Advanced Computer Graphics
Collision Detection

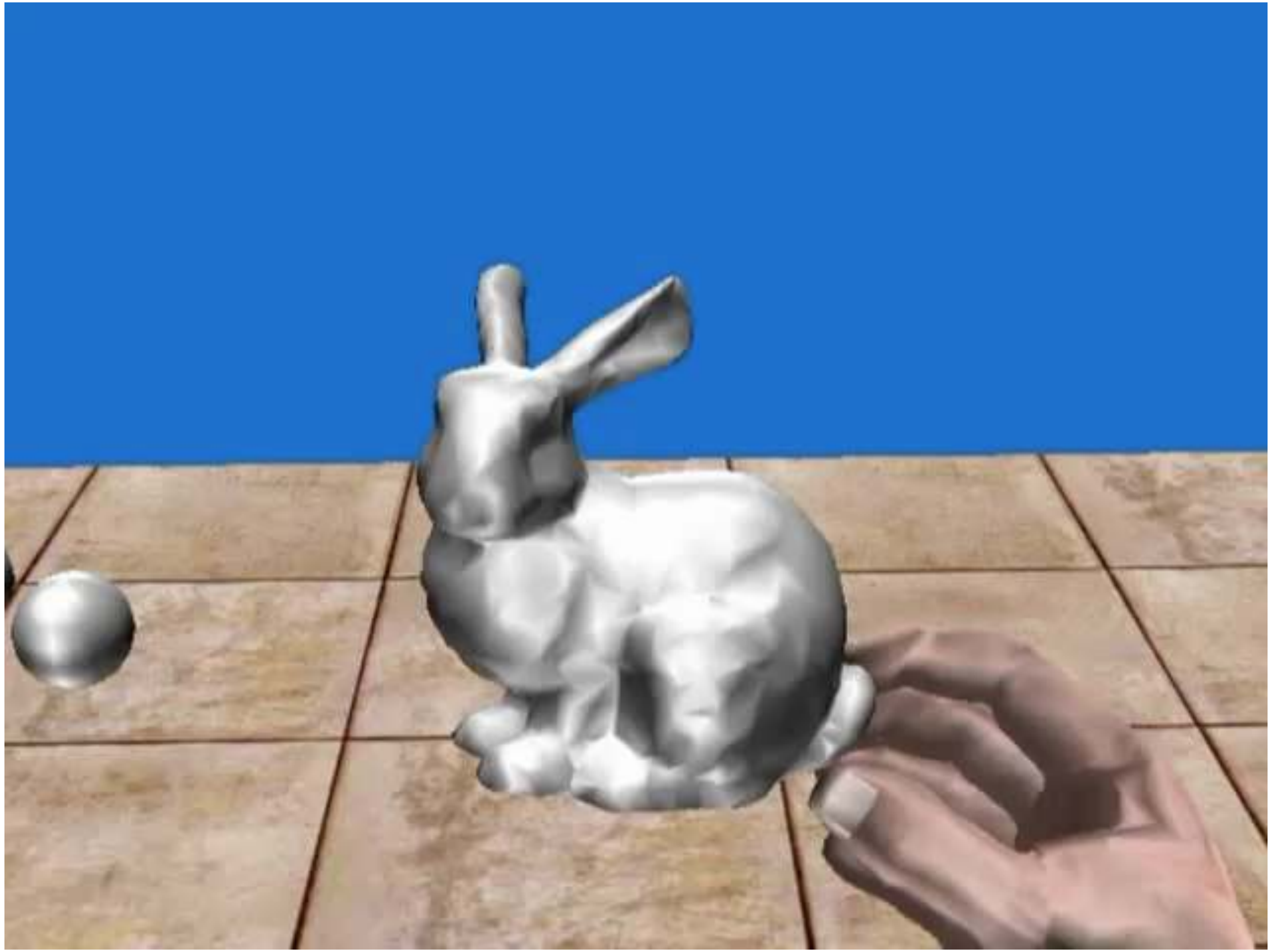
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Examples of Applications

Virtual
Prototyping ,
Digital Twins,
Assembly
Simulation

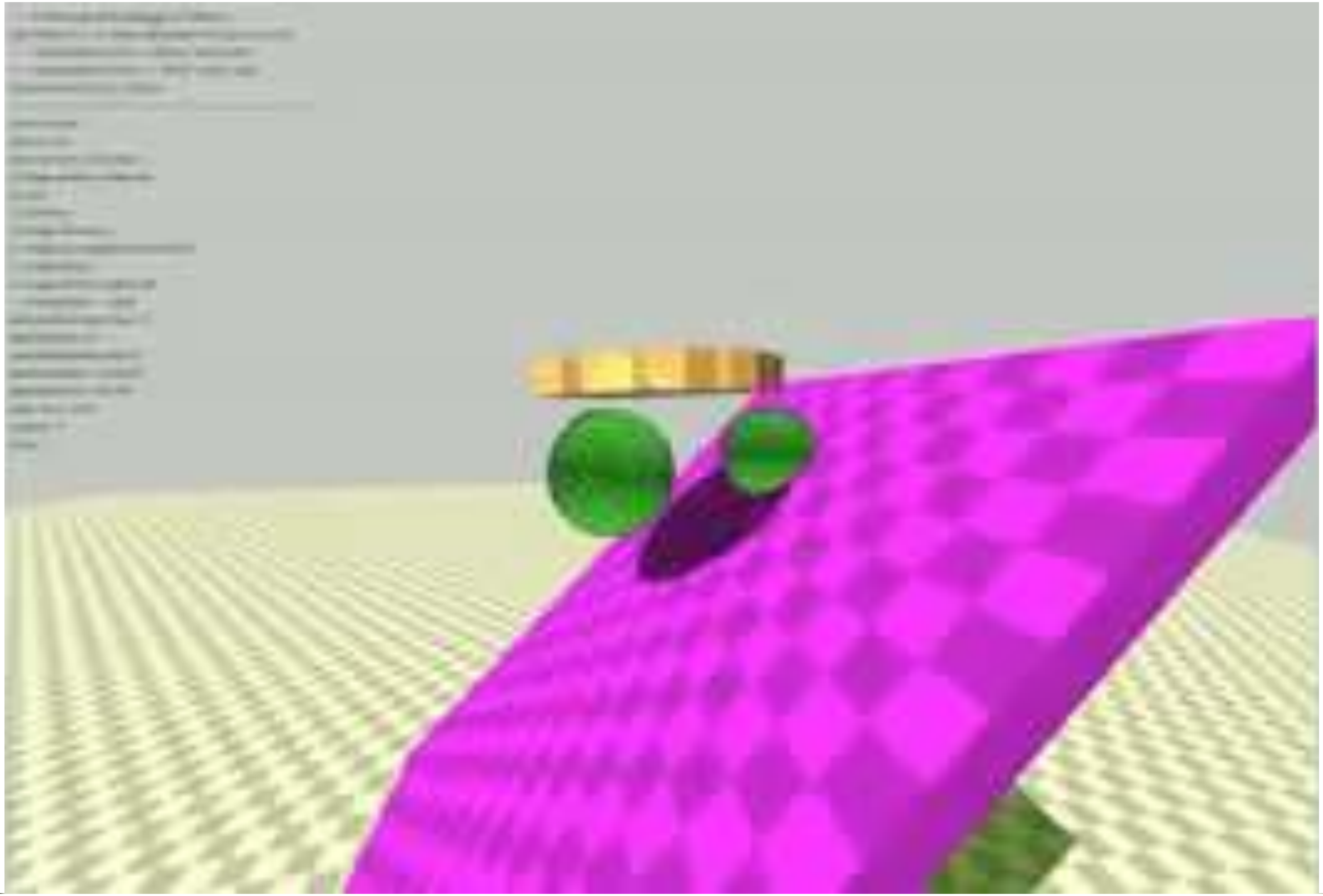
Examples of Applications

Natural User Interaction in Virtual Reality



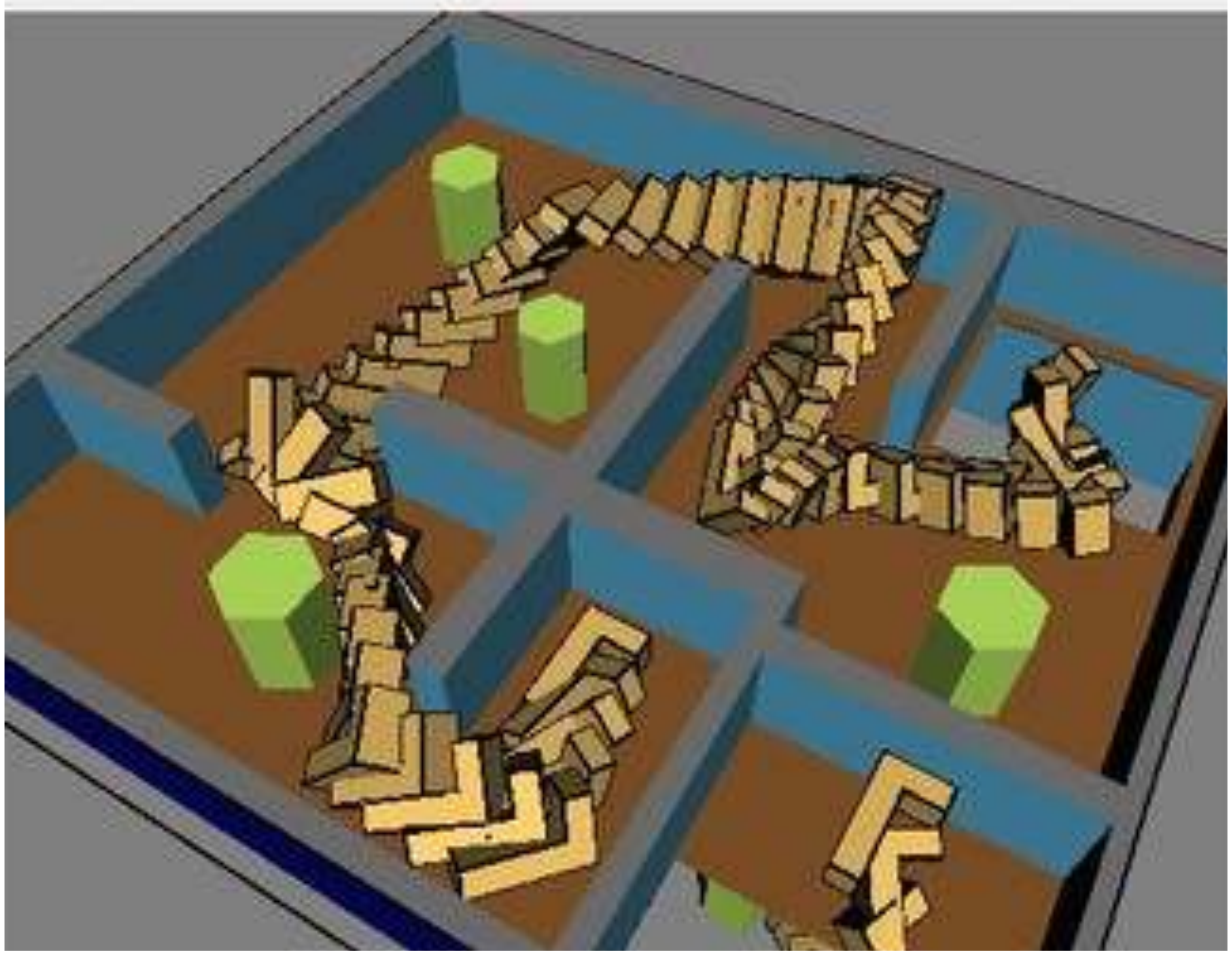
Examples of Applications

Physically-Based Simulation in Games and VR



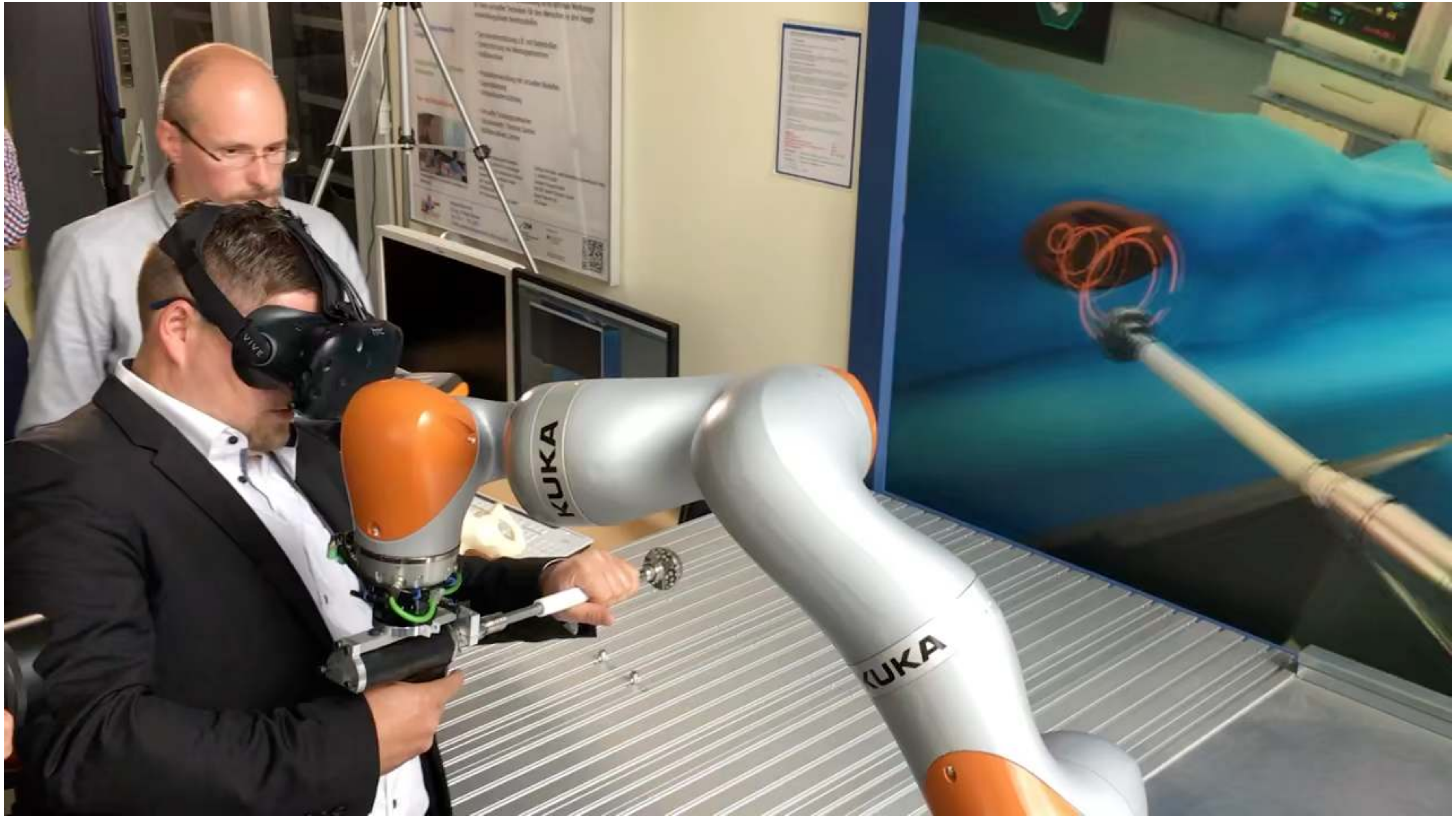
Examples of Applications

Robotics: path planning
(piano mover's problem)



