Computer Graphics
- Ray-Tracing III -

Hendrik Lensch

Overview

- **Last lecture**
  - spatial acceleration structures
  - Grids, Octrees, BVH, KD-Tree
  - static scenes only

- **Today**
  - updates for dynamic scenes

Ray Tracing Dynamic Scenes

- **Problem: Changing geometry requires updating indices**
  - While maintaining interactivity
- **Very little research prior to 2006**
  - Efficient dynamic data structures: so far not in realtime
    - From computational geometry (i.e. kinetic data structures)
  - Animation with predefined motion [Glassner'88, Gröller'91, …]
  - Exclude dynamic primitives [Parker'99]
  - Constant time rebuild [Reinhard'00]
  - Divide and conquer [Lext'00, Wald02]
- **Last two years:**
  - Motion Compensation [Guenther06, Guenther06]
  - Updated BVH [Wächter06, Lauterbach06, Woop06, Wald06]
  - Fast rebuild [Wald06, Popov06, Ize06, Wald06, Havran06, Hunt06]
  - Partial rebuild [Yoon07]

Ray Tracing Dynamic Scenes

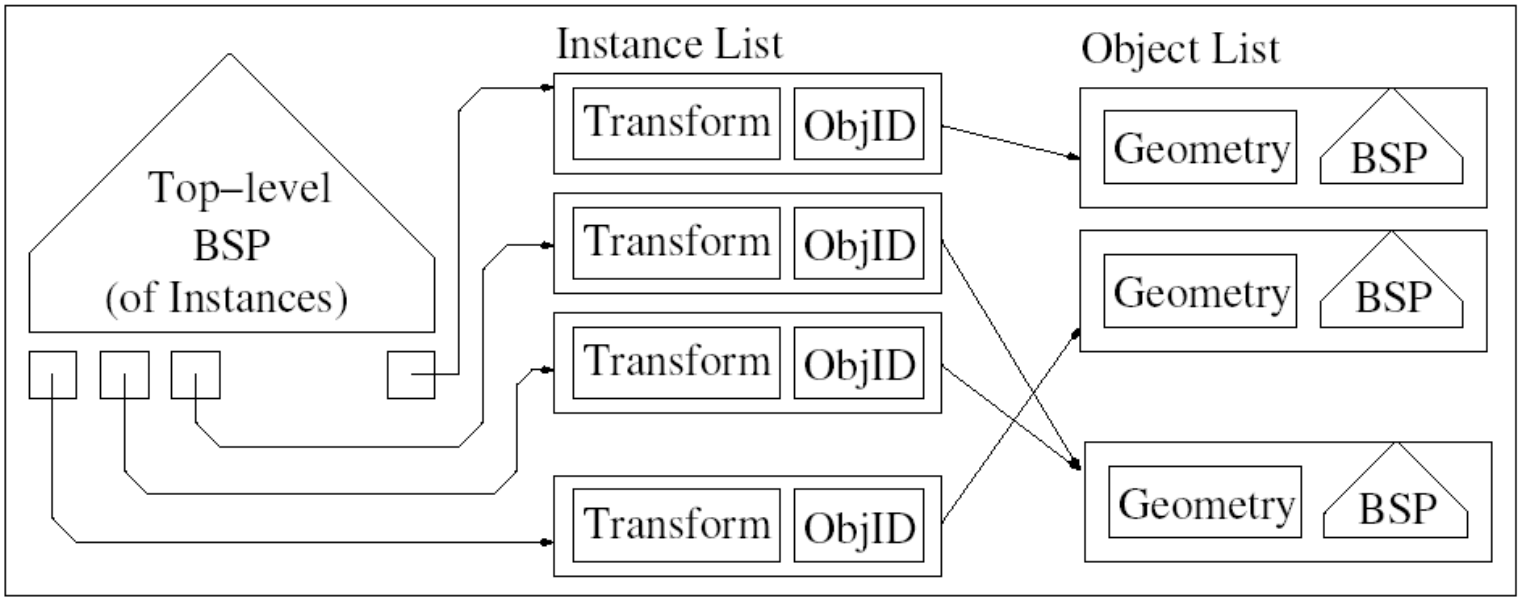
- **Different Types of Motion**
  - Static: No changes
  - Structured: Affine transformations for groups of primitives
  - Continuous: Adjacent geometry stays adjacent during animation
  - Unstructured: Arbitrary or random movements of primitives
- **General Framework**
  - What does the application know about the motion?
  - How do we communicate that to the renderer? (➔ API)
  - How to efficiently and effectively use this information?
- **General Approaches**
  - Partition scene depending on type of motion
  - Build index valid for longer time (fuzzy indicies)
  - Rebuild entire index each frame
  - Lazy building of tree (on-the-fly only where needed)
  - Update index each frame

Partitioning: Divide & Conquer

- **Observation**
  - Very often a simple approach is sufficient
  - Building hierarchical index structures requires O(n logn)
    - Divide and conquer reduces complexity
- **Categorize primitives into independent groups/objects**
  - Static parts of a scene (often large parts of a scene)
  - Structured motion (affine transformations for groups of primitives)
  - Anything else
- **Select suitable approach for each group**
  - Do nothing
  - Transform rays instead of primitives
  - Only update index structure for remaining part of the scene

Divide & Conquer Approach

- **Two-level index structure**
  - Find relevant object along the ray
  - Transform ray (efficient SSE code)
  - Find primitives within object
  - Same kd-tree traversal algorithms in both cases
- **Results in some run-time overhead: k times O(log n)**



Implementation

- **KD-tree building algorithms**
  - Static & structured motion
    - Build once with sophisticated and slow algorithm [Havran'01]
    - Optimize for traversal (as low as 1.5 intersection per ray)
  - Unstructured Motion
    - Will be used for single or few frames
    - Balance construction and traversal time
      - E.g. allow more primitives in leaf nodes
  - Top-Level tree over objects:
    - Significantly more efficient than for primitives
    - Possible splitting planes for kd-tree are already given (Bbox)

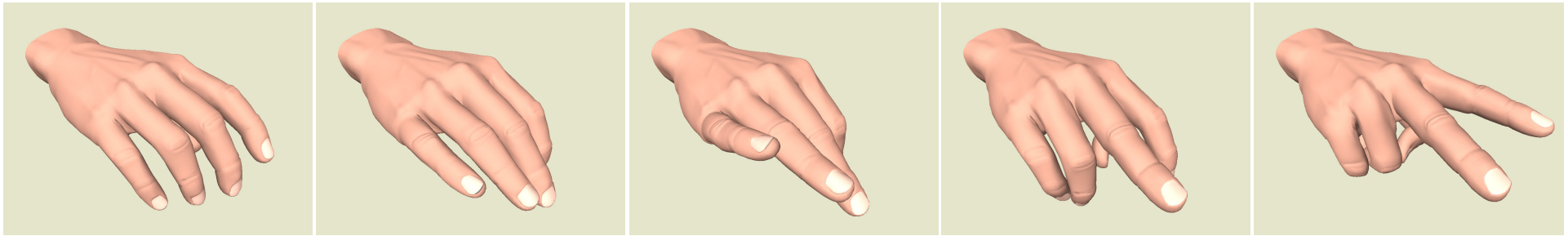
Implementation

- **Index Structure Updates**
  - Static: Done
  - Structured Motion
    - Update transformation
    - Update of top-level index with transformed bounding boxes
  - Unstructured Motion
    - Rebuild local index
    - Schedule top-level update, iff bounding box changed

Updating the Index Structure

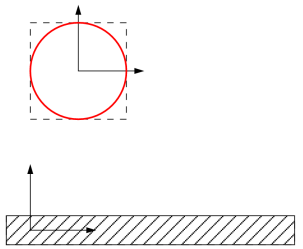
- **IDEA**
  - „Make dynamic scenes static”

- **Assumptions:**
  - Deformation of a base mesh (constant connectivity)
  - All frames of animation known in advance
  - Continuous motion

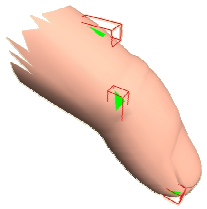


Method Overview

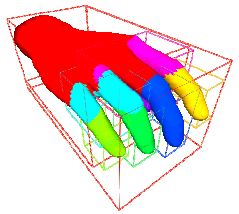
- **Motion decomposition**
  - Affine transformations
  - \+ residual motion



- **Fuzzy kd-tree**
  - Handles residual motion

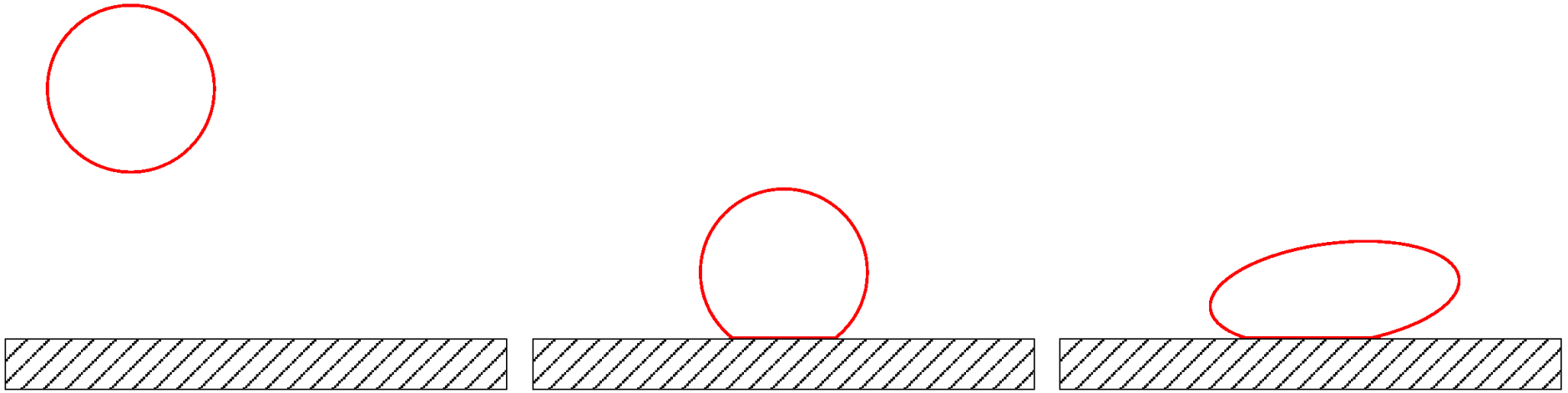


- **Clustering**
  - Exploit local coherent motion



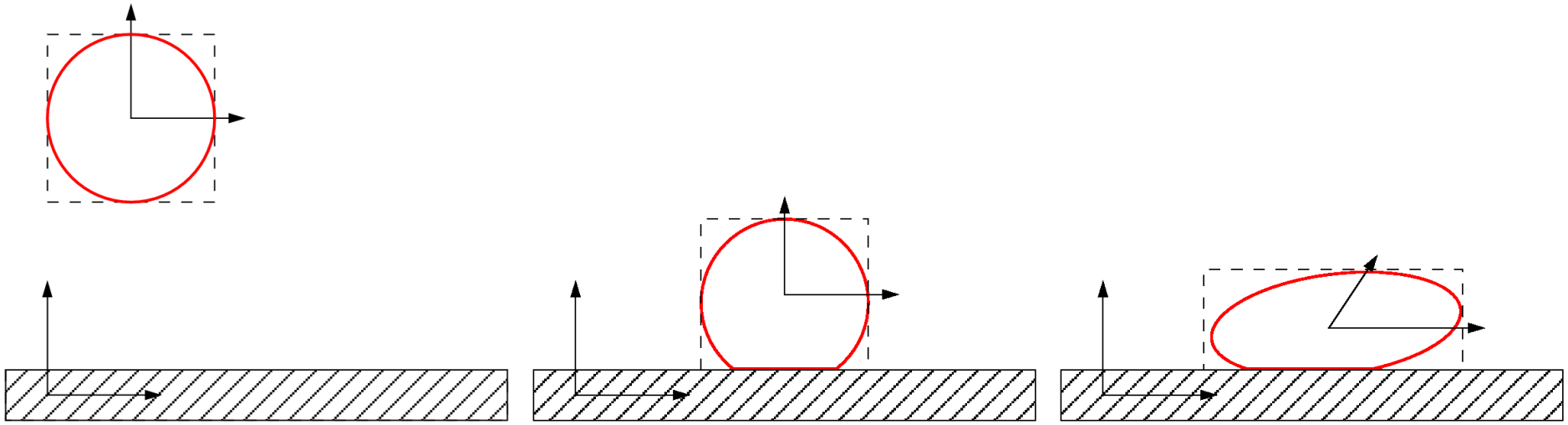
Motion Decomposition

- **Dynamic scene: ball thrown onto floor**

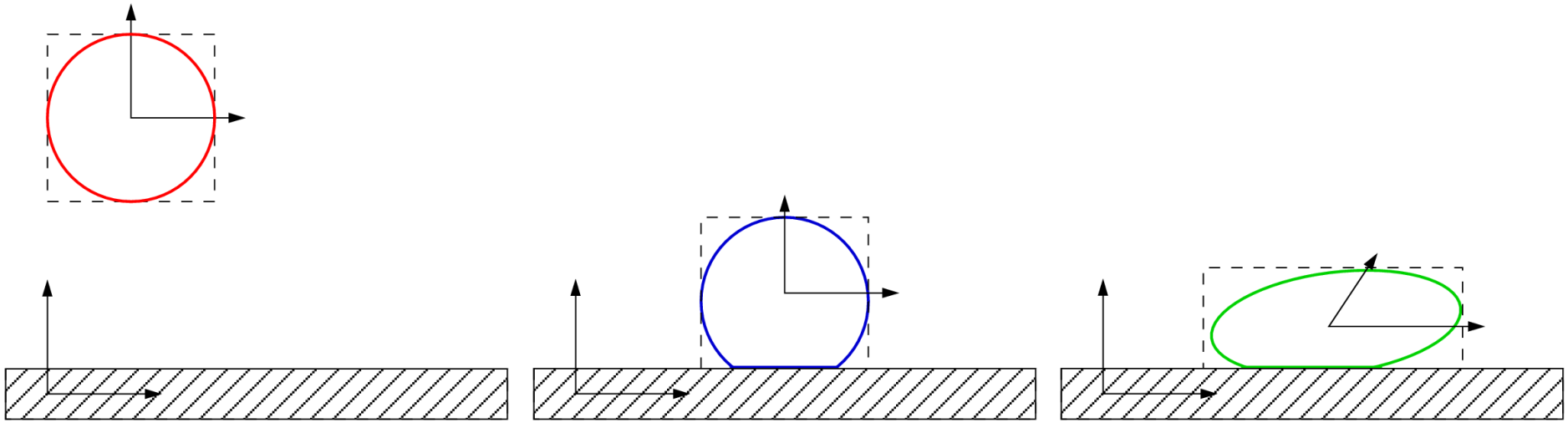


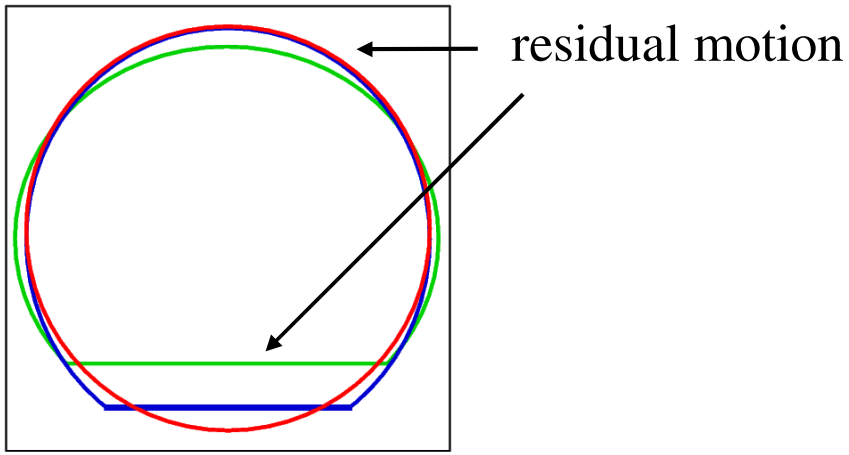
Motion Decomposition

- **Affine transformations**
  - Approximate deformations
  - Include shearing (3rd frame)

