Advanced Computer Graphics
Towards Realtime Ray-Tracing

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

Parallelization

- Simple (trivial) parallelization:
  - "Course grain" parallelization = distribution among multiple CPUs/Cores
  - → hence also *"thread-level parallelism"* (TLP)
  - Implementation:
    - Multiple threads (≈ processes), shared memory
    - Multiple processes are distributed among multiple computers, copy scenes onto all of the computers
    - Every process/thread receives an image tile as work packet
    - Pro: no synchronization necessary (only at the very end)
- *Dynamic Load Balancing*:
  - Divide the image into $k \cdot n$ tiles, n = # procs, k = 10 ... 100
  - Every processor (worker) fetches the next work packet (an image tile) from the pool as soon as it is finished with the old one
  - Hence the (wrong) saying: “Ray tracing is embarrassingly parallel."
- More on this in the lecture about massively parallel algorithms

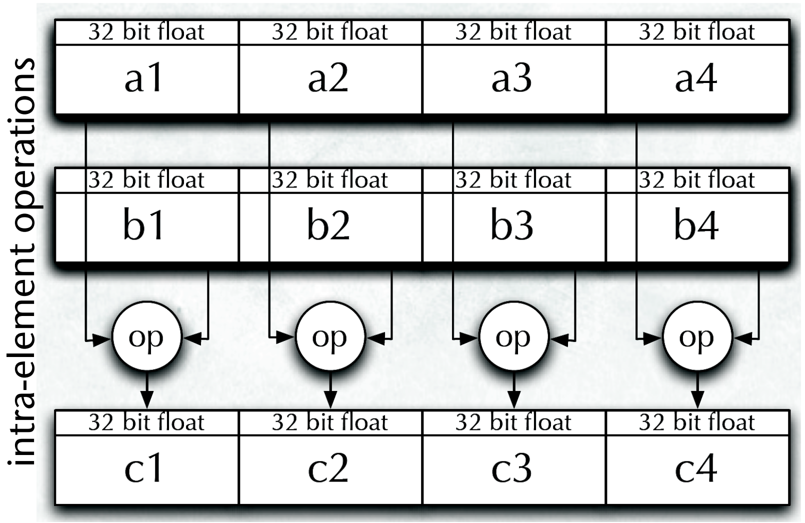
- Another type of parallelization: *Instruction-Level Parallelism* (ILP)
- Example:

```
int a = x + y;          // process 1
int b = u + v;          // process 2
int c = a + b;          // wait for proc 1 & 2
```

- Note:
  - Nowadays, CPUs & Compilers do this on their own
- Gets us nowhere with the *k*d-tree (for example):
  - Work per node on the traversal =
    - Load the float
    - Branch (for axis splitting x, y, z)
    - Div. & Add.
    - Branch (which child first)
  - Branches cancel out ILP

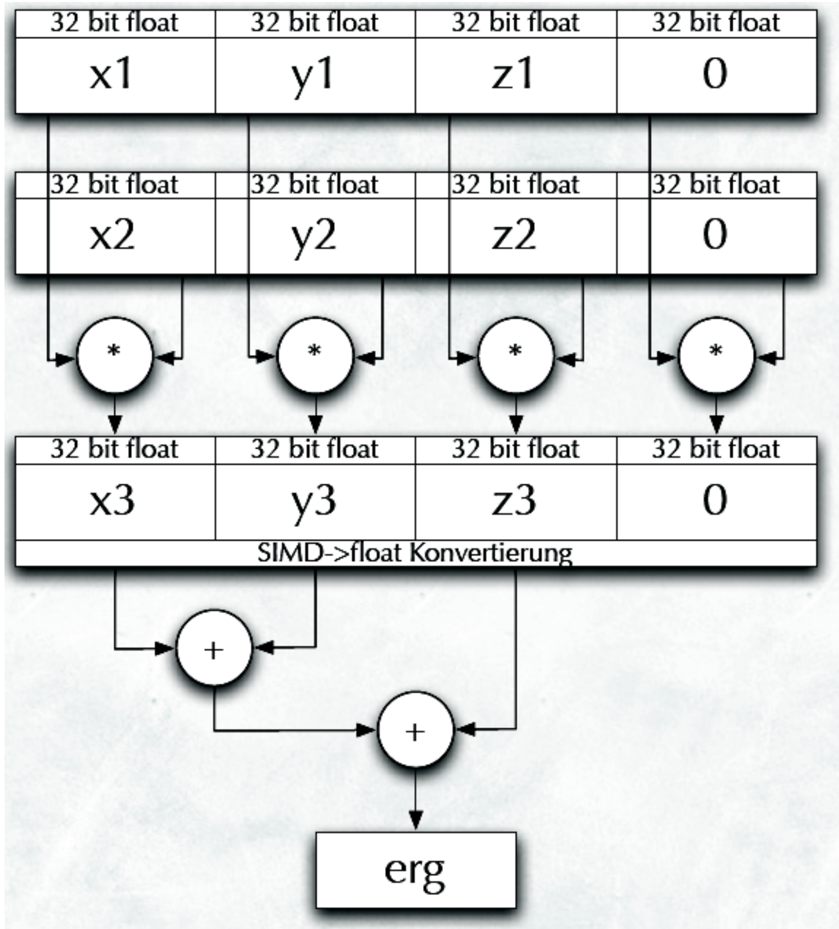
- Yet another type of parallelization: *data parallelism*
  - SIMD (*single instruction multiple data*) parallelism
  - All registers (Float/Int) of a CPU are present in 4-fold
    → registers = 4-vectors
  - An operation can be simultaneously applied on all 4 components
  - I.e.: all computer operations are equally time-consuming, regardless of whether on a single float or 4-vector

- Typical SIMD instruction set (AltiVec, SSSE):
  - All Float/Int operations (Add., Mult., Comp., Round., Load/Store, ...) work component-wise on a pair of vectors (*intra-element operations*)
  - Inter-element operations (permute, pack/unpack, merge, splat, ...)
  - "Horizontal" operations = reductions (horizontal subtract, add, ...)
  - More complex operations: dot product (SSE4)

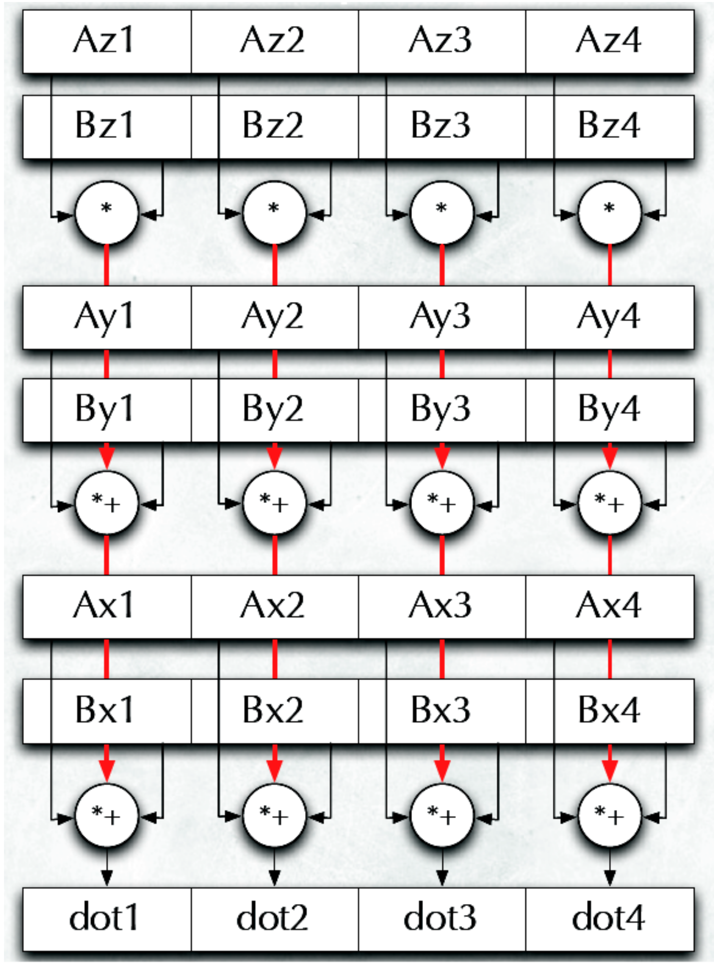


Example of a 3D Scalar Product

1 Scalar Product

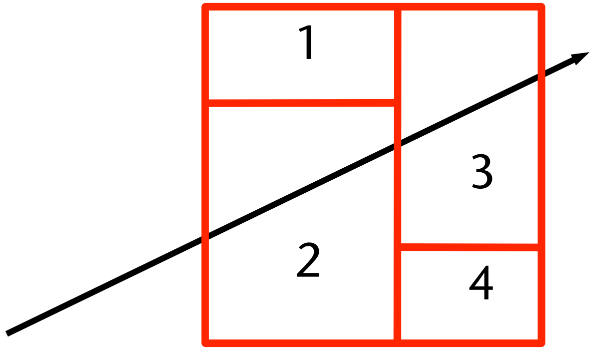


4 Scalar Products

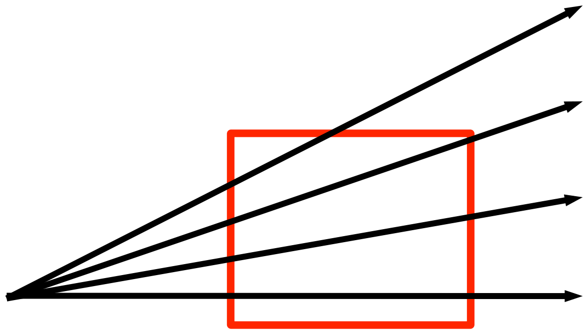


Application to *kd*-tree Traversal

1. Variant: 1 Ray, 4 Objects
   - Problem: Data "objects" must be of the same type
   - Control flow must be the same



2. Variant: 4 Rays (*Ray Packet*), 1 Object
   - Data "objects" are all the same
   - Enough rays are present
   - In order for the control flow to be the same, the rays have to be located as closely as possible to one another