Examples of Applications

Force Feedback for Medical Immersive Training Simulators



Examples of Applications

Force Feedback for Medical Immersive Training Simulators

Collision Detection Within Simulations

- Main loop:

  Move objects

  Check collisions

  Handle collisions (e.g., compute penalty forces)

- Collisions pose two different problems:
  1. Collision detection
  2. Collision handling (e.g., physically-based simulation, or visualization)
- In this chapter: only collision detection

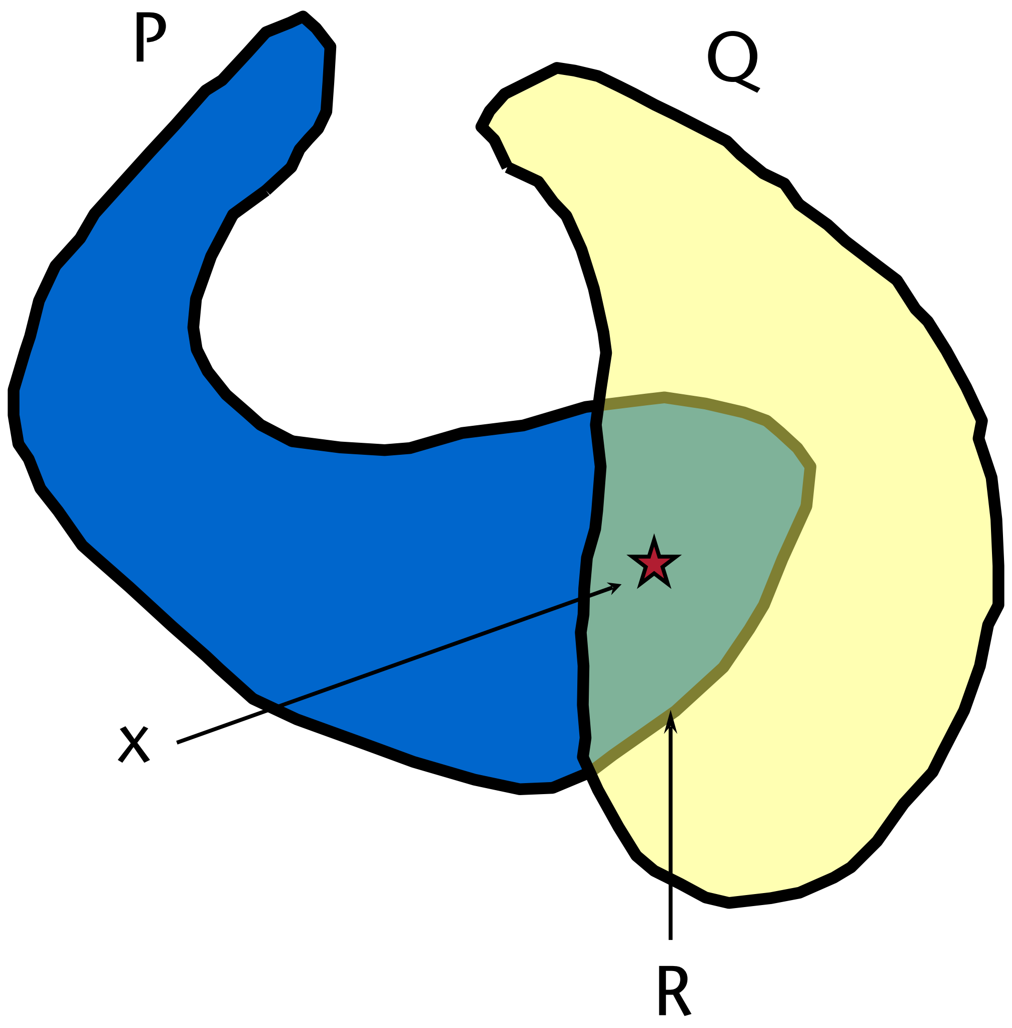
Definitions

- Given polyhedrons $P, Q \subseteq \mathbb{R}^3$
- The detection problem:

  P and Q collide $\Leftrightarrow$

  $P \cap Q \neq \emptyset \Leftrightarrow$

  $\exists x \in \mathbb{R}^3 : x \in P \wedge x \in Q$

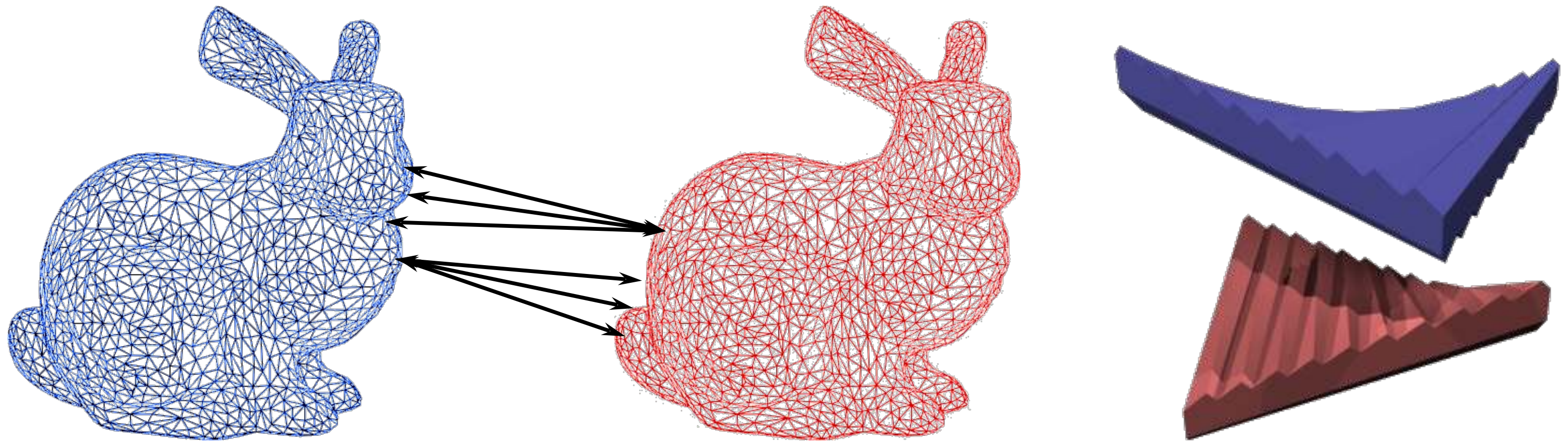


- The construction problem:

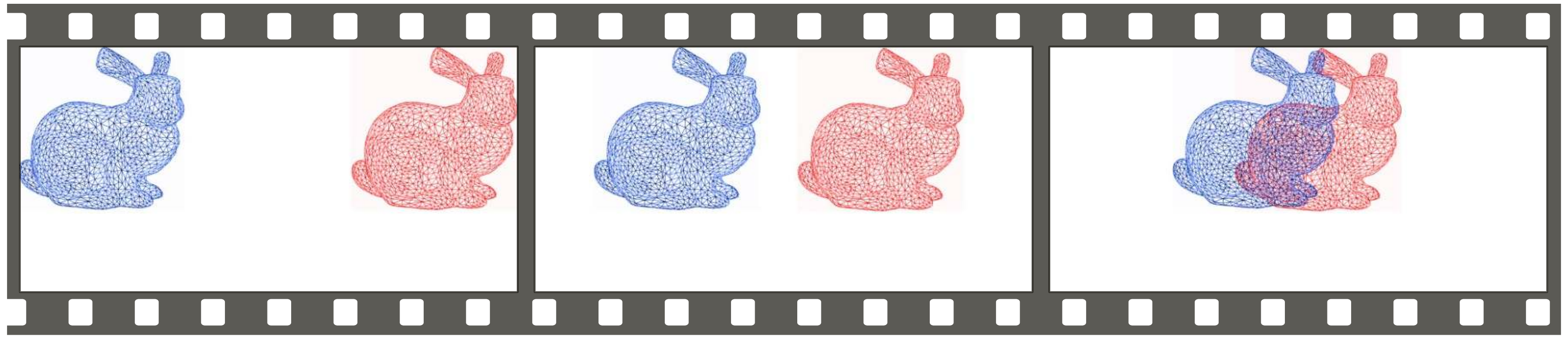
  compute $R := P \cap Q$

Why is Collision Detection Hard?

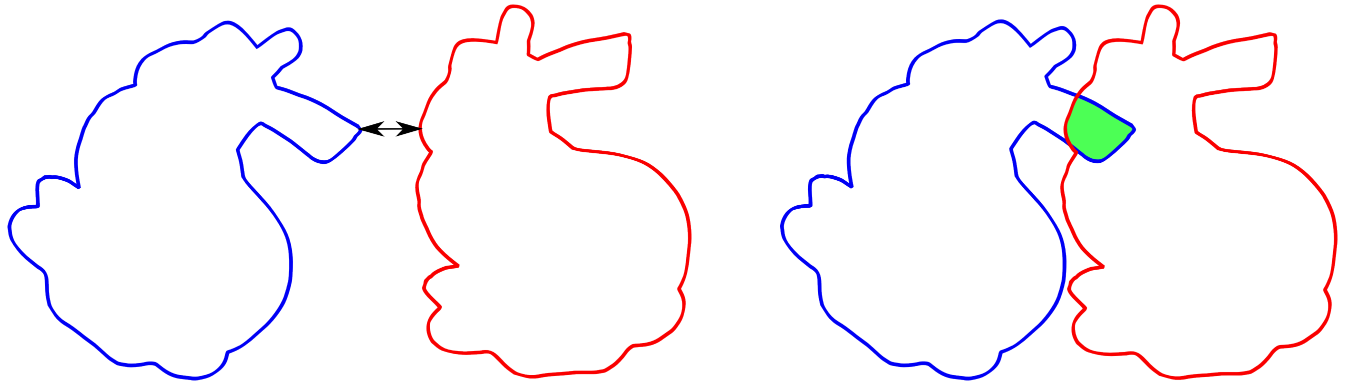
1. All-pairs weakness:



2. Discrete time steps:



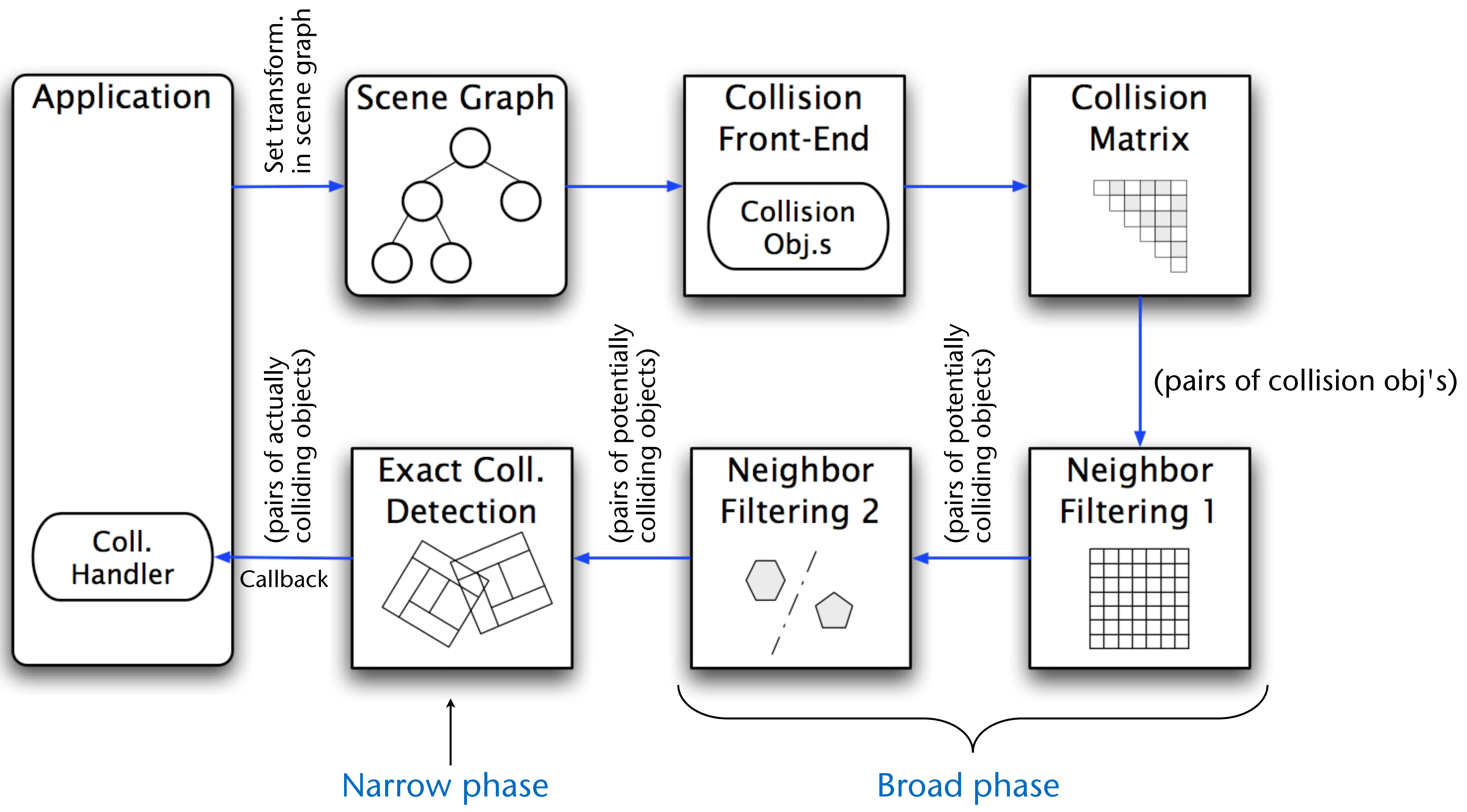
3. Efficient computation of proximity / penetration:



Requirements on Collision Detection

- Handle a large class of objects
- Lots of moving objects (1000s in some cases)
- Very high performance, so that a physically-based simulation can do many iterations per frame (at least 2x 100,000 polygons in <1 millisec)
- Return a contact point ("witness") in case of collision
  - Optionally: return *all* intersection points
- Auxiliary data structures should not be too large (<2x memory usage of originial data)
  - Preprocessing for these auxiliary data structures should not take too long, so that it can be done at startup time (< 5sec / object)

The Collision Detection Pipeline



The Collision Matrix

- Interest in collisions is specific to different applications/modules:
  - Not all modules in an application are interested in all possible collisions;
  - Some pairs of objects collide all the time, some can never collide;
- Goal: prevent unnecessary collision tests
  $\Rightarrow$ Collision Matrix
- The elements in this matrix comprise:
  - Flag for collision detection
  - Additional info that needs to be stored from frame to frame for each pair for certain algorithms (e.g., the separating plane)
  - *Callbacks* in die Module

Obj	1	2	3	4	5	6	7	8
1		x	x	x	x			
2					x			
3					x			x
4								x
5								x
6								x
7								x
8								x

Methods for the Broad Phase

- Broad phase = one or more filtering step
  - Goal: quickly filter pairs of objects that cannot intersect because they are *too far away* from each other $\longrightarrow$ output: PCO's (potentially colliding objects)
- Standard approach:
  - Enclose each object within a bounding box (bbox)
  - Compare the 2 bboxes for a given pair of objects
- Assumption: n objects are moving