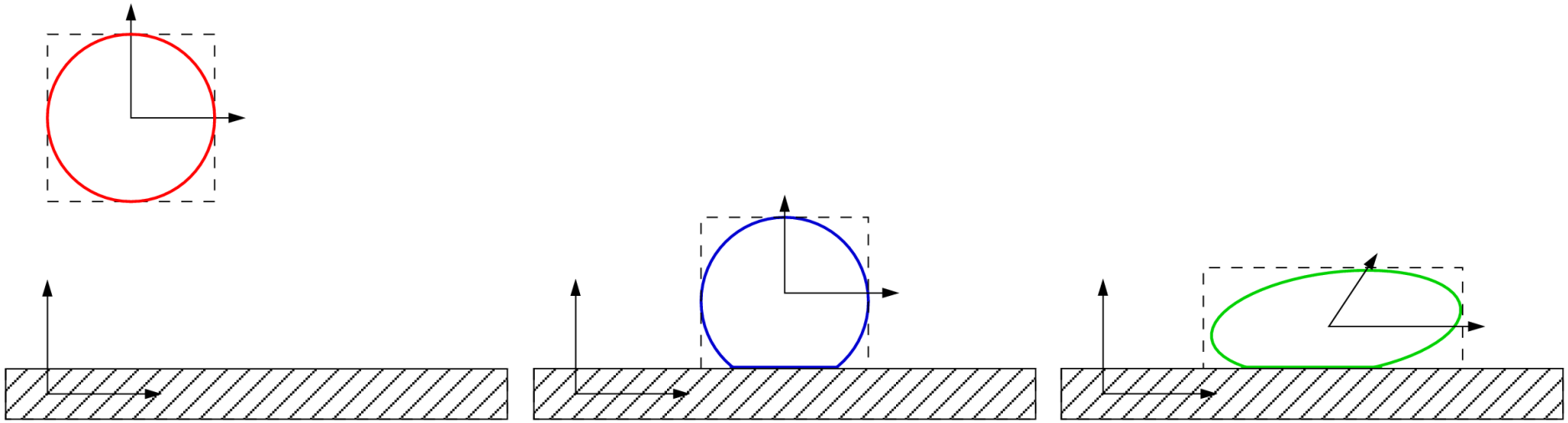


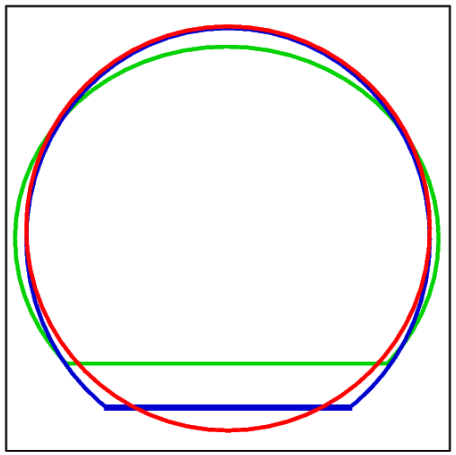
Motion Decomposition

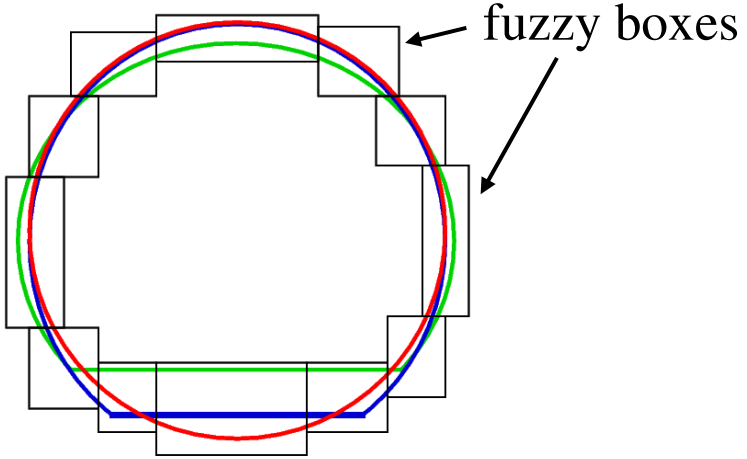




Motion Decomposition

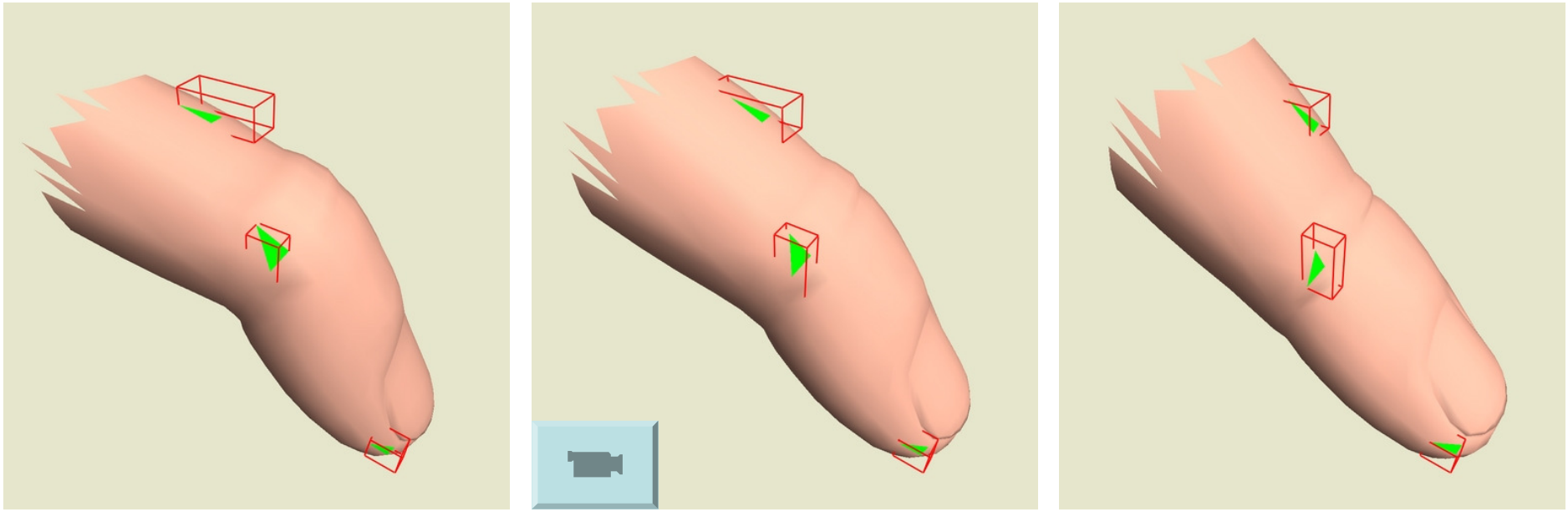




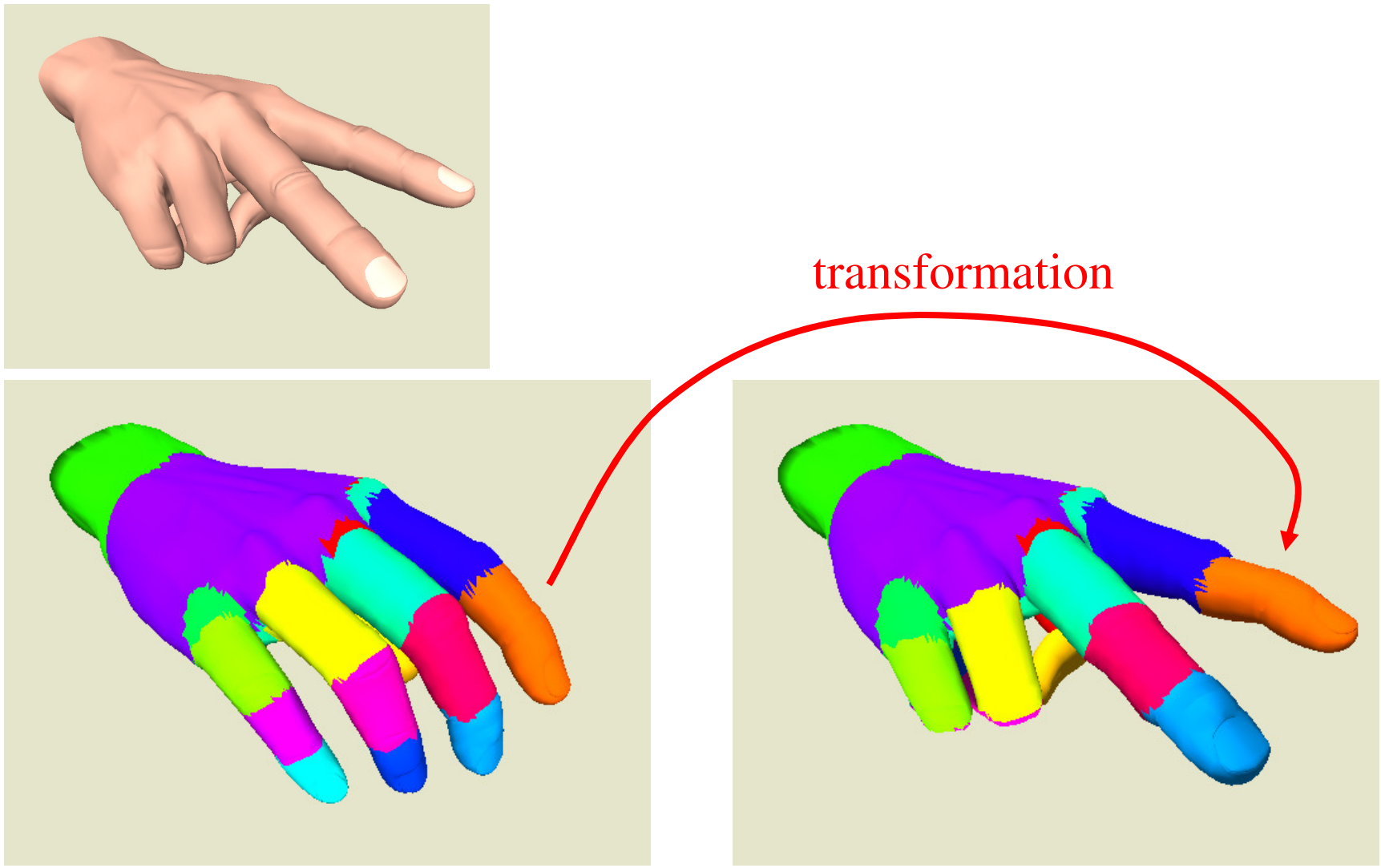


Fuzzy KD-Tree

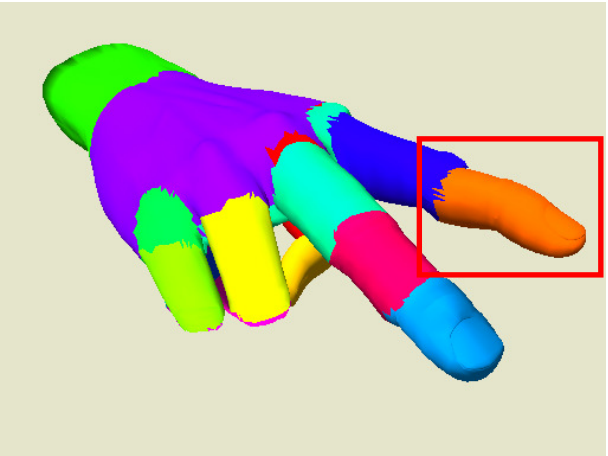
- **Handles residual motion**
- **KD-Tree over the fuzzy boxes of triangles**
  - Valid over complete animation

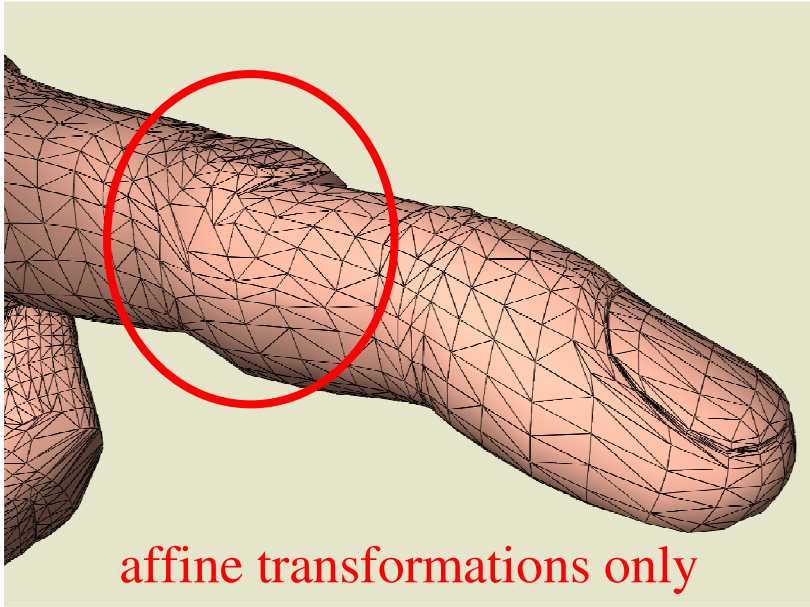


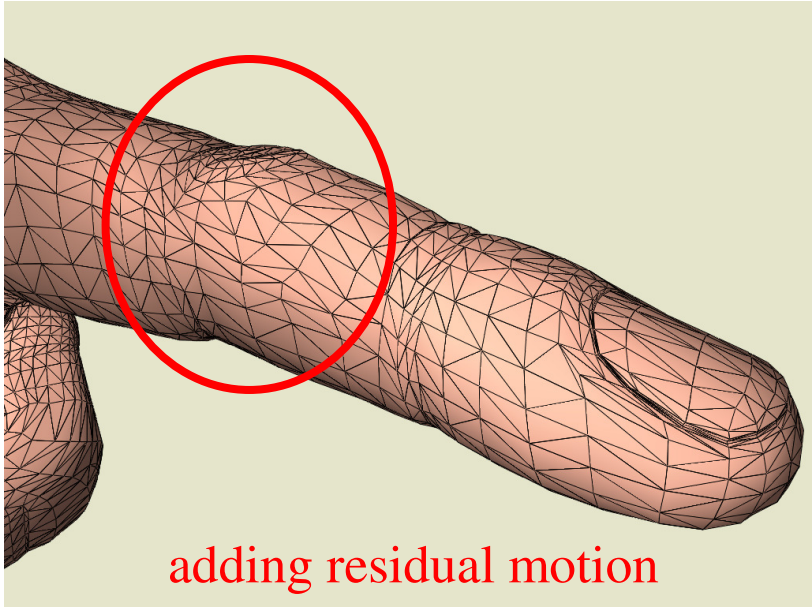
Illustration



Illustration





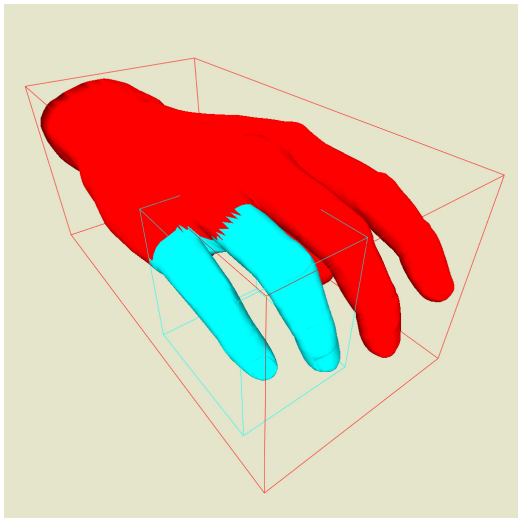


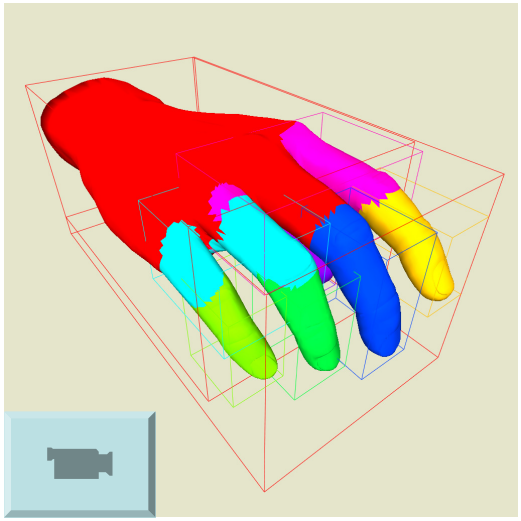
Details: Clustering

- **Efficient ray tracing:**
  - Requires small fuzzy boxes
  - Must minimize residual motion
  - Should cluster coherently moving triangles
    - Results in few object
- **Many clustering algorithms**
  - But mostly for static meshes
  - Not designed for ray tracing
- **Develop new one**
  - Based on Lloyd relaxation

Clustering Algorithm

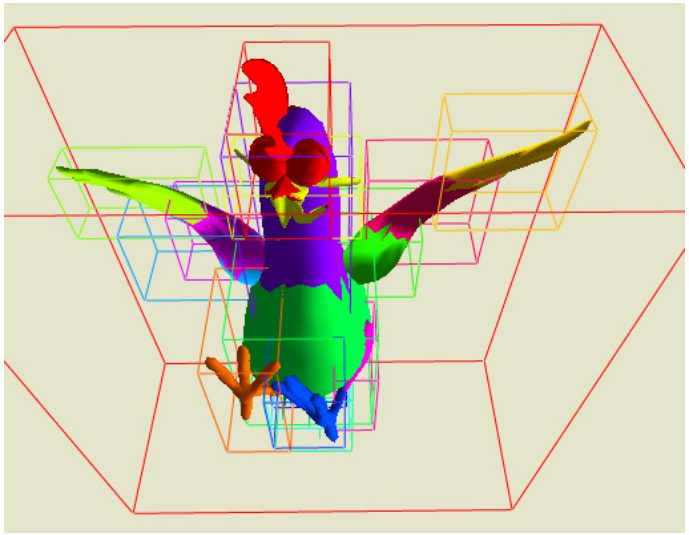
- **Start with one cluster (all triangles)**
- **Lloyd relaxation:**
  - Find transformations for clusters
    - Linear least squares problem
  - Recluster triangles
    - By choosing transformation with minimal error
  - Until convergence
    - No triangles move between clusters
- **Insert new cluster**
  - Seeded by triangle with highest residual motion
- **Until improvement below threshold**
  - Measured by summing all error terms

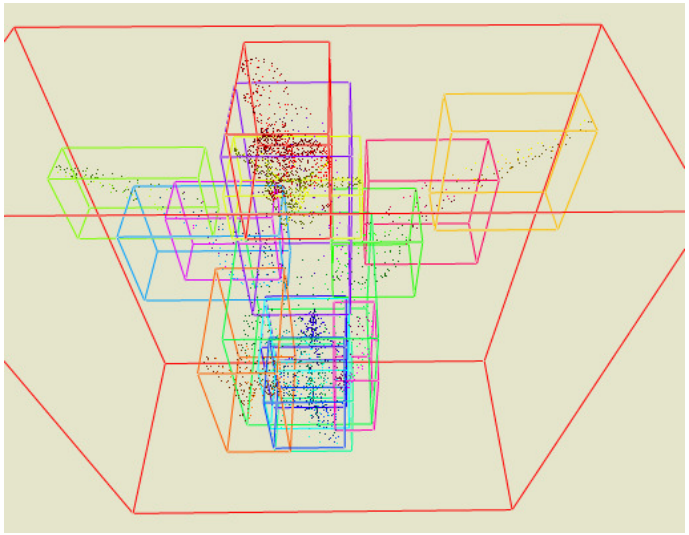




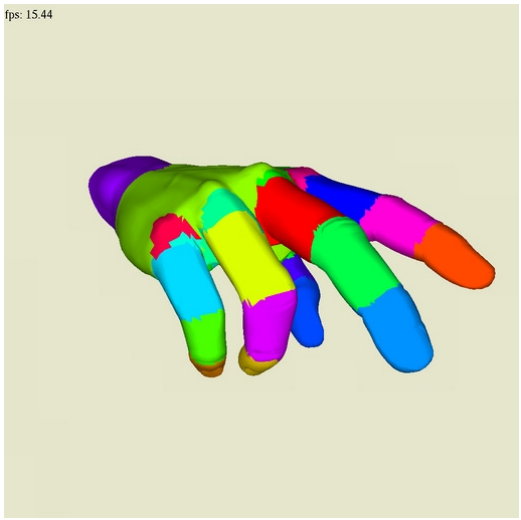
Ray Tracing: Two-level Approach

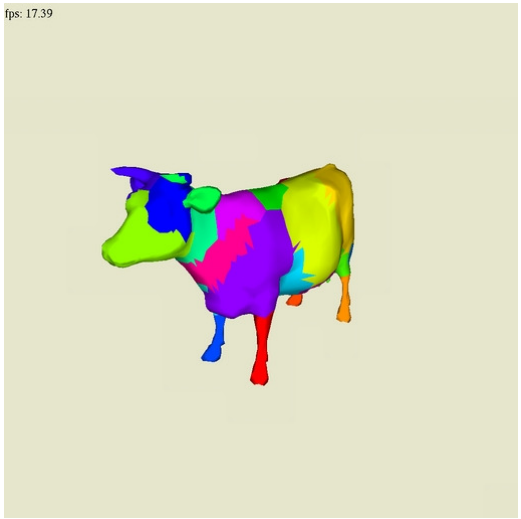
- **Build top-level kd-tree over current cluster bounds**
- **Transform rays into local coordinate system**
  - Inverse affine transformation of cluster from motion decomposition
- **Traverse fuzzy kd-tree of cluster**