SIMD-Algorithm for Intersection Test Ray-Packet & Box

- Reminder: cut the ray successively against the slabs

```
// A/B = linke/rechte Seite der Bbox
// d = Richtungsvektor, O = Aufpunkt des Strahls
// d' = 1 / d
// alle Operationen, auch min/max, sind komponentenweise!
```

$$t_{\text{min}} = -\infty$$

$$t_{\text{max}} = \infty$$

```
loop a = x, y, z:
```

$$t_1 = (\bar{A}_a \ominus O_a) \odot d'_a$$

$$t_2 = (\bar{B}_a \ominus O_a) \odot d'_a$$

$$t_{\text{min}} = \max(\ \min(t_1, t_2), t_{\text{min}}\ )$$

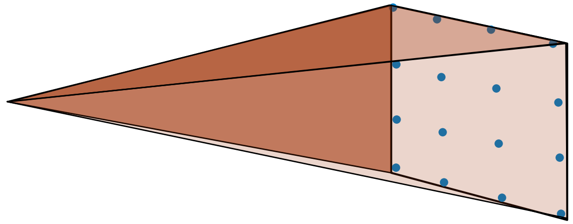
$$t_{\text{max}} = \min(\ \max(t_1, t_2), t_{\text{max}}\ )$$

```
return ! all_ge(t_min, t_max) && all_le(t_max, 0)
```

Returns 1, if all 4 components of t_{min} are larger than the respective components in t_{max}

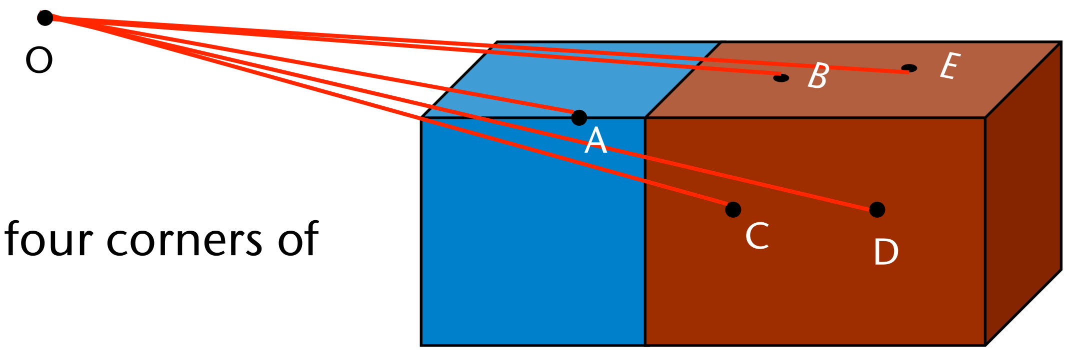
Frustum Tracing in the *kd*-Tree [2005]

- Goal: more than only 4 rays at a time
- Trace the whole ray bundle through the *kd*-tree
- Idea: represent ray bundles as frustum



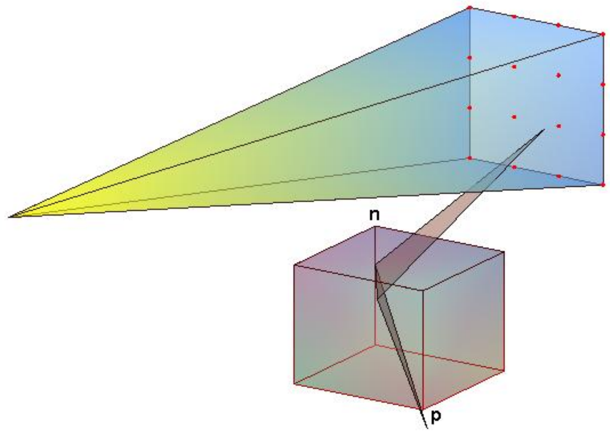
- Up until now: during traversal, a decision was made for only one ray e.g.: "only the left subtree" / "only the right subtree"
- Whith packet/frustum tracing: make an "OR" decision for all rays
  - E.g.: if 1 ray meets the left subtree → trace the entire packet through the left subtree; ditto for the right subtree

- First (problematic) idea:, check only the corner rays
- Counterexample:

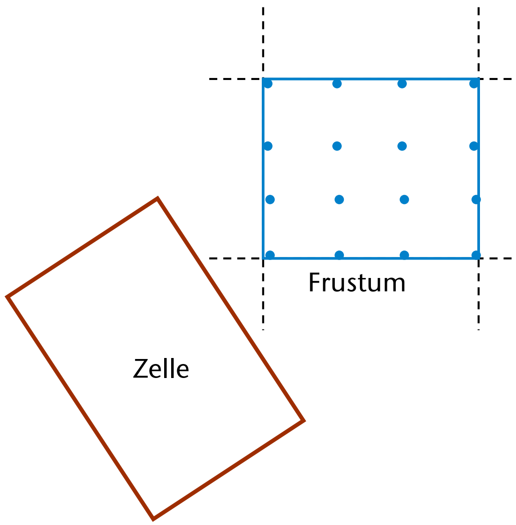


  - Rays B, C, D, E are the four corners of the ray bundle
  - Ray A is located in the same plane as B and C
  - All 4 corner rays intersect only the right cell; but ray A intersects the left!

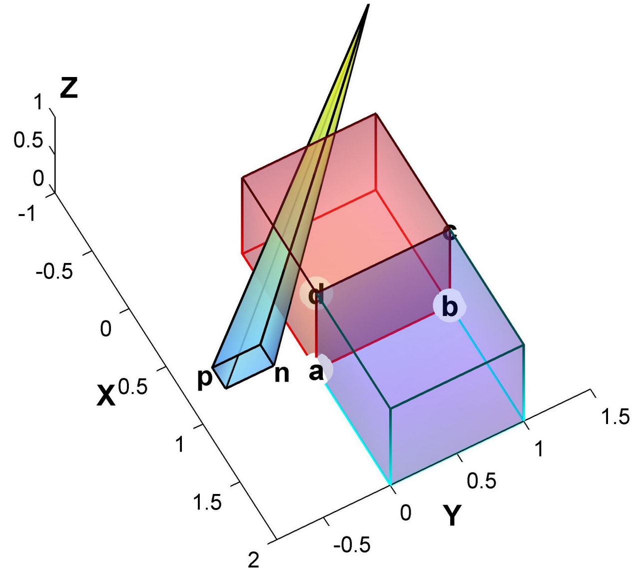
- Better idea:
  - Use the technique of view frustum culling
  - Test: box (= *k*d-tree cell) intersects frustum? (frustum = BV of the ray packet)
  - Possible algorithm: just like with view frustum culling [Möller, see VR lecture]

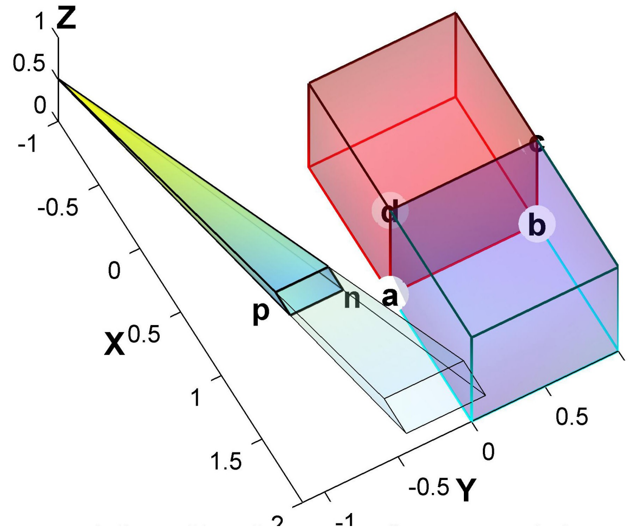


- Problems:
  - Frustum here is long & small → many "false positives"
  - We're doing too much work:
    - We already know that the frustum intersects the father cell!

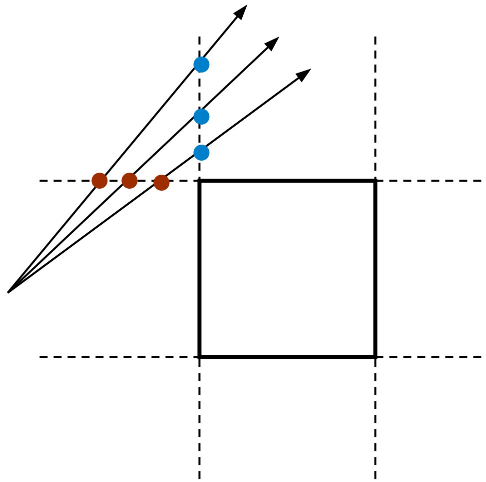


- Idea: test frustum against the splitting plane ("*inverse frustum culling*")
- Example:
  - $\mathbf{d}^i$ = direction of the rays
  - $\forall i: \ \mathbf{d}_x^i > 0$
  - Frustum intersects the father
  - Let the splitting plane be x=1
  - Let the y-coord. of all of the intersection points of all rays < y-coord. of the cell (*)
  - Case differentiation:
    - $\forall i: \ \mathbf{d}_y^i < 0$ → only the red child cell
    - $\forall i: \ \mathbf{d}_y^i > 0$ → only the blue child cell
- Note: here the four corner rays are really sufficient!





