
Convex hull
[Lin et. al., 2001]



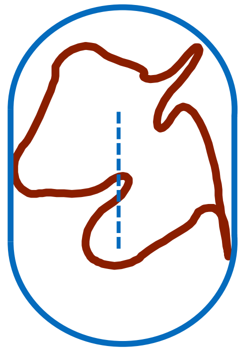
Sphere
[Hubbard, 1996]



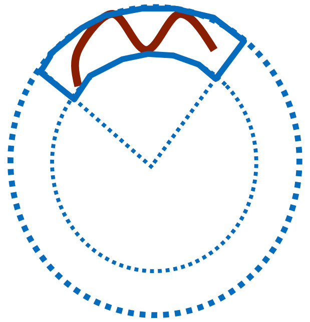
Prism
[Barequet, et al., 1996]



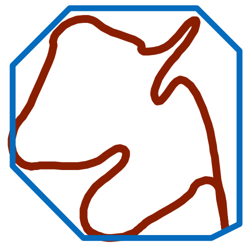
OBB (oriented bounding box)
[Gottschalk, et al., 1996]



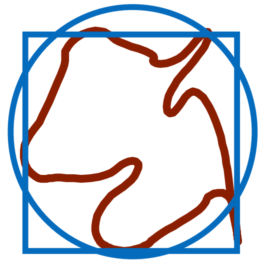
Capsule
[Larsen, 1999]



Spherical shell
[...]



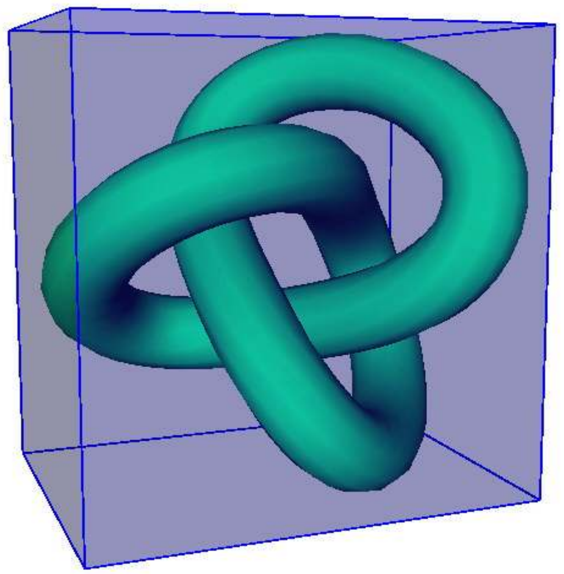
k-DOPs / Slabs
[Zachmann, 1998]

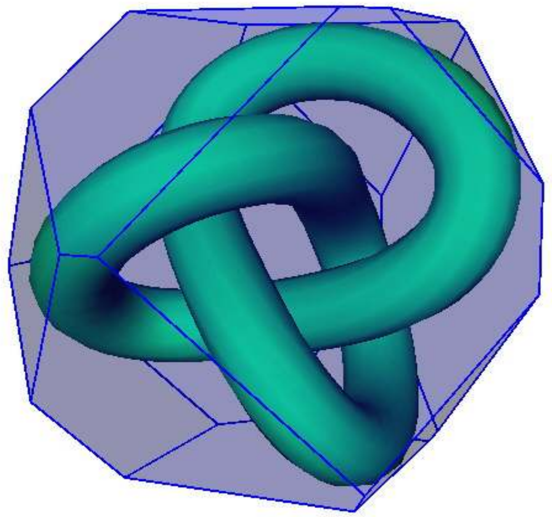


Intersection of
several, other BVs

Examples of k-DOPs

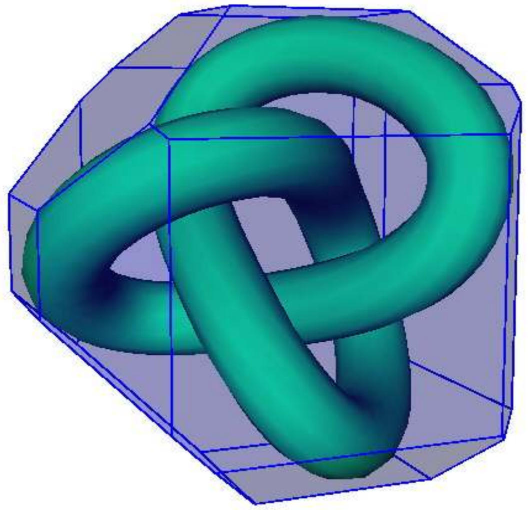
6-DOP
(AABB)

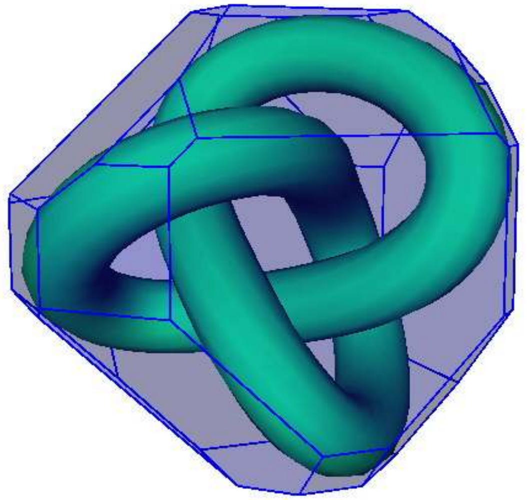




14-DOP

18-DOP





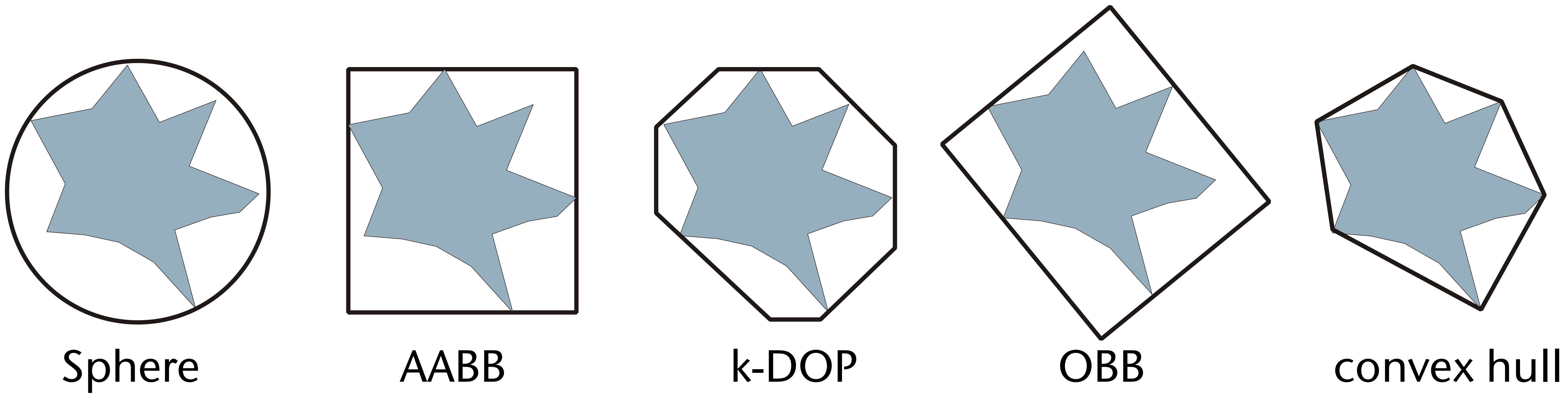
26-DOP

More information in the course "Virtual Reality and Simulation"

Qualitative comparison

Better approximation,

higher build and intersection costs



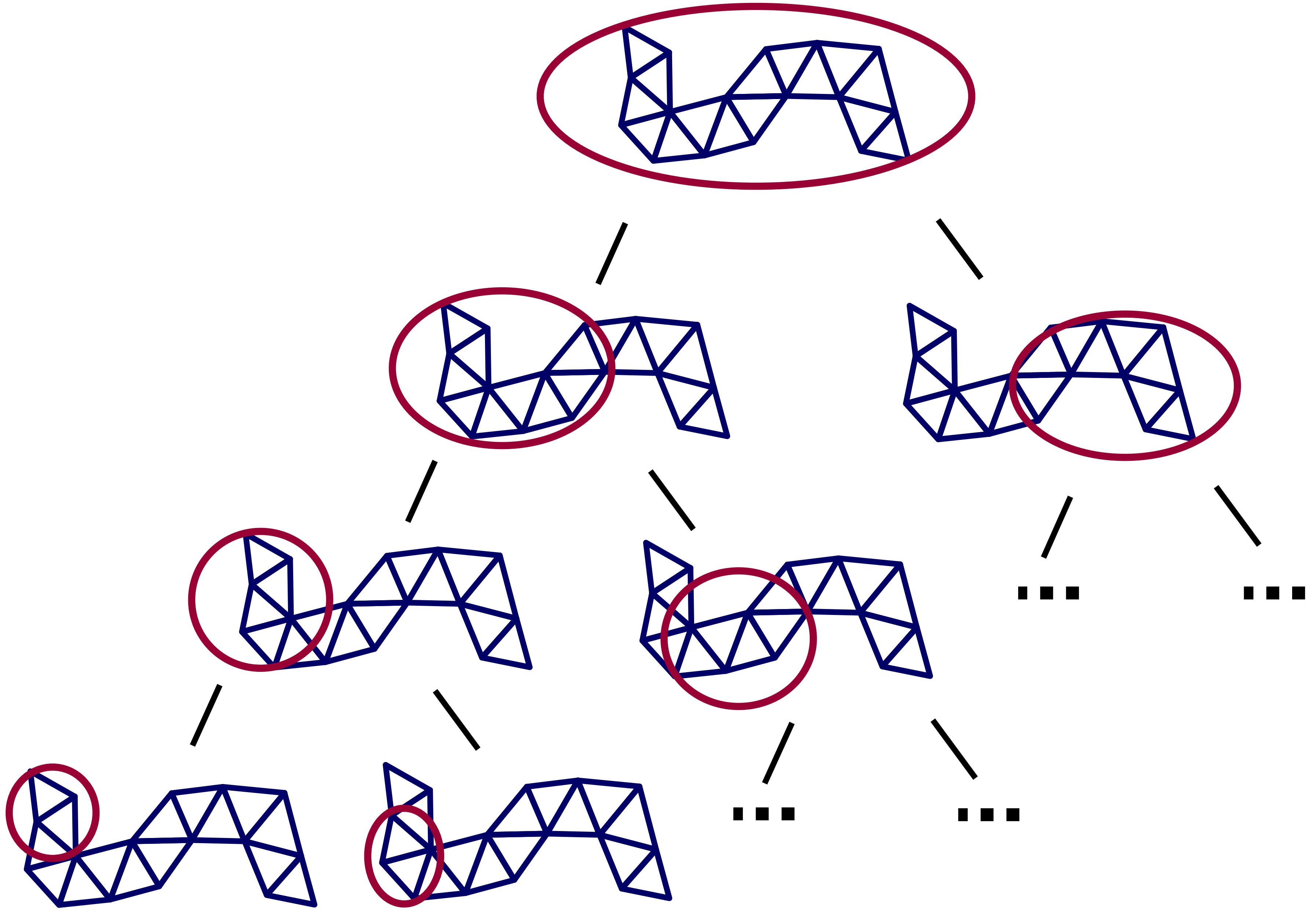
Smaller computational costs for overlap test ,