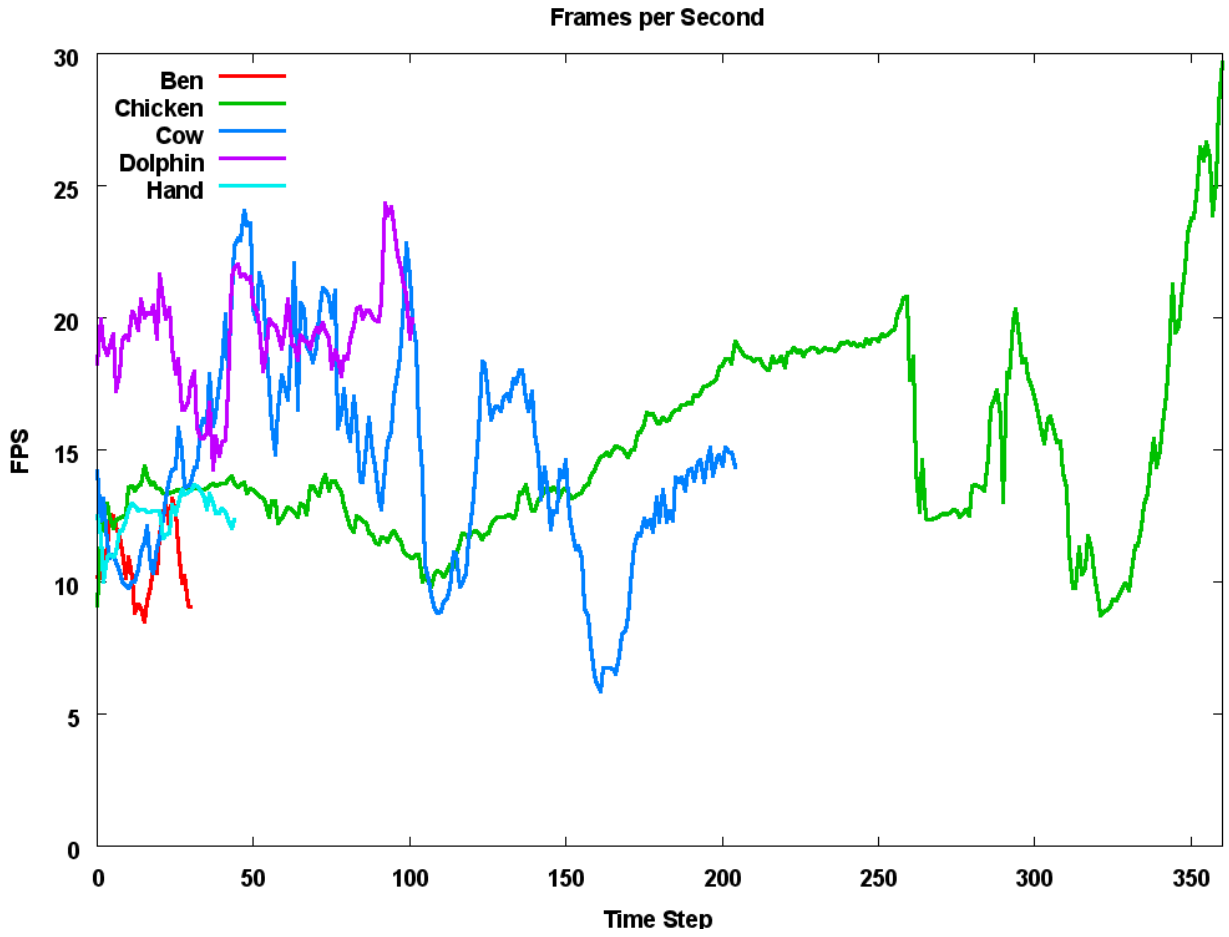
Video





Ray Tracing Performance

- **Single CPU**
  - (Opteron 2.8 GHz)
- **1024×1024 px**
- **Incl. shading**



Comparison to Static KD-Tree

- **Baseline: separate static kd-tree per frame**

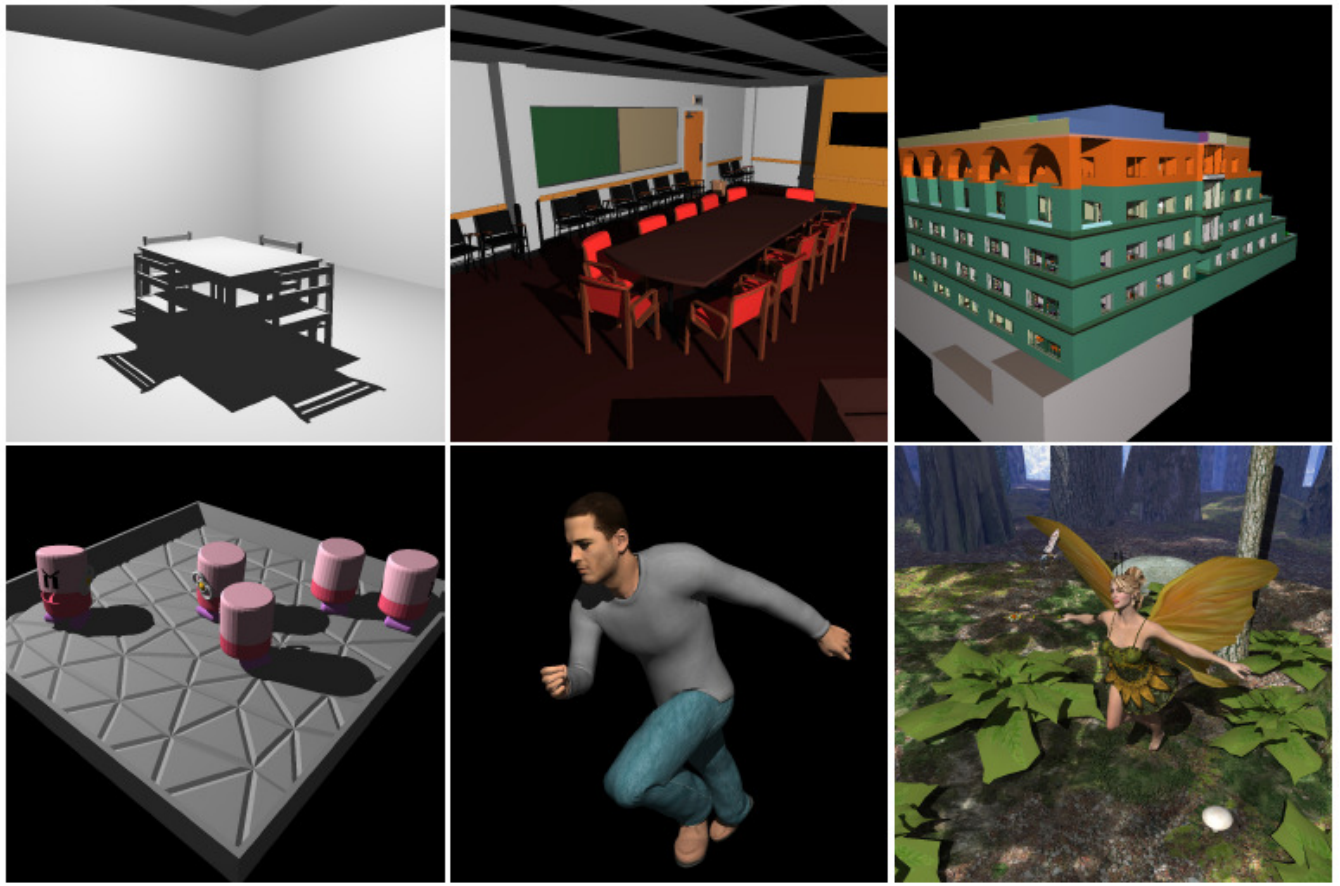
- **Traversal steps**
  - Factor 1.5 - 2
- **Intersections**
  - Factor 1.2 – 2, Cow 4, Chicken 6
- **Average fps**
  - Factor 1.2 – 2.6, alone two-level kd-tree costs ca. 30%
- **Memory**
  - only one fuzzy kd-tree (+ transformation matrices)
  - vs. #frames static kd-trees

Updating Spatial Index Structures

Bounding Volume Hierarchies

- **Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies [Wald, TOG06/07]**
- **Build binary BVH hierarchy**
  - Using variation of SAH algorithm
- **Fast packet traversal**
  - Test first ray against box of child node
    - If hit, immediately traverse child node
  - Test frustum of rays against box of child node
    - If miss, do not traverse child node
  - Otherwise, test all ray until hit is found
    - Do not traverse in case of no hit
  - Optimization, store order of child nodes for ever axis
    - Allows for more effective early ray termination
  - Can use fast SIMD computation for packets of rays

Test Scenes



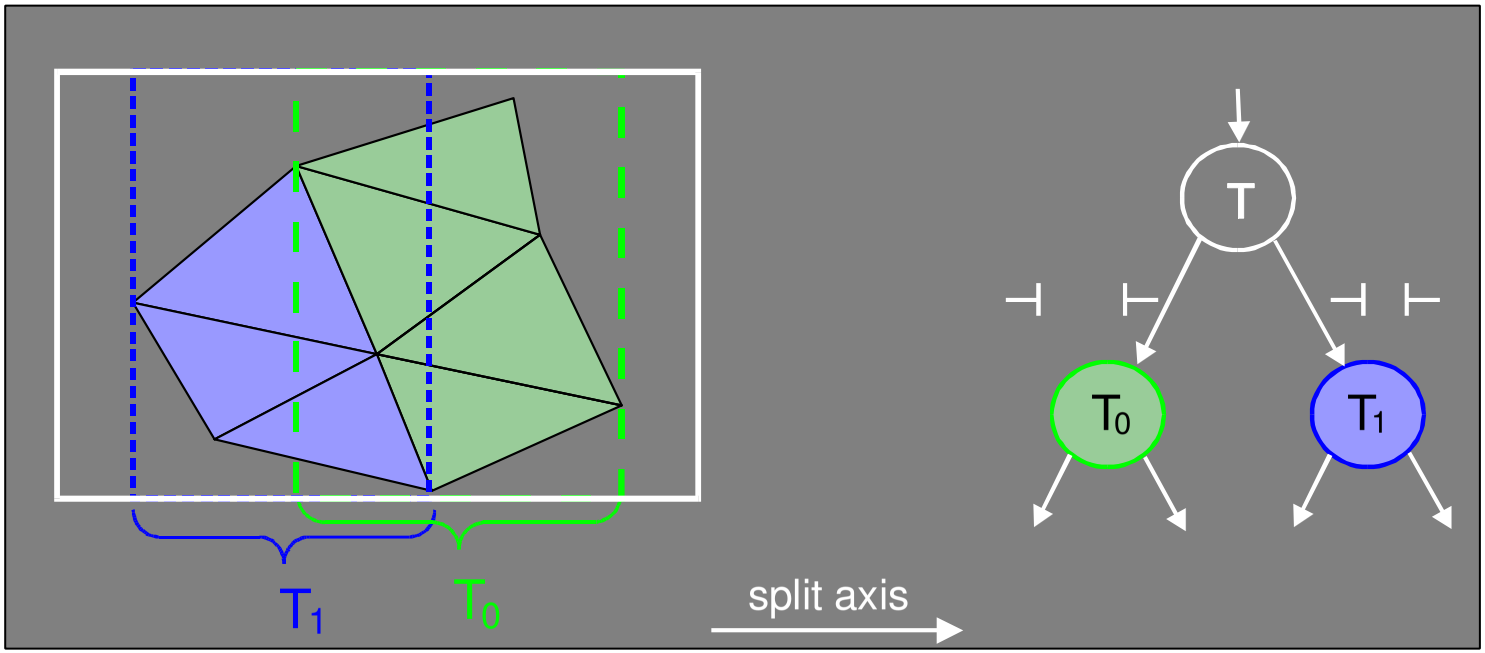
BHV Updates

- **Choose a Good Initial Pose**
  - Avoids accidental closeness of triangles that separate later
  - Usually not a big issue
- **Good simple approach**
  - Test various poses from (known) animation
- **Build BHV over entire animation**
  - Build BHV hierarchy with respect to best partitioning over all animation frames
  - Usually not much improvement
    - Shows that hierarchies stay good during animations

Bounding KD-Trees:
Mixing Bounding Volumes and KD-trees

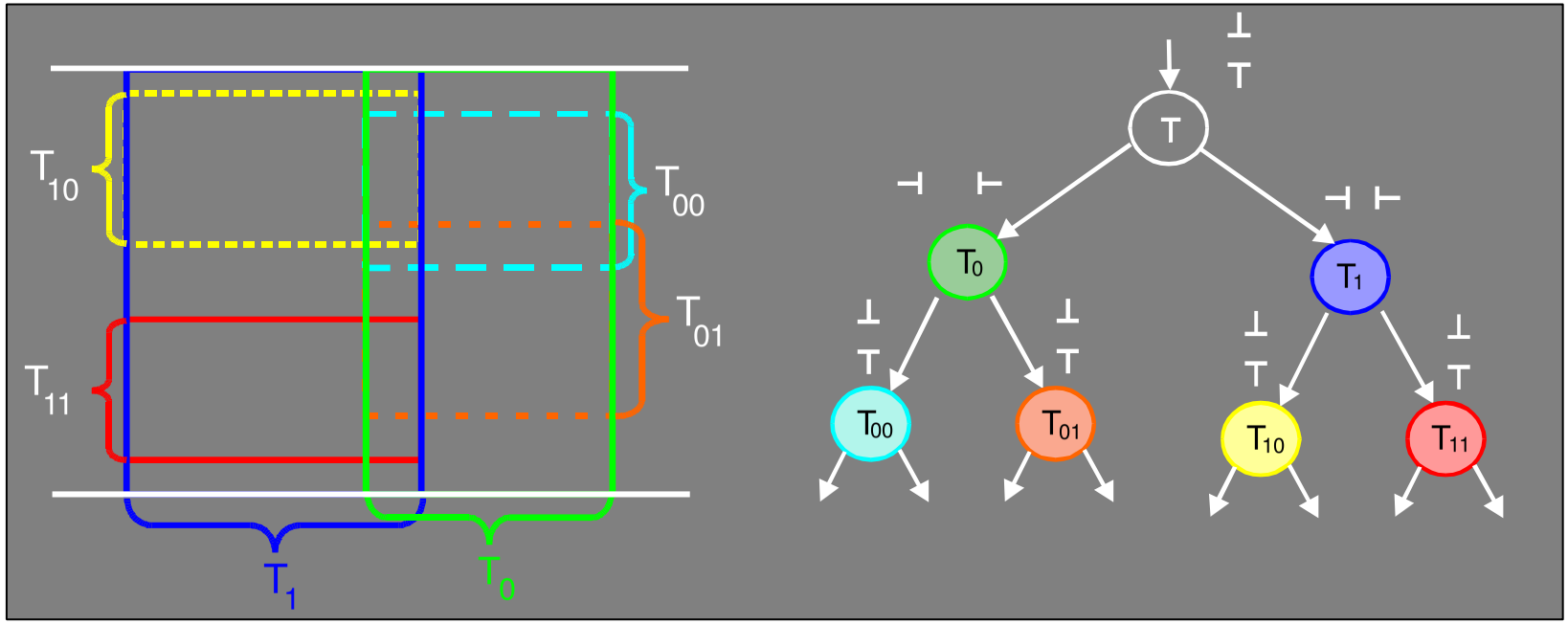
Definition of B-KD Trees

- **B-KD Tree (Bounded KD-Tree)**
  - Binary Tree
  - 1D bounding intervals for each child
  - Leaf nodes point to a single primitive



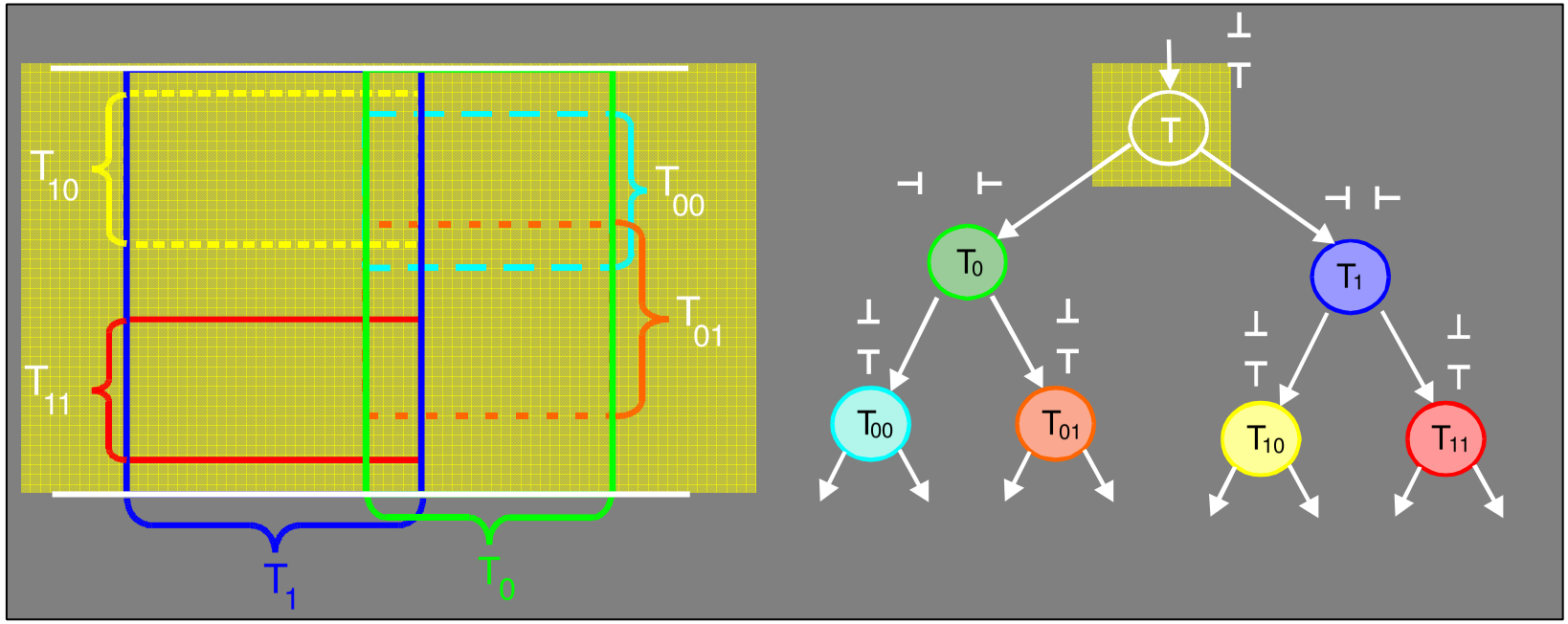
B-KD Tree Subdivision

- **Bounding Volume Hierarchy (partially unbounded)**
- **Each node can be associated with a full bounding box**
- **Bounds may overlap**
  - Primitives in single leaf nodes
  - More traversal steps as for KD Tree
  - Support for dynamic scenes