➢ *Brute-force* method needs to compare O(n^2) bboxes

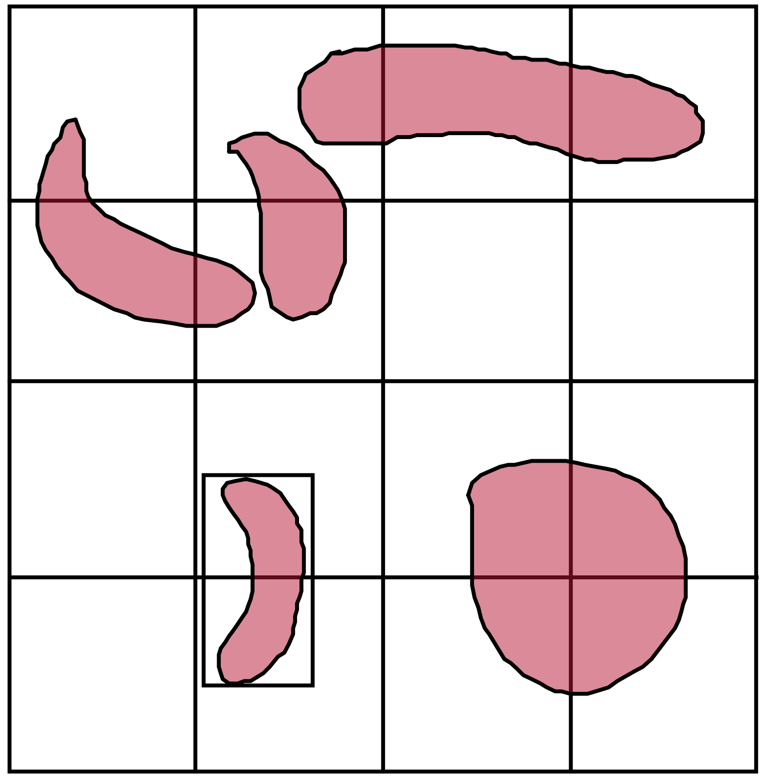
- Goal: determine neighbors more efficiently

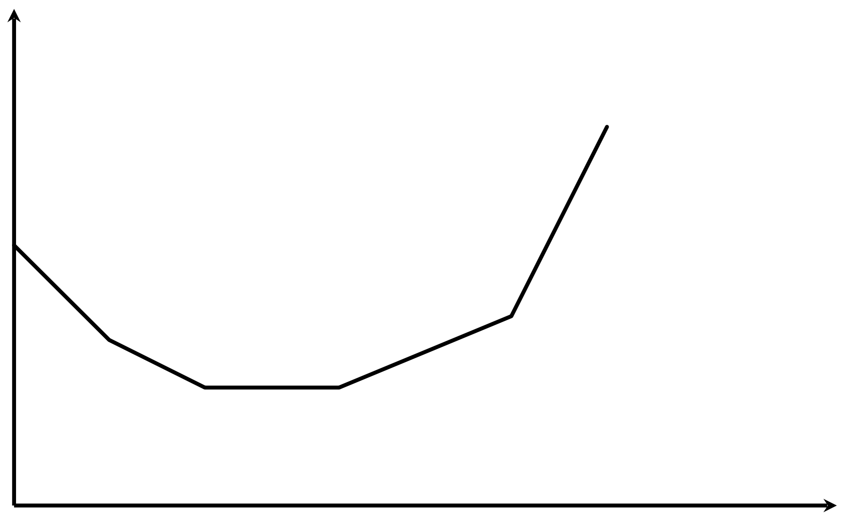
➢ 3D grid, sweep plane techniques ("sweep and prune"), feature tracking on

The 3D Grid

1. Partition the "universe" by a 3D grid
2. For each obj: determine cell occupancy by bbox
3. Find potentially colliding pairs (PCP):
   - Data structure here: hash table (!)
   - Collision in hash table $\longrightarrow$ pairs are a PCP
4. When objects move, update grid

- The trade-off:
  - Fewer cells = larger cells
    - Distant objects are still "neighbors"
  - More cells = smaller cells
    - Objects occupy more cells
    - Effort for updating increases
  - Rule of thumb: cell size ≈ avg obj diameter

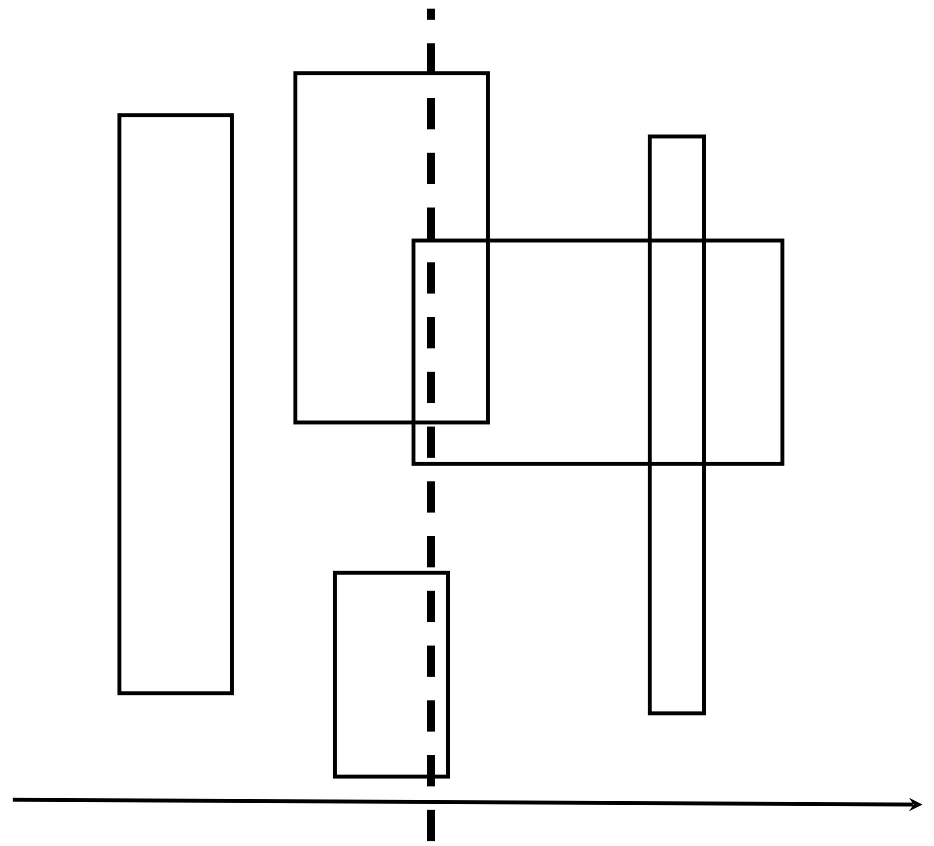




The Plane Sweep Technique (aka Sweep and Prune)

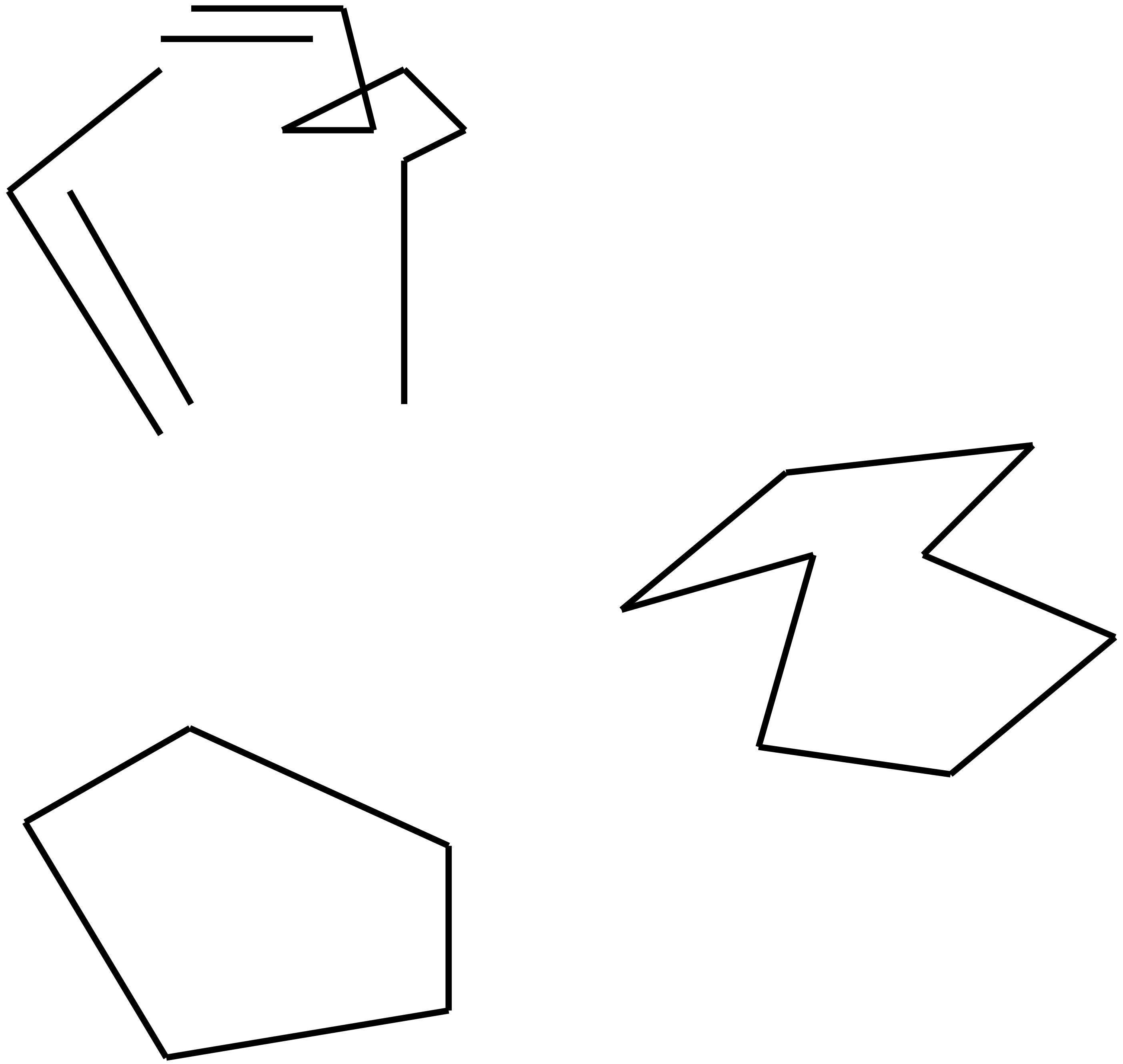
- The idea: sweep plane through space perpendicular to the X axis
- The algorithm:

```
sort the X coordinates of all boxes
start with the leftmost box
keep a list of active boxes
loop over x-coords (= left/right box borders):
  if current box border is the left side (= "opening"):
     check this box against all boxes in the active list
     add this box to the list of active boxes
  else (= "closing"):
     remove this box from the list of active boxes
```



Classes of Objects

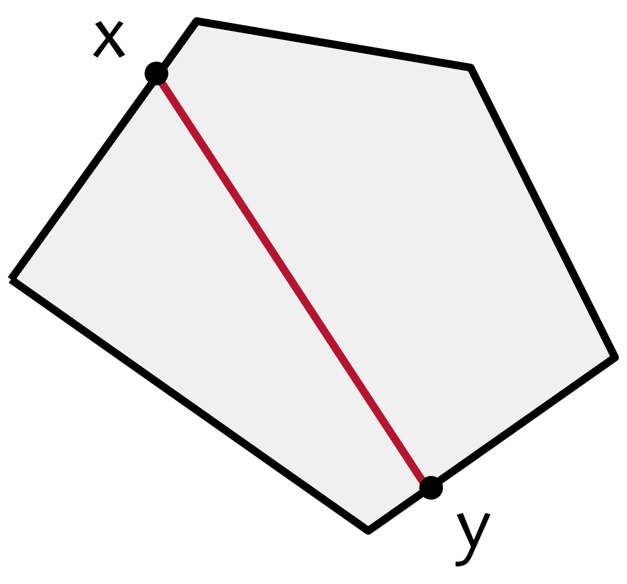
- Polygon soups
  - Not necessarily closed
  - Duplicate polygons
  - Coplanar polygons
  - Self-penetrations
  - Holes
- Closed and simple (no self-penetrations)
- Convex
- Deformable / rigid



Collision Detection for Convex Objects

- Definition of "convex polyhedron":

$$P \subset \mathbb{R}^3 \ \text{convex} \Leftrightarrow$$
$$\forall x, y \in P : \overline{xy} \subset P \Leftrightarrow$$
$$P = \bigcap_{i=1\ldots n} H_i \quad , H_i = \text{half-spaces}$$

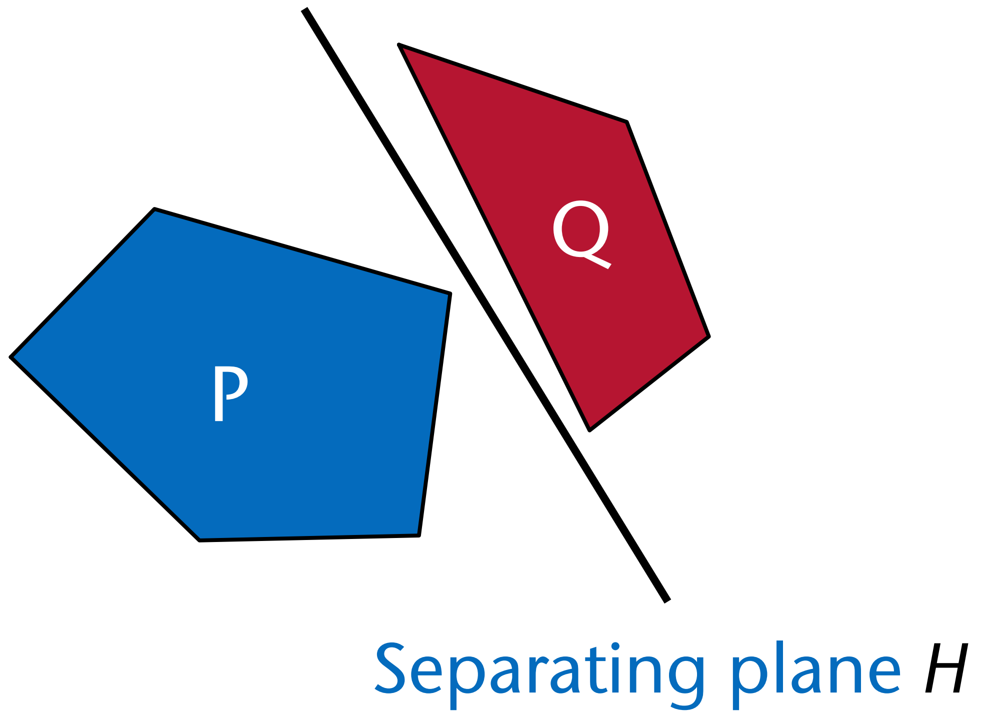


- A condition for "non-collision":