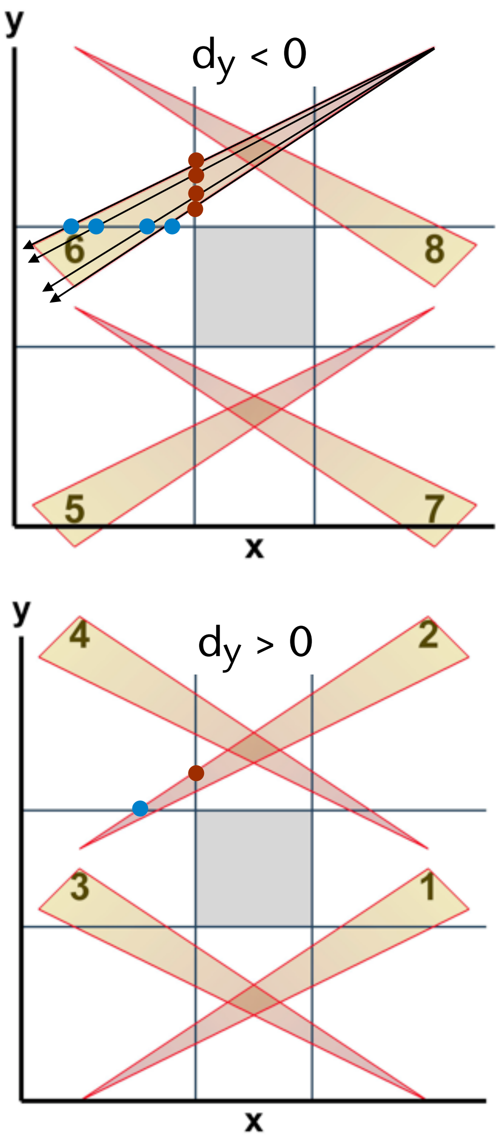
- Problem: there are still *"false positives"*
- Goal: a more exact box-frustum test that is still suitable for SIMD
- First idea: extend the intersection test box-ray to 4 rays
  - Reminder: test the ray against the series of slabs
    - We obtain one "*t entry*" and one "*t exit*" per ray
- Problem: could lead to *"false negatives"*
  - Example: see the example three slides earlier
  - Here "false negative" = test says "frustum is not intersecting," but it actually is!



- Idea: project frustum onto xy plane und test there
  - One does not have to identify the 2 border rays in 2D; simply execute the calculations with all 4 (projected) corner rays (is just as expensive, since SIMD)
  - Let y_i^{entry} be the y-coord. of the "enter" intersection point of the rays (in 2D) with the planes of the boundary sides y=const of the AABB
  - Ditto y_i^{exit}
  - Ditto for x $\rightarrow x_i^{\text{entry}}, x_i^{\text{exit}}$
  - There are 8 cases, 2 tests are sufficient:

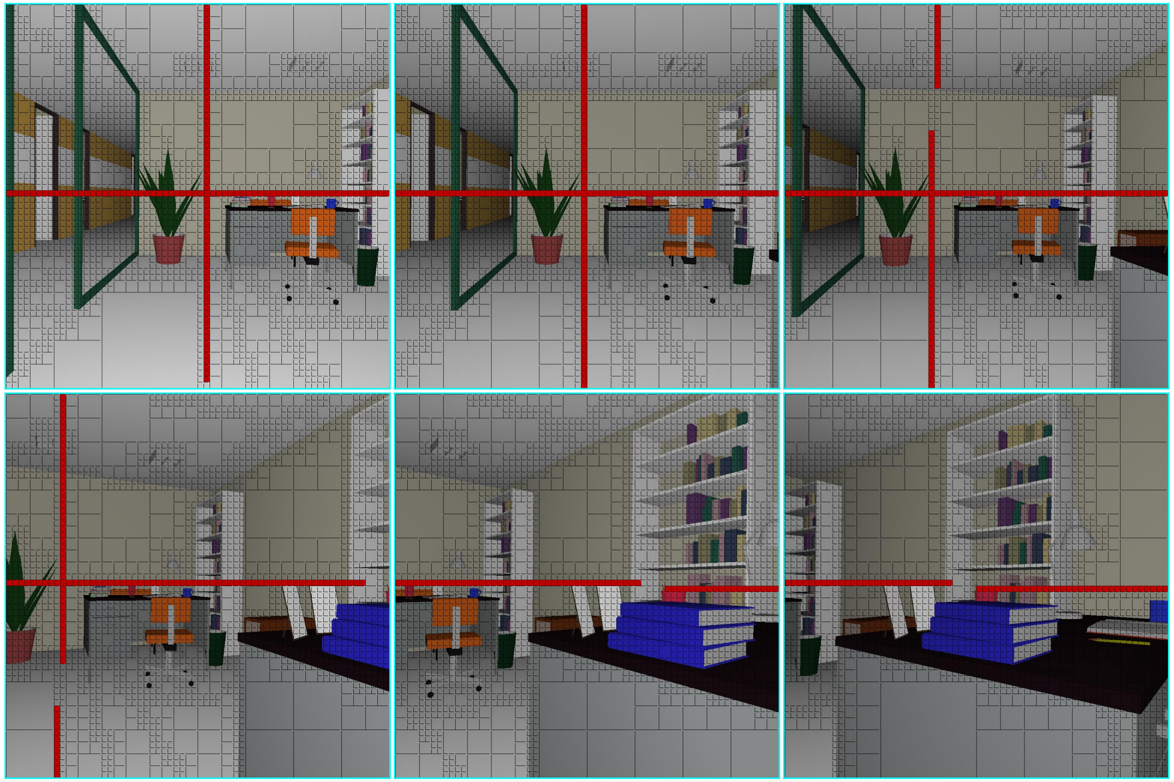
$$\min\{y_i^{\text{entry}}\} > \max\{x_i^{\text{exit}}\} \ \vee \qquad (1,3,6,8)$$

$$\min\{x_i^{\text{entry}}\} > \max\{y_i^{\text{exit}}\} \qquad (2,4,5,7)$$



Adaptive Tile / Frustum Splitting

- Start with "large" ray bundles (= frusta) as "primary rays"
- Try to traverse the *k*d-tree
- Split the frustum, if the conditions (*) for the frustum-cell test are no longer given



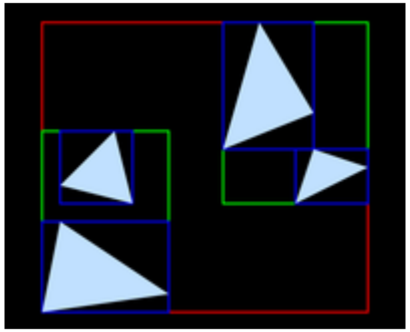
(Courtesy Reshetov et al.)

Dynamic Scenes

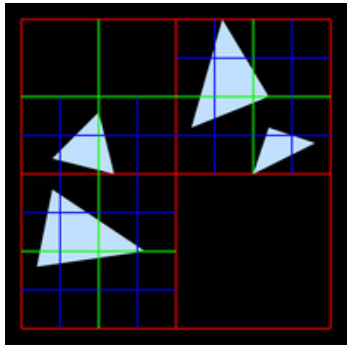
- Problem:
  - All vertices move (animation/simulation)
  - *K*d-tree/grid/BVH become invalid (virtually all acceleration data structures)
- Naïve idea:
  - Build acceleration data structures anew in every frame (after the new positions of the vertices have been calculated)
  - One can do this with a grid, but it's too expensive for all other acceleration data structures

What is so special about grids?

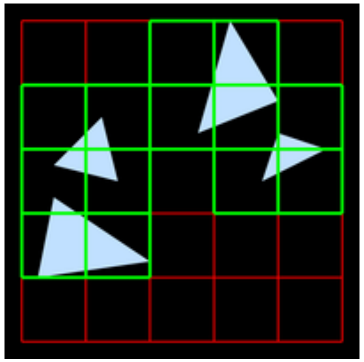
- Since the 70s: many *acceleration data structures*



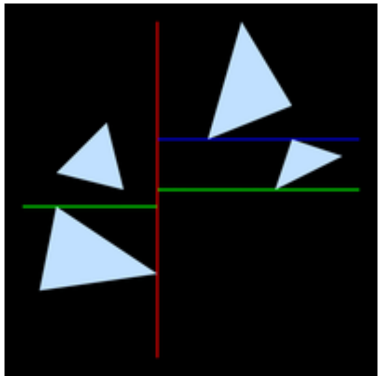
BVH



Octree



Grid

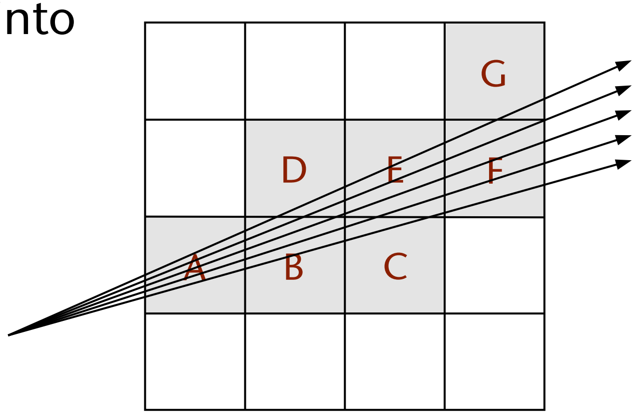


Kd-tree

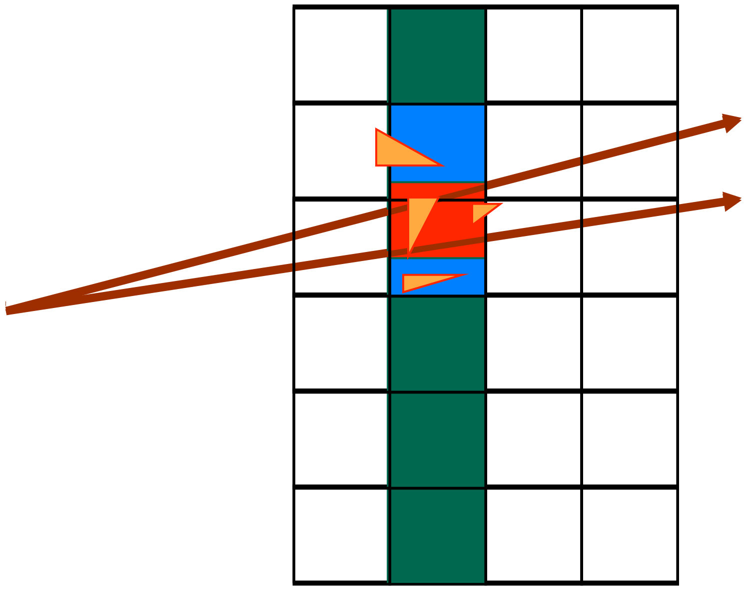
- Of all of the above, only the grid is non-hierarchical!