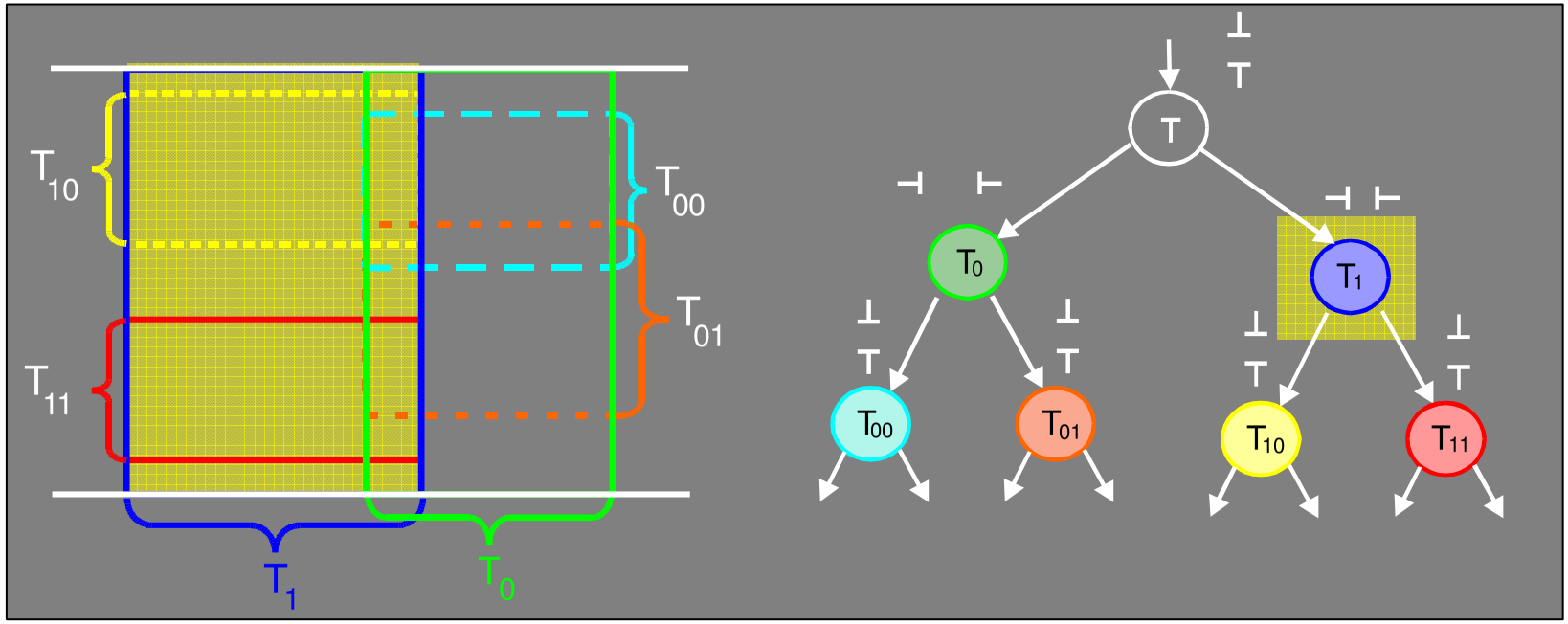
B-KD Tree Subdivision

- **Bounding Volume Hierarchy (partially unbounded)**
- **Each node can be associated with a full bounding box**
- **Bounds may overlap**
  - Primitives in single leaf nodes
  - More traversal steps as for KD Tree
  - Support for dynamic scenes



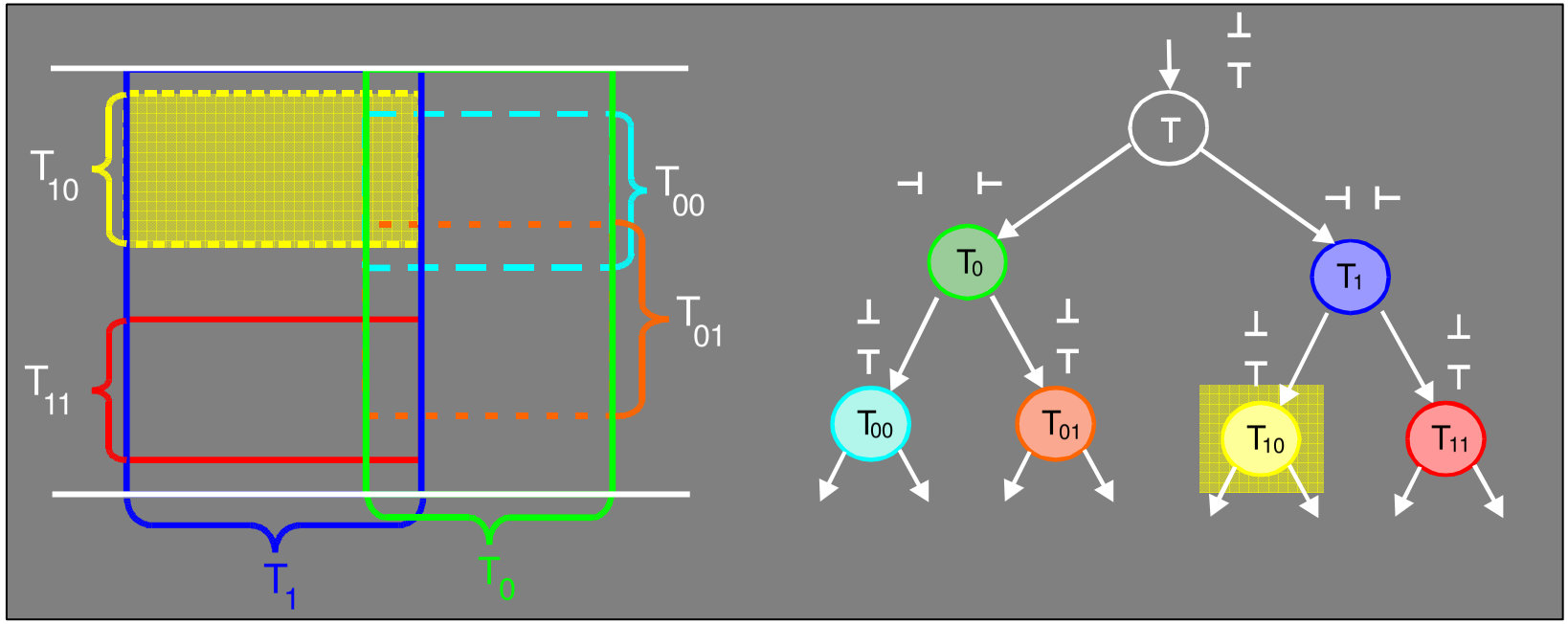
B-KD Tree Subdivision

- **Bounding Volume Hierarchy (partially unbounded)**
- **Each node can be associated with a full bounding box**
- **Bounds may overlap**
  - Primitives in single leaf nodes
  - More traversal steps as for KD Tree
  - Support for dynamic scenes



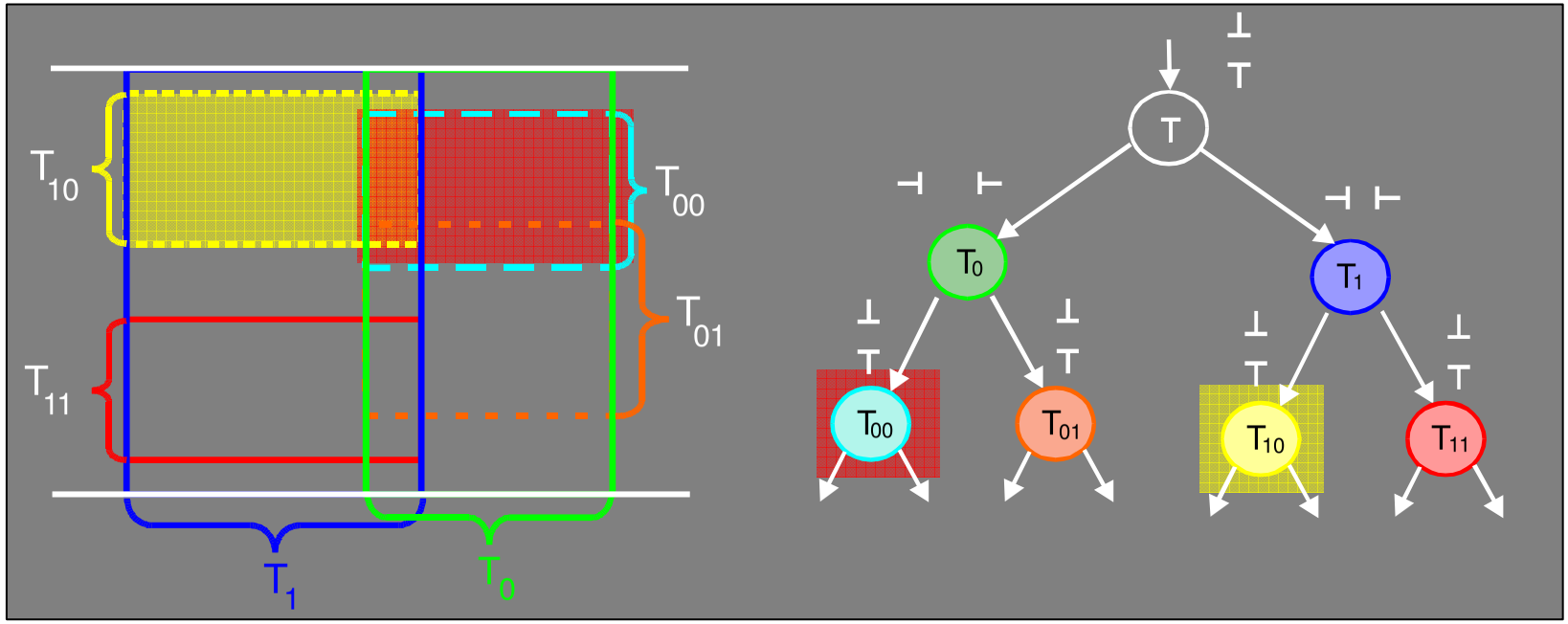
B-KD Tree Subdivision

- **Bounding Volume Hierarchy (partially unbounded)**
- **Each node can be associated with a full bounding box**
- **Bounds may overlap**
  - Primitives in single leaf nodes
  - More traversal steps as for KD Tree
  - Support for dynamic scenes



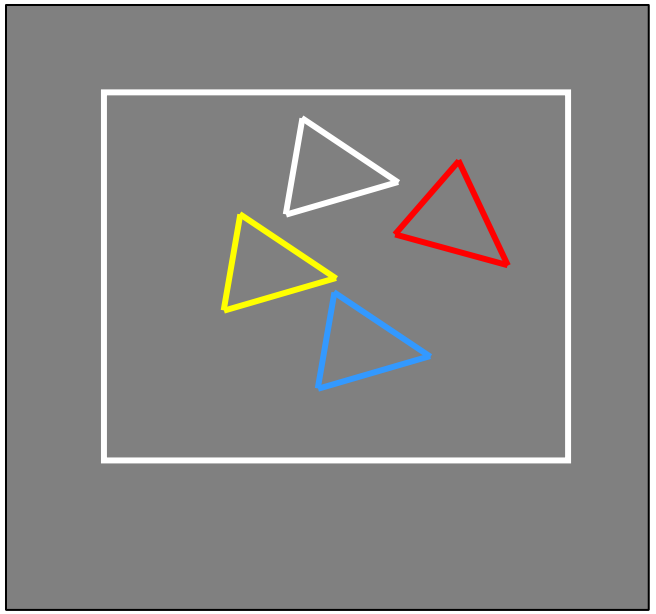
B-KD Tree Subdivision

- **Bounding Volume Hierarchy (partially unbounded)**
- **Each node can be associated with a full bounding box**
- **Bounds may overlap**
  - Primitives in single leaf nodes
  - More traversal steps as for KD Tree
  - Support for dynamic scenes



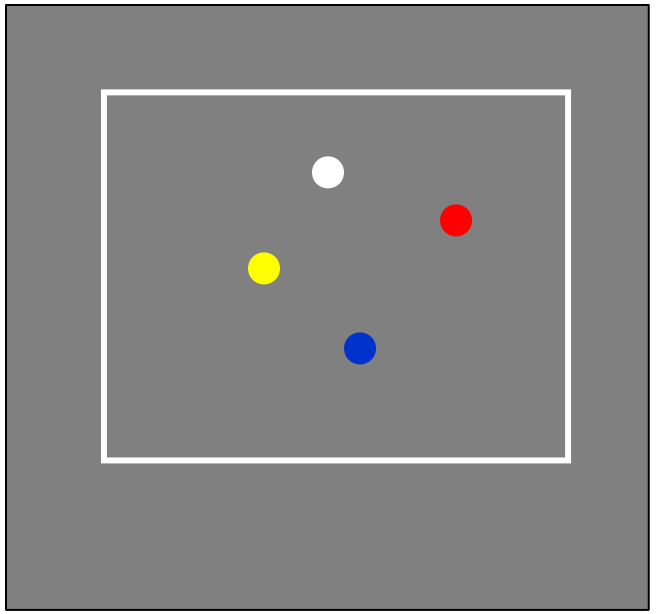
B-KD Tree Construction

- **If #primitives > 1 then**



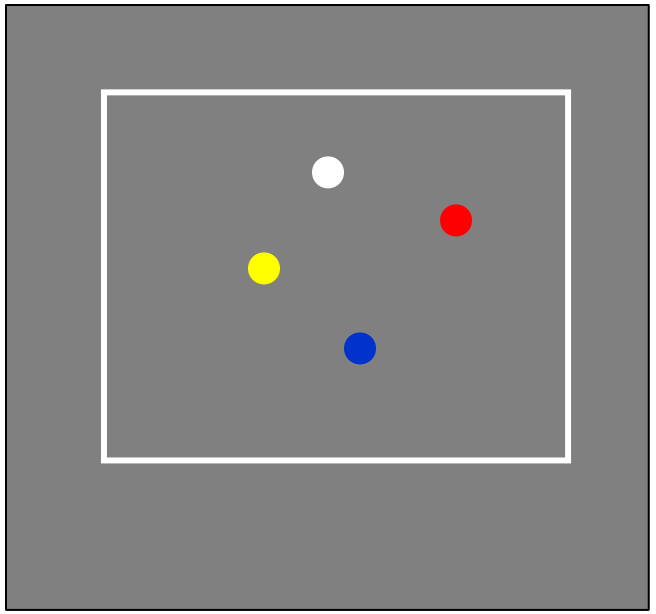
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass



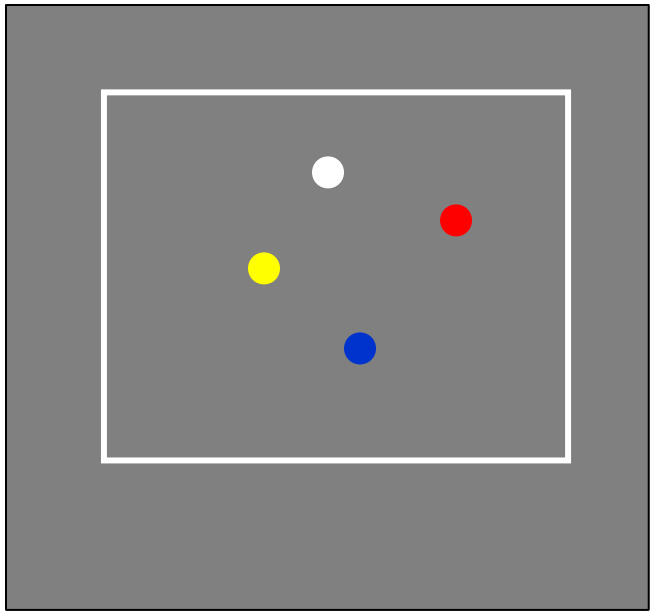
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Spatial Median
  - Object Median



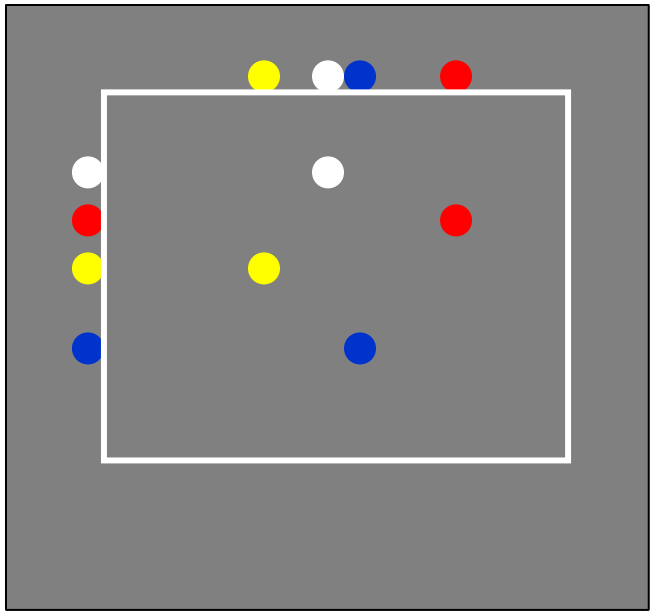
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - ~~Spatial Median~~
  - ~~Object Median~~



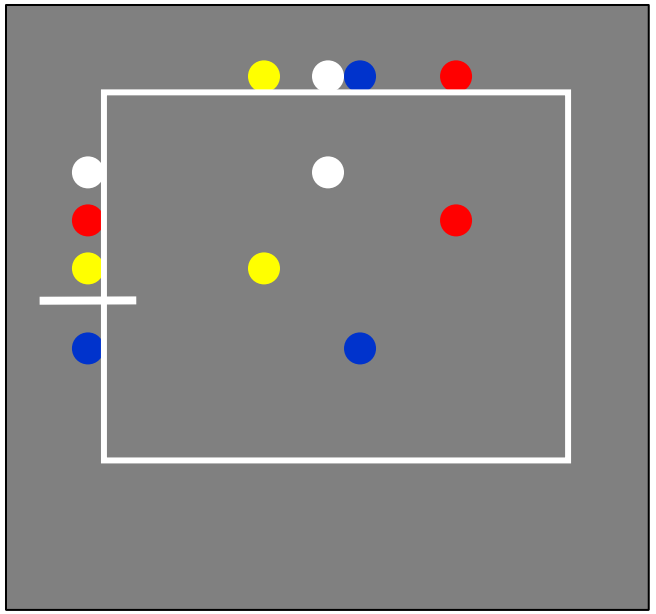
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions



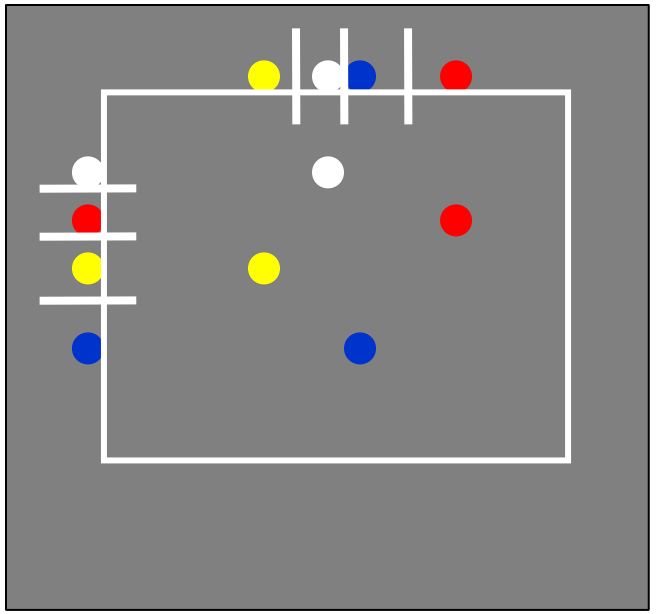
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions
  - Partitions can be determined by splitting a list at a position



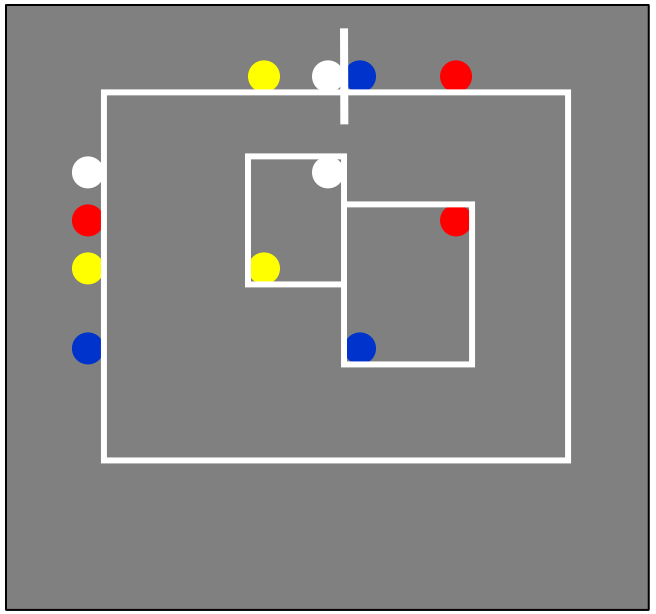
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions
  - Partitions can be determined by splitting a list at a position
  - Build all possible partitions in all three dimensions