more false positives

The Bounding Volume Hierarchy (BVH)

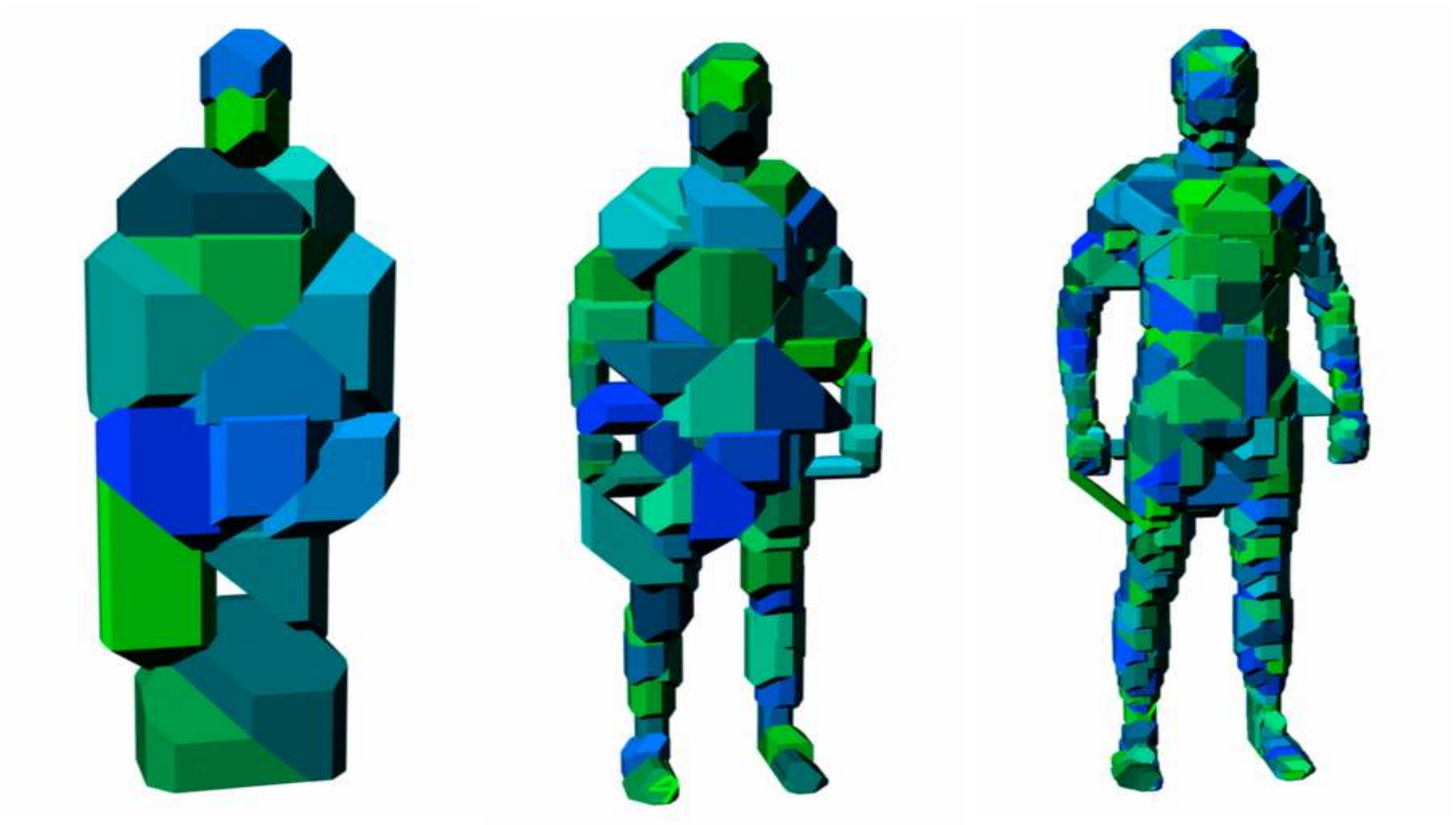
- Definition: a BVH over a set of primitives, $\mathcal{P}$, is a tree where each node is associated with
  - a subset of $\mathcal{P}$; and
  - a BV $\mathcal{B}$, that encloses all primitives in this subset.
- Remark:
  - Often, we use the BV as a synonym for the *node* in the BVH
  - Primitives are usually *stored only* at leaf nodes
    - Feel free to experiment; exceptions might make sense
  - Usually, the set of primitives is *partitioned*, i.e., let $\mathcal{P}_i$ = the subset of primitives associated with the node $\mathcal{B}_i$, then all $\mathcal{P}_i$ are disjoint
    - Again, feel free to experiment

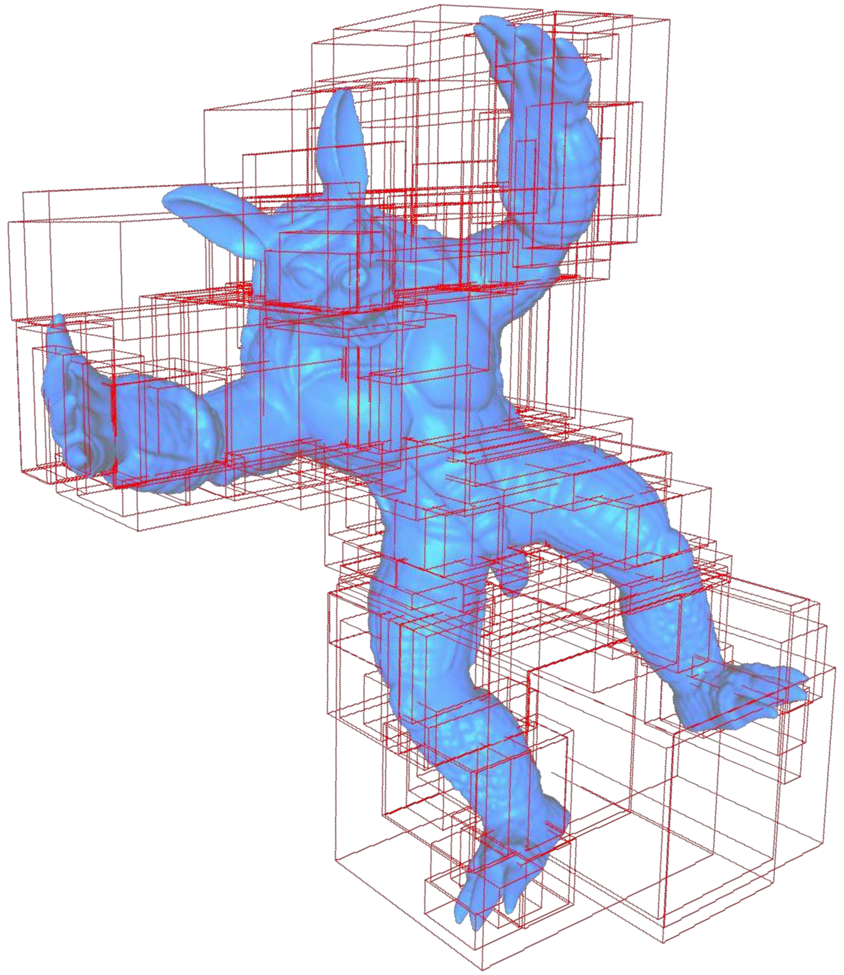
- Schematic example:

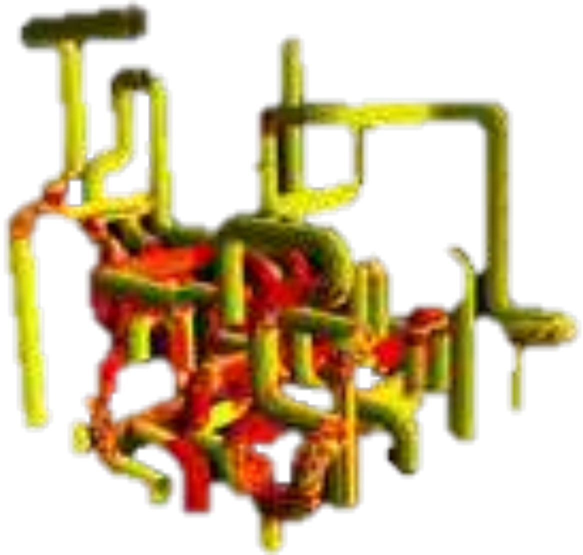


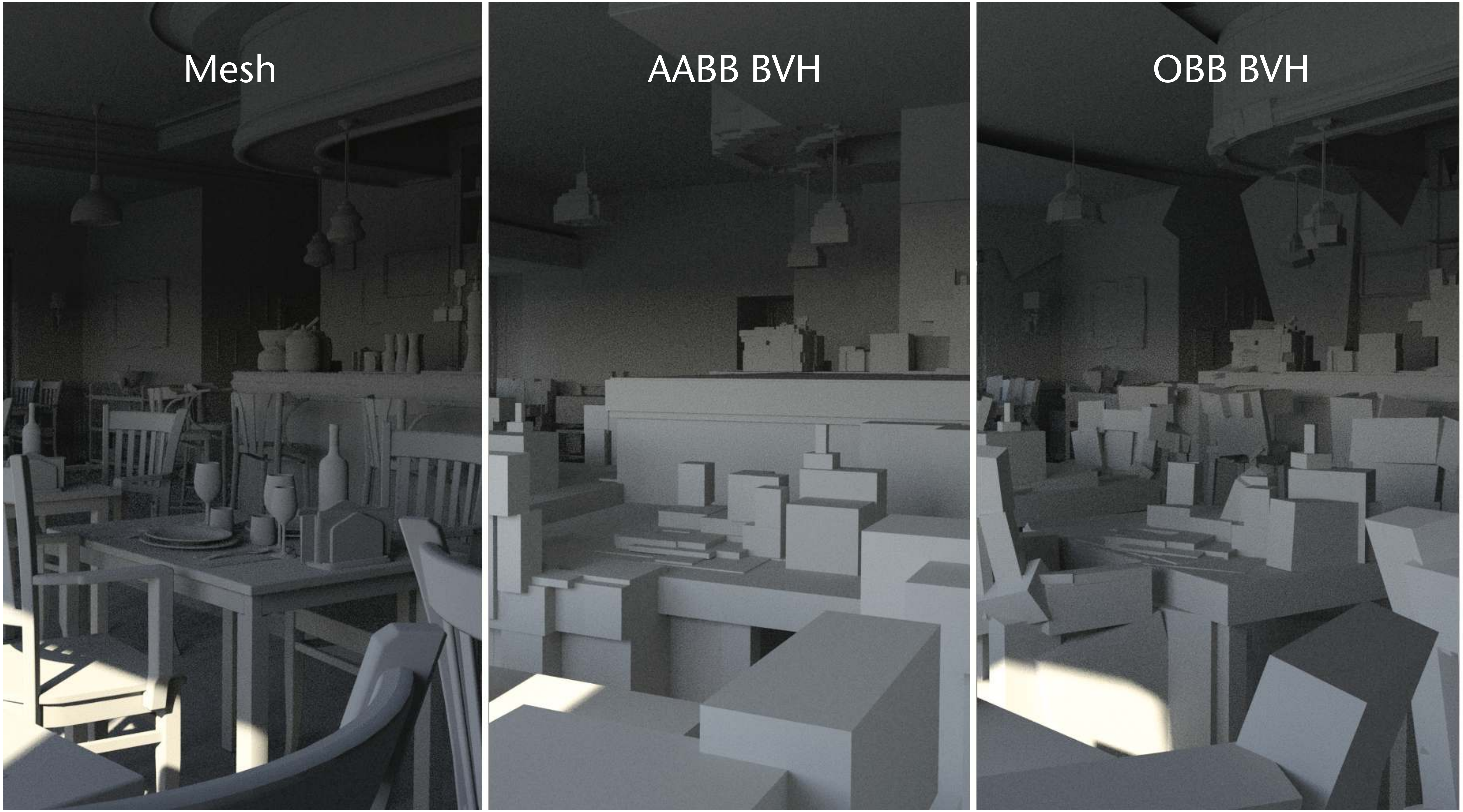
- Parameters & variations:
  - The kind of BV used
  - *"Arity"* (degree of the nodes)
  - Stopping criterion (in particular, number of triangles per leaf)
  - Criterion for partitioning the primitives (guiding the construction)