P and Q are "linearly separable" $:\Leftrightarrow$

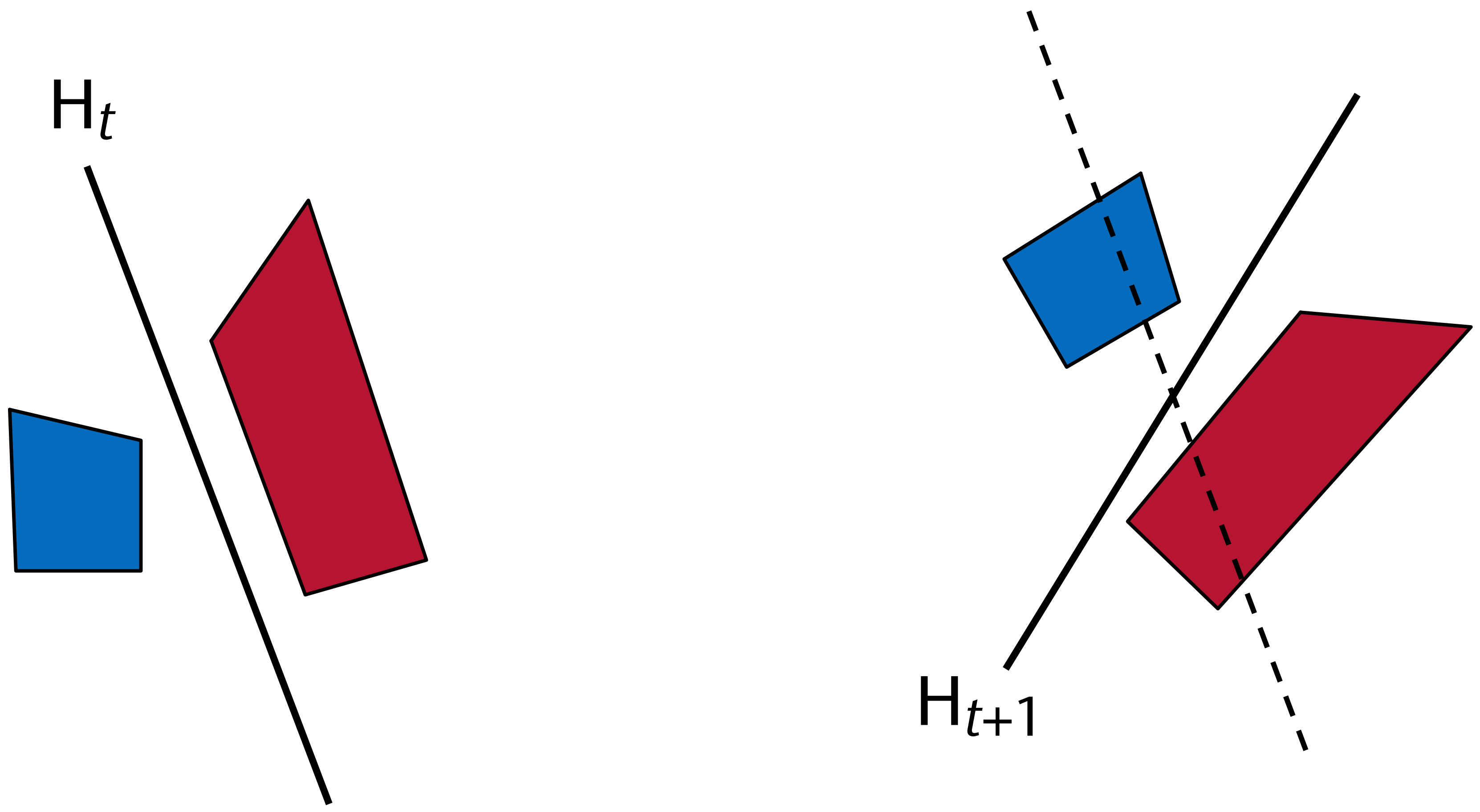
$$\exists \text{ half-space } H : P \subseteq H^- \wedge Q \subseteq H^+ :\Leftrightarrow$$

$$\exists \mathbf{h} \in \mathbb{R}^4 \ \forall \mathbf{p} \in P, \mathbf{q} \in Q : \ (\mathbf{p}, 1) \cdot \mathbf{h} > 0 \ \wedge \ (\mathbf{q}, 1) \cdot \mathbf{h} < 0$$

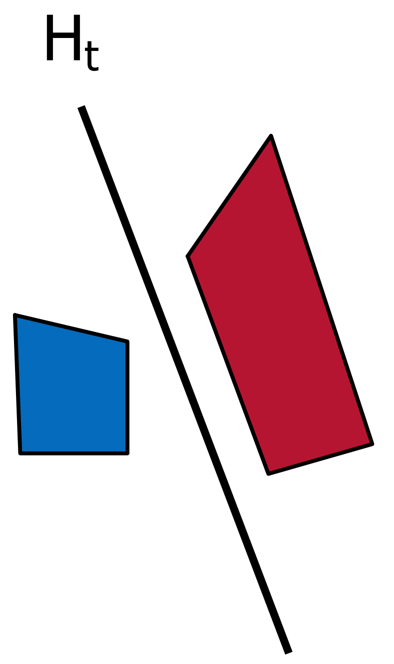


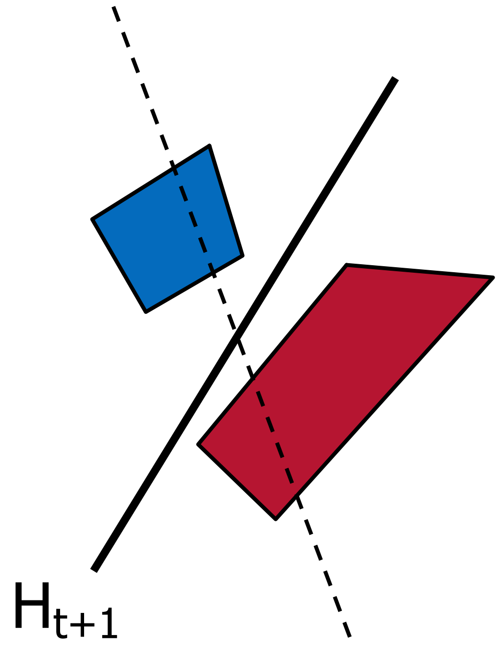
The "Separating Planes" Algorithm

- The idea: utilize temporal coherence →
  if E_t was a separating plane between *P* and *Q* at time *t*, then the new separating plane H_{t+1} is probably not very "far" from H_t (perhaps it is even the same)



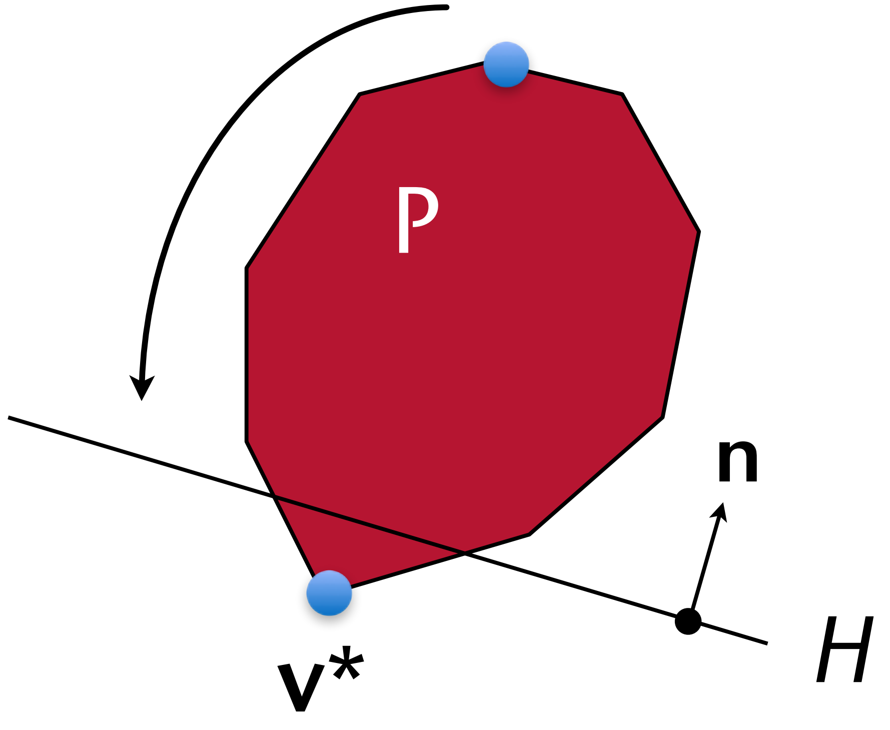
```
load Ht = separating plane between P & Q at time t
H := Ht
repeat max n times
    if exists v ∈ vertices(P)  on the back side of H:
        rot./transl. H such that v is now on the front side of H
    if exists v ∈ vertices(Q)  on the front side of H:
        rot./transl. H such that v is now on the back side of H
    if there are no vertices on the "wrong" side of H, resp.:
        return "no collision"
if there are still vertices on the "wrong" side of H:
    return "collision"    {could be wrong}
save  Ht+1 := H   for the next frame
```





How to Find a Vertex on the "Wrong" Side Quickly

- The brute-force method:
  test all vertices **v** whether $f(\mathbf{v}) = (\mathbf{v} - \mathbf{p}) \cdot \mathbf{n} > 0$
- Observation:
  1. *f* is linear in v_x, v_y, v_z,
  2. P is convex ⇒ *f*(*x*) has (usually) exactly *one* minimum over all points **x** on the surface of P, consequently ..
  3. $\exists^1 \mathbf{v}^* \; : \; f(\mathbf{v}^*) = \min$
- The algorithm (steepest descent on the surface wrt. *f*):
  - Start with an arbitrary vertex **v**
  - Walk to that neighbor **v'** of **v** for which $f(\mathbf{v}') = \min.$ (among all neighbors)
  - Stop if there is no neighbor **v'** of **v** for which $f(\mathbf{v}') < f(\mathbf{v})$



Updating the Candidate Plane, H

- In the following, represent all vertices **p** as (**p**, 1), i.e., use *homogeneous coords*
- We want $\forall \mathbf{p} \in P : \mathbf{h} \cdot \mathbf{p} > 0$ and $\forall \mathbf{q} \in P : \mathbf{h} \cdot \mathbf{q} < 0$
- Let $\bar{P} \subseteq P$ be the "offending" points for a given plane **h**, i.e. $\forall \mathbf{p} \in \bar{P} : \mathbf{h} \cdot \mathbf{p} < 0$
- Define a cost function $c = c(\mathbf{h}) = -\sum_{\mathbf{p} \in \bar{P}} \mathbf{h} \cdot \mathbf{p}$
- Change **h** so as to drive *c* down towards 0
- Gradient descent: change **h** by negative gradient of *c*, i.e. $\mathbf{h}' = \mathbf{h} - \frac{d}{d\mathbf{h}} c(\mathbf{h})$
- Cost fct *c* is linear in **h**, so $\frac{d}{d\mathbf{h}} c = -\sum_{\mathbf{p} \in \bar{P}} \mathbf{p}$
- Therefore, $\mathbf{h}' = \mathbf{h} + \eta \sum_{\mathbf{p} \in \bar{P}} \mathbf{p}$, with η = "learning speed" (usually $\eta \ll 1$)
- In practice, one decelerates, i.e., $\eta' = 0.97\eta$, to prevent cycling
- (For object Q, some signs need to be changed)

- Perceptron Learning Rule (known in machine learning for a long time):
  whenever we find $\mathbf{p} \in P$ with $\mathbf{h} \cdot \mathbf{p} < 0$, update $\mathbf{h}$ using $\mathbf{h}' = \mathbf{h} + \eta \mathbf{p}$.
  (Analog for Q, with some signs reversed.)
- Theorem:
  If *P*, *Q* are linearly separable, then repeated application of the perceptron learning rule will terminate after a finite number of steps.
- Corollary:
  If *P*, *Q* are linearly separable, then the algorithm will find a separating plane in a finite number of steps.

  (When algo terminates, none of *P*, *Q*'s vertices are on the wrong side. I.e., each step brings *H* closer to the solution.)

Proof of the Theorem

- Let **h*** be a separating plane, w.l.og. $\|\mathbf{h}^*\| = 1$
- There is a d, such that $\forall p \in P : \mathbf{h}^* \cdot \mathbf{p} \geq d > 0 \ , \ \forall q \in Q : \mathbf{h}^* \cdot \mathbf{q} \leq -d < 0$
  - Such a value d is called the "margin" of **h***
- Assume further **h*** is optimal w.r.t. the margin d (i.e., has the largest margin)
- Let $V = P \cup \{-\mathbf{q} \,|\, \mathbf{q} \in Q\}$
  - Thus, P, Q is linearly separable $\Leftrightarrow$

$$\forall p \in P : \mathbf{h} \cdot \mathbf{p} > 0 \ \wedge \ \forall q \in Q : \mathbf{h} \cdot \mathbf{q} < 0 \quad \Leftrightarrow \forall v \in V : \mathbf{h} \cdot \mathbf{v} > 0$$

- Let $\mathbf{v} \in V$ be an "offending" vertex in k-th iteration
- After k iterations, $\mathbf{h}^k = \mathbf{h}^{k-1} + \eta\mathbf{v} = \mathbf{h}^{k-2} + \eta\mathbf{v}' + \eta\mathbf{v} = \ldots = \eta \sum_{\mathbf{v}\in V} k_v \mathbf{v}$ where k_v = #iterations in which **v** was the offending vertex
- Consider $\mathbf{h}^*\mathbf{h}^k$:

$$\mathbf{h}^* \cdot \mathbf{h}^k = \mathbf{h}^* \cdot \left(\eta \sum_{\mathbf{v}\in V} k_v \mathbf{v}\right) = \eta \sum_{\mathbf{v}\in V} k_v \mathbf{h}^* \cdot \mathbf{v} \geq \eta d \sum_{\mathbf{v}\in V} k_v = \eta d k$$

- Now, we use a trick to find a lower bound on $|\mathbf{h}^k|$:

$$\|\mathbf{h}^k\|^2 = \|\mathbf{h}^*\|^2 \cdot \|\mathbf{h}^k\|^2 \geq \|\mathbf{h}^* \cdot \mathbf{h}^k\|^2 = \eta^2 d^2 k^2$$

- Now, find an upper bound
- Let $D = \max_{\mathbf{v} \in V} \{ \|\mathbf{v}\| \}$
- Consider one iteration:

$$\|\mathbf{h}^k\|^2 - \|\mathbf{h}^{k-1}\|^2 = \|\mathbf{h}^{k-1} + \eta \mathbf{v}\|^2 - \|\mathbf{h}^{k-1}\|^2$$

$$= \|\mathbf{h}^{k-1}\|^2 + 2\eta \mathbf{h}^{k-1}\mathbf{v} + (\eta \mathbf{v})^2 - \|\mathbf{h}^{k-1}\|^2$$

$$< 0 + \eta^2 D^2$$

- Taking this over *k* iterations:

$$\|\mathbf{h}^k\|^2 < k\eta^2 D^2 + \|\mathbf{h}^0\|^2$$

- Putting lower and upper bound together gives:

$$\eta^2 d^2 k^2 \leq \|\mathbf{h}^k\|^2 \leq k\eta^2 D^2$$

- Solving for k:

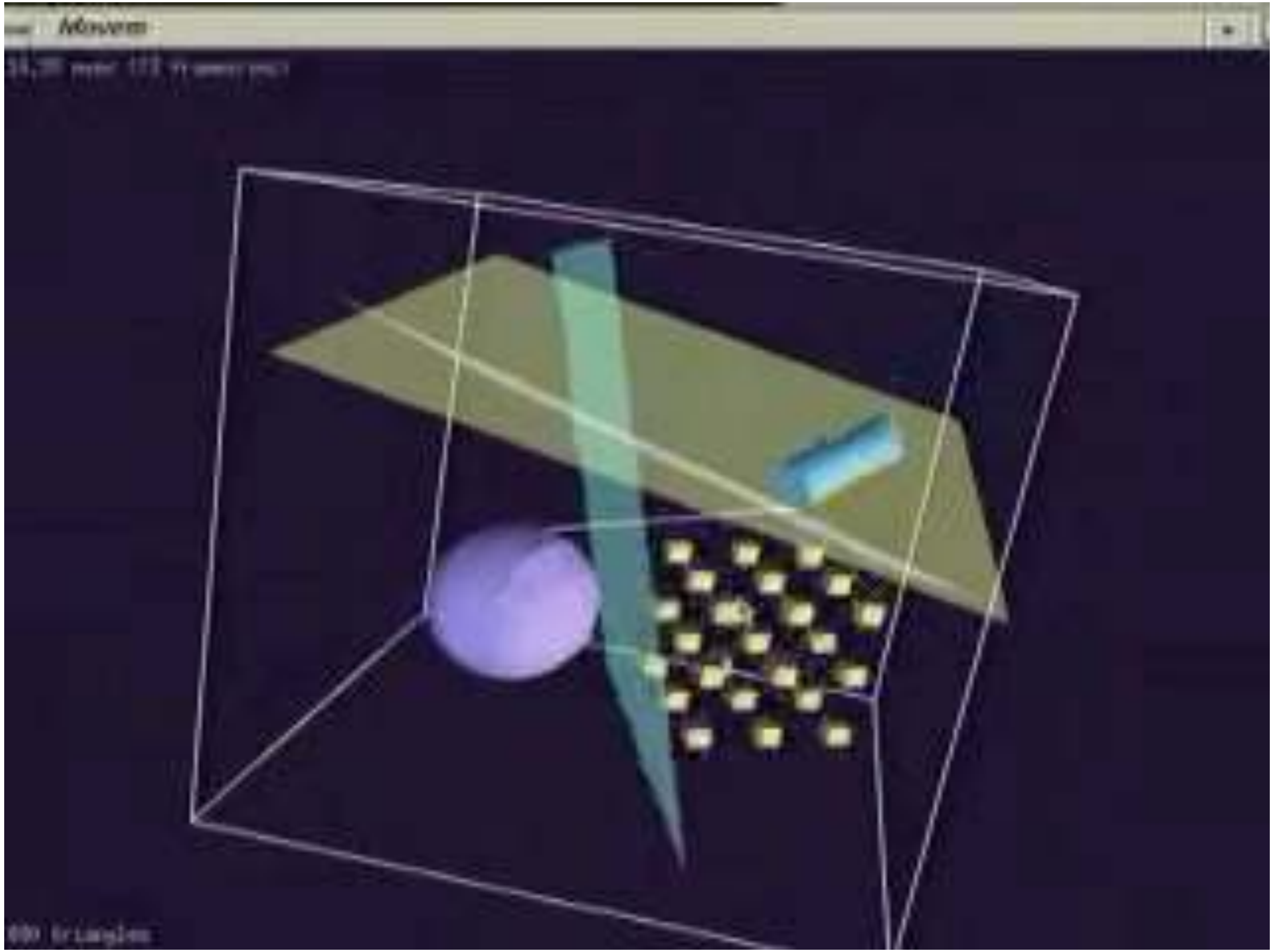
$$k \leq \frac{D^2}{d^2}$$

- In other words, the factor $\frac{D^2}{d^2}$ gives a hint, how many iterations could be needed; i.e., to some extent, $\frac{D}{d}$ is a measure of the "difficulty" of the problem (except, we don't know d or D in advance)

Properties of this Algorithm

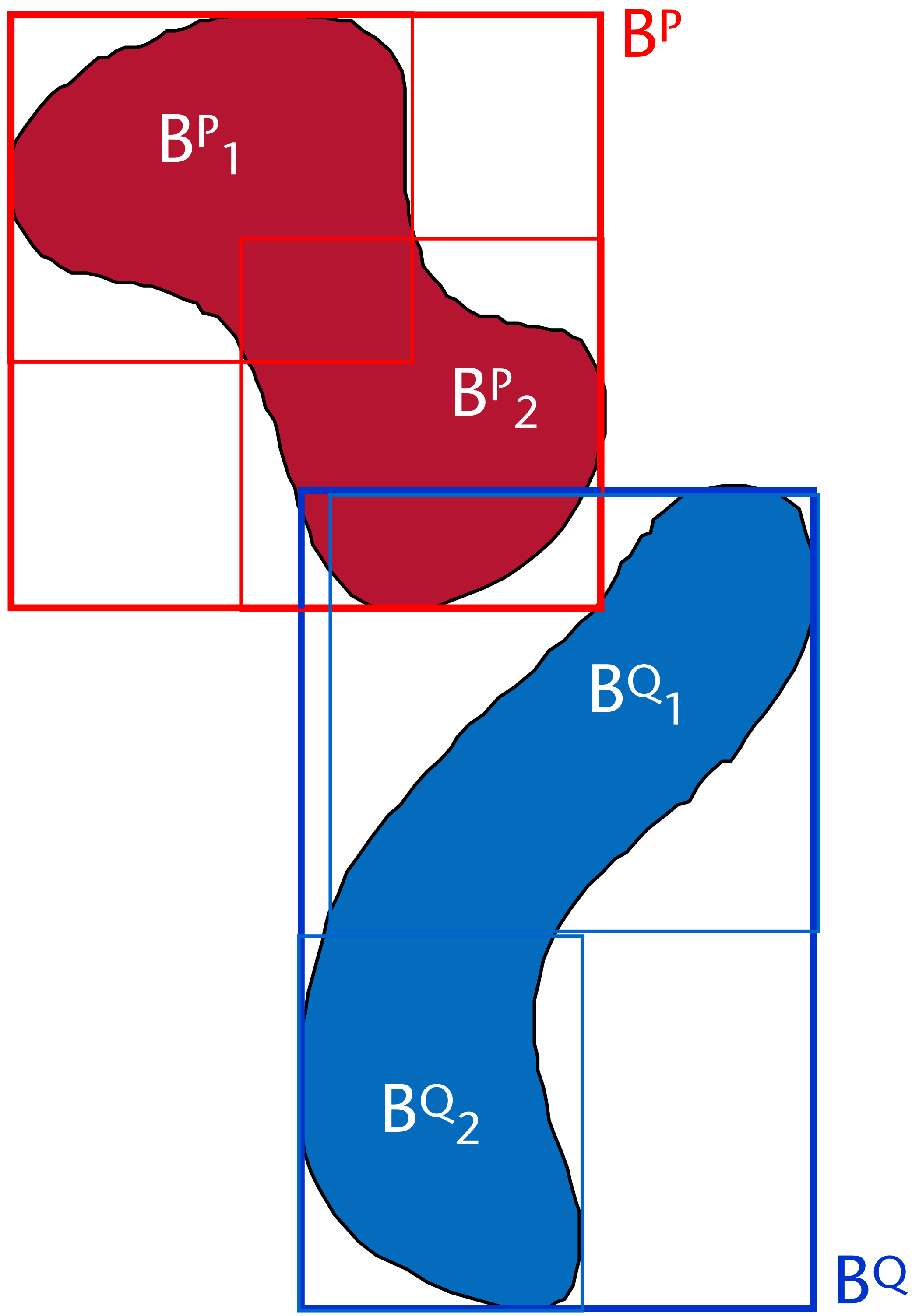
- \+ Expected running time is in O(1)!
  The algo exploits *frame-to-frame coherence*:
  if the objects move only very little, then the algo just checks whether the old separating plane is still a separating plane;
  if the separating plane has to be moved, then the algo is often finished after a few iterations.
- \+ Works even for deformable objects, so long as they stay convex
- – Works only for convex objects
- – Could return the wrong answer if P and Q are extremely close but not intersecting (bias)
- Research question: can you find an un-biased (deterministic) variant?

Visualization



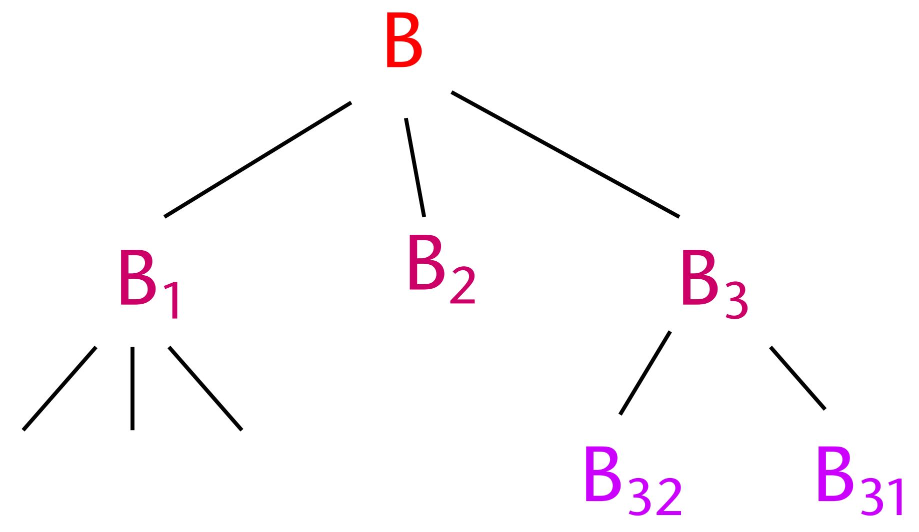
Hierarchical Collision Detection

- *The* standard approach for "polygon soups"
- Algorithmic technique: divide & conquer

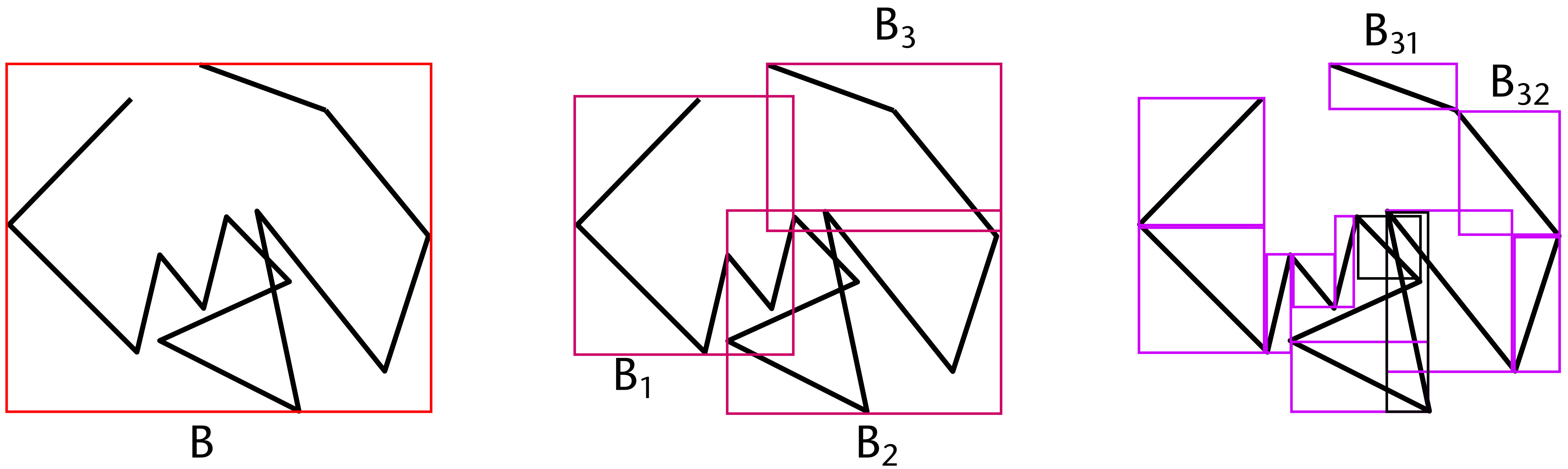


The Bounding Volume Hierarchy (BVH)

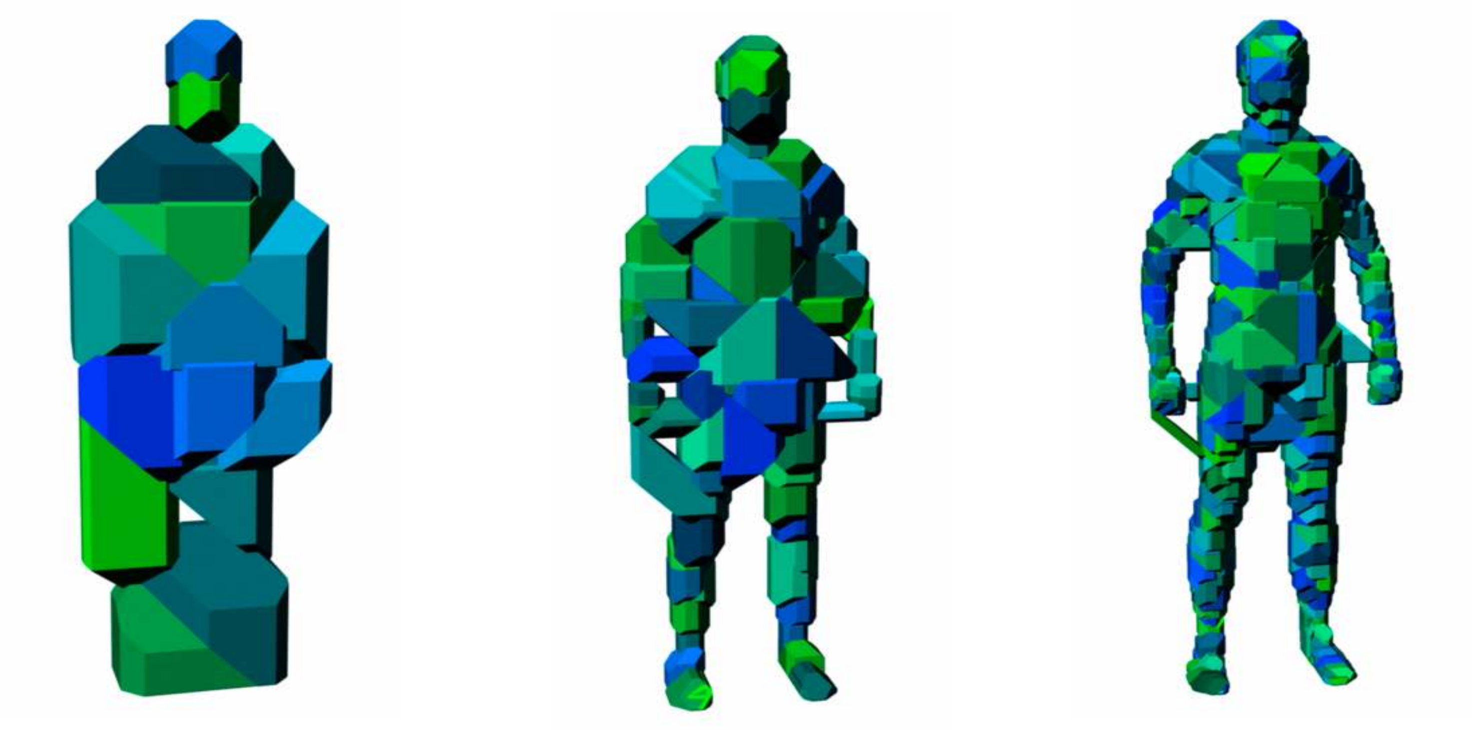
- Constructive definition of a bounding volume hierarchy:
  1. Enclose all polygons, P, in a bounding volume BV(P)
  2. Partition P into subsets $P_1, \ldots, P_n$
  3. Rekursively construct a BVH for each P_i and put them as children of P in the tree