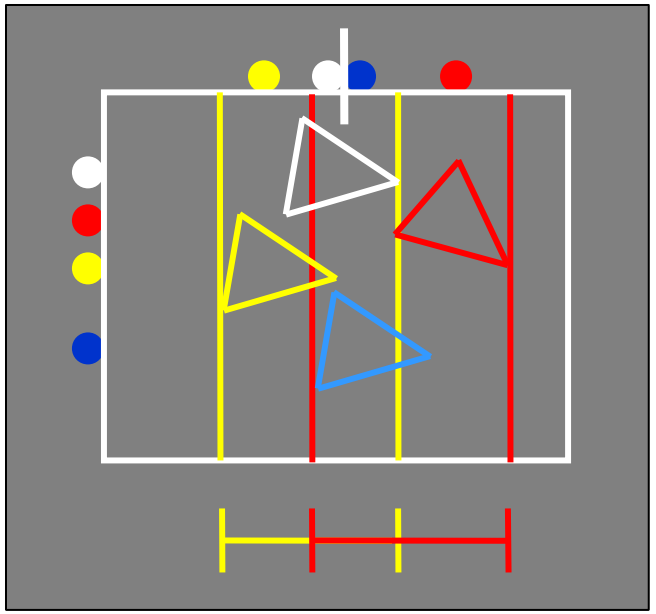
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions
  - Partitions can be determined by splitting a list at a position
  - Build all possible partitions in all three dimensions
  - Find the partitioning with smallest SAH cost



B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions
  - Partitions can be determined by splitting a list at a position
  - Build all possible partitions in all three dimensions
  - Find the partitioning with smallest SAH cost
  - Create node and recurse



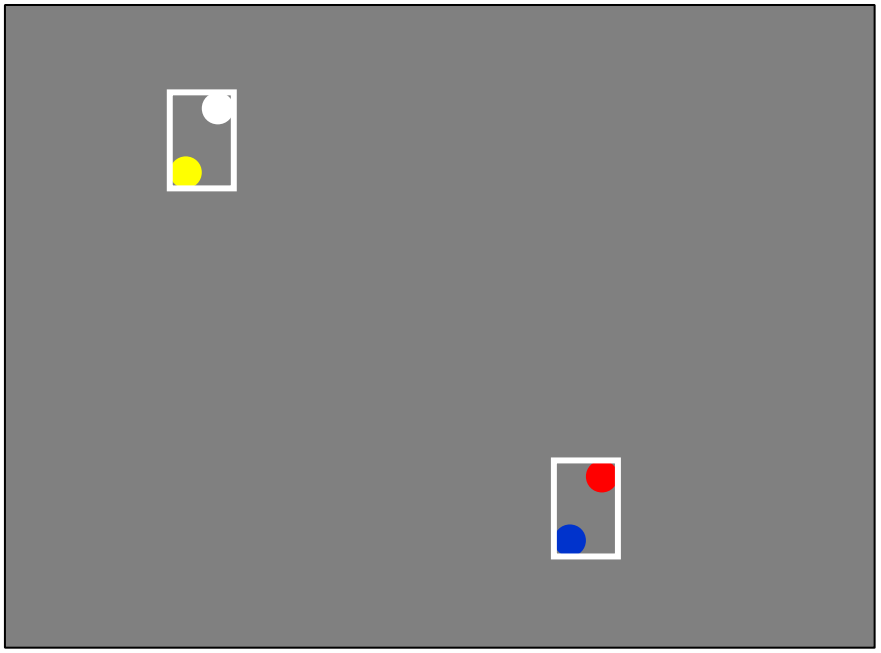
B-KD Tree Construction

- **If #primitives > 1 then**
  - Compute center of mass
  - Sort geometry along all three dimensions
  - Partitions can be determined by splitting a list at a position
  - Build all possible partitions in all three dimensions
  - Find the partitioning with smallest SAH cost
  - Create node and recurse
- **Else if #primitives = 1 then**
  - Create leaf node

B-KD Tree Construction

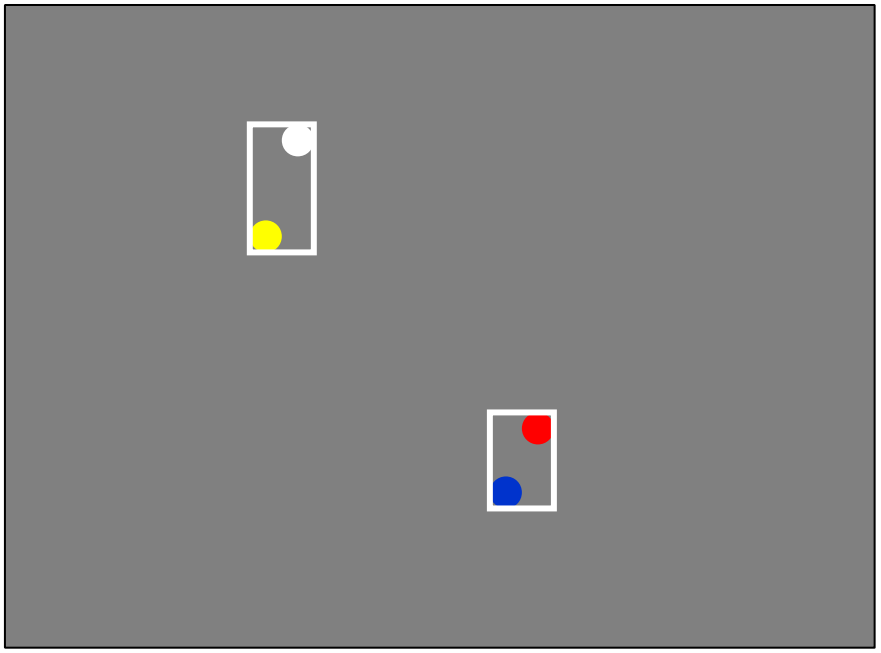
- **Rendering Performance**
  - 20% to 100% better than center splitting approaches
- **Two-level B-KD Trees**
  - Top-level B-KD tree over object instances
  - Bottom-level B-KD tree for each object
- **On changed object geometry**
  - B-KD tree bounds are updated from bottom up
  - B-KD tree structure remains constant
  - Linear updating complexity

Examples



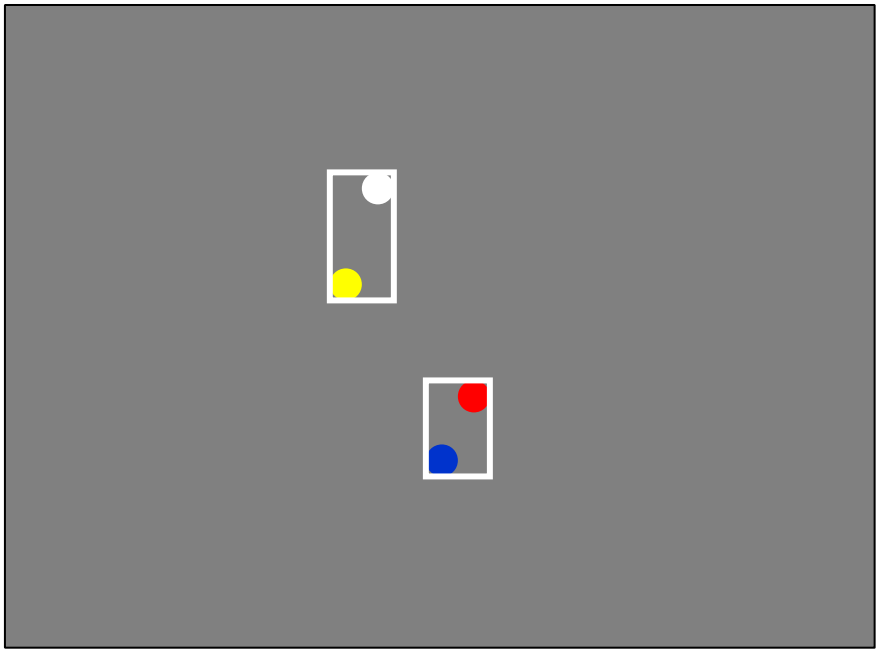
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



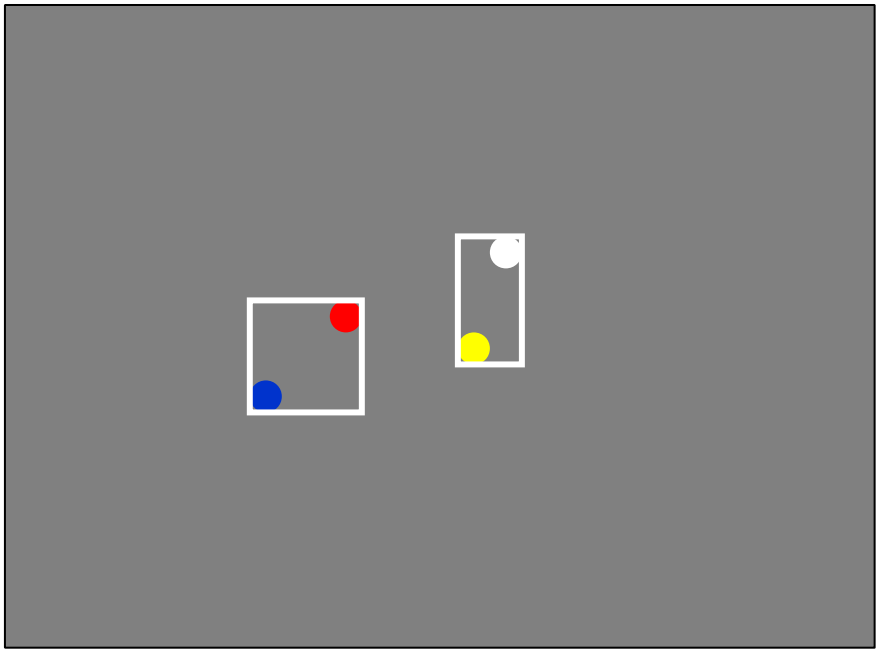
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



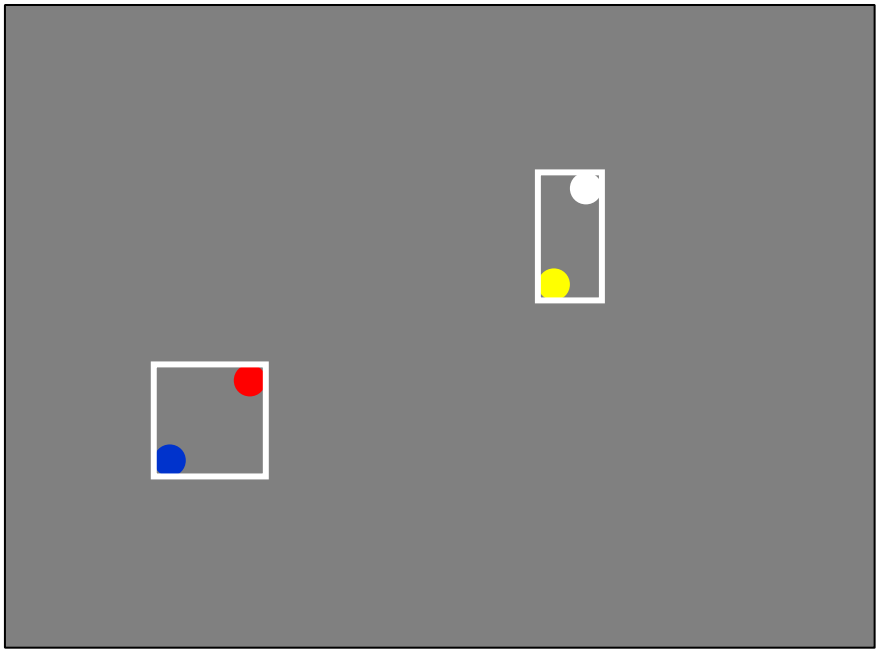
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



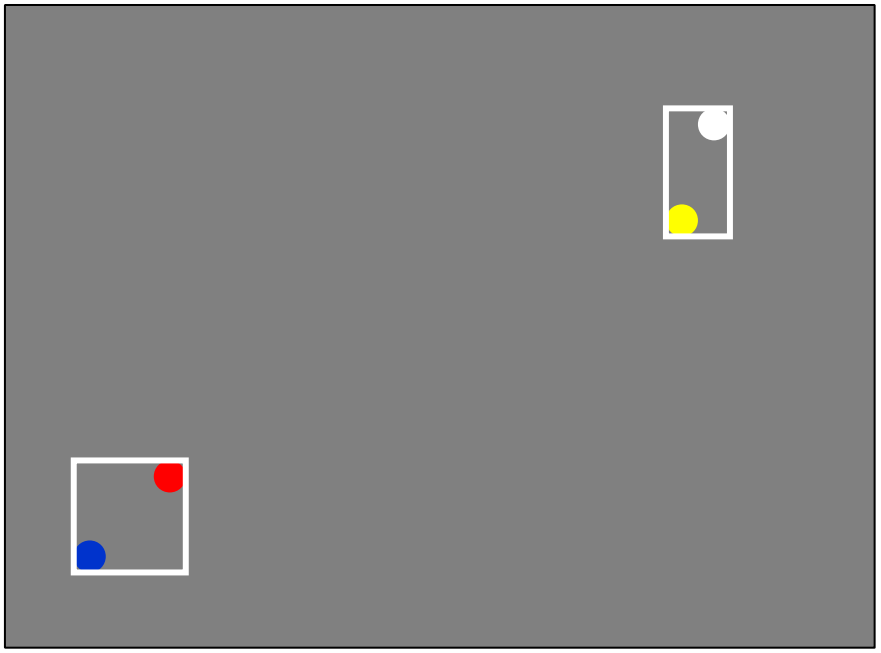
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



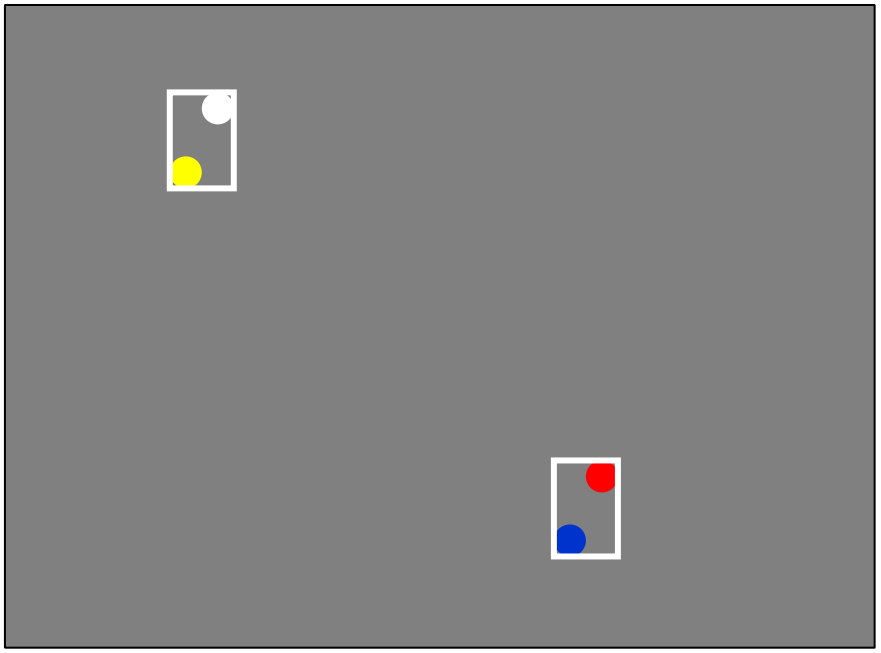
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



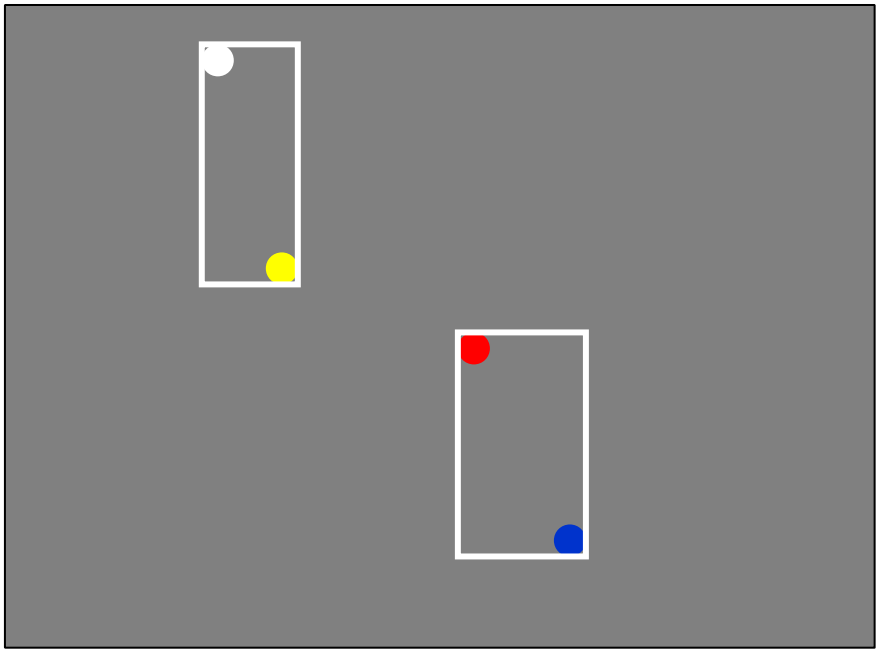
- **Bounding approaches perform well for**
  - Continuos motion
  - Structure of motion must match tree structure
  - E.g. skinned meshes, characters, water surfaces, ...