Examples and Visualizations of All the Boxes on a Level of a BVH

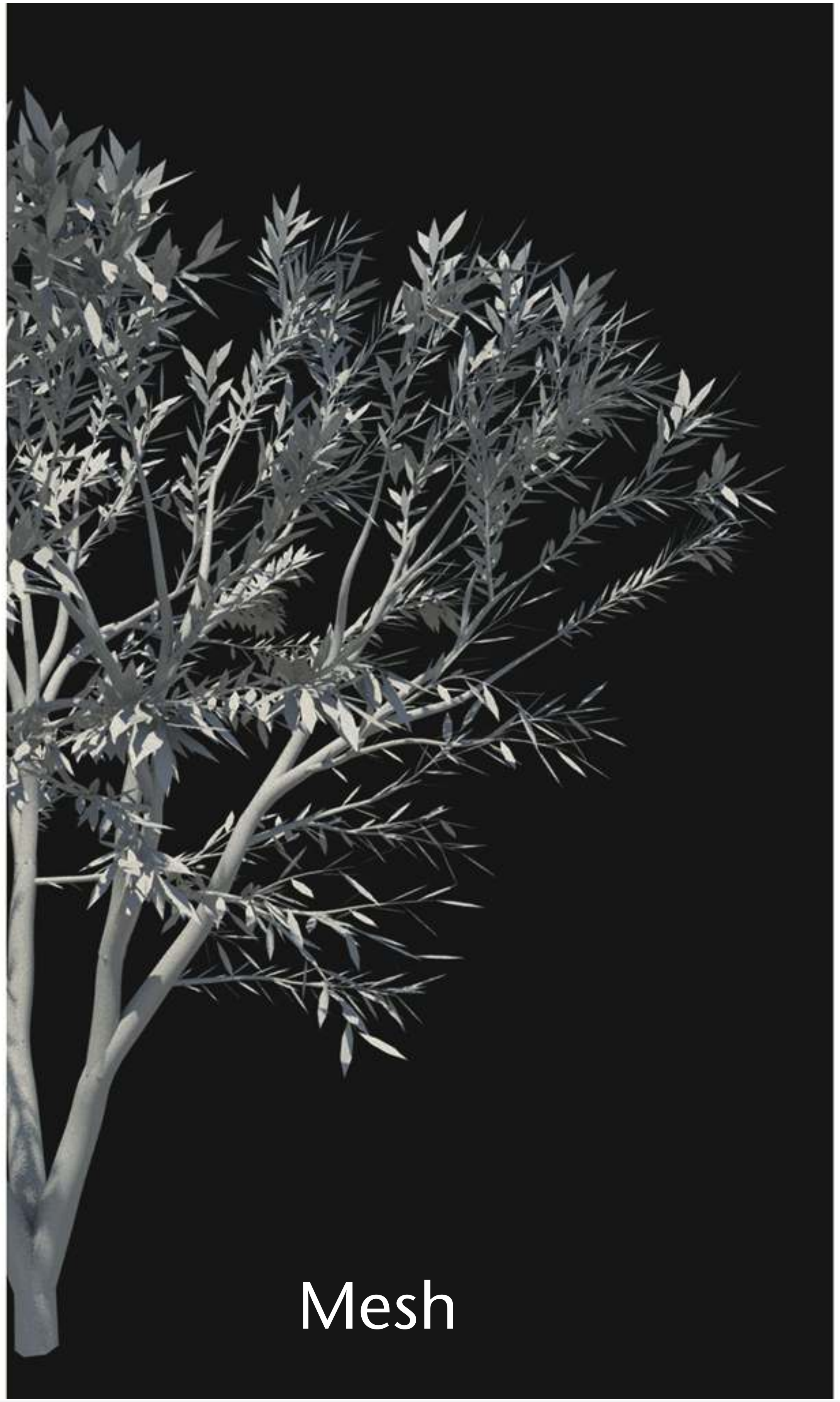


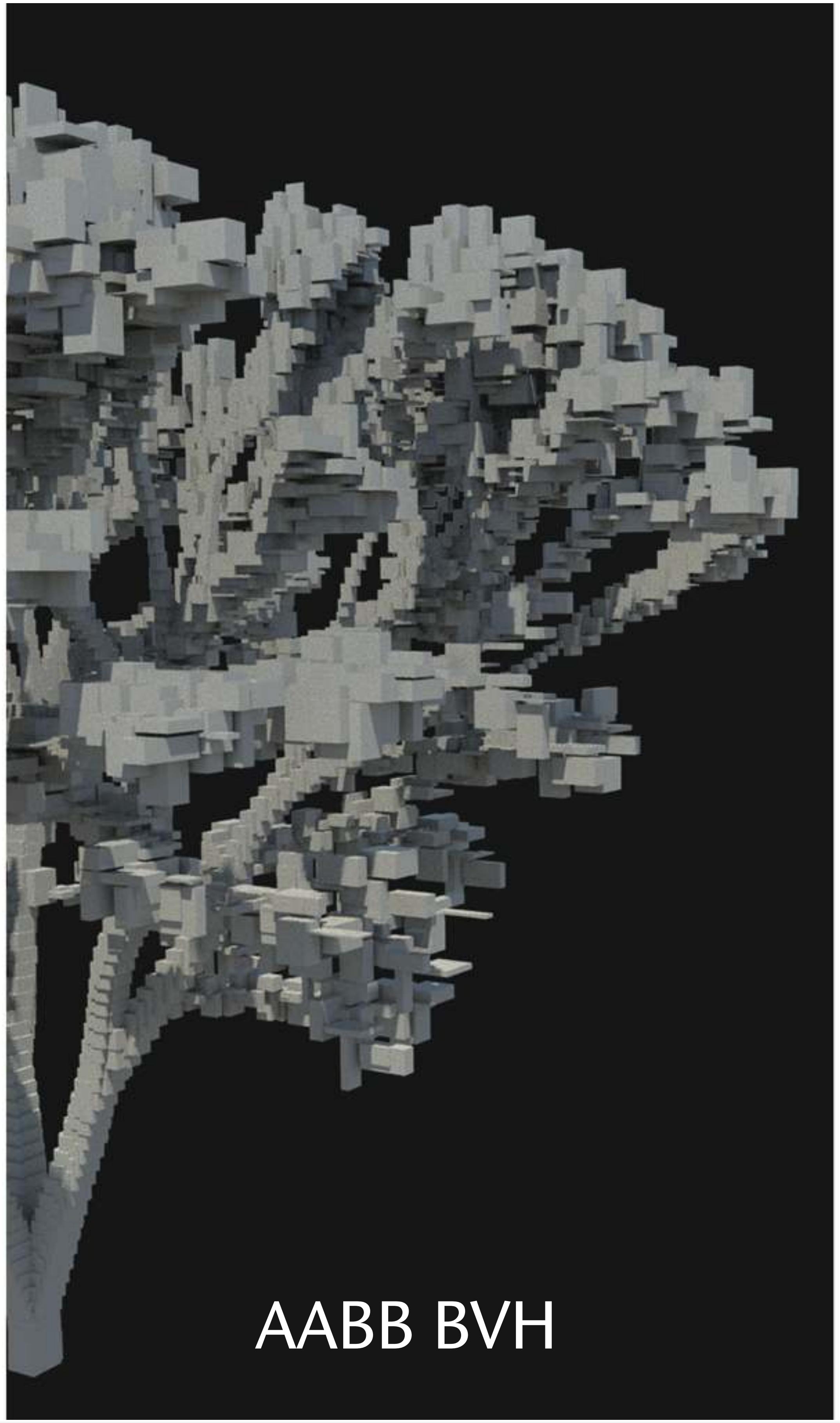






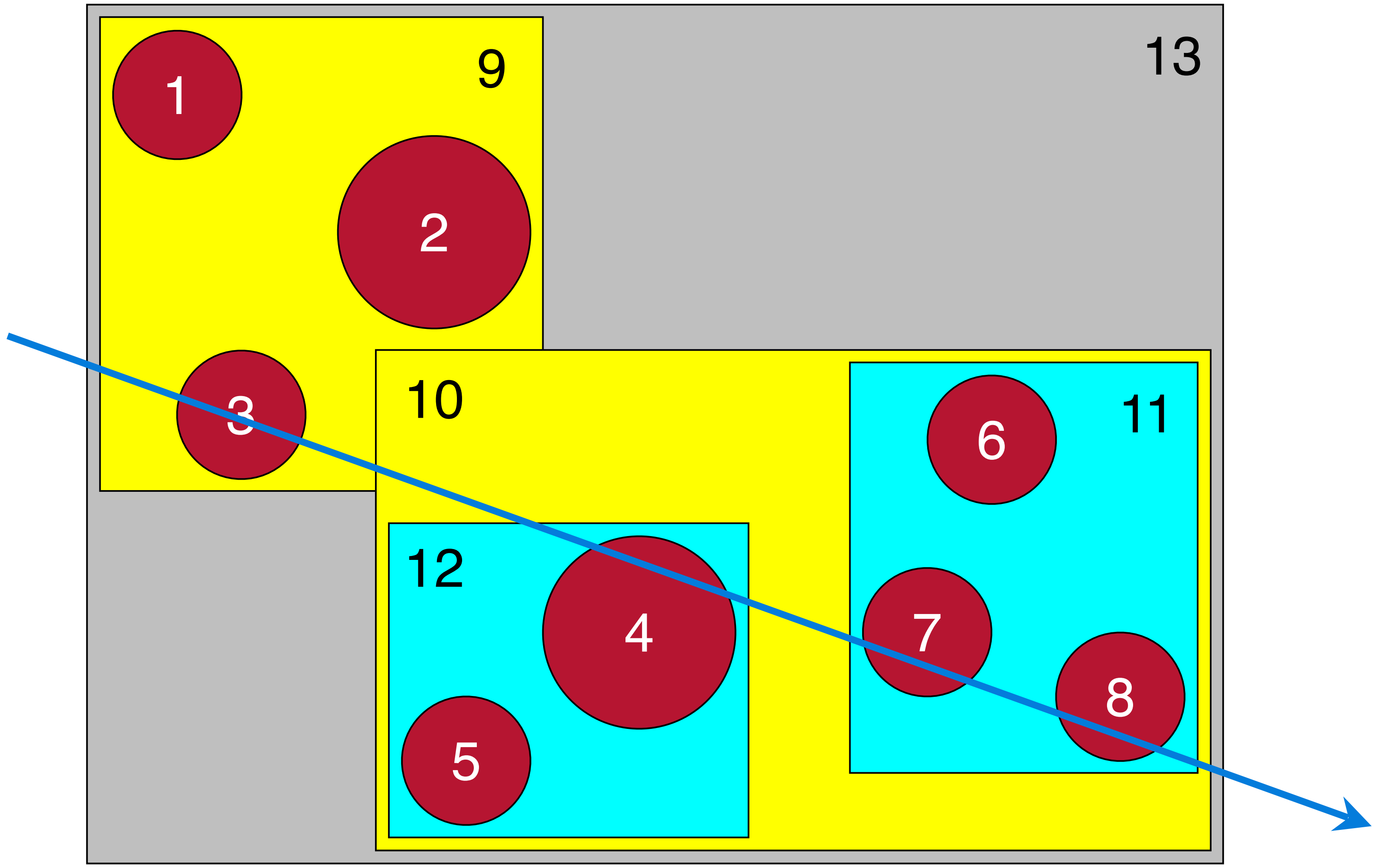
[Vitsas et al., 2023]