Coherent Grid Traversal [2006]

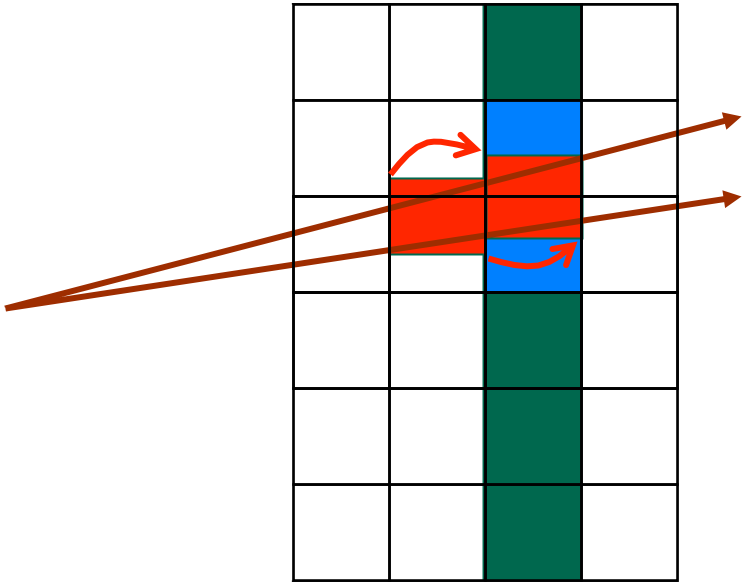
- Goal: accelerate ray packets by using a grid (with SIMD)
- Problem: traversal is incompatible with packet tracing
  - In which order does one visit the cells? ABCD or ABDC?
  - Incremental traversal algorithms (midpoint, 3D DDA) are no longer SIMD-capable when rays diverge
    - Decision variables for different rays in the same packet differ!
  - Splitting up packets degenerates quickly into single-ray traversals
- Idea:
  - Packets do not work with a grid…
  - … but frusta do.



- Determine the coordinate axes to which the given ray packet is "the most perpendicular;" planes perpendicular to that are called E_i
- Determine the upper/lower/left/right frustum plane for the ray packet
  - The upper frustum plane should be chosen such that an intersection with an E_i plane produces a horizontal line
  - Determine the other frustum planes analogously
- Traverse the grid with the frustum layer by layer
  - Determine an "overlap box" between frustum und grid level
  - Round up to integer (i.e., grid) indices → covered cells
  - Intersect rays with all triangles in covered cells



- One can maintain the overlap box incrementally, from layer to layer
  - Trivial, since the bounding planes of the frustum are known and parallel to the axes
  - Per step, a total of only 4 additions are needed (= 1 SIMD operation)
    - Independent of the number of rays in the frustum!



- Combine that with SIMD frustum culling in order to remove triangles that don't intersect the frustum

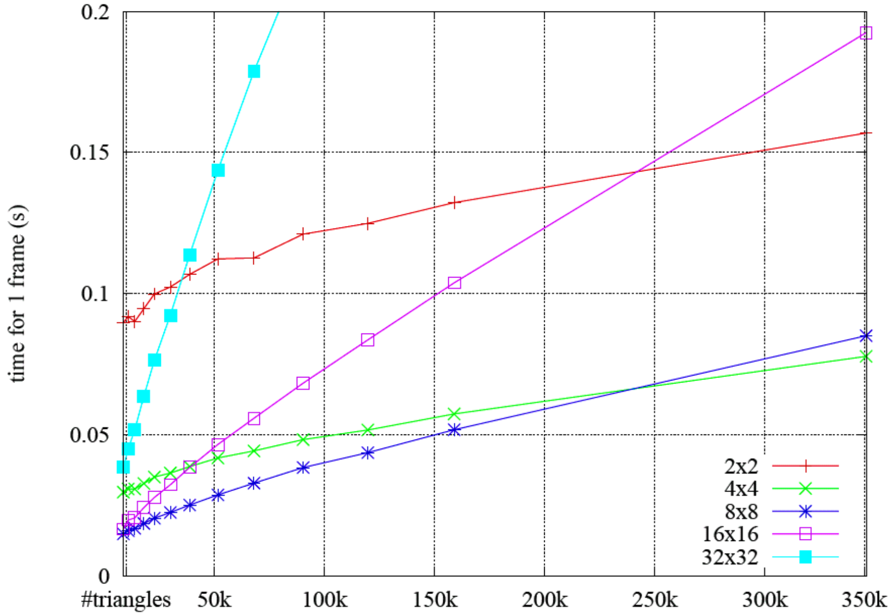
Your Thesis Topic ? ...

Notes

- Expensive setup phase
  - Calculate frustum, setup of incremental algorithms
- Very cheap update step from layer to layer
- Very well suited to dynamic scenes:
  - Rebuilding = few milliseconds for ~100.000 triangles (1 Proc)
  - Rebuilding is easy to parallelize: 10 MTris in ~150 ms (16 Opterons)
- Just as few intersection calculations (ray-obj.) as with *k*d-tree
- Small con: one *must* use mailboxes (MB)
  - Grid w/o FC & MB : 14 M ray-tri intersections
  - Grid with FC & MB : .9 M ray-tri intersections (14x less)
  - Kd-tree : .85M ray-tri intersections (5% less than with grid)
- All in all: grid is only ~2x slower than BVH and *k*d-tree, but for that we get dynamic scenes!

Towards Choosing the Optimal Packet Size

- The costs of the traversal steps are mostly independent of the number of rays →
  - Larger packets = more potential for amortization (pro)
- More rays/packets = larger frustum →
  - More visited cells, more triangles that must be tested against all rays in the packet (con)
- *"Sweet spot"*:
  - The best is 4x4 (green) or 8x8 (blue)

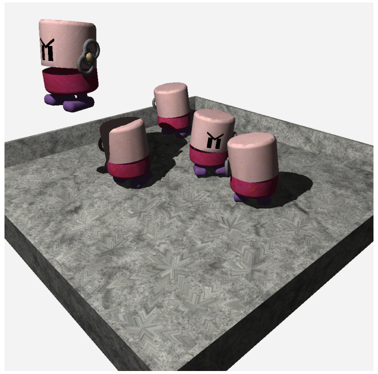


Results

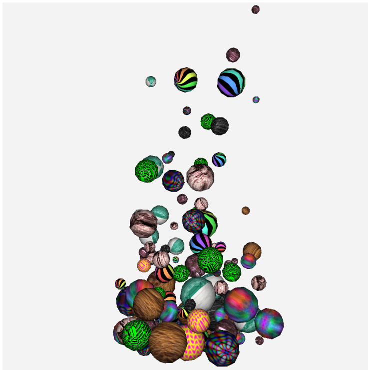
- Dual-Xeon 3.2GHz, 1024x1024, without shading, pure animation