Examples



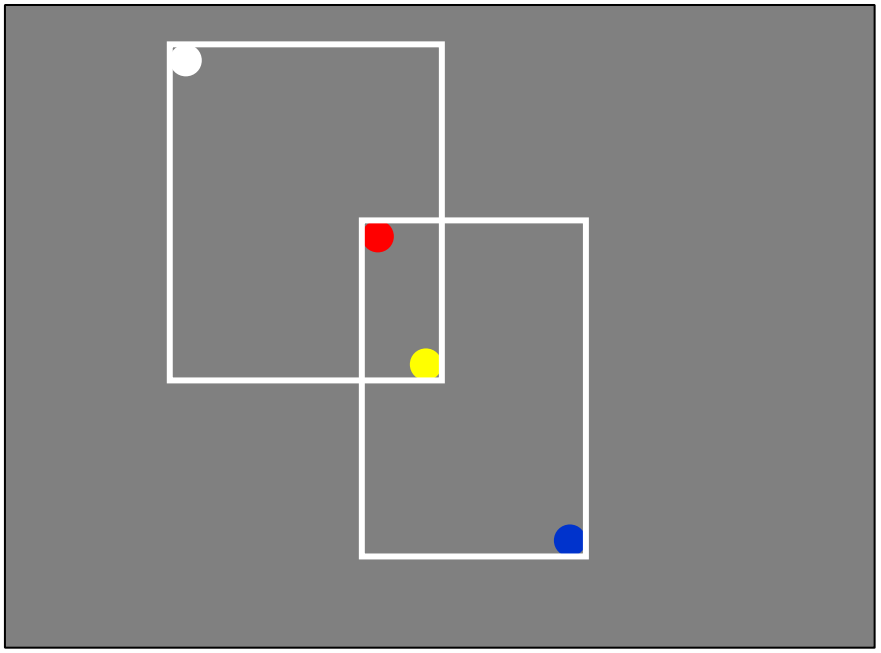
- **Bounding volume approaches are less efficient for**
  - Non-continuos motion
  - Structure of motion does not match tree structure
  - High traversal cost due to large overlapping boxes

Examples



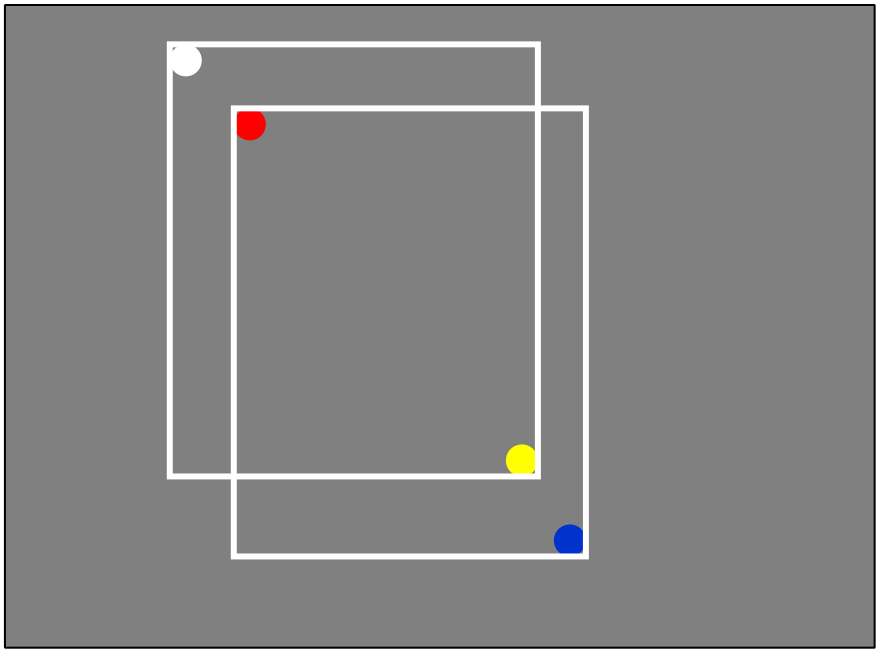
- **Bounding volume approaches fail for**
  - Non-continuos motion
  - Structure of motion does not match tree structure
  - High traversal cost due to large overlapping boxes

Examples



- **Bounding volume approaches fail for**
  - Non-continuos motion
  - Structure of motion does not match tree structure
  - High traversal cost due to large overlapping boxes

Examples



- **Bounding volume approaches fail for**
  - Non-continuos motion
  - Structure of motion does not match tree structure
  - High traversal cost due to large overlapping boxes

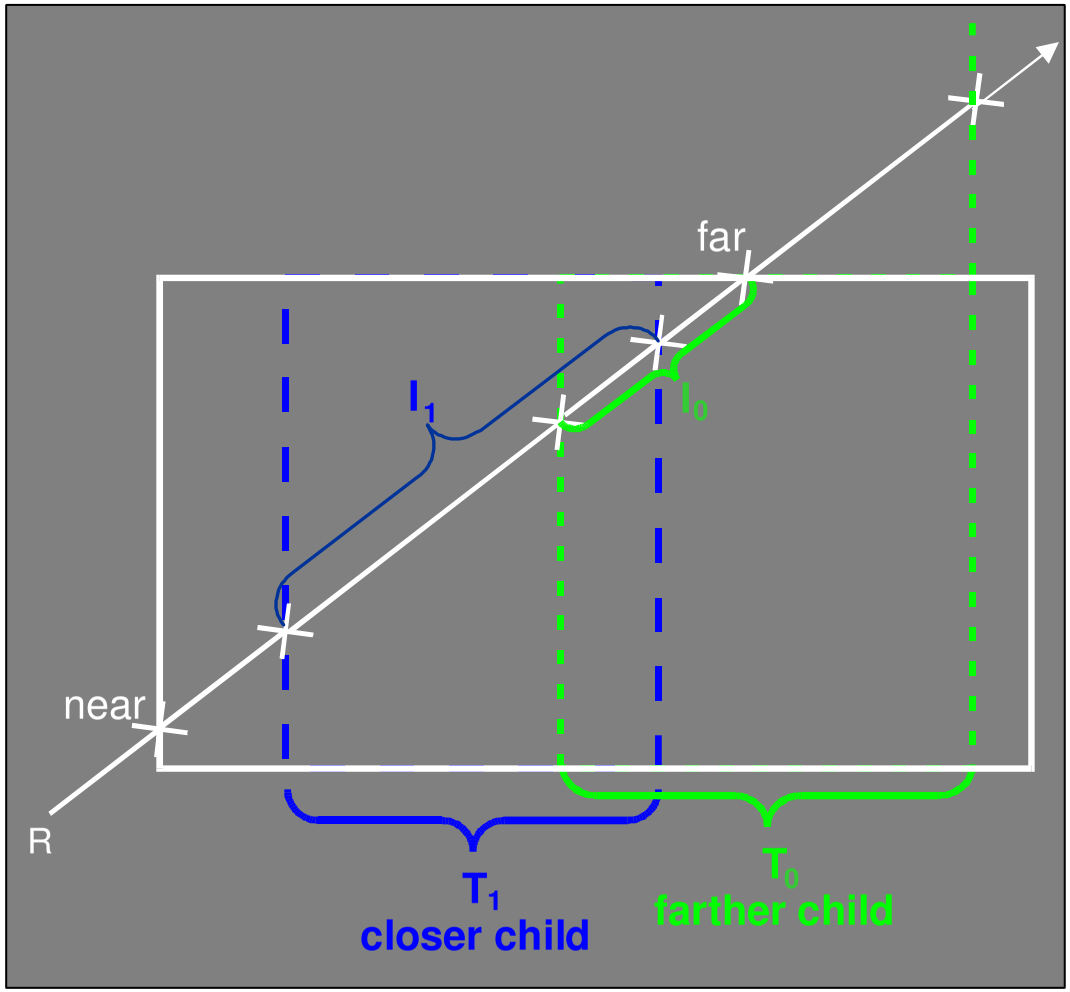
Traversal of B-KD Trees

Traversal of B-KD Trees

- Early ray termination
- Clipping of near/far interval against both bounding intervalls
- Take closer child, push farther child to stack
- Traversal order does not affect correctness

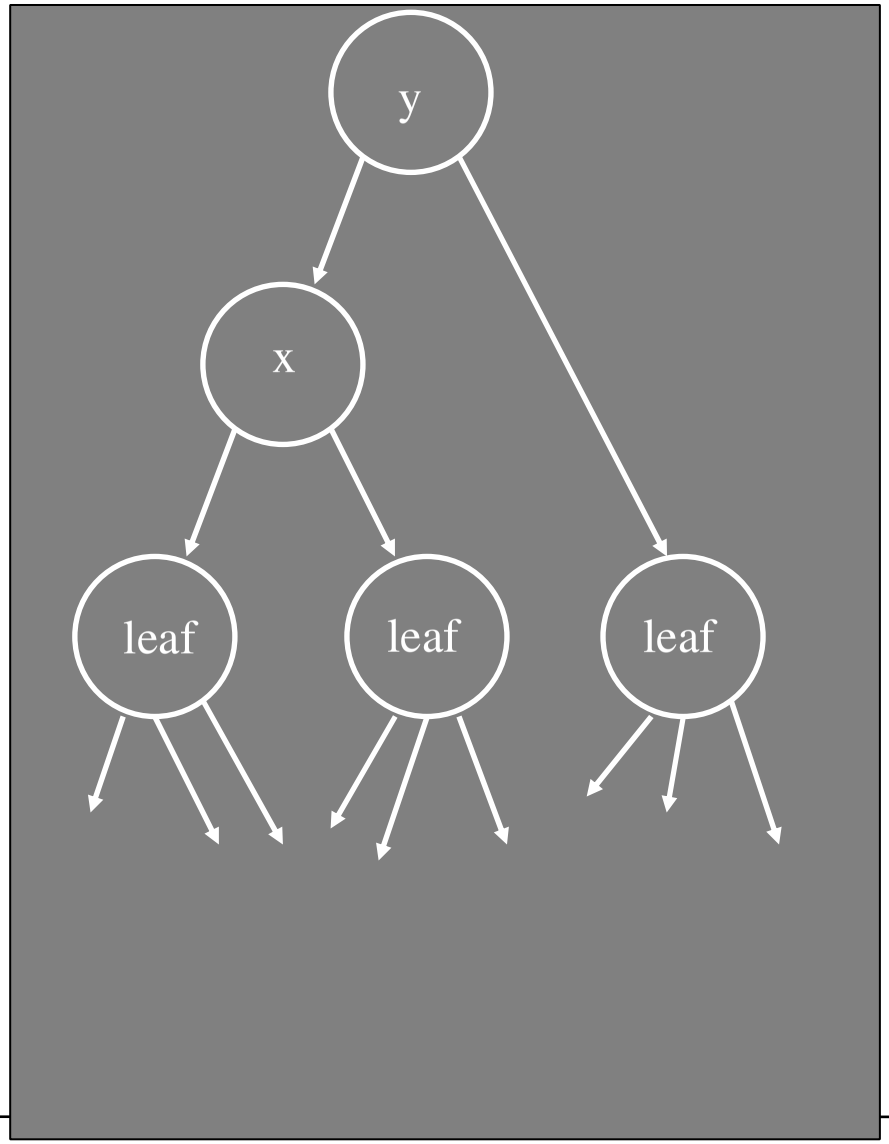
Complexity

- 4x computational cost of KD tree traversal step
- 2x stack memory



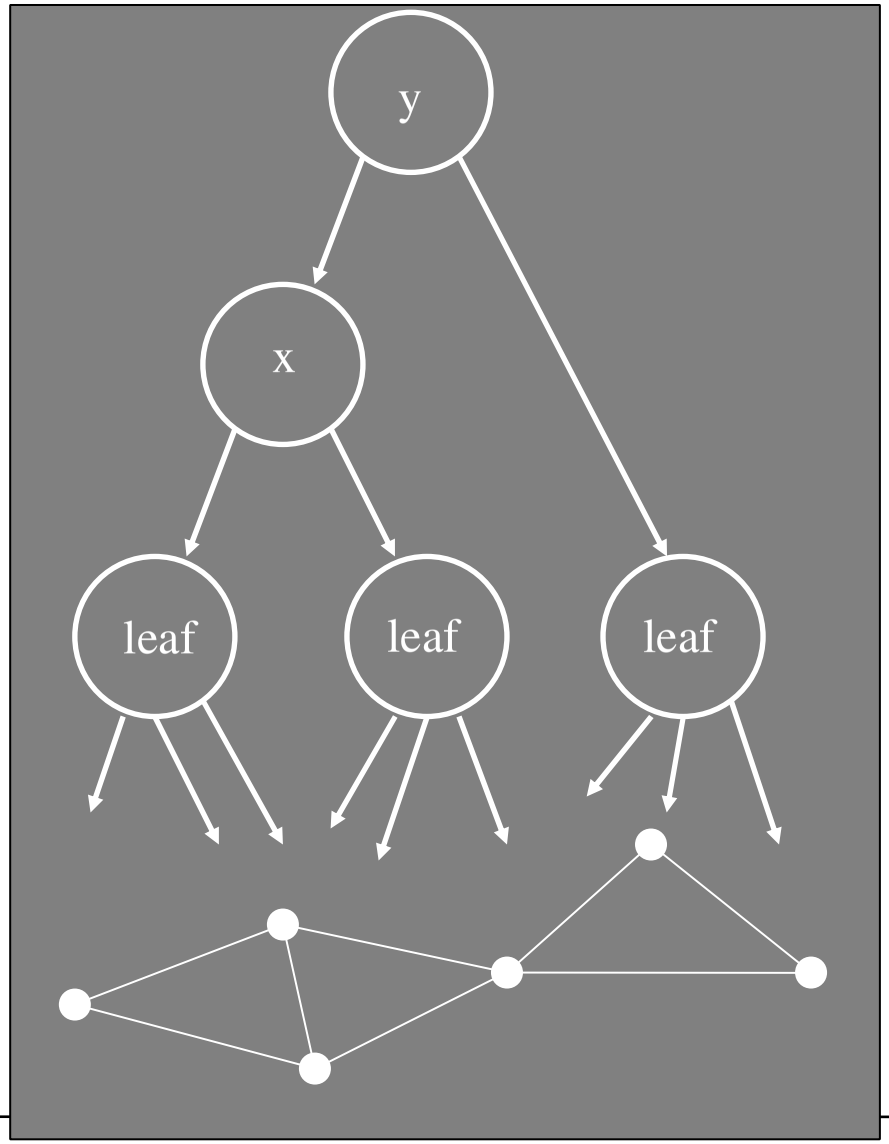
Update of B-KD Trees

- **Leaf Node**



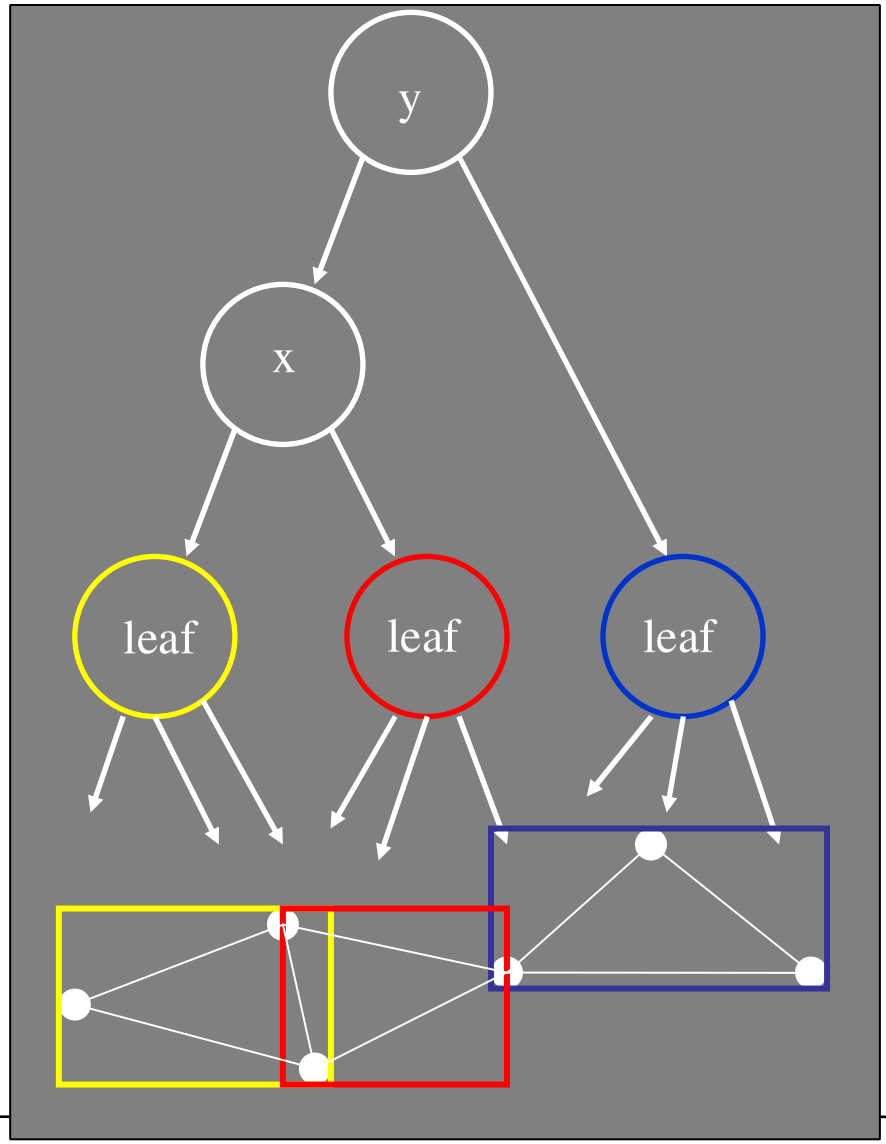
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices



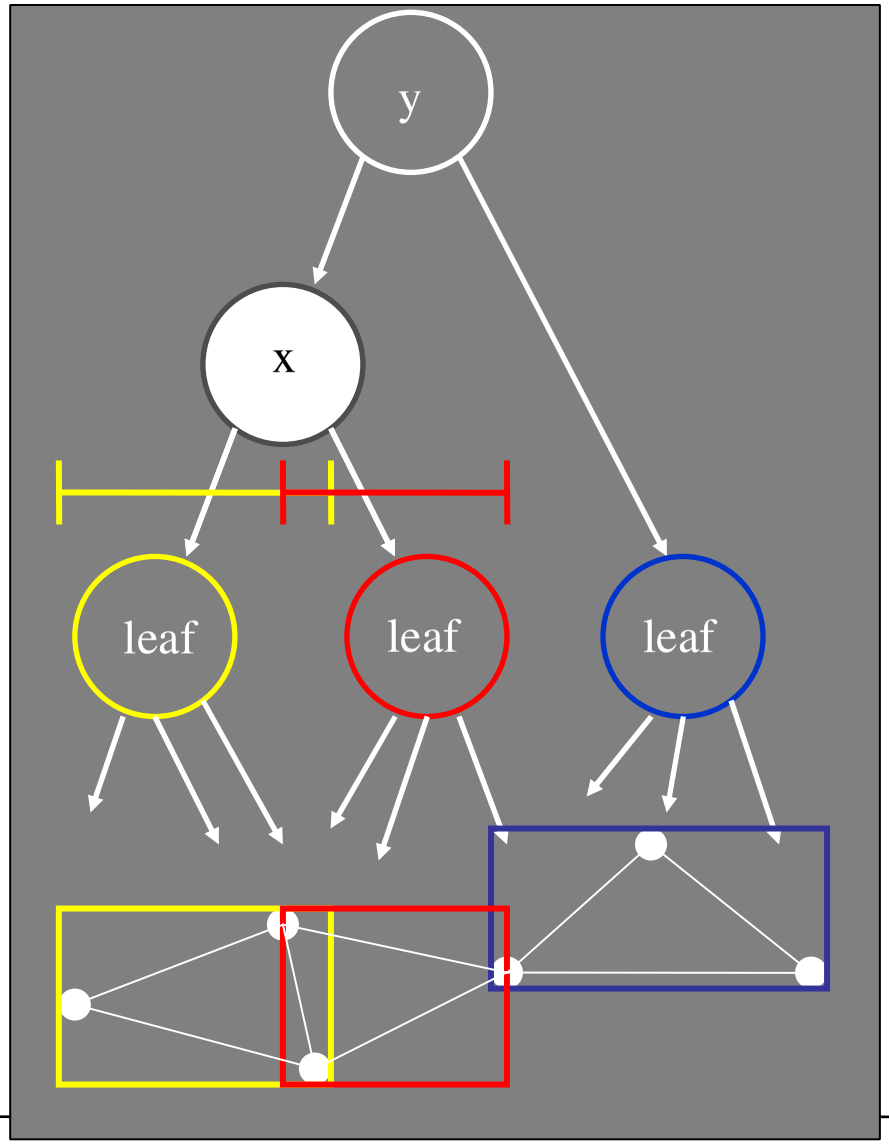
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices
  - Compute leaf boxes