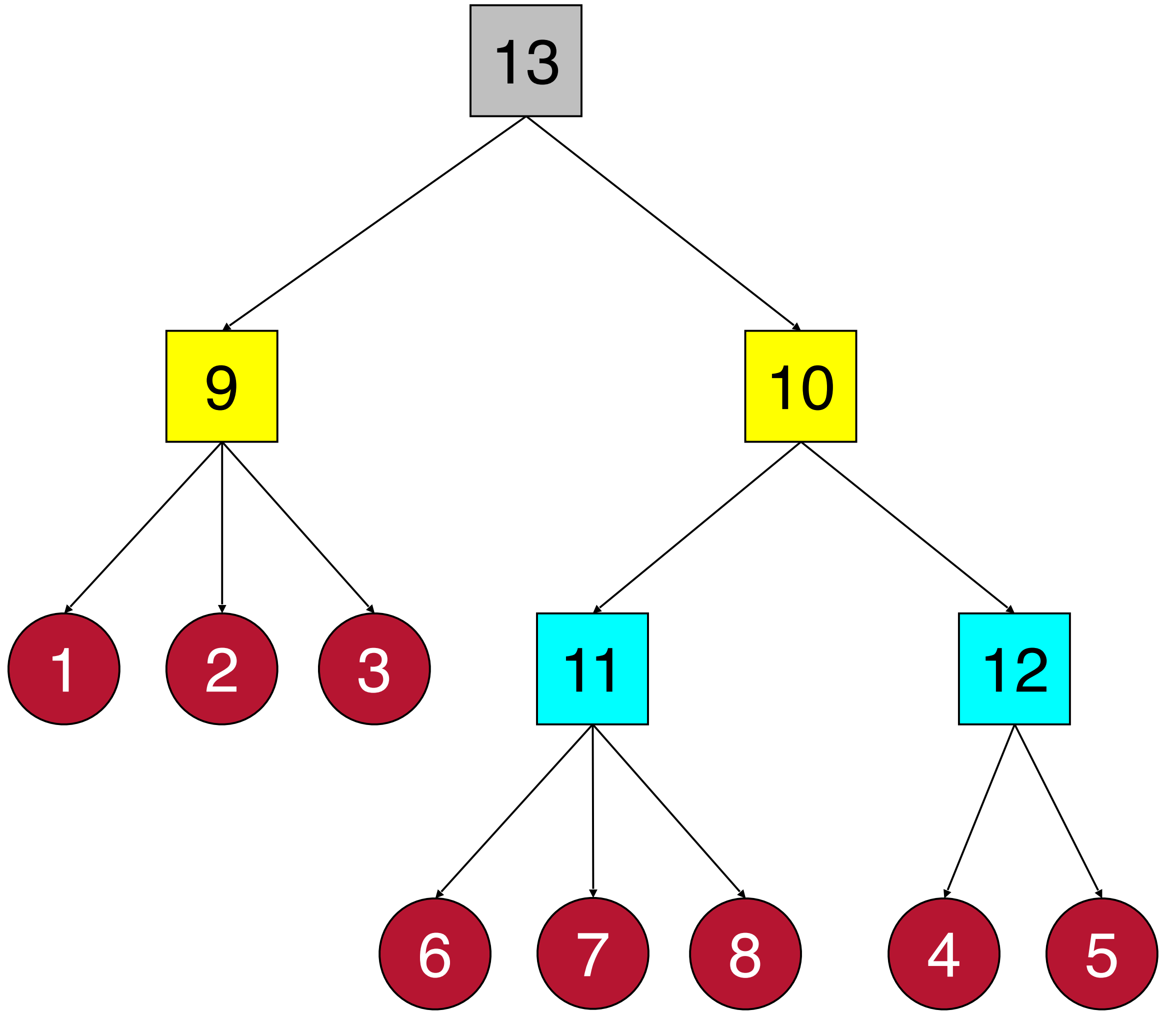

Mesh


AABB BVH

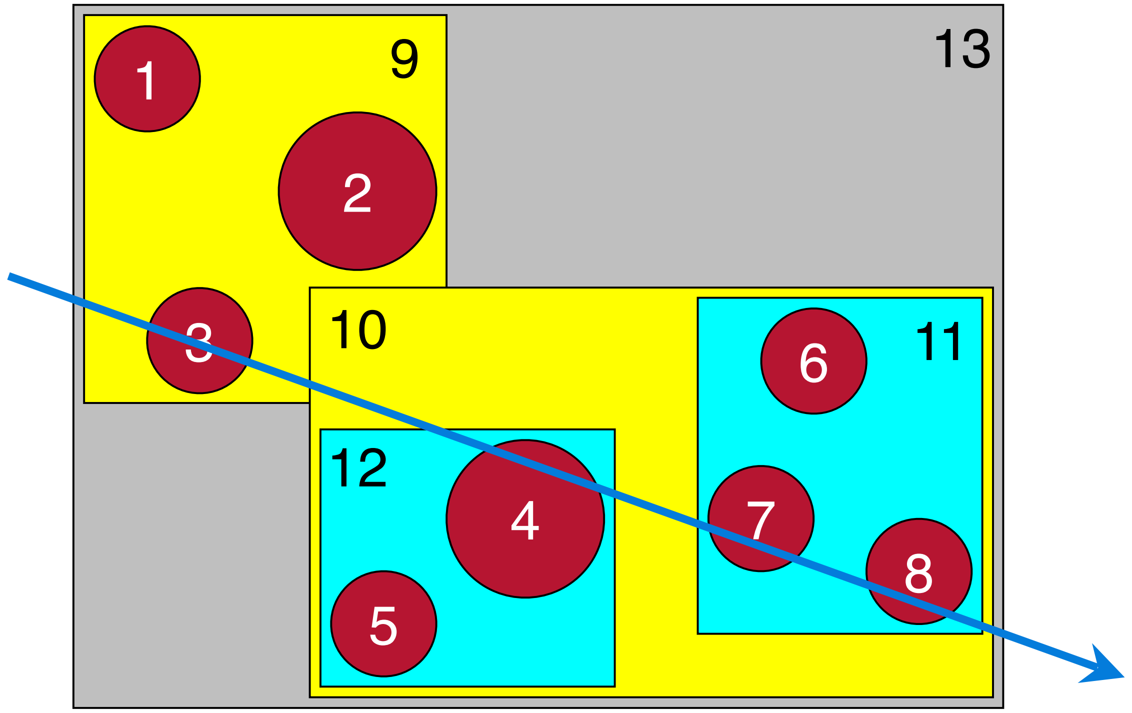

OBB BVH

Example for the Traversal of a BVH with a Ray



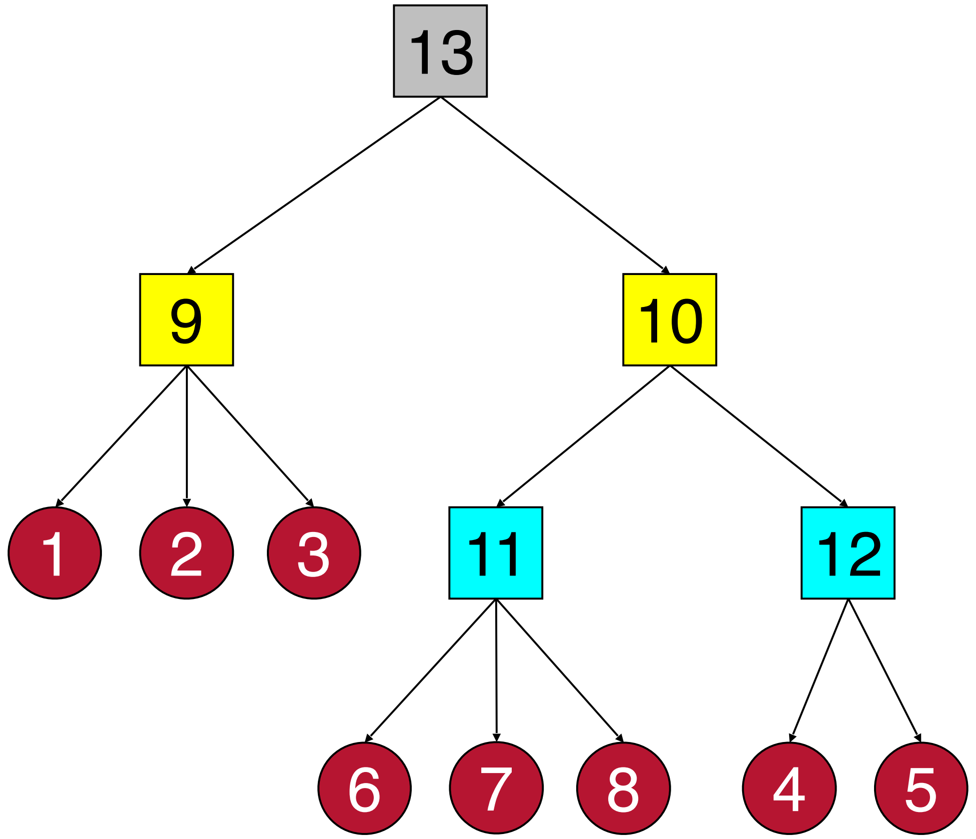


- Test box 13 ⟶ yes
  - Test box 9 ⟶ yes
    - Test obj 1 ⟶ no
    - Test obj 2 ⟶ no
    - Test obj 3 ⟶ yes
  - Test box 10 ⟶ yes, but intersection point is farther away



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

- Question: why did we start with BV 9?



A 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 (= *lower bound on distance of enclosed primitives*)
- Consequence: should not use simple recursion / stack any more
- Use **priority queue**