"Hand"
16K triangles
34.5/15.3 fps

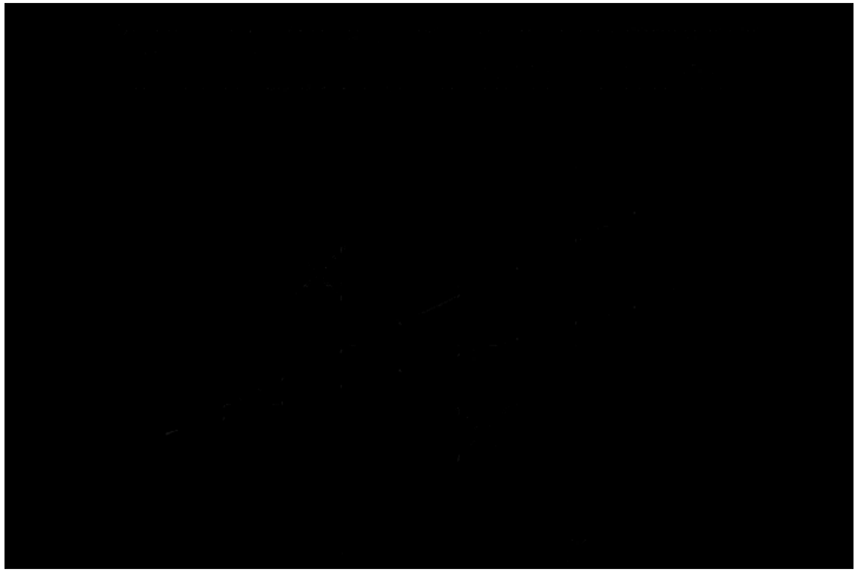
"Runner"
78K triangles
15.8/7.8 fps



"Toys"
11K triangles
29.3/10.2 fps



"Marbles"
8.8K triangles
57.1/26.2 fps



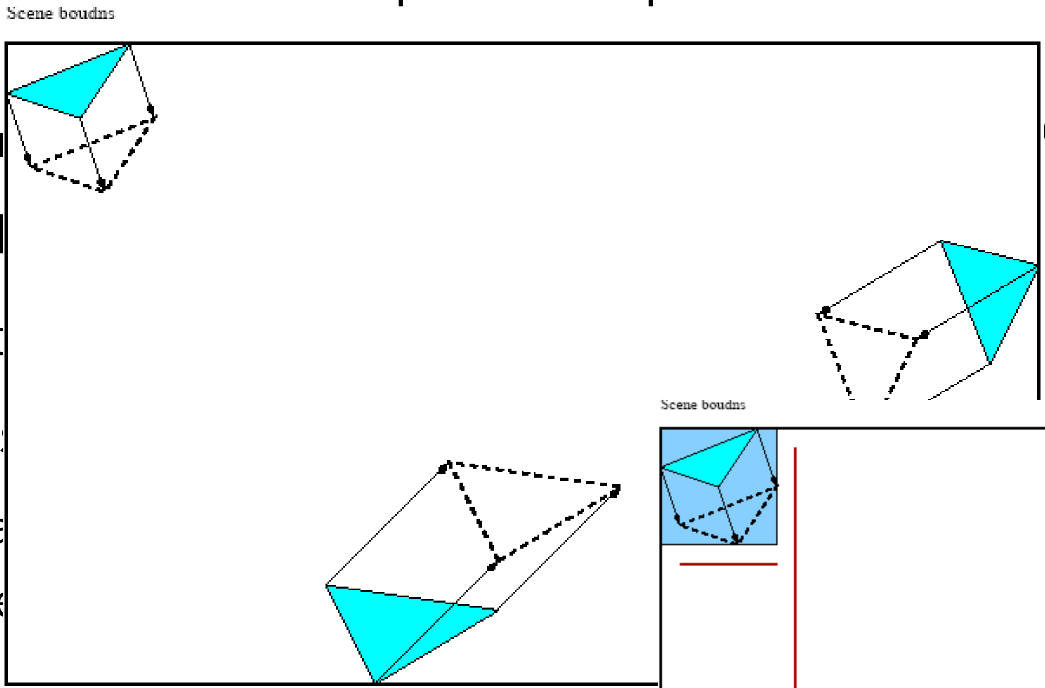
X/Y fps:
X=raycast only
Y=raycast+shade+texture+shadows

- Video (fairy)
  - 174k tris, 1024x1024 Pixels, 16-core Opteron (180 GFLOPs)
    - CELL = 256 GFLOPs
    - ATI X1900 ~ 1000 GFLOPs
  - 3.4 fps (raycast only)
  - 1.2 fps (raycast + shade + texture + shadows)

Space-Time *kd*-Trees [2006]

- Idea: if we know all of the positions of a triangle during the course of the animation ...
  - ... then we can enclose the space of the positions of the triangle in only one BV

  →Every trian

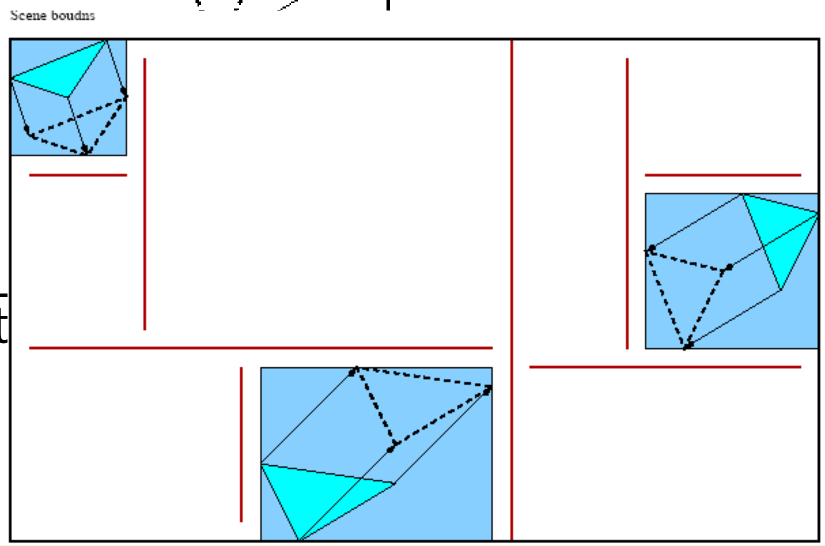


V)

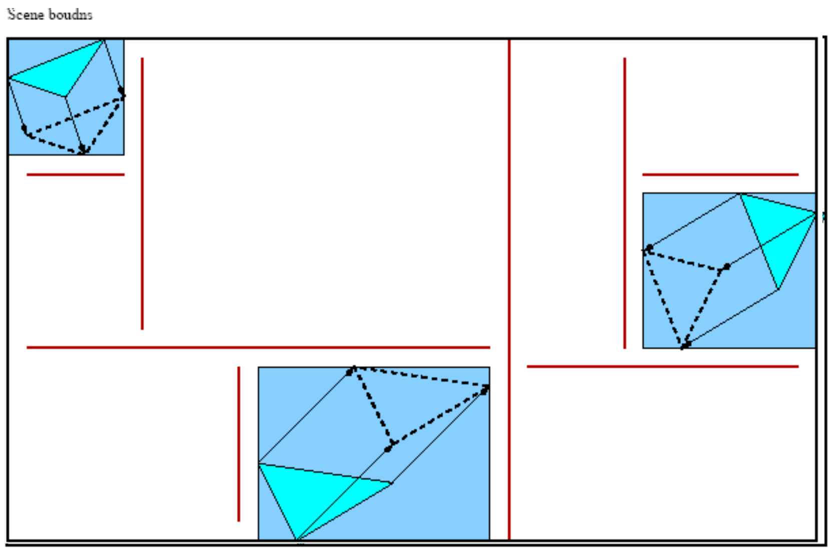
- Then build

  boxes:

  - Is correct
  - As alway
    - Beforeha
      the curre
- But: it is only performant if the space-t



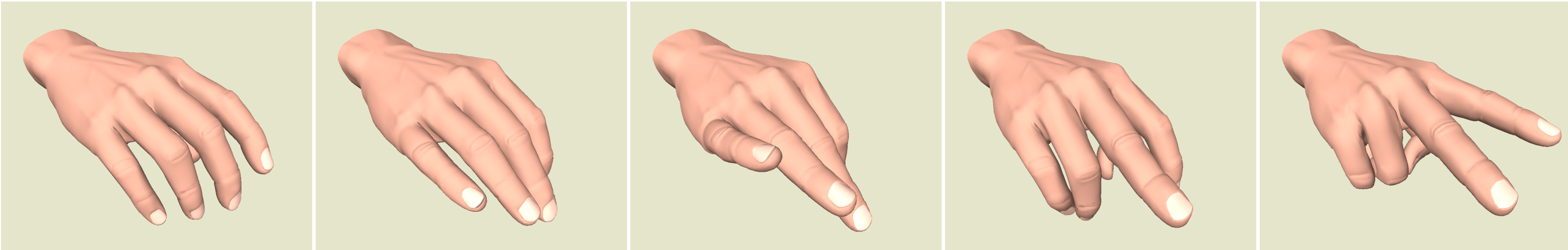
- Idea: if we know all of the positions of a triangle during the course of the animation ...
  - ... then we can enclose the space of the positions of the triangle in only one BV
- Every triangle receives a space-time box (or space-time BV)
- Then build only one kd-tree over all of the space-time BVs:
  - Is correct throughout the entire time span
  - As always, do a ray test at the leaves
    - Beforehand, one needs only to calculate the position of the vertices at the current point in time



- But: this is only efficient if the space-time boxes are small!

Idea of Motion Decomposition

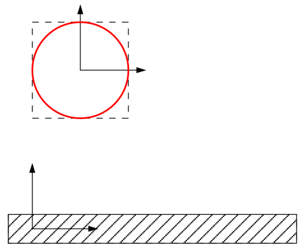
- Observation:
  - Many "real" animations are "mostly" hierarchical
  - … plus a small residual deformation
- Example:



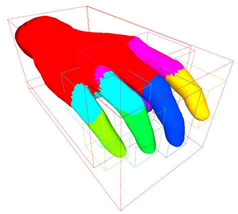
- Consider only special deformations:
  - Base Mesh Deformation (constant connectivity)
  - All frames in the animations are known ahead of time
  - Locally coherent movement

Overview of the Method

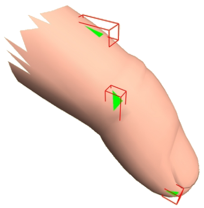
- Motion decomposition
  - Affine transformation + *residual motion*