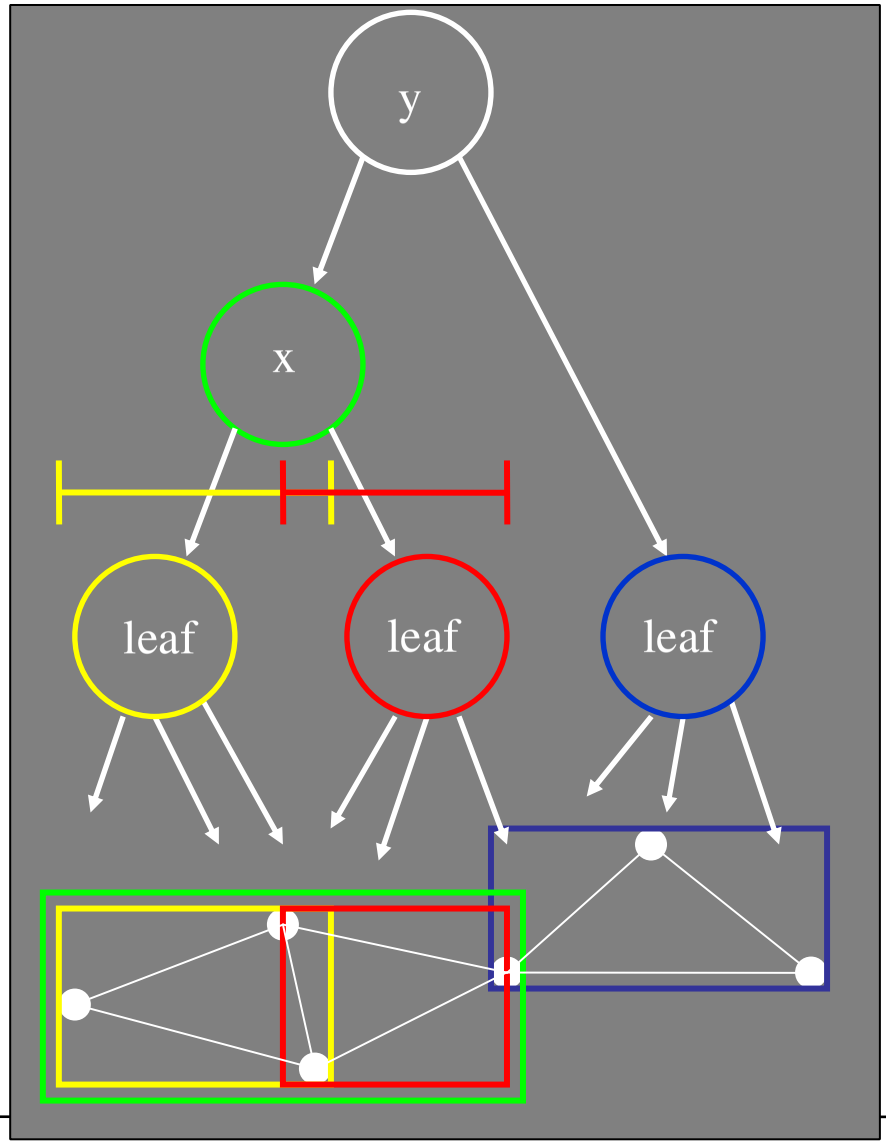
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices
  - Compute leaf boxes
- **Inner Node**
  - Update 1D node bounds



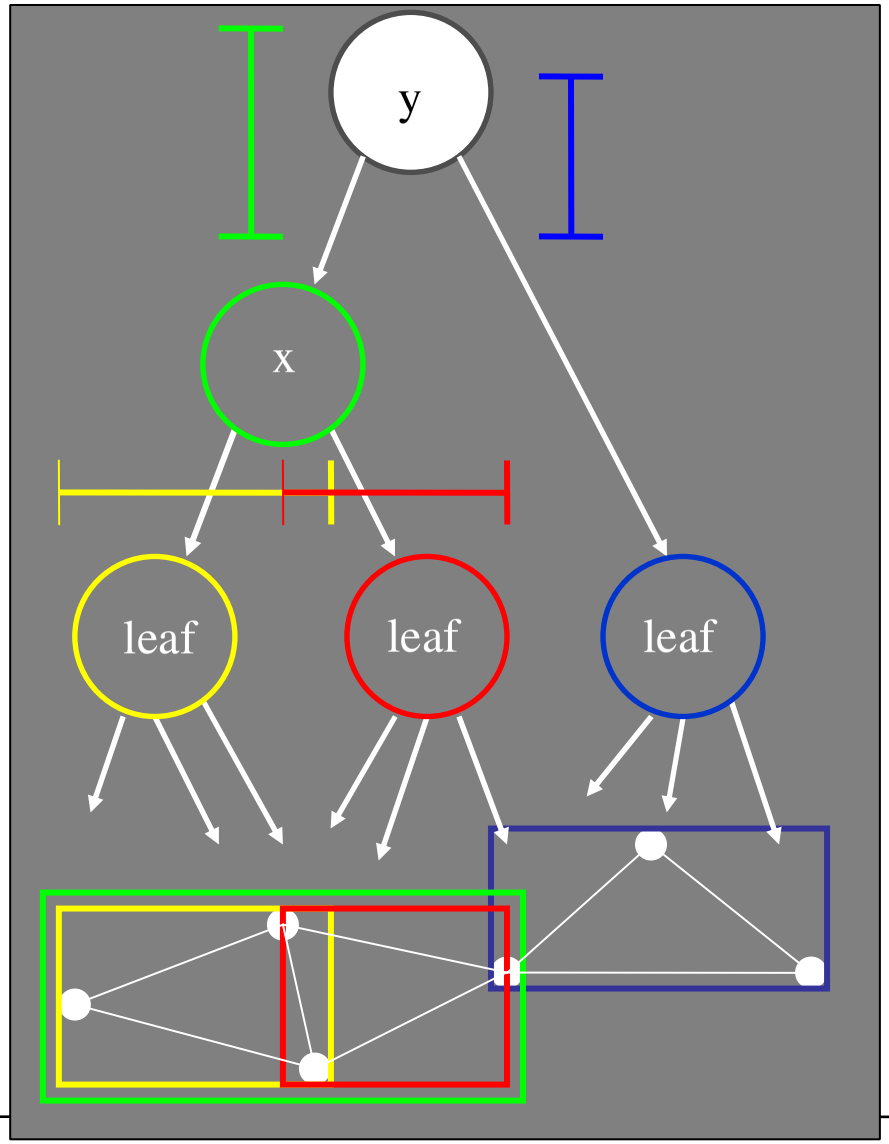
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices
  - Compute leaf boxes
- **Inner Node**
  - Update 1D node bounds
  - Merge boxes of both children



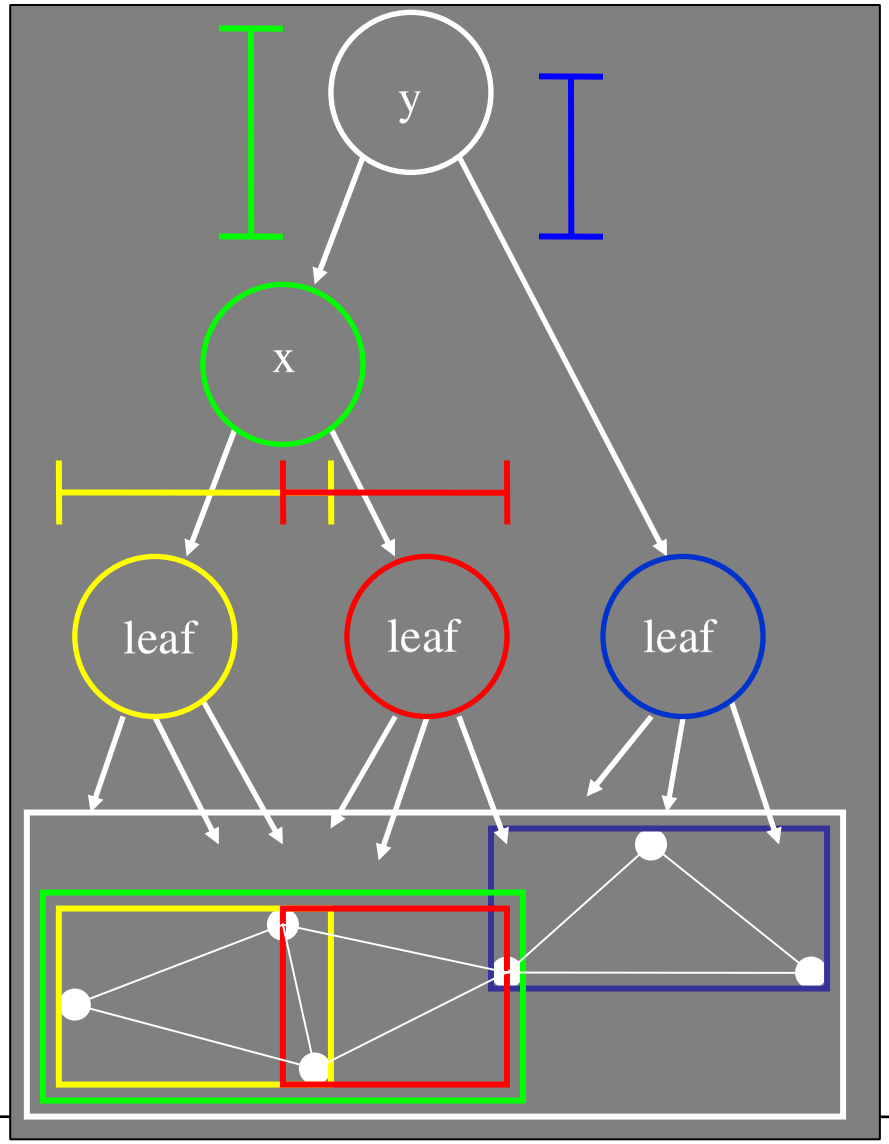
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices
  - Compute leaf boxes
- **Inner Node**
  - Update 1D node bounds
  - Merge boxes of both children



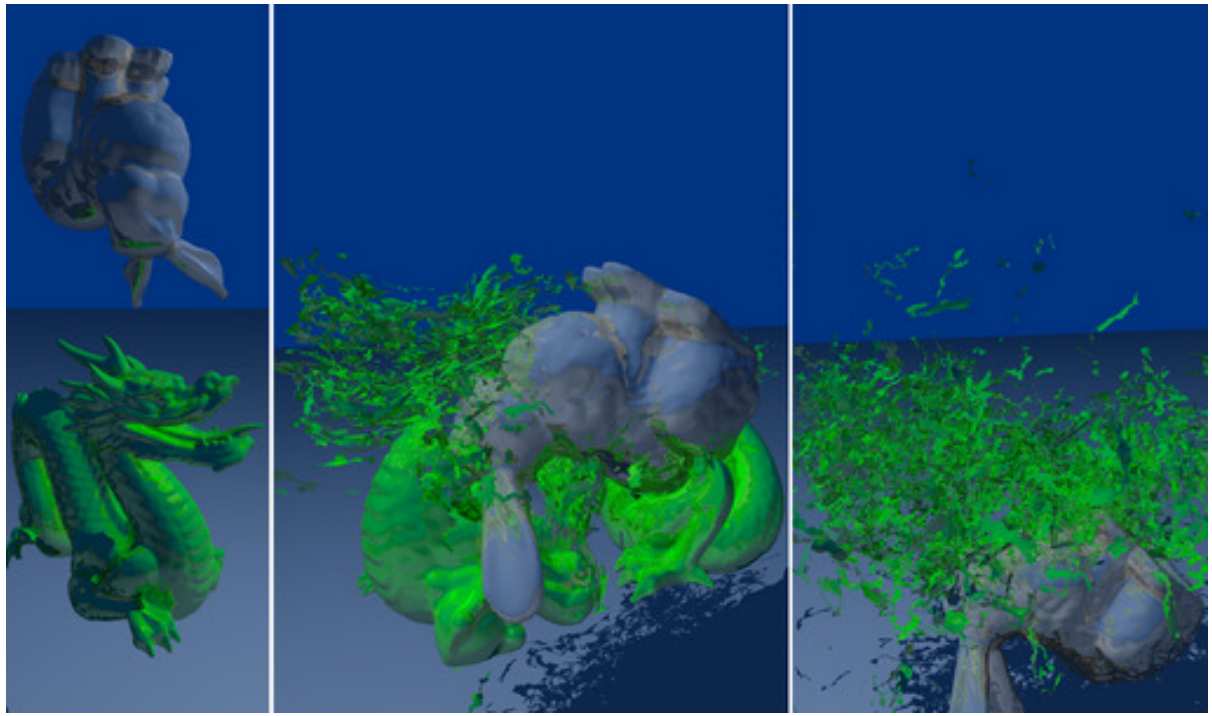
Update of B-KD Trees

- **Leaf Node**
  - Fetch vertices
  - Compute leaf boxes
- **Inner Node**
  - Update 1D node bounds
  - Merge boxes of both children



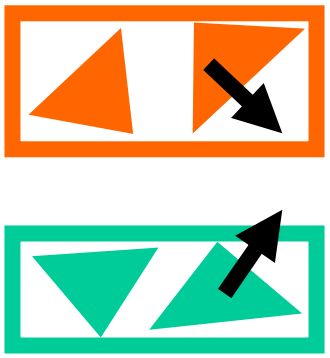
Fastest Current Dynamic Ray-tracing

- **following slides from [Yoon et al. EGSR 2007]**
  "Ray Tracing Dynamic Scenes using Selective Restructuring"
- **massive dynamic scenes**
- **based on BVH**



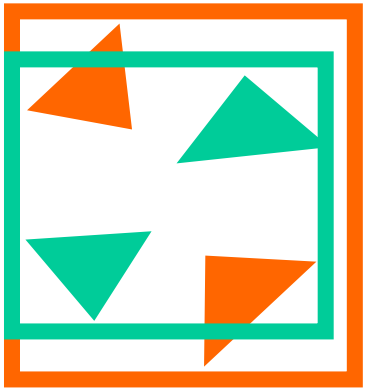
Two BVH Update Methods

Frame 1



BV refitting

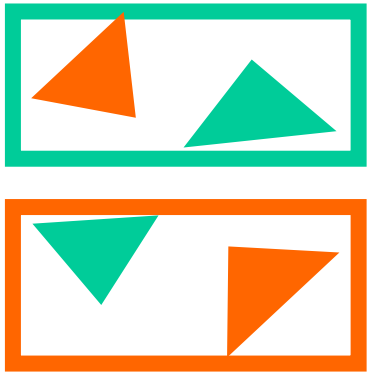
- **O(n)**
- **Poor-quality BVs**



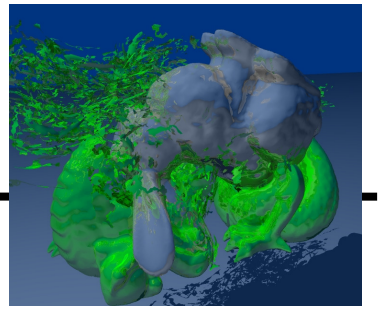
BV reconstruction

- **O(n log n)**
- **Good-quality BVs**

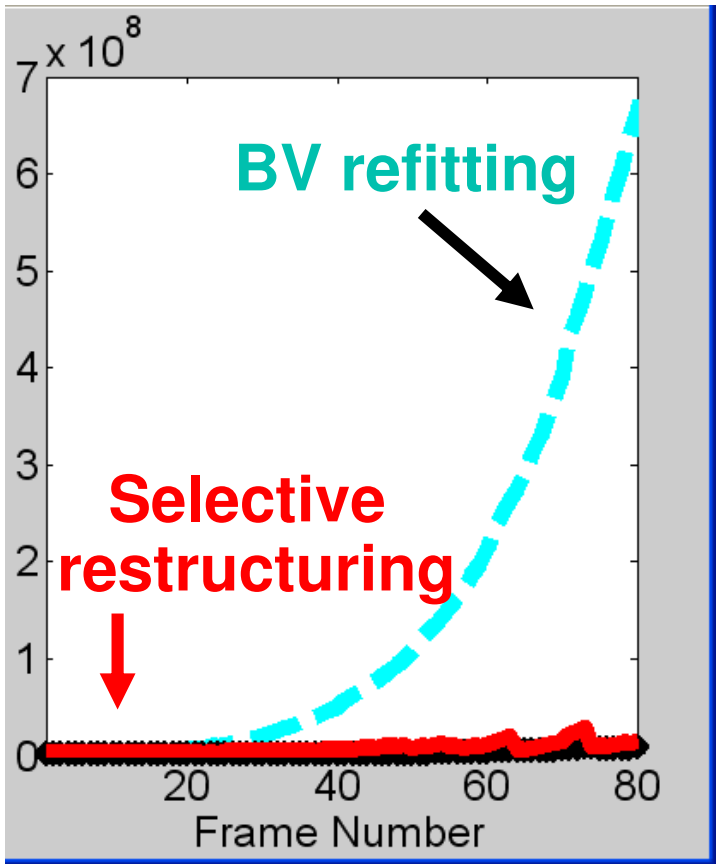
Frame 2



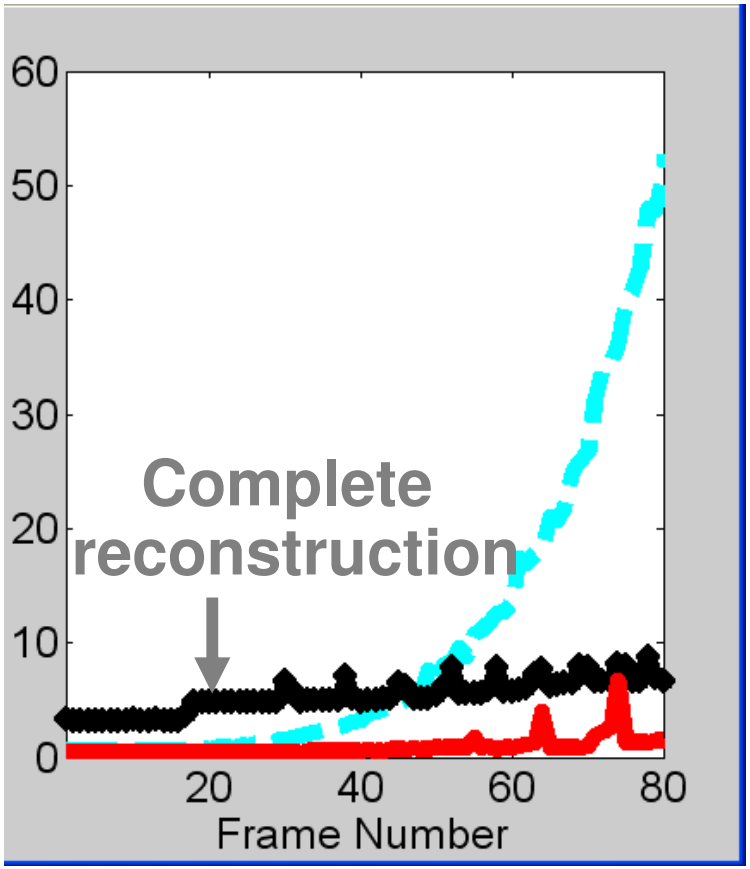
Example of Exploding Dragon Model



of intersections

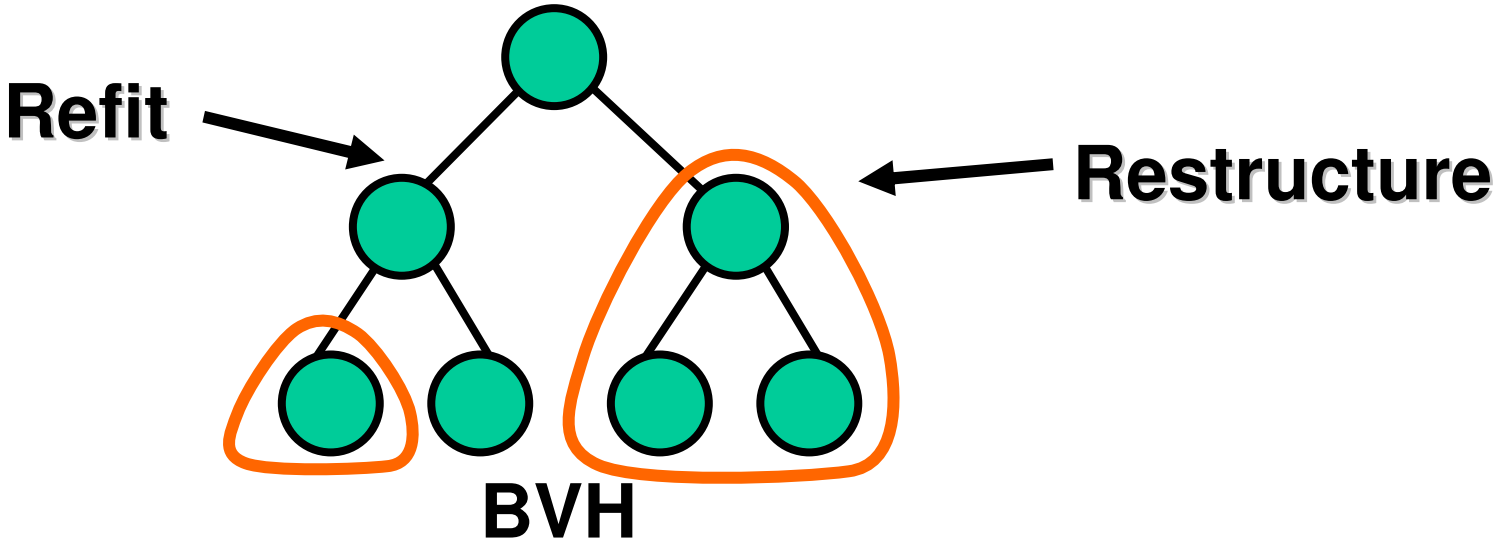


Ray tracing time (sec): construction + traversal



Selective Restructuring

- **selective restructuring based on two metrics**
  - culling efficiency
  - restructuring benefit