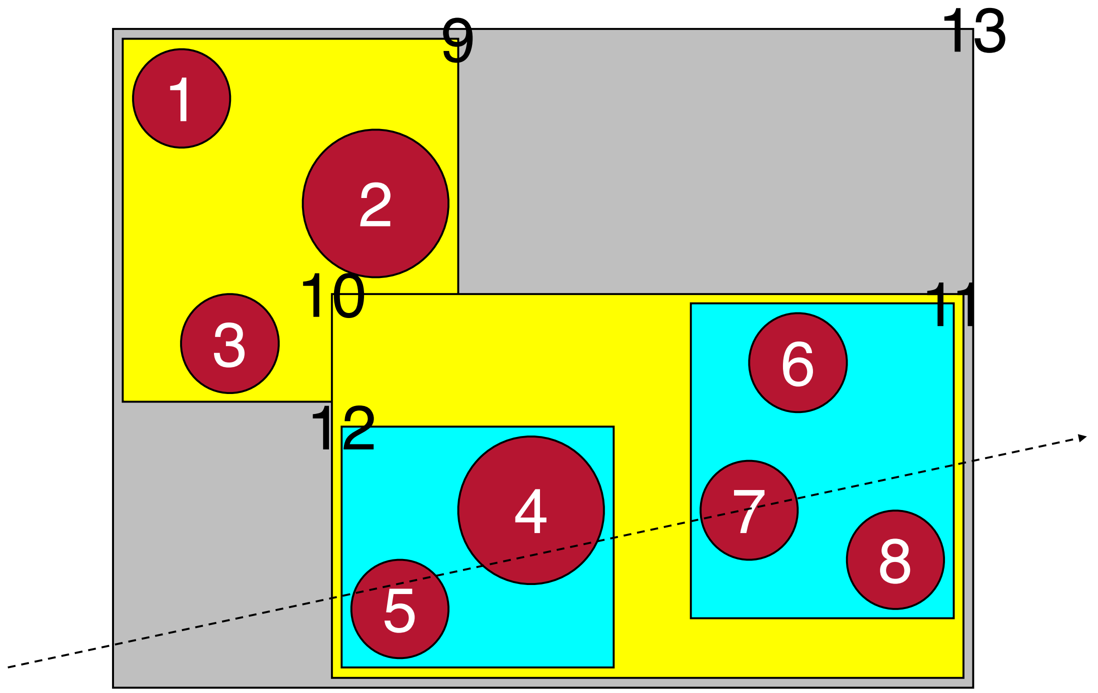
Algorithm

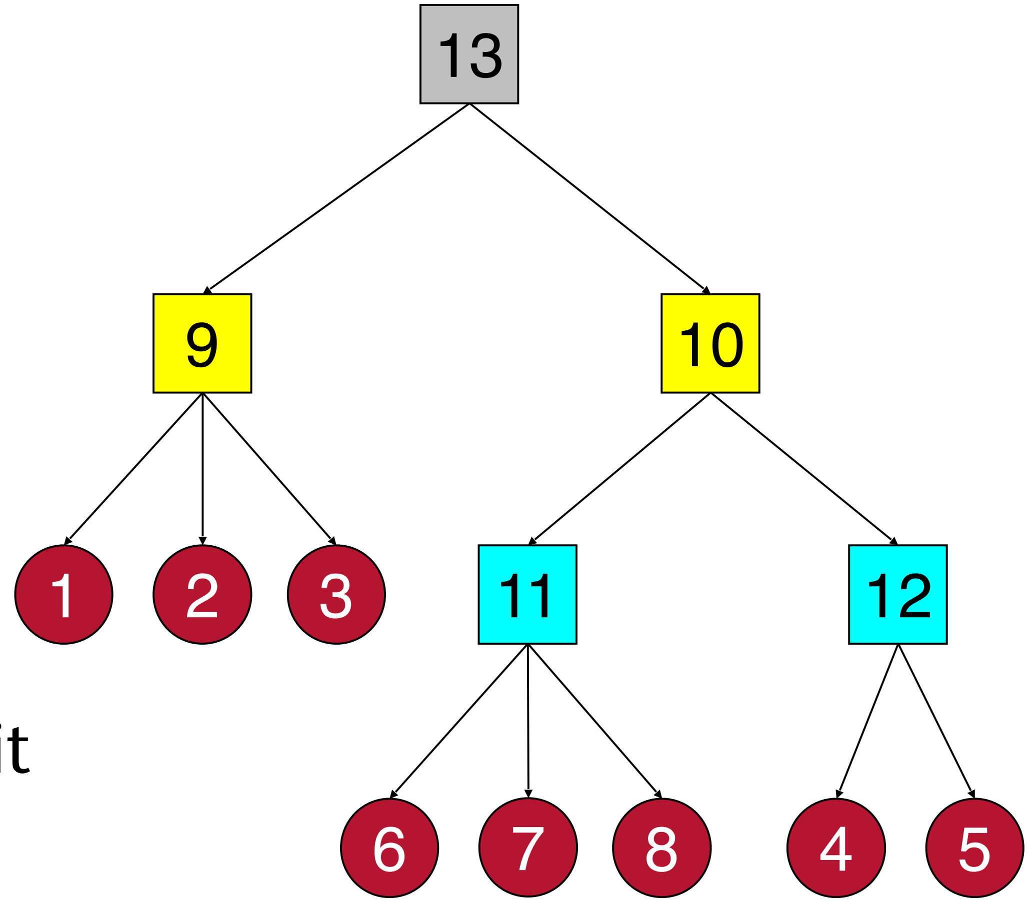
- Maintain a p-queue
  - Contains all BVs (= BVH nodes) that still need to be visited
  - Sorted by their distance from ray origin (along ray)

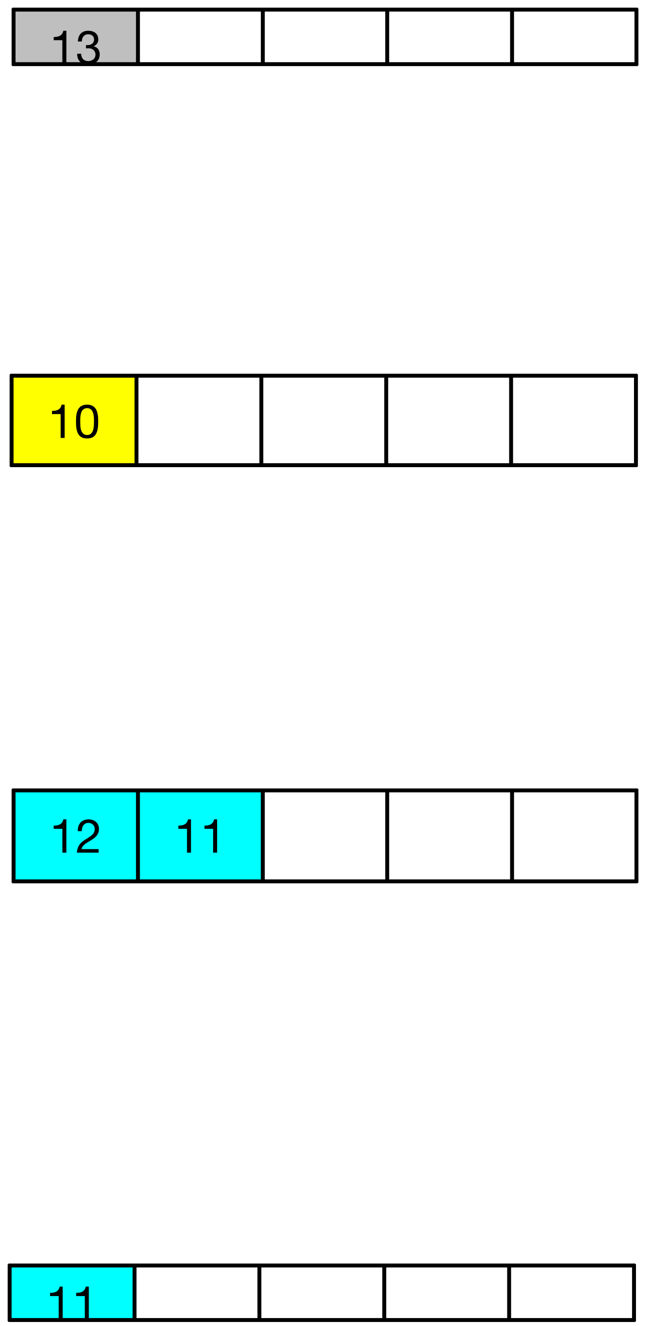
```
Pqueue q ← init with root
closest_hit = ∞
while q not empty:
  node ← extract front from q          // = nearest BV
  if dist(node) <= closest_hit:        // else: skip this subtree
    if node is leaf:
      intersect ray with all polygons in node
      update closest_hit, if any polygon is closer
    else                               // inner node
      forall children of node:
        if ray intersects child:
          insert child in q with its distance
```

Example

- Insert root
- 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
- Pop front ⟶ 12
  - Test with 4 → yes, save hit
  - Test with 5 → yes, closer ⟶ save hit