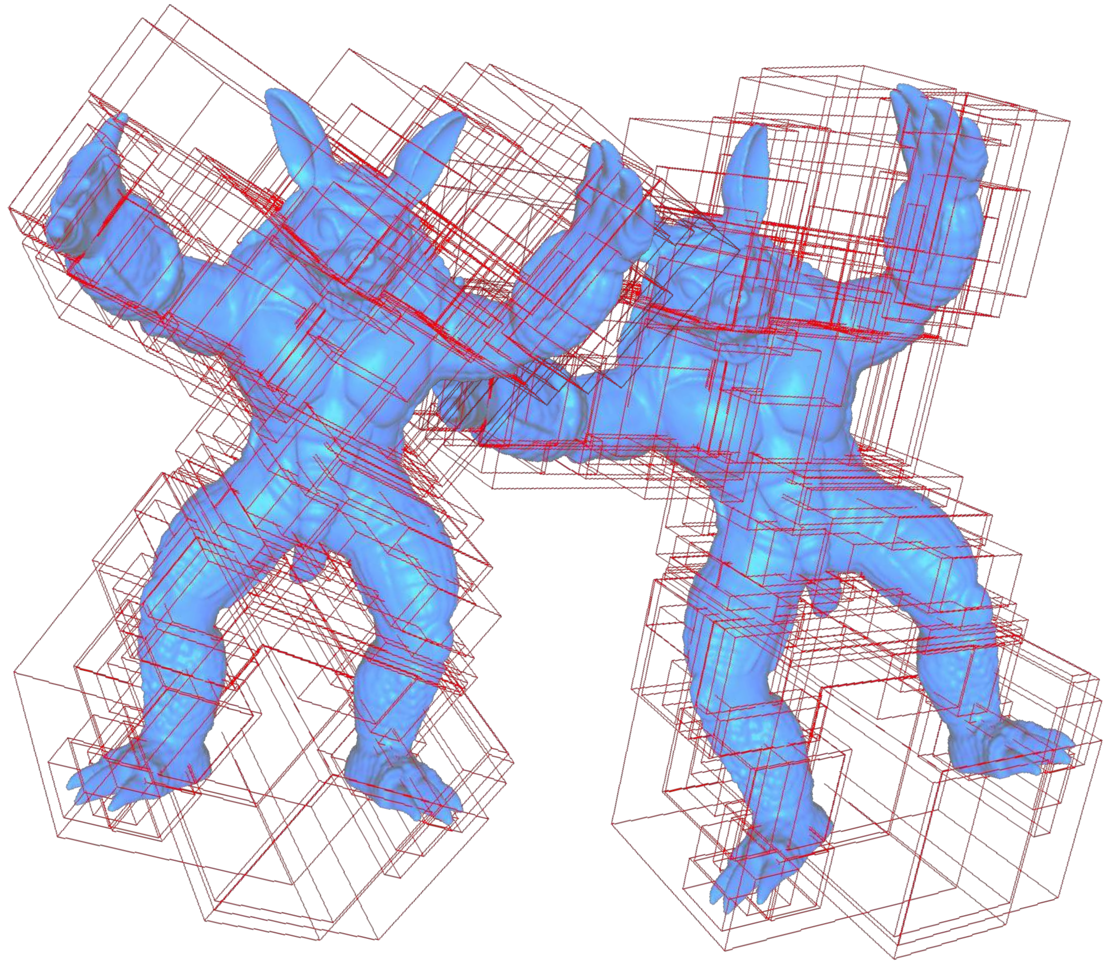


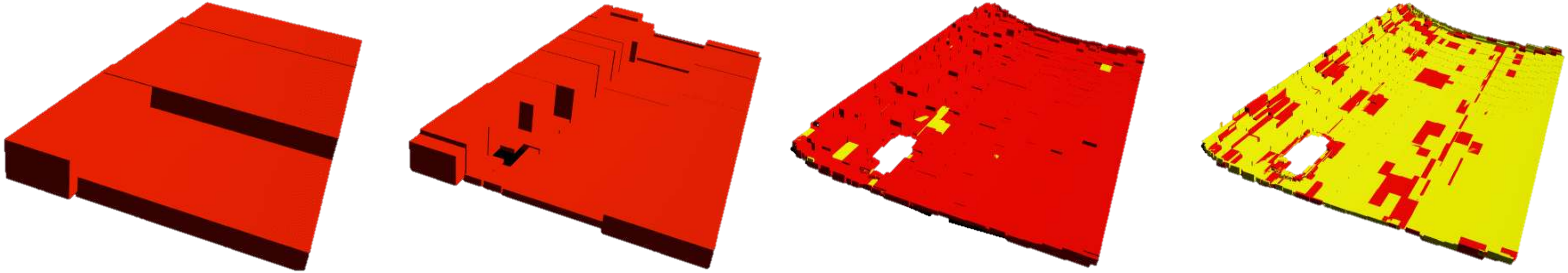
- Typical arity = 2 or 4



Visualizations of different levels of some BVHs



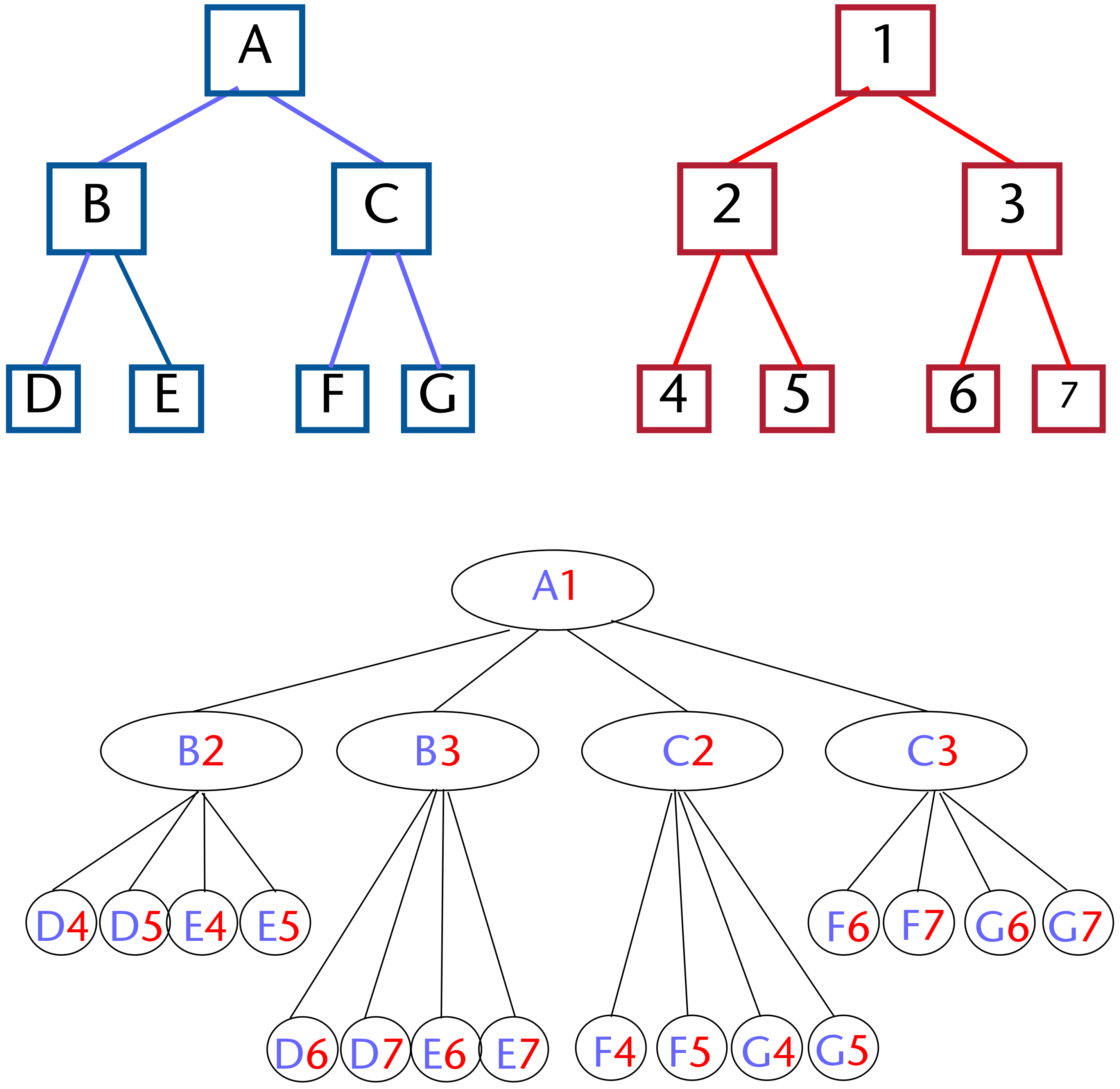




The General Hierarchical Collision Detection Algo

- Simultaneous traversal of two BVHs:

```
traverse( node X, node Y ):
if X,Y do not overlap:
    return
if X,Y are leaves:
    check polygons
else
    for all children pairs:
        traverse( Xi, Yj )
```



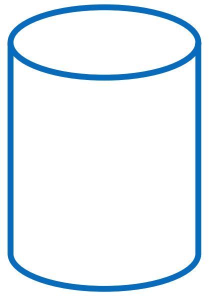
Bounding Volume Test Tree (BVTT)
(only a conceptual(!) tree, never actually stored)

Different Kinds of Bounding Volumes

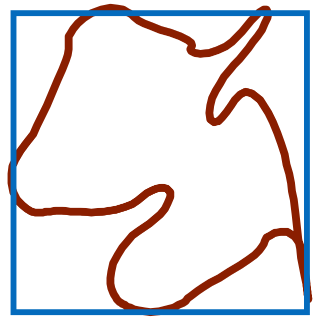
Requirements (for collision detection):

- *Very* fast overlap test $\longrightarrow$ "simple" BVs
  - Even if BVs have been translated/rotated
- Little overlap among BVs on the same level in a BVH (i.e., if you want to cover the whole space with the BVs, there should be as little overlap as possible) $\longrightarrow$ "*tight BVs*"

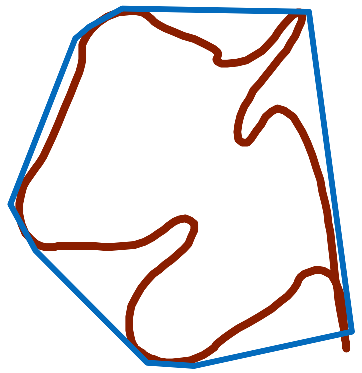
Different Kinds of Bounding Volumes



Cylinder
[Weghorst et al., 1985]

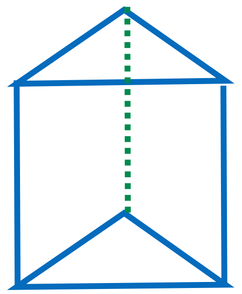


Box, AABB (R*-trees)
[Beckmann, Kriegel, et al., 1990]

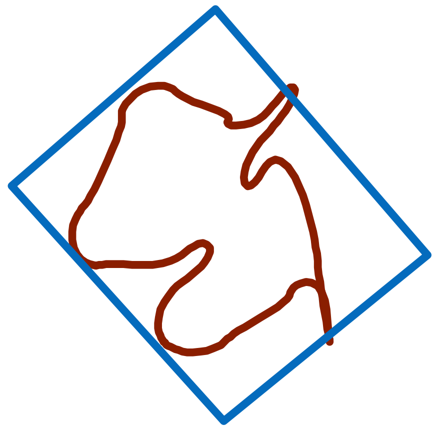


Convex hull
[Lin et. al., 2001]

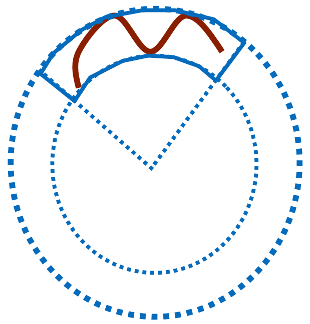
Sphere
[Hubbard, 1996]



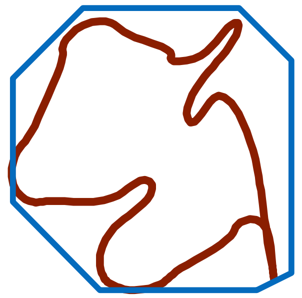
Prism
[Barequet, et al., 1996]



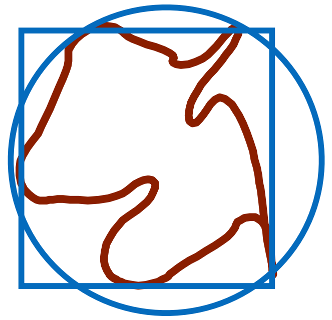
OBB (oriented bounding box)
[Gottschalk, et al., 1996]



Spherical shell
[Manocha, 1997]



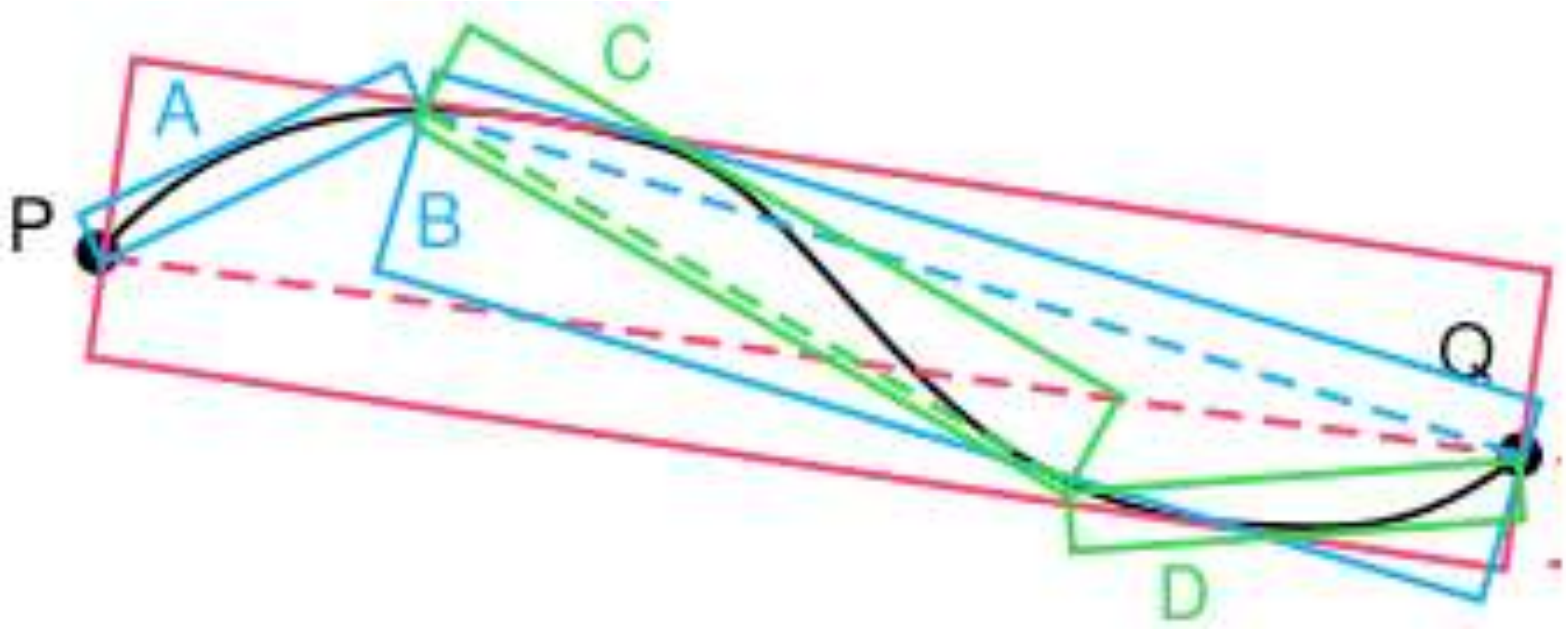
k-DOP / Slabs
[Zachmann, 1998]



Intersection of
several BVs

The Wheel of Re-Invention

- OBB-Trees: have been proposed already in 1981 by Dana Ballard for bounding 2D curves, except they called it "strip trees"



- AABB hierarchies: have been invented(?) in the 80-ies in the spatial data bases community, except they call them "R-tree", or "R*-tree", or "X-tree", etc.