- Clustering
  - So that all vertices within a cluster move locally coherently

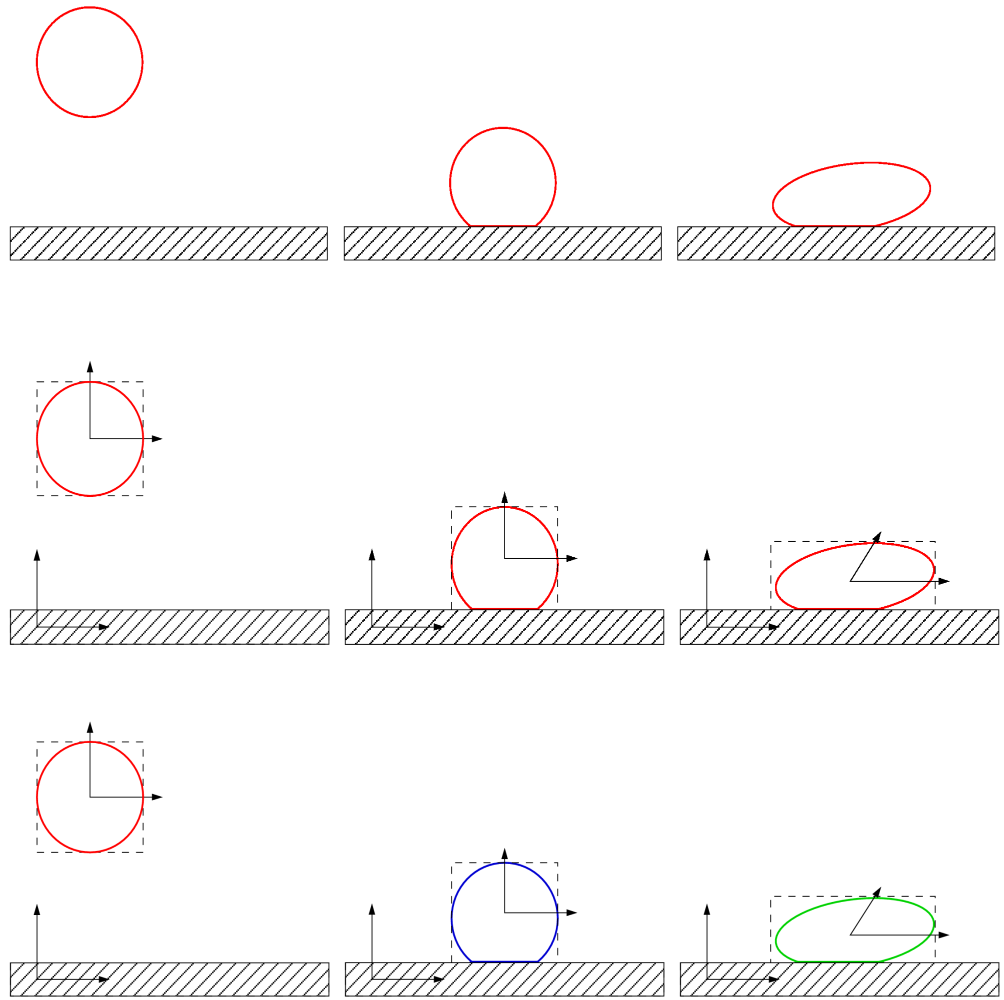


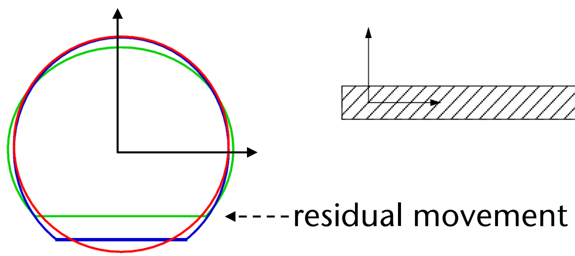
- Space-time *k*d-tree
  - For the treatment of residual movement



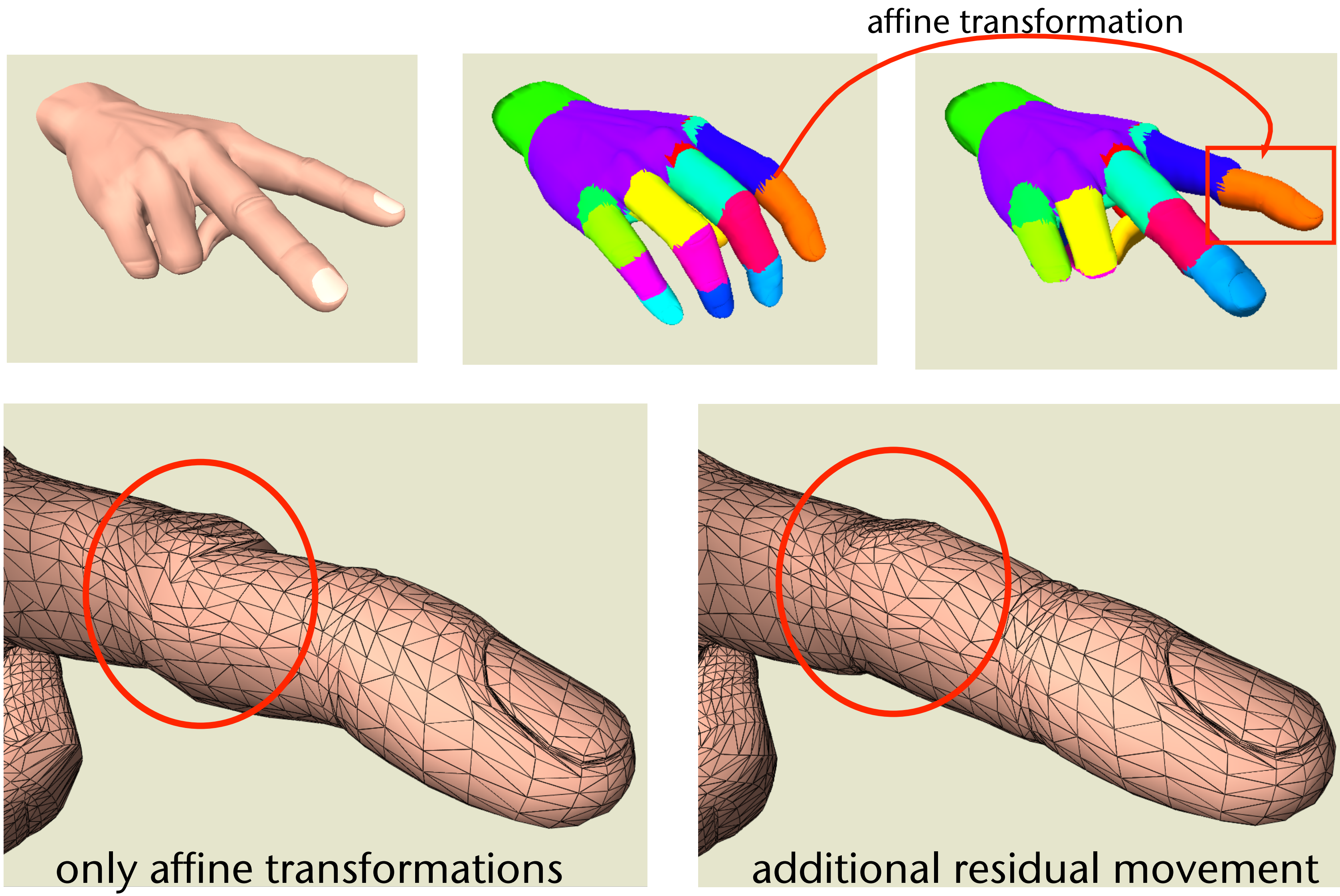
Movement Decomposition

- Dynamic scenes:
  Ball thrown onto the floor
- Affine Transformation
  - With shearing for the "squash" effect
- Extraction of the residual movement