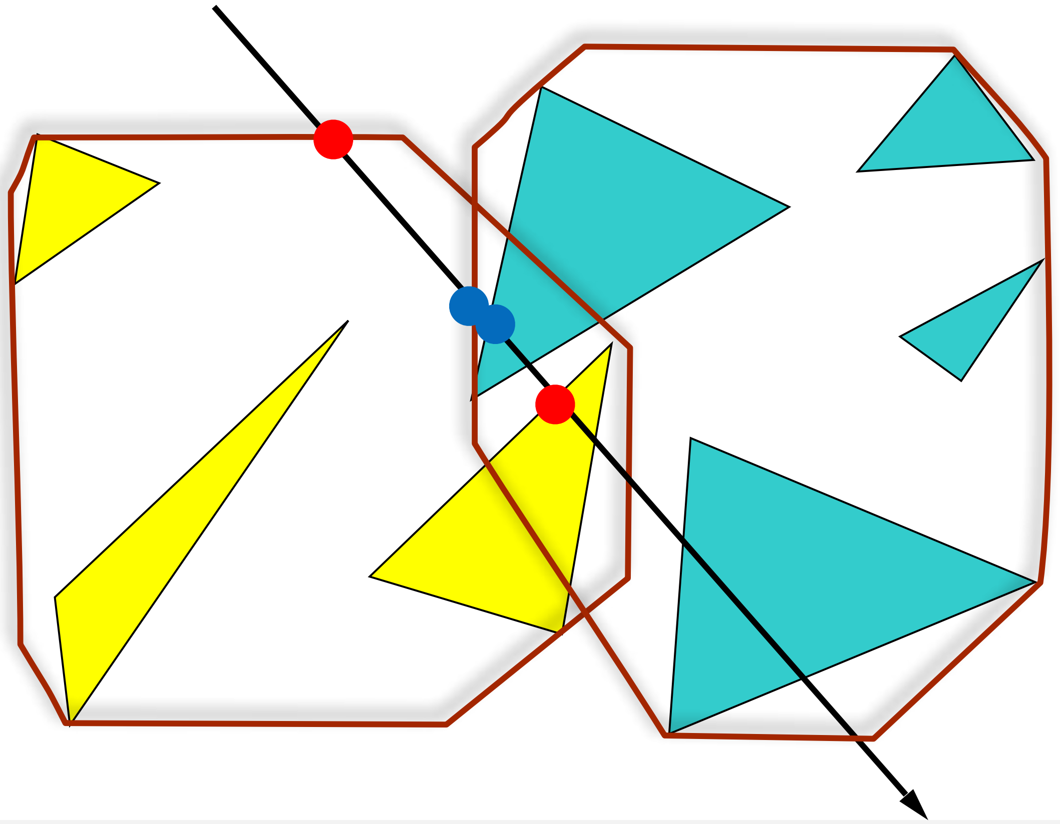






Remarks

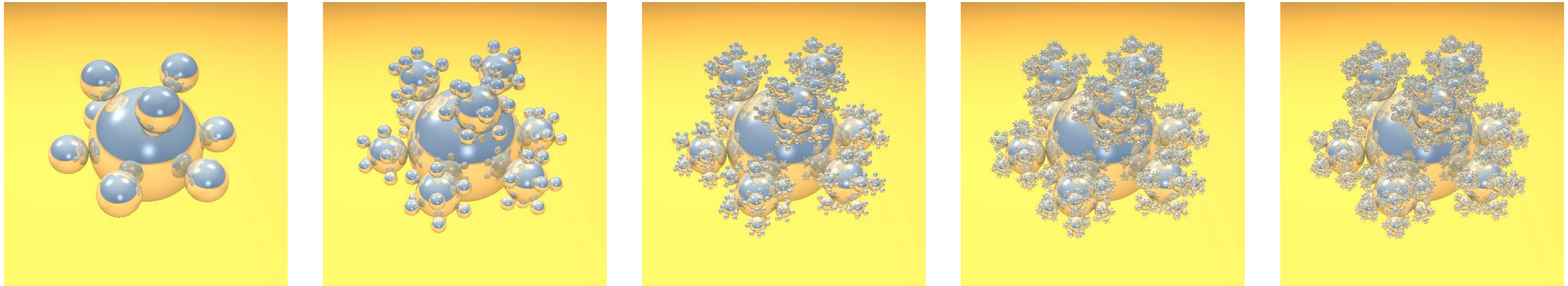
- Observation: we don't need a complete ordering among the BV's in the priority queue, because in each step, we only need to extract the BV that has the *closest* intersection (among all others in the queue)
- Efficient implementation of a p-queue: heap
- Insertion of an element, and extracting the front $\longrightarrow$ O(log k) (where k = #elements in the p-queue)
- **Warning:** the closest ray-BV intersection and the closest ray-primitive intersection can occur in different BV's!



Complexity of BVH Traversal Along a Ray

- Assumptions (rather strong):
  - On each level in the BVH, all pairs of BV's are intersection-free
  - During construction, the polygon list is always partitioned at the median
- One BVH traversal for a single ray query: $O(\log n)$
- More precisely:

Performance Gain?

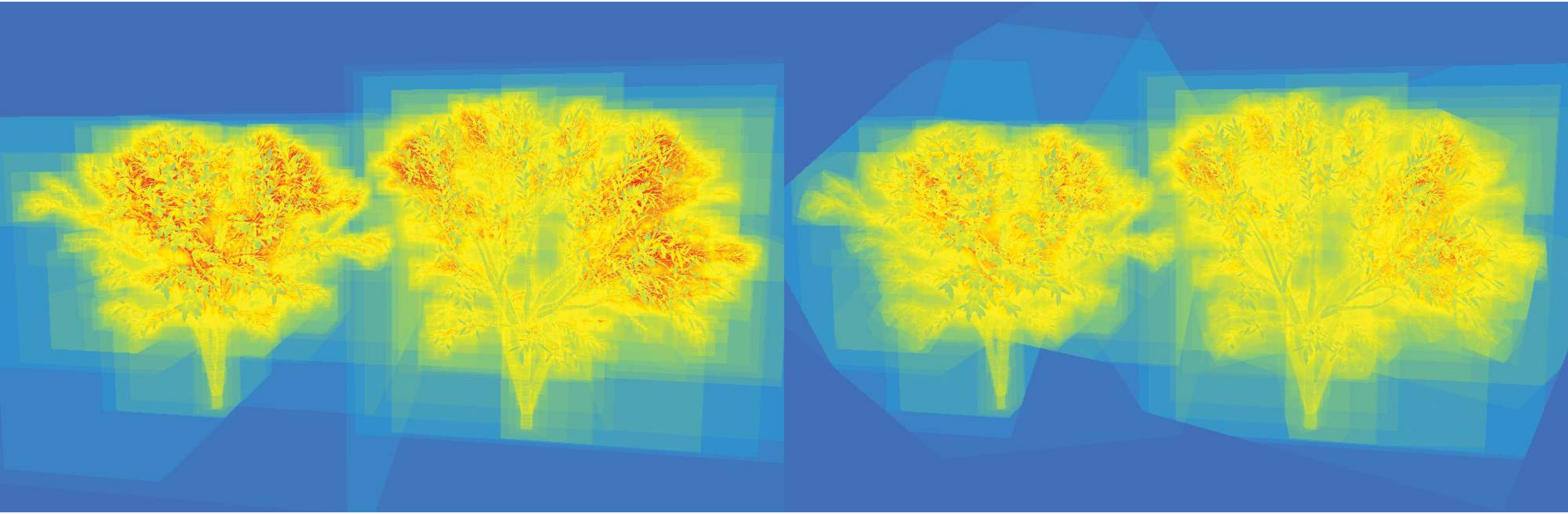


# spheres	10	91	820	7381	66430
Brute-force	2.5	11.4	115.0	2677.0	24891.0
With BVH	2.3	2.8	4.1	5.5	7.4

Rendering times in seconds, Athlon XP 1900+ [Markus Geimer]

Performance Comparison AABB vs OBB Hierarchy for Raytracing

Number of BV intersections during BVH traversal for the primary ray



AABB hierarchy

OBB hierarchy

Note that a single BV-ray intersection calculation is more expensive for OBB's

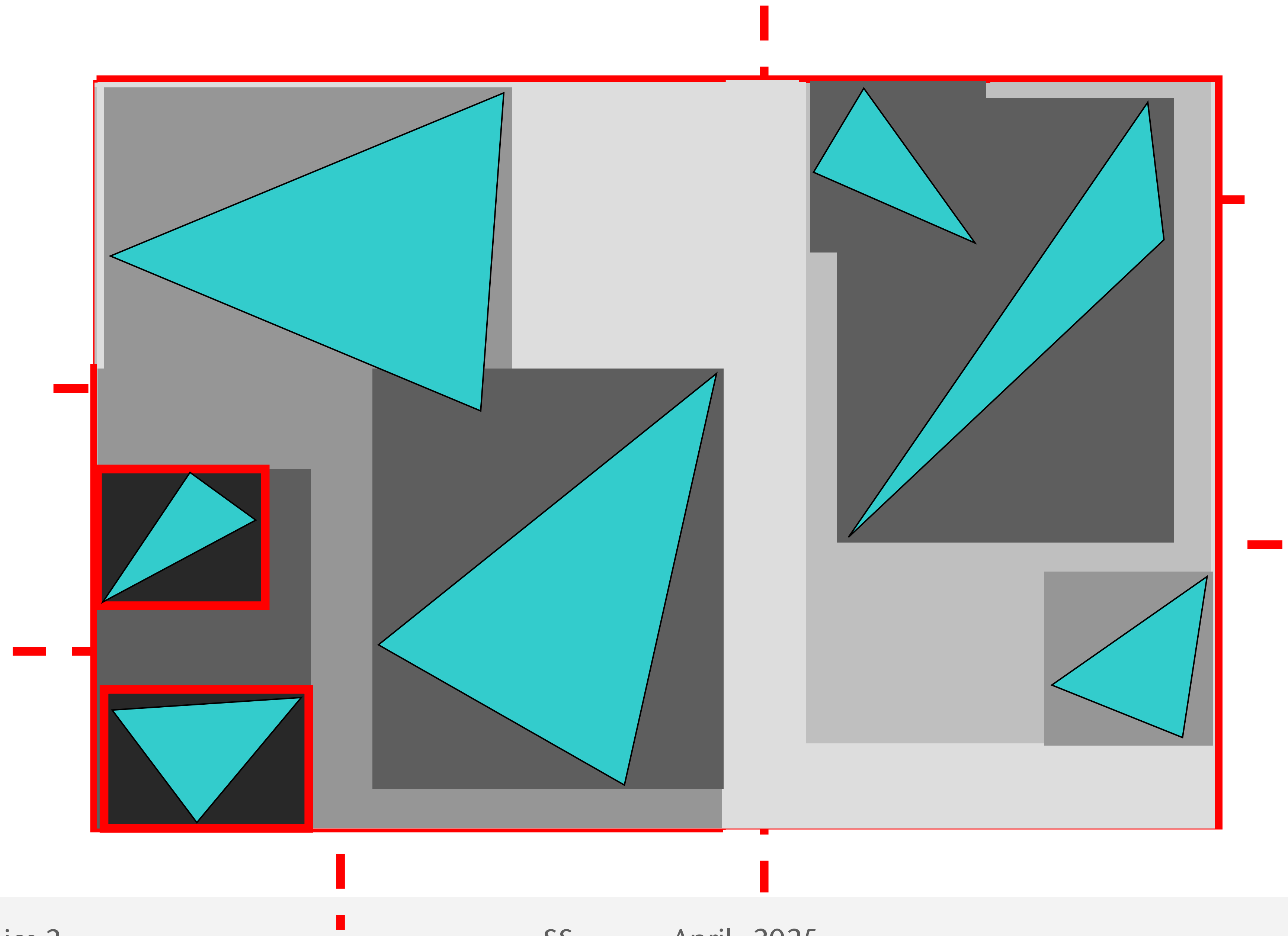
The Construction of BV Hierarchies

- There are many possible principles:

1. Given by modeling process (e.g., in form of a 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. Iterative Insert:
   - Start with empty tree, iteratively add polygons, let each polygon "sift" through the tree [Goldsmith/Salmon]
4. **Top-down**:
   - Partition the set of primitives recursively
   - Problem: how to partition the set?