Relationship Between Type of BV and Runtime

- In case of rigid collision detection (BVH construction can be neglected):

$$T = N_V C_V + N_P C_P$$

N_V = number of BV overlap tests
C_V = cost of one BV overlap test
N_P = number of intersection tests of primitives (e.g., triangles)
C_P = cost of one intersection test of two primitives

- In case of deformable objects (BVH must be updated):

$$T = N_V C_V + N_P C_P + N_U C_U$$

N_U / C_U = number/cost of a BV update

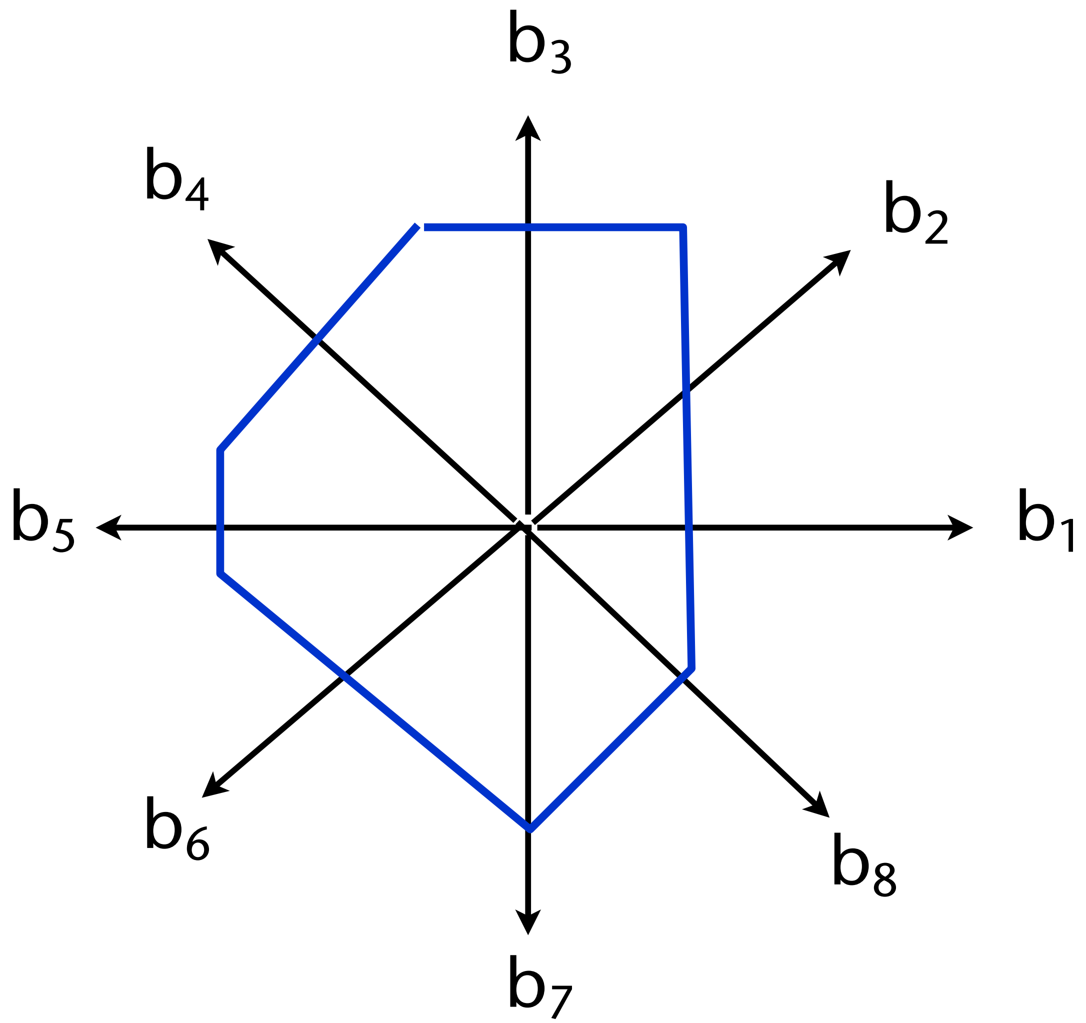
- As the kind of BV gets tighter, N_V (and, to some degree, N_P) decreases, but C_V and (usually) C_U increases

Discretely Oriented Polytopes (k-DOPs)

- Definition of *k*-DOPs:
  Choose *k* fixed vectors $\mathbf{b}_i \in \mathbb{R}^3$, with *k* even, and $\mathbf{b}_i = -\mathbf{b}_{i+k/2}$.
  We call these vectors generating vectors (or just generators).

  A *k*-DOP is a volume defined by the intersection of *k* half-spaces:

$$D = \bigcap_{i=1..k} H_i \quad , \quad H_i : \mathbf{b}_i \cdot x - d_i \leq 0$$



Note: this is just a sketch in 2D! in 3D graphics, the generators should be evenly spaced over the unit sphere!

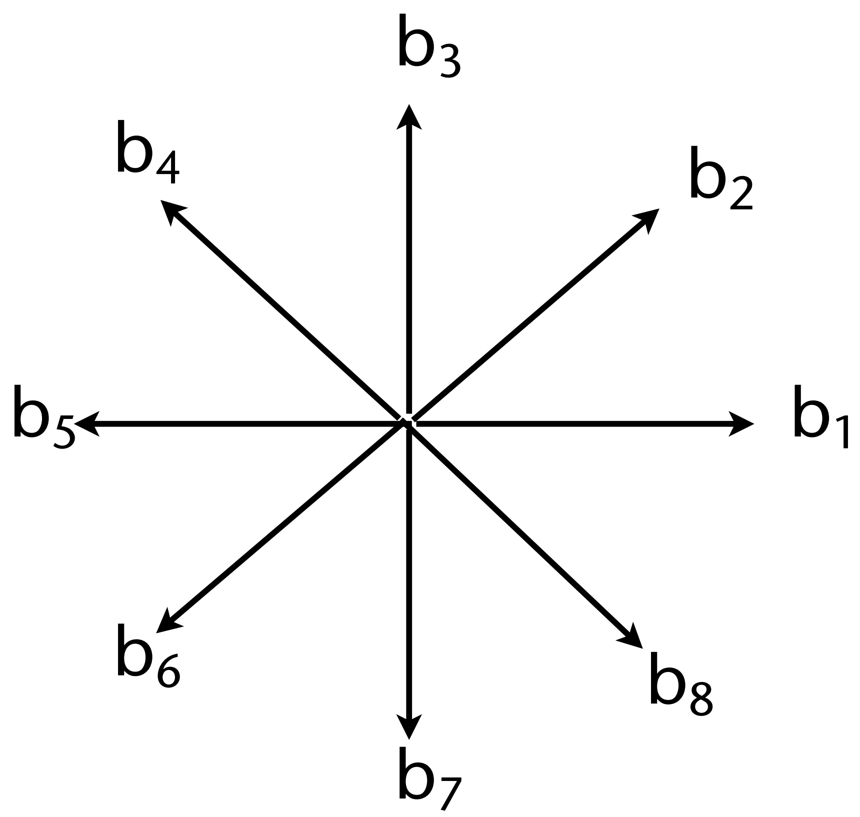
- Note: a *k*-DOP is completely described by $D = (d_1, \ldots, d_k) \in \mathbb{R}^k$

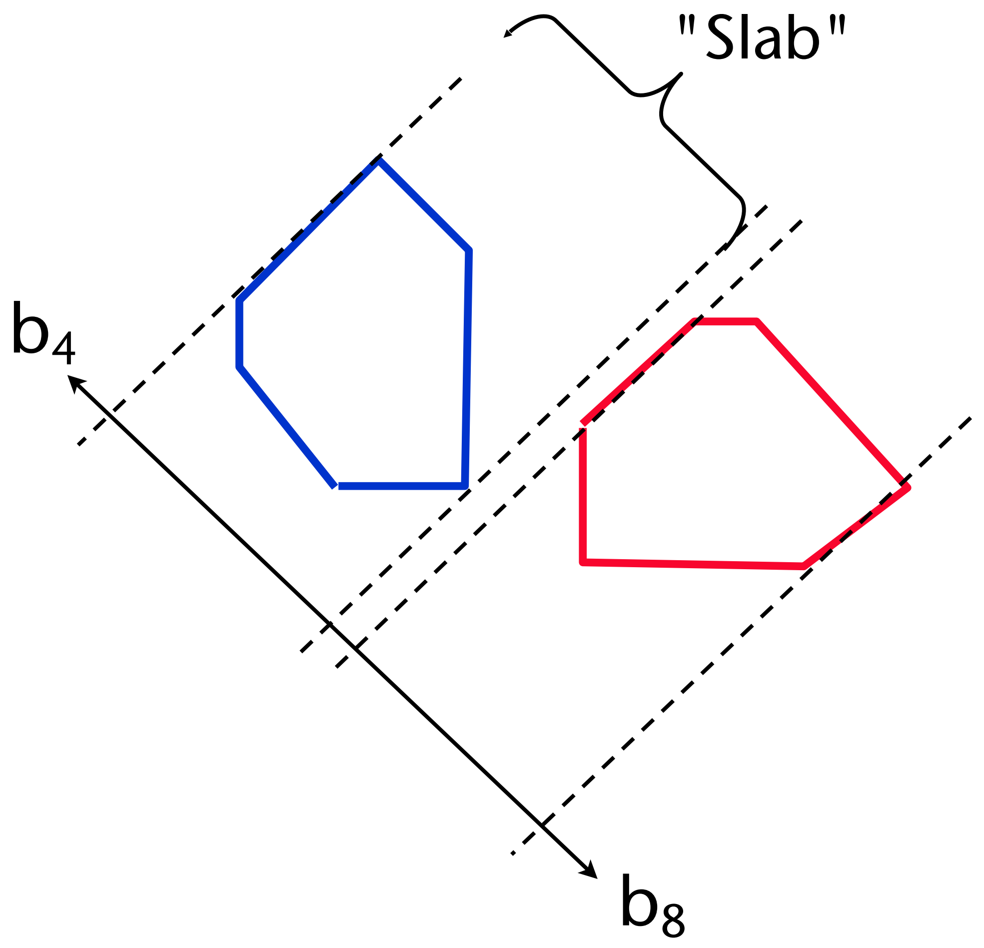
- The overlap test for two (generator-aligned) *k*-DOPs:

$$D^1 \cap D^2 = \varnothing \Leftrightarrow$$

$$\exists i = 1, .., \frac{k}{2} : \left[d_i^1, d_{i+\frac{k}{2}}^1 \right] \cap \left[d_i^2, d_{i+\frac{k}{2}}^2 \right] = \varnothing$$

i.e., it is just *k*/2 interval tests, like this one:



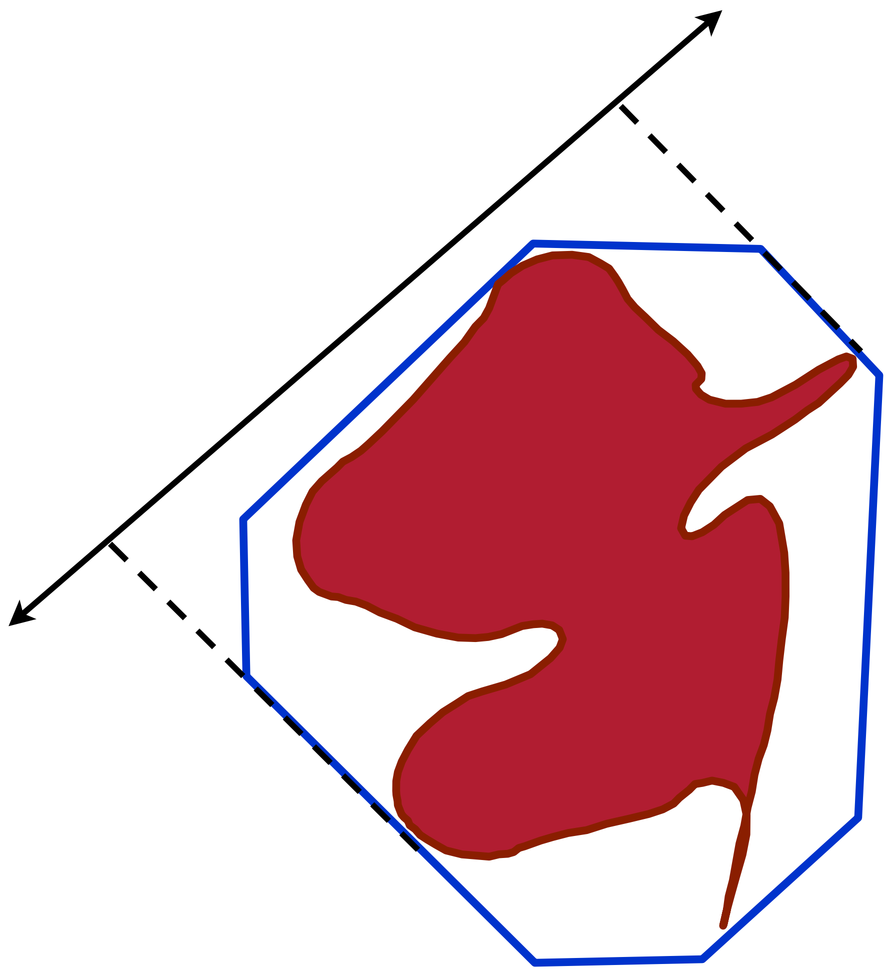


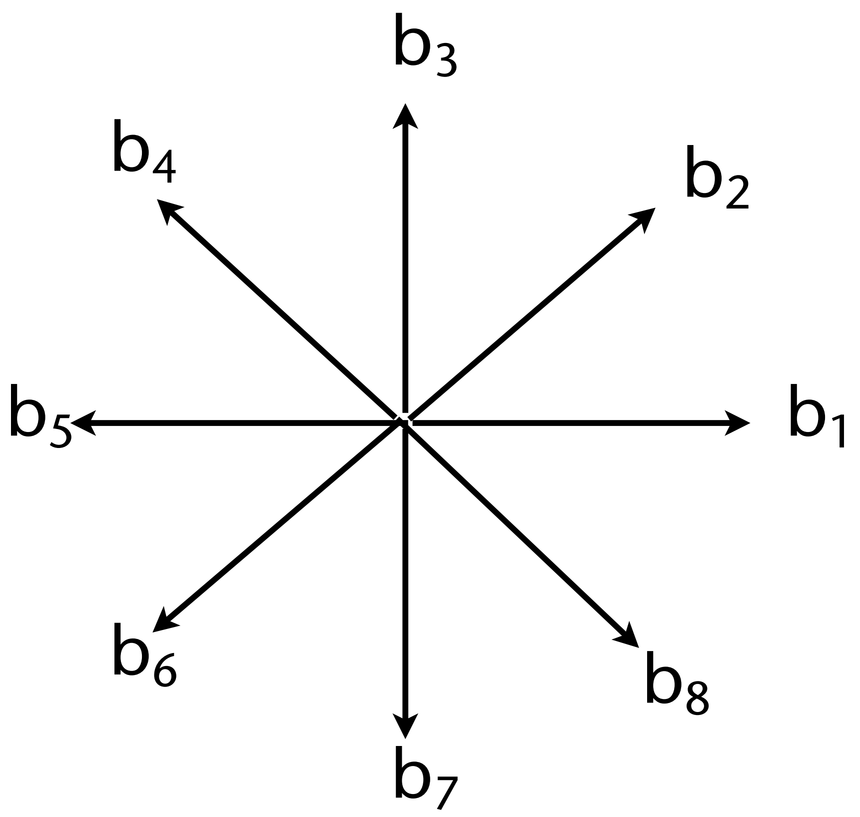
- Note: this is just a generalization of the simple AABB overlap test!