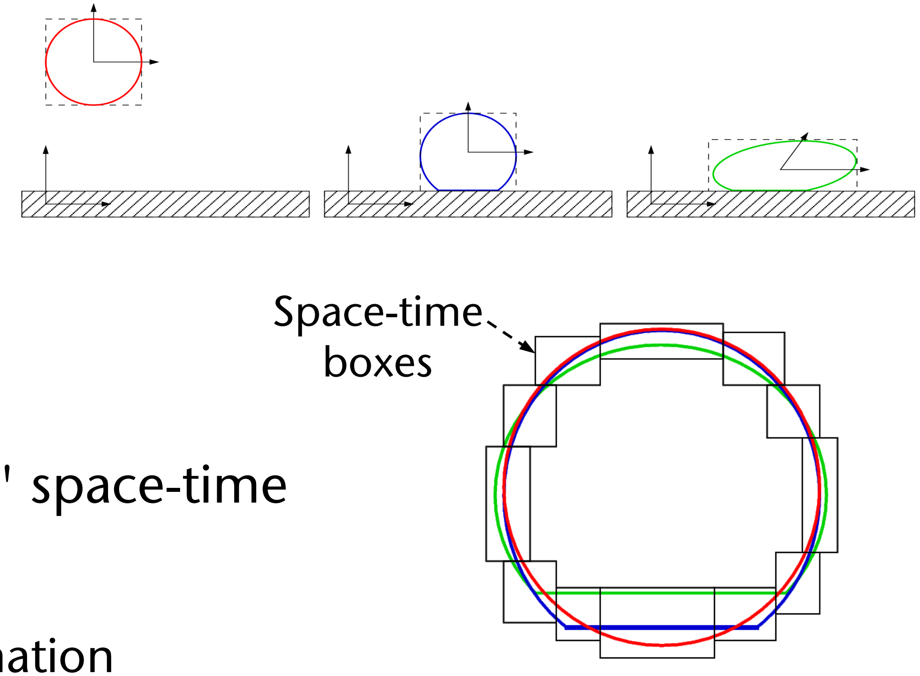


Example

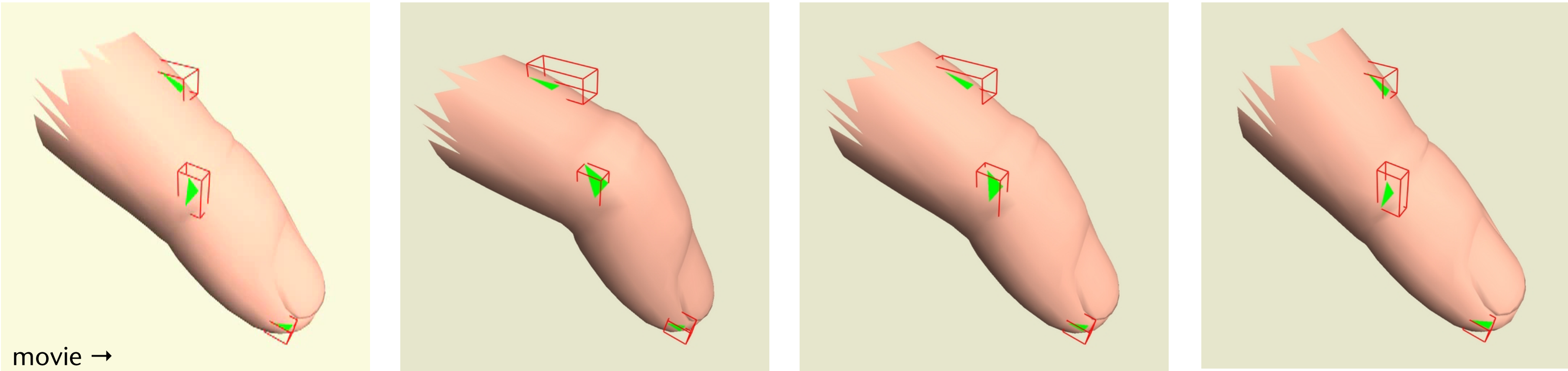

affine transformation
only affine transformations
additional residual movement

Space-time kd-Tree

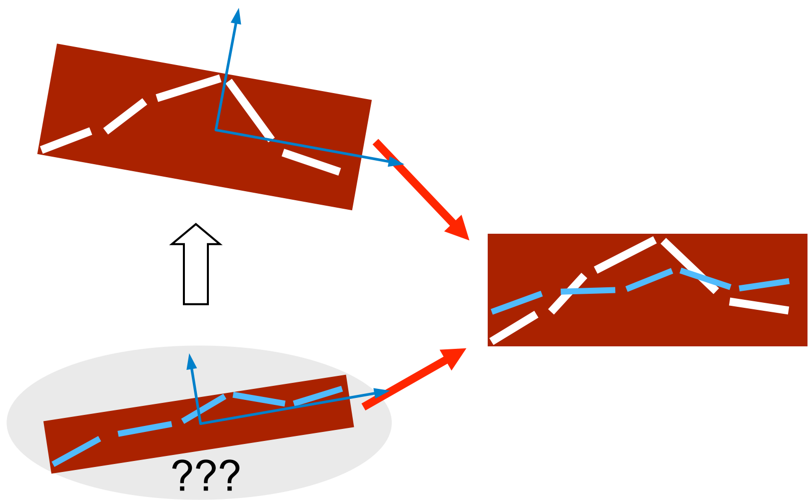
- Enclose the polygons in the common coordinate system in space-time BVs that are "large enough"



- *Build k*d-tree over the triangles' space-time boxes
  - thereby valid for the entire animation



- Assumption: the model is already divided into subsets, whereby all triangles in the subset move "similarly"
  - Now how does one calculate an affine transformation for the entire subset from one point in time t_1 to another t_2?
  - → Compute PCA over vertices at point t_1, PCA at point t_2 → two coordinate systems, affine transformation in between
- How does one group the triangles?
  - → Clustering



- Choose w.l.o.g. the coordinate system at point t_0 (first key frame) as a common coordinate system for all vertices in the same cluster
- Assumption: a vertex **v** is member of a cluster that moves from point t_0 to t_1 with the affine transformation $M(t_1)$:

$$\mathbf{v}(t_1) = M(t_1) \cdot \mathbf{v}(t_0) + \delta(t_1)$$

whereby $\mathbf{v}(t_0)$ = position at t_0 (= rest position),
and $\delta(t_1)$ = residual movement

- That is:

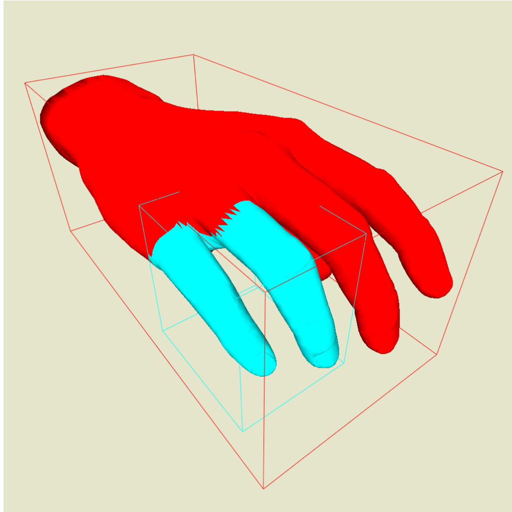
$$M(t_0) = I \ , \quad \delta(t_0) = 0$$

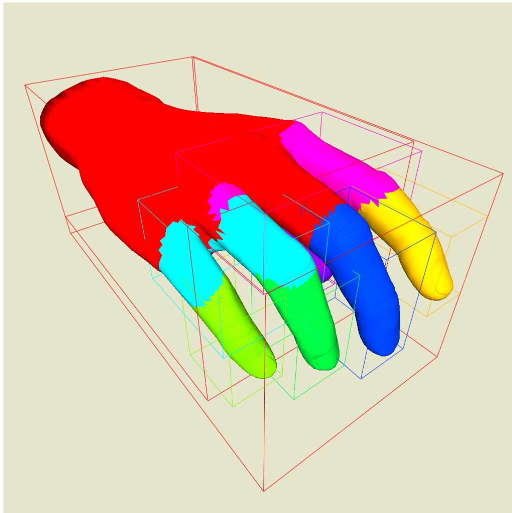
- Transformation into the common coordinate system:

$$\tilde{\mathbf{v}} := M^{-1}(t_1) \cdot \mathbf{v}(t_1) = \mathbf{v}(t_0) + M^{-1} \cdot \delta(t_1)$$

The Clustering Algorithm

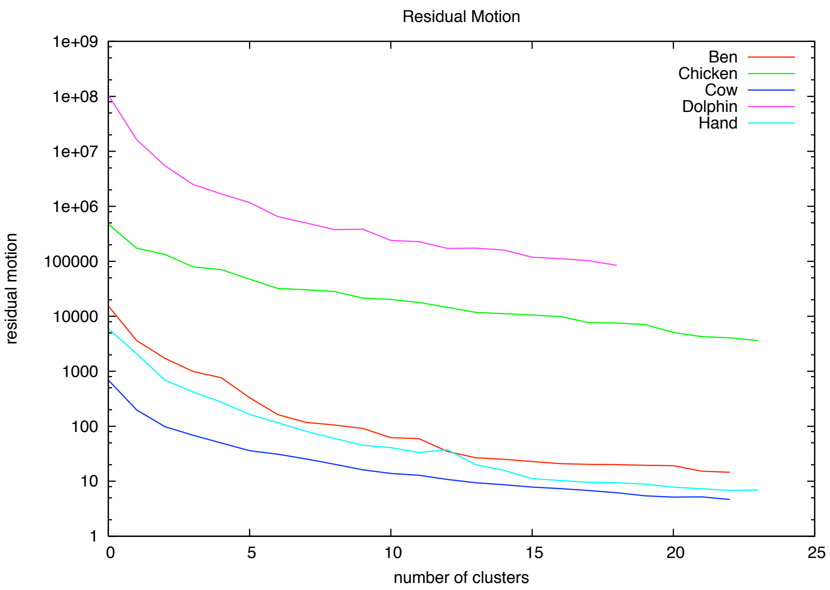
- Goal: efficient ray tracing
  - Thus, try to minimize the size of the space-time boxes
  - Cluster triangles that move "similarly"
  - Trade-off between number of clusters und size of the space-time boxes
- Clustering by the k-means algorithm
  - Represent triangles by their mid-points
  - Use straight-forward Euclidean distance
    - Other distance measures are conceivable
- On determining the number of clusters:
  - Begin with 1 cluster (= all triangles)
  - Find affine transform. for every cluster
  - Insert new cluster
    - Initialize with the triangle with the largest residual movement
  - Until enhancement is below a threshold



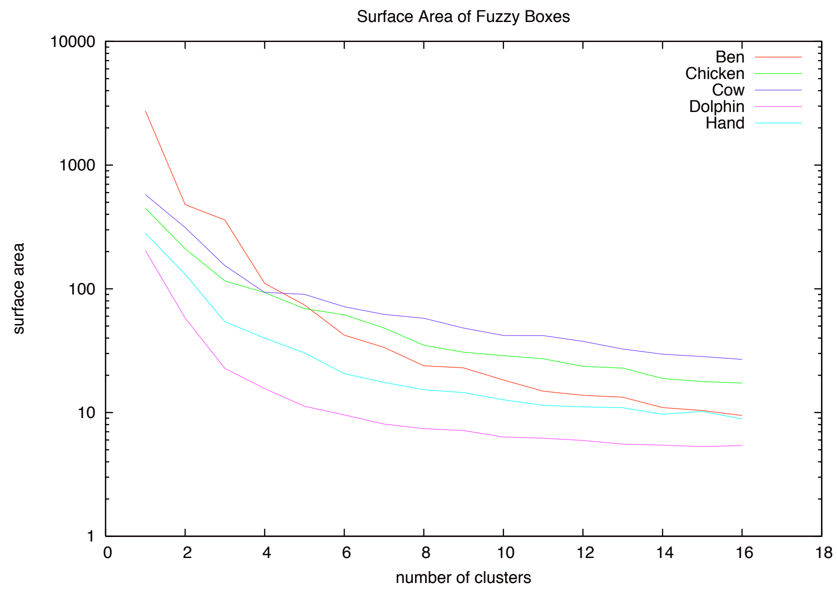


- Termination criterion:

Residual Movement

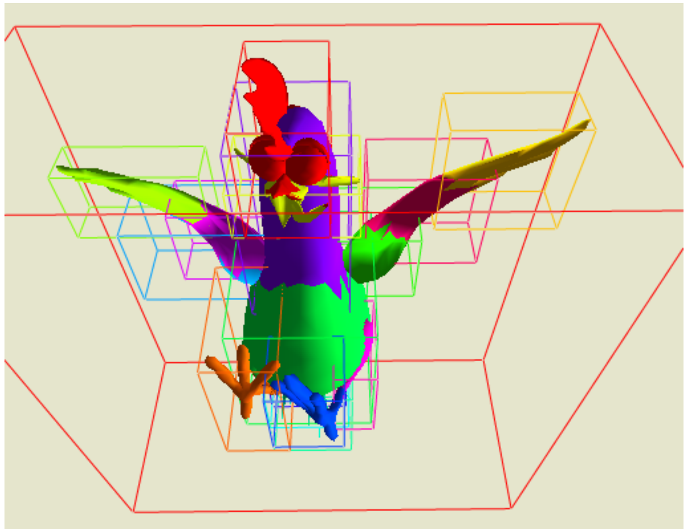


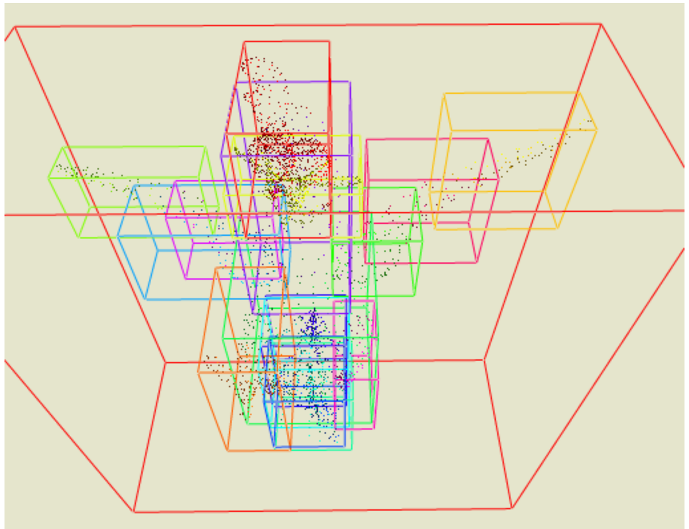
Surfaces



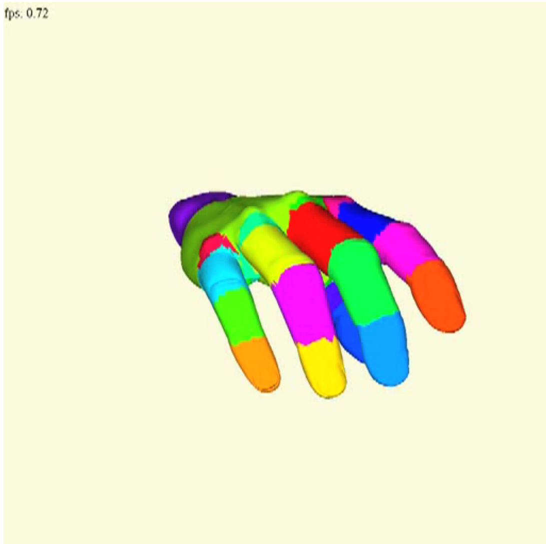
The Two-Level Data Structure

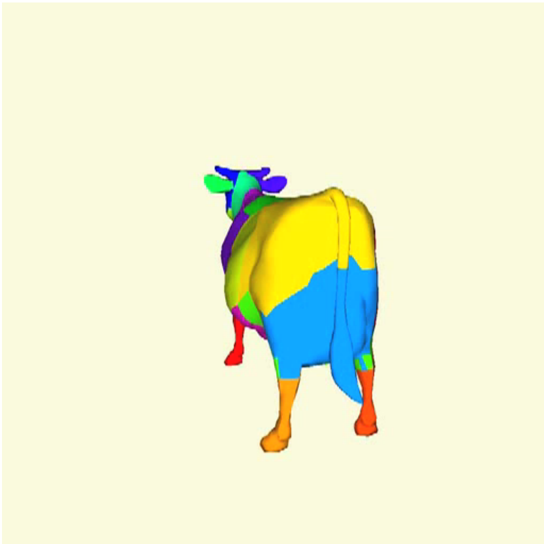
- Test the ray against the root BV of each cluster
- For each cluster that is hit:
  - Transform the ray into the local coordinate system of that cluster
  - Traverse its space-time *k*d-tree

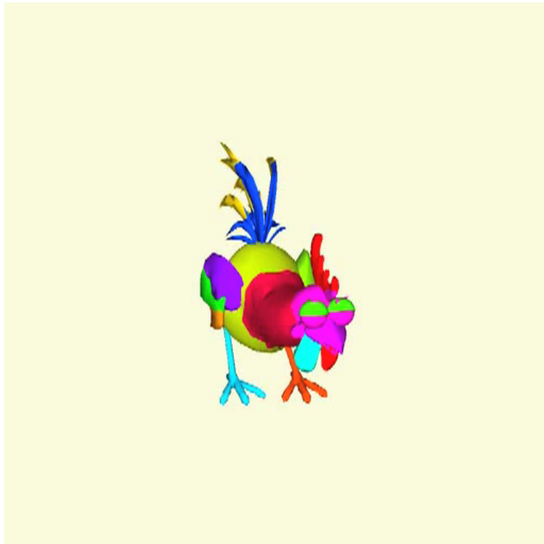




Videos

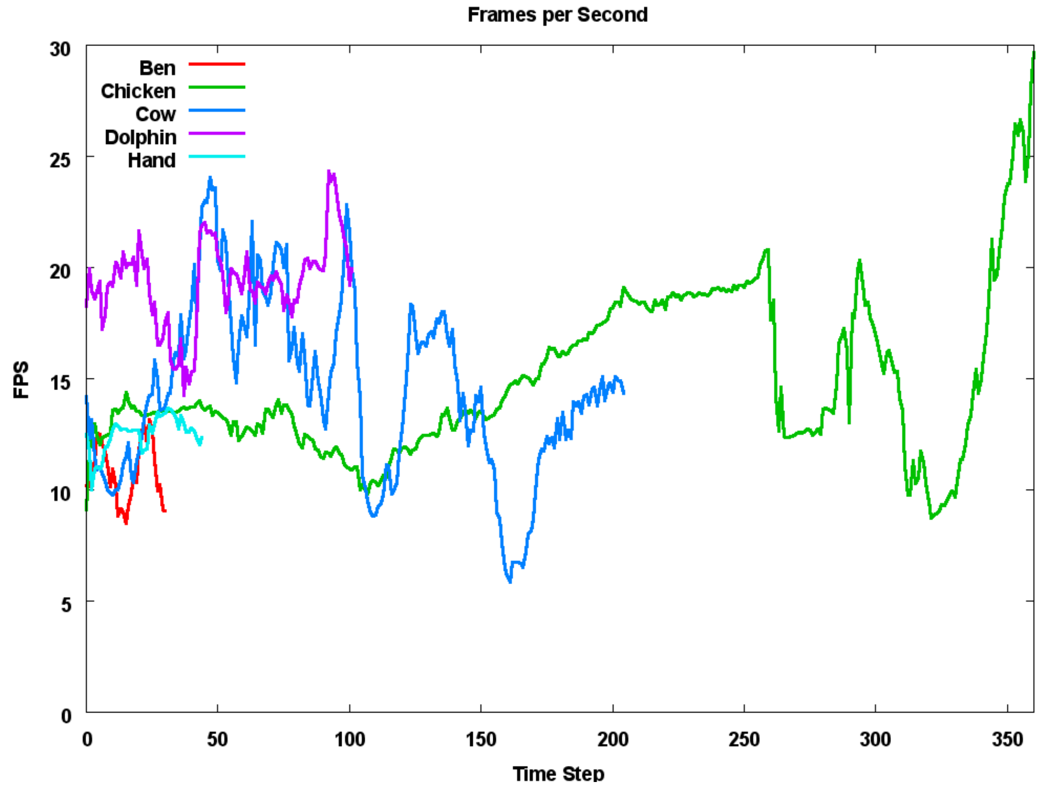






Performance

- One CPU (Opteron 2.8 GHz), 1024×1024 px, including shading(?):



- With texturing, shading, shadows: 2.2 fps
  - Compare to static *k*d-tree: 4.1 fps

Comparison to a Static *k*d-Tree

- In comparison with a new (static) *k*d-tree per frame, which all were constructed before the ray-tracing of the animation
- Traversal steps: factor 1.5–2 more with the space-time *k*d-tree
- Intersection calculations (with triangles): factor 1.2–6 more
- Average frames/sec: factor 1.2–2.6 slower
  - Not including the time it takes for the construction of the static *k*d-trees!
- Memory:
  - Only 1 space-time *k*d-tree
  - Against # frames many static *k*d-trees

Notes

- Fully compatible with other *k*d-tree techniques
  - E.g. frustum tracing
- The authors name their boxes at the leaves "*fuzzy boxes*" and they call the *k*d-tree "*fuzzy kd-tree*" — but this is just nonsense
  - The data structure has nothing to do with the concept of "*fuzziness*" known from fuzzy logic
  - The idea of space-time BV's has been around for quite awhile
- The whole thing only works with so-called "*articulated bodies*"
  - That is, if one builds clusters over the set of triangles such that the movement within the clusters is similar, then the clusters have to be small in terms of volume