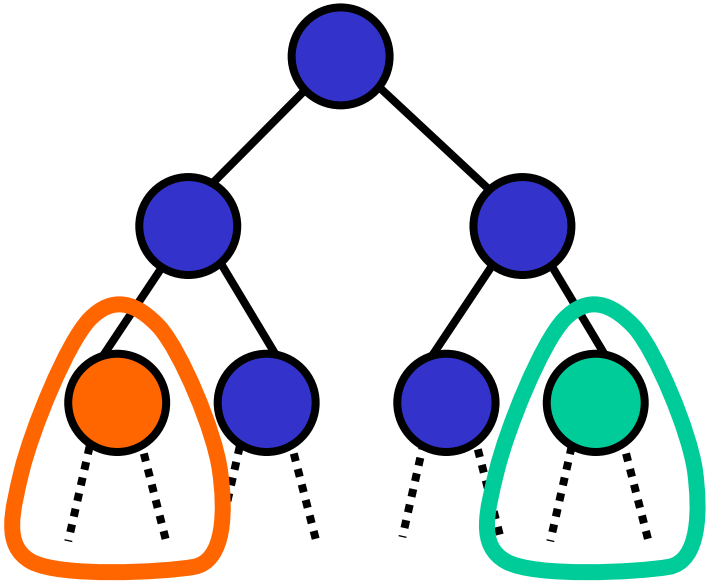
Probabilistic BVH Metrics for Ray Tracing

- **Culling efficiency**
  - Quantifies the quality of any sub-BVHs
  - Measures the expected # of intersection tests for a ray

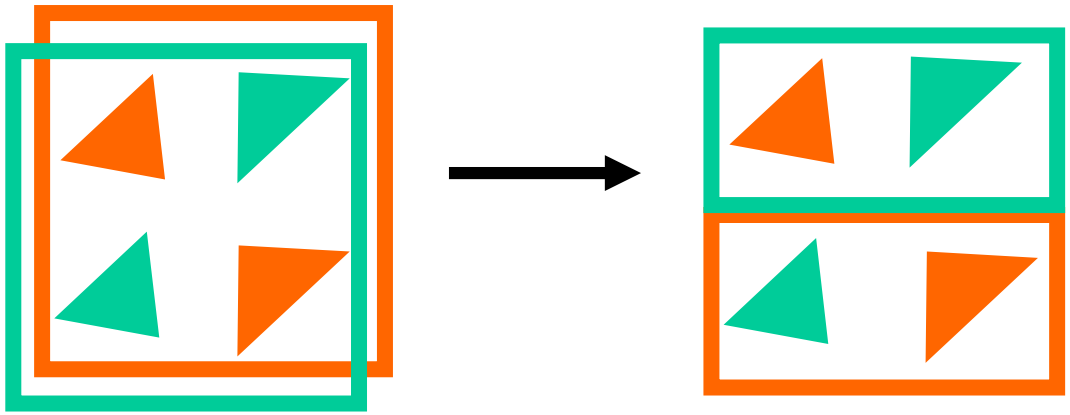
- **Restructuring benefit**
  - Predicts the performance improvement
  - Measures improved culling efficiency when restructuring sub-BVHs

Major Observation

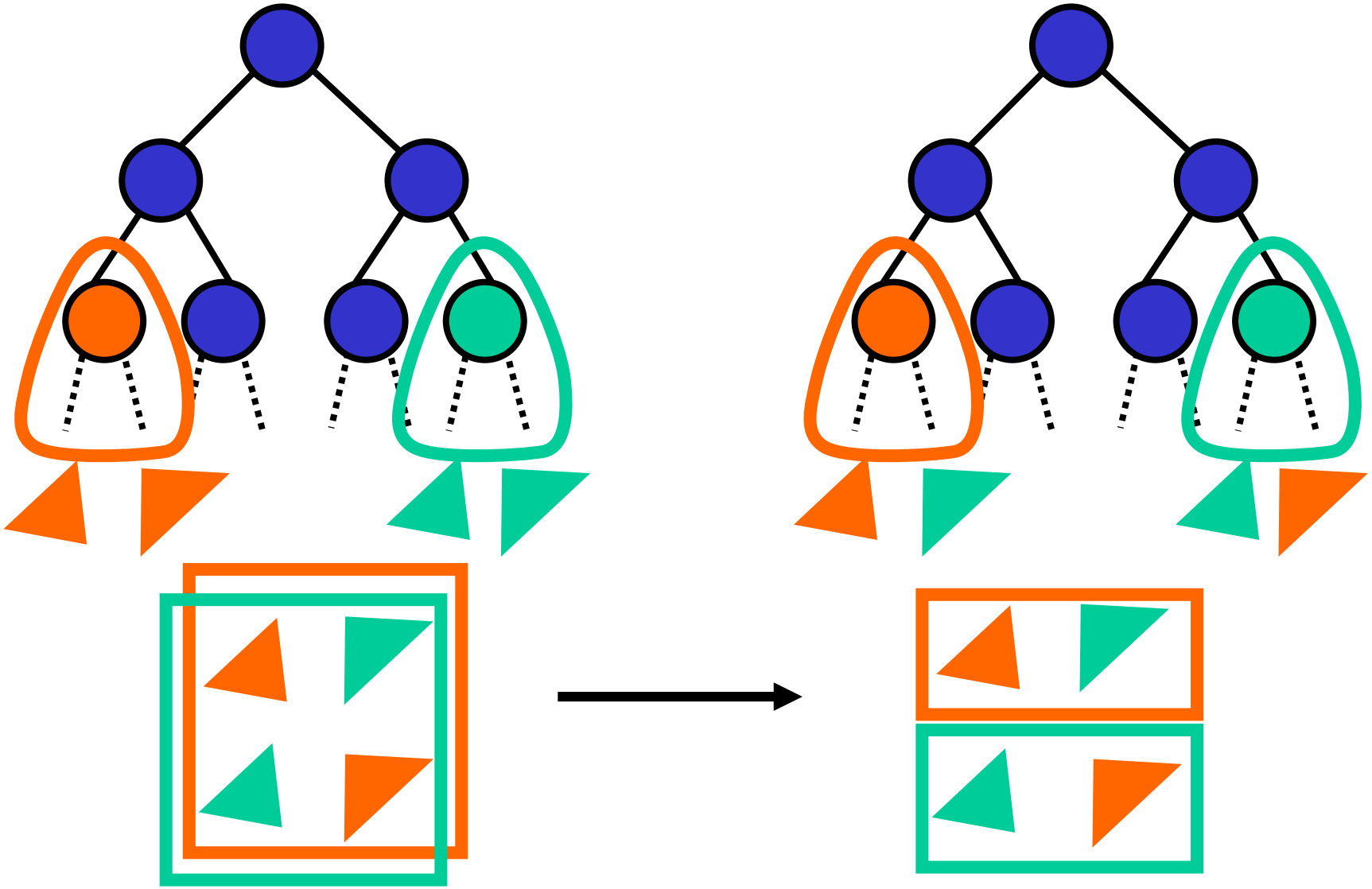
- **Restructuring any two nodes with BV overlaps can improve the culling efficiency**
  - Assumes that restructuring operation will remove all the BV overlaps



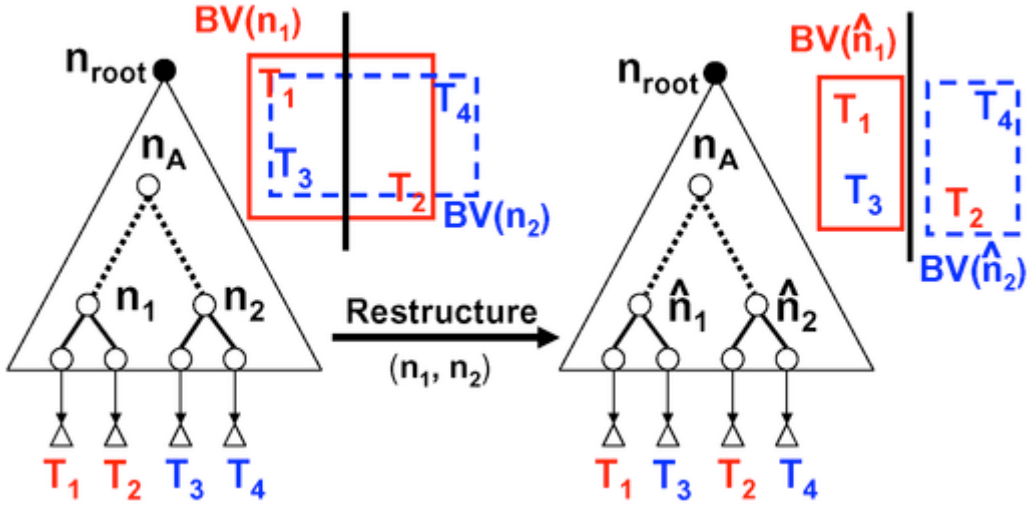
A BVH



Restructuring any Pair of Sub-trees



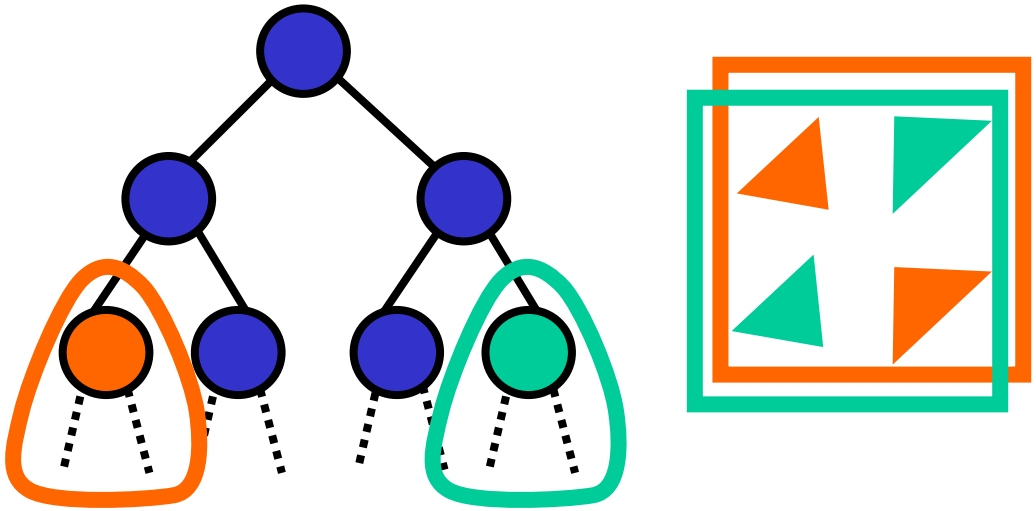
Restructuring Sub-trees



- **complexity of restructuring n_1 + n_2 O(k log k), with k = |n_1| + |n_2|**
- **compare to O(p log p), with p = |n_A|**
- **How to find good candidates?**

Detecting BV Overlaps

- **Brute-force method**
  - Requires $O(m^2)$ where m is # of BVs
- **Hierarchical traversal and culling**
  - top-down



Overview of Selective Restructuring Algorithm

- **Computes restructuring candidates**
  - Detects nodes with BV overlaps during hierarchy traversal
- **Restructure node pairs with higher restructuring benefits greedily**
  - Improves the performance of ray tracing

Culling Efficiency Model

- **Measure C(n) for how many intersection test are likely for the sub-BVH(n) given BVH(n) was hit.**
- **#intersections of BVH(n) recorded from previous frame**

$$C(n) = \begin{cases} |n|_{tri} * C_{leaf}, & \text{if } n \text{ is a leaf node} \\ 1 + P(Left(n)) * C(Left(n)) + \\ \quad P(Right(n)) * C(Right(n)), & \text{otherwise,} \end{cases}$$

(compare to KD-tree heuristic)

- **Estimated cost for jointly considering two sub-trees a and b**

$$C(n) = C(a \cup b) = \frac{Area(a)}{Area(n)} C(a) + \frac{Area(b)}{Area(n)} C(b).$$

Estimated Cost after Restructuring

- **Given n_1 and n_2 to be restructured**
- **n_c the region in common to both**
- **Let n_1 be the sub-tree before and $\hat{n}_1$ after restructuring**
- **n_1^r is the approximated reduced region after restructuring**
  - assume all overlap removed
  - subtract half the overlap surface plus plane parallel to cutting plane
- **The cost benefit can be approximated**

$$\Delta C(n_1) = C(n_1) - C(\hat{n}_1) \approx C(n_1) - C(n_1^c) \approx C(n_1) - \frac{Area(n_1^c)}{Area(n_1)} C(n_1)$$

$$\approx C(n_1) \frac{Area(n_1^r)}{Area(n_1)}.$$

Final Cost Benefit (RM-IQ)

- **consider probability that BVH(n) is hit (previous frame)**

$$R(n_1) = C(n_1)\frac{Area(n_1^r)}{Area(n_1)}P(n_1).$$

- **restructure n_1 and n_2 only if**

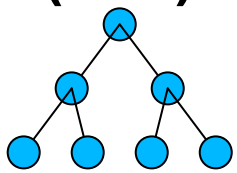
$$R(n_1) + R(n_2) > f_{rebuild}(n_1, n_2)$$

benefit outperforms restructuring cost

$$f_{rebuild}(n_1, n_2) = c_{rebuild} k \log k \qquad k = |n_1|_{tri} + |n_2|_{tri}$$

Top-Down Traversal

- **Create list of pairs**
- **avoid O(n^2)**

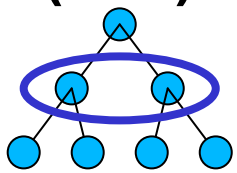


Algorithm 1 Selective Restructuring Algorithm

```
1:  // Hierarchical Refinement Phase
2:  PairQ ← (Left(n_Root), Right(n_Root)) // Add initial pair to the queue
3:  while (! PairQ.empty ()) do
4:      Pair ← PairQ.front ();
5:      EvaluateRM-IQ (Pair); // Evaluate RM-IQ based on the overlap
6:      // Check whether restructuring benefit of a pair is larger than the refining cost
7:      if (WorthRefine(Pair)) then
8:          PairQ ← Refine (Pair);
9:      end if
10:     Propagate (Pair); // Propagate RM-IQ value of Pair to pairs, which are refined to Pair
11: end while
12: // Restructuring Phase
13: PairH ← All created pairs // Add the created pairs to the pair heap
14: while (! PairH.empty ()) do
15:     Pair ← PairH.front ();
16:     // Restructure benefit should be larger than the restructuring cost
17:     if (WorthRebuild (Pair)) then
18:         (n_1, n_2) ← Pair
19:         RestructureTwoSubBVHs (n_1, n_2); // Described in Sec. 3.2
20:     end if
21: end while
```

Top-Down Traversal

- **Create list of pairs**
- **avoid O(n^2)**

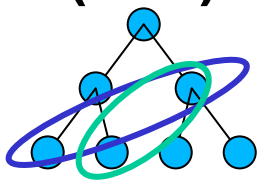


Algorithm 1 Selective Restructuring Algorithm

```
1:  // Hierarchical Refinement Phase
2:  PairQ ← (Left(n_Root), Right(n_Root)) // Add initial pair to the queue
3:  while (! PairQ.empty ()) do
4:      Pair ← PairQ.front ();
5:      EvaluateRM-IQ (Pair); // Evaluate RM-IQ based on the overlap
6:      // Check whether restructuring benefit of a pair is larger than the refining cost
7:      if (WorthRefine(Pair)) then
8:          PairQ ← Refine (Pair);
9:      end if
10:     Propagate (Pair); // Propagate RM-IQ value of Pair to pairs, which are refined to Pair
11: end while
12: // Restructuring Phase
13: PairH ← All created pairs // Add the created pairs to the pair heap
14: while (! PairH.empty ()) do
15:     Pair ← PairH.front ();
16:     // Restructure benefit should be larger than the restructuring cost
17:     if (WorthRebuild (Pair)) then
18:         (n_1, n_2) ← Pair
19:         RestructureTwoSubBVHs (n_1, n_2); // Described in Sec. 3.2
20:     end if
21: end while
```

Top-Down Traversal

- **Create list of pairs**
- **avoid O(n^2)**



Algorithm 1 Selective Restructuring Algorithm

```
1:  // Hierarchical Refinement Phase
2:  PairQ ← (Left(n_Root), Right(n_Root)) // Add initial pair to the queue
3:  while (! PairQ.empty ()) do
4:      Pair ← PairQ.front ();
5:      EvaluateRM-IQ (Pair); // Evaluate RM-IQ based on the overlap
6:      // Check whether restructuring benefit of a pair is larger than the refining cost
7:      if (WorthRefine(Pair)) then
8:          PairQ ← Refine (Pair);
9:      end if
10:     Propagate (Pair); // Propagate RM-IQ value of Pair to pairs, which are refined to Pair
11: end while
12: // Restructuring Phase
13: PairH ← All created pairs // Add the created pairs to the pair heap
14: while (! PairH.empty ()) do
15:     Pair ← PairH.front ();
16:     // Restructure benefit should be larger than the restructuring cost
17:     if (WorthRebuild (Pair)) then
18:         (n_1, n_2) ← Pair
19:         RestructureTwoSubBVHs (n_1, n_2); // Described in Sec. 3.2
20:     end if
21: end while
```

Next Lecture

- **Light transport**

- **Due to VMV 2007 conference:**
- **Change of room for Thursday only:**

MPI, room 024 !!

- **shorter lecture**