Example for the Top-Down Construction of a BVH

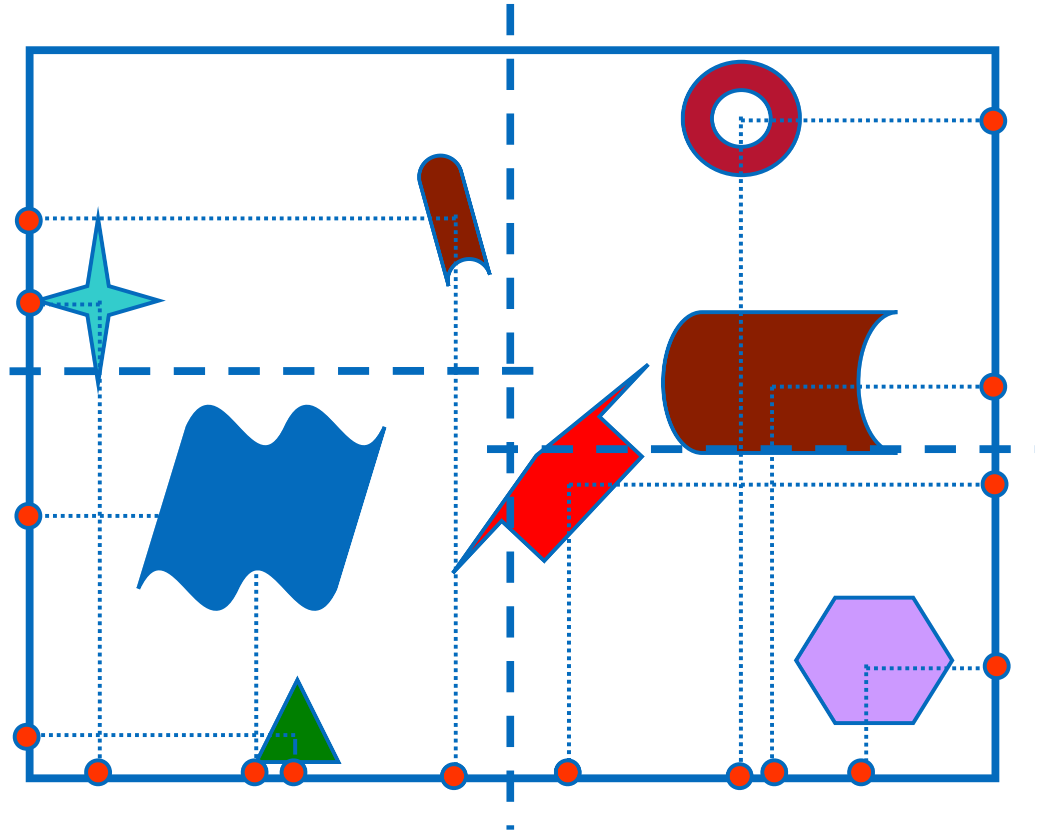
- Enclose each object (= primitives) by an elementary BV (e.g., AABB)
- In the following, work only with those elementary BVs
- Partition the set of objects in two sub-sets
- Recurse

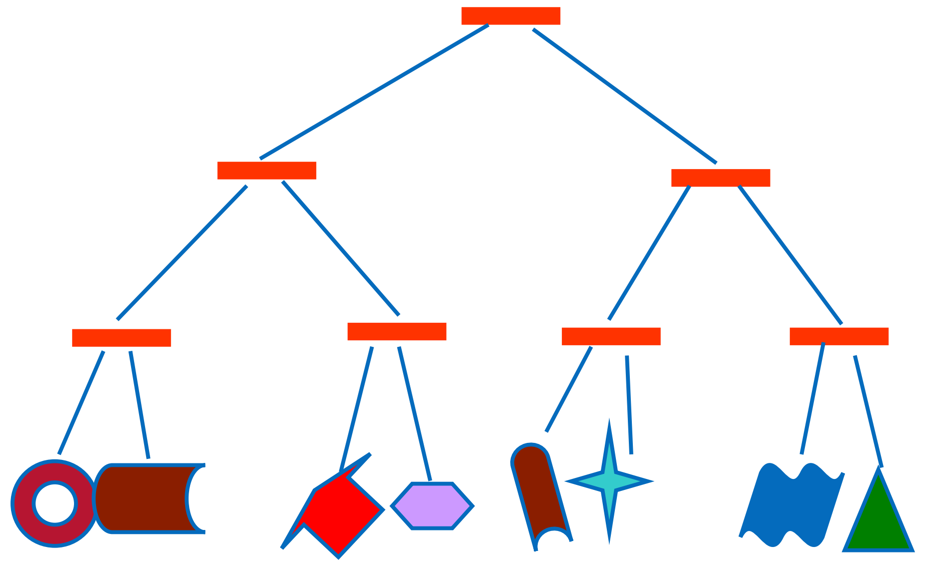


Optional

Simplest Heuristic for Partitioning: Median Cut

1. Construct elementary BVs 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





A Better BVH Construction Method

- Given a set of polygons, what is their optimal partitioning? (optimal with respect to raytracing performance)

- Use the Surface Area Heuristic (SAH):
partition polygon set B into subsets B_1 and B_2 such that

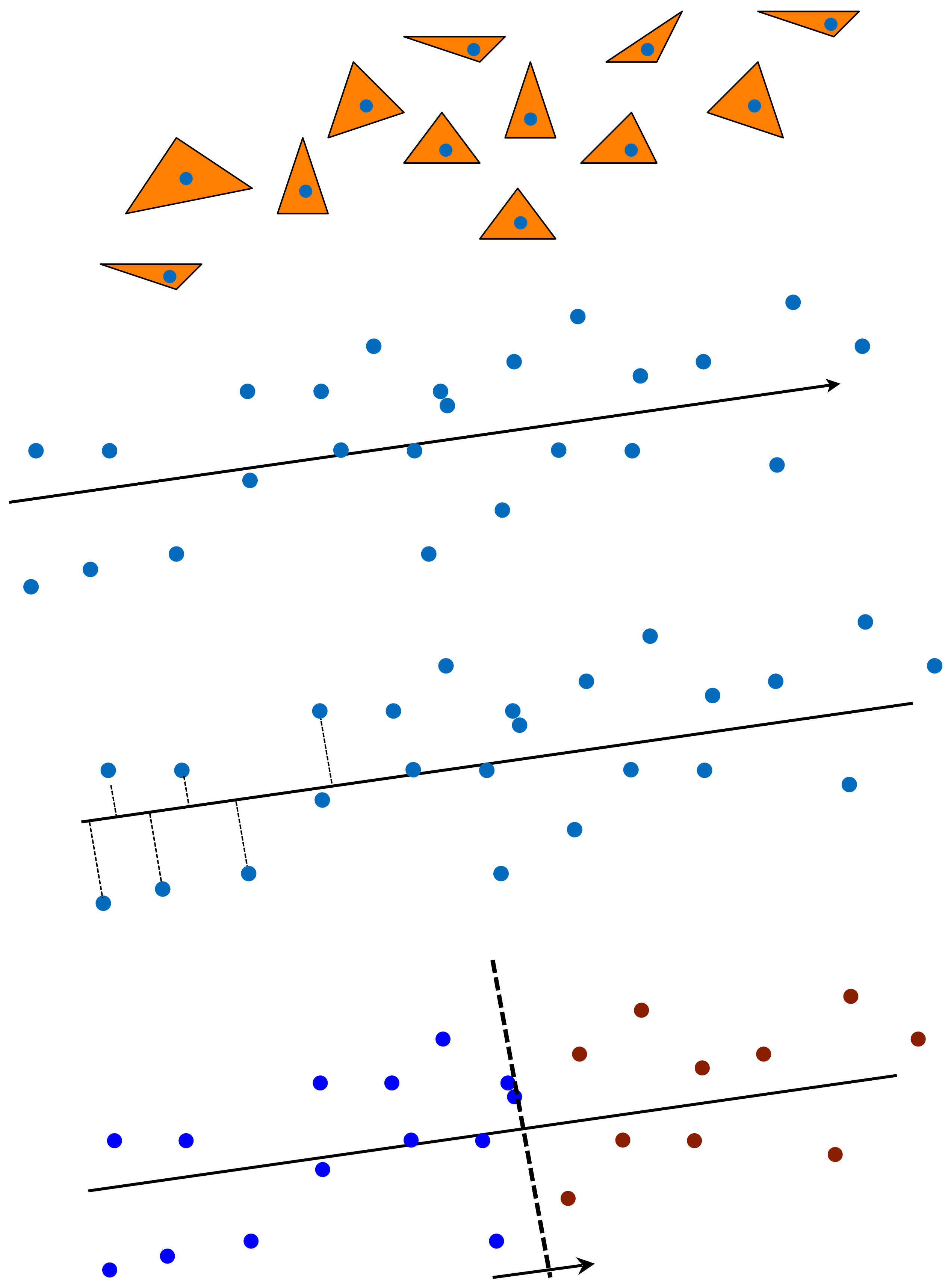
$$C(B) = \text{Area}(B_1) \cdot N(B_1) + \text{Area}(B_2) \cdot N(B_2)$$

attains its minimum

- Optimum could be achieved by exhaustive search: consider all possible subsets $B_1 \in \mathcal{P}(B)$ and $B_2 = B \setminus B_1$
  - Not practical
- Current "best" way: use a method similar to kd-tree construction

The Plane Sweep Method to Construct Good BVHs

1. Represent all polygons by their midpoints
2. Calculate axis of largest extent (using PCA)
3. Project all midpoints onto that axis and sort
4. Search minimum of $C(B)$ by plane sweep



- Running time:

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

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

where α is the proportion of polygons that end up in the "left" child BV, and assuming α is bounded (e.g., between 0.1 and 0.9)

- Remarks:
  - Stopping criteria are the same as for the kd-tree
  - Top-down methods usually lead to better BVHs than iterative ones