- Computation of a k-DOP, given a polygon soup with vertices $\mathcal{V} = \{\mathbf{v}_0, \ldots, \mathbf{v}_n\}$
- For each i = 1, .., k, compute

$$d_i = \max_{j=0,\ldots,n} \{\mathbf{v}_j \cdot \mathbf{b}_i\}$$

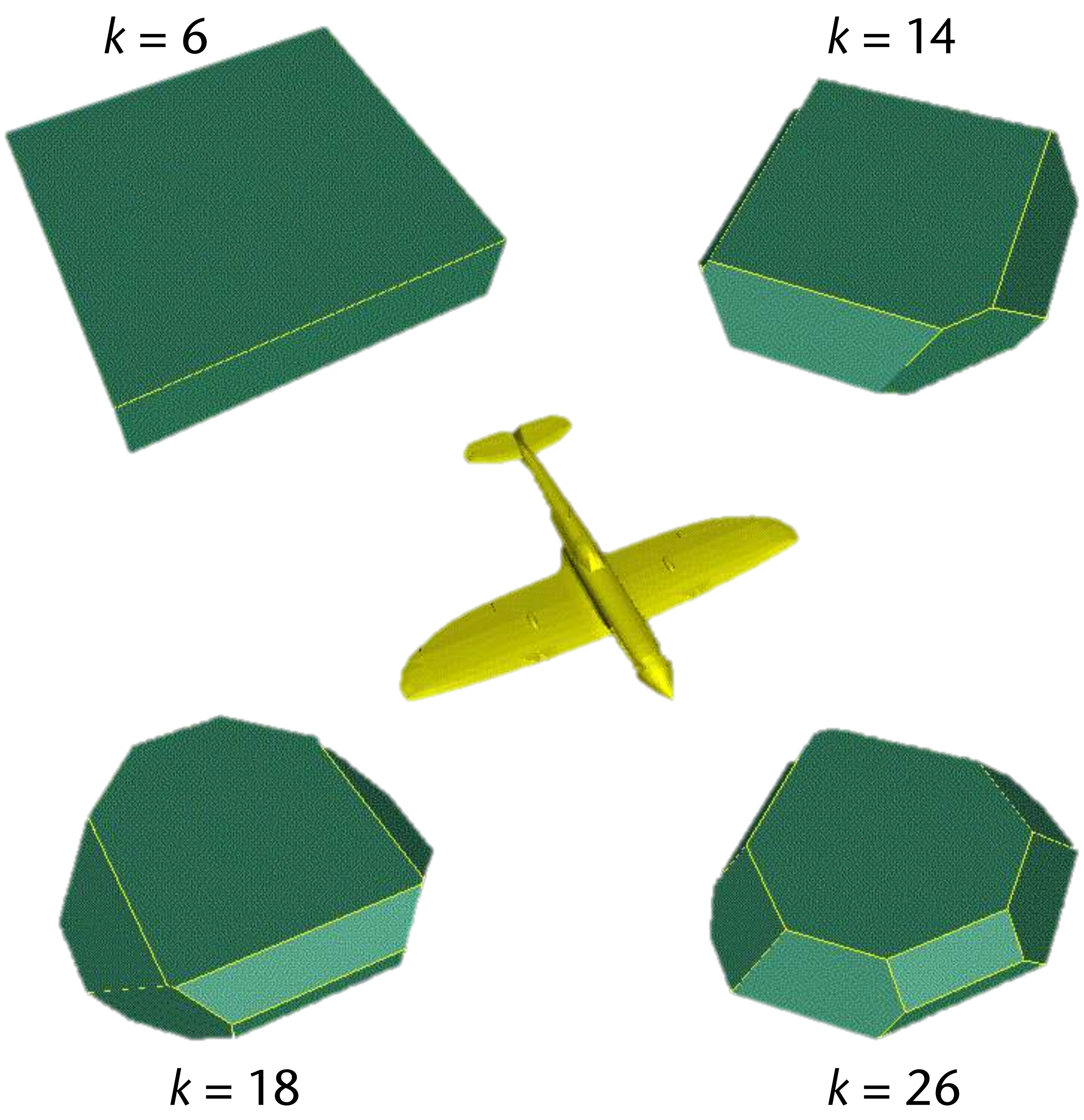
(assuming $||\mathbf{b}_i|| = 1$)





Some Properties of k-DOPs

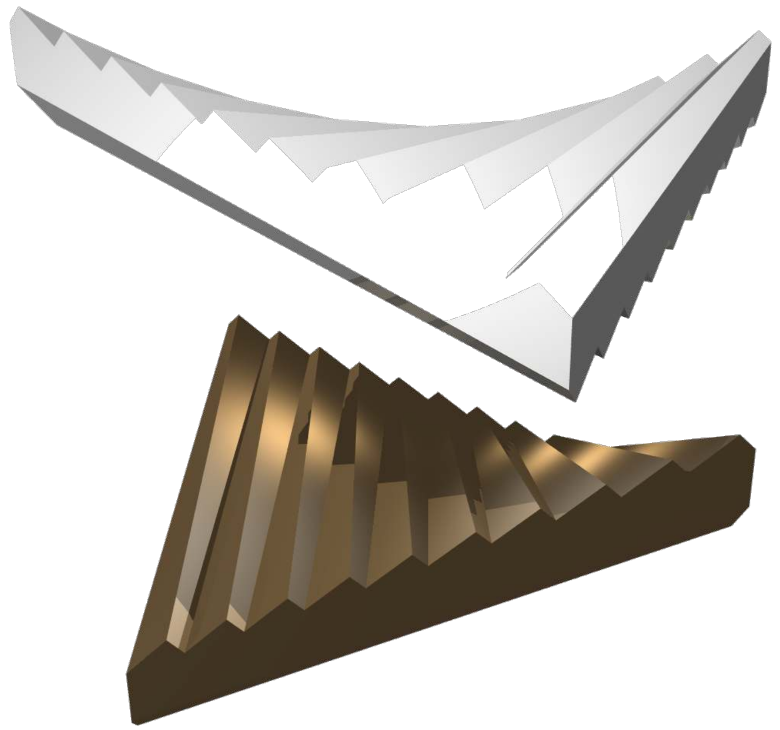
- AABBs are special DOPs
- The overlap test takes time $\in O(k)$, k = number of orientations
- With growing k, the convex hull can be approximated arbitrarily precise



How to Deal With Non-Aligned (Rotated) DOPs?

- When using k-DOPs for BVH's for collision detection, usually the DOPs in those hierarchies are calculated in object space, but later rotated in world space
- Approach (w/o details):
  - Precompute (at the beginning of kDOP-BVH traversal) a rotation matrix from *A*'s object space into *B*'s object space
  - Using that rotation matrix and a generic, generator-aligned kDOP, precompute a transformation matrix for the kDOP's in BVH *A*
  - Before testing a pair of (non-aligned) kDOP's in the two BVH's, enclose the kDOP *D* from *A* in a new kDOP *D*' that is generator-aligned w.r.t. *B*'s generators
  - Then perform the standard overlap test doing $k/2$ interval overlap tests

Parallel Collision Detection (*k*Det)



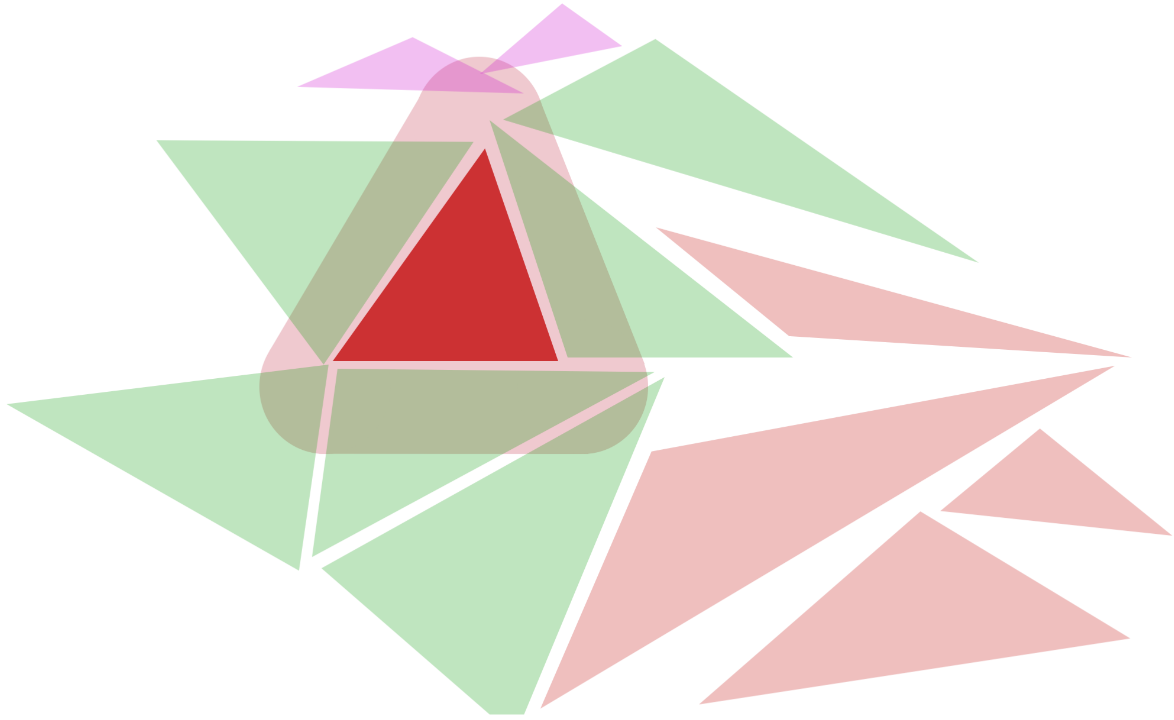
- Problem: all-pairs weakness, i.e., O(n^2) in worst-case
- Goals:
  1. Parallelize polygon pair finding
  2. Characterization of objects *not* exposing all-pairs weakness
- Approach:
  1. Algorithm using a hierarchy of grids (bottom-up traversal)
  2. Geometric predicate involving Minkowski sums of triangles and balls showing $O(n)$ intersecting pairs of triangles

Preliminary Considerations

- What are the root causes for $O(n^2)$ coll.det. time?

1. Polygons are two-dimensional manifolds embedded in 3D $\longrightarrow$ can be stacked arbitrarily tightly without intersections
2. In "stair cases"-like objects, polygons can have arbitrarily large aspect ratio
   - Aspect ratio = $\frac{\text{long side}}{\text{short side}}$ of its enclosing bbox

- Definition of "k-free sparsity":
  Consider a set A of triangles and a triangle $T \in A$;
  T is called k-free, iff the #tris "close" to $T \leq k$,
  where we only count triangles if they are
  "larger than" or as large as T



- If all A is k-free, then tris can't get "too close" to each other

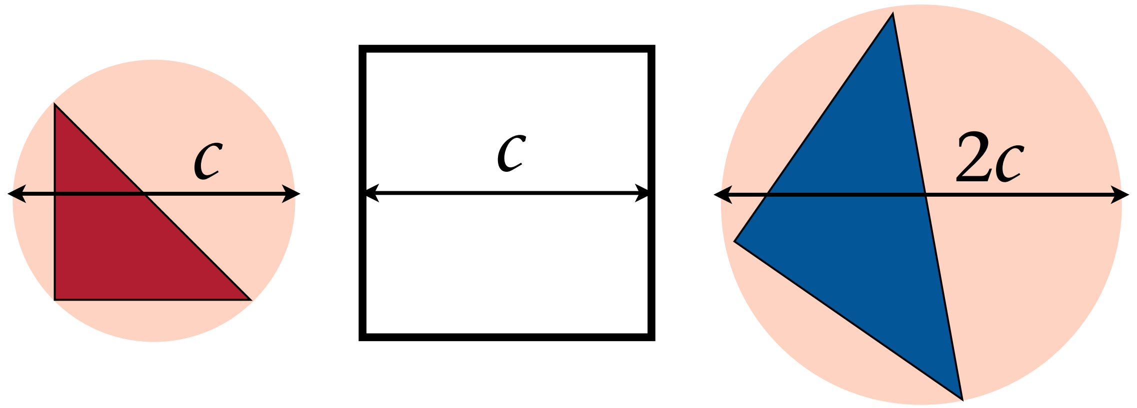
- Theorem [Weller 2017]:
  Let A be a k-free set of triangles; let T be a triangle not in A.
  Then T intersects at most a *constant number* of larger tris in A.
  More precisely, T intersects at most $3k$ larger tris from A.
- Proof: see the "Computational Geometry" course.

Populating the Hierarchy of 3D Grids

- Let $d(T)$ = diameter of circumcircle of triangle T, $d_{\text{min}} = \min\{d(T) \mid T \in A\}$
- Construct hierarchy of grids (partitioning the same bbox of the object)
- "Lowest" level has cell size d_{min}, next level has cell size $2 \cdot d_{\text{min}}$, etc.
- For T, determine its level l such that

$$2^{k-l} d_{\text{min}} \leq d(T) \leq 2^{k-l+1} d_{\text{min}}$$

- Insert T in all cells it occupies *on level* l
- I.e., cells of size c contain only triangles with $d \geq c$, but not $d \geq 2c$
- As usual, we store each level as a hash table



Checking One Polygon for Intersections

- Given polygon $p \in$ A, and hierarchy of grids containing polygons from B
- Traverse levels of grid upwards, until intersection is found or top level reached
-

```
checkIntersection( pgon p, multi-grid for B ):
determine level l for p
forall levels l .. lmax:
  forall cells ck on level l overlapping bbox(p):
    forall polygons qj in ck:
      check (p,qj) for intersection
```

The Complete Algorithm

- When checking polygons from A, consider only *larger* polygons in B
  - For checking polygons from A, build a multi-level 3D grid for all polygons from B
- Then check polygons from B against larger polygons in A

```
checkColl( obj A, obj B ):
in parallel forall pᵢ ∈ A:
  insertInMultiGrid( pᵢ )
in parallel forall qᵢ ∈ B:
  checkIntersection( qᵢ )
clear multi-grid
in parallel forall qᵢ ∈ B:
  insertInMultiGrid( qᵢ )
in parallel forall pᵢ ∈ A:
  checkIntersection( pᵢ )
```

Correctness

- If $p \in$ A and $q \in$ B intersect, then
  - Either, $q \leq p$ and the intersection will be found during the first upsweep phase;
  - Or, $p \leq q$ and detection occurs during second upsweep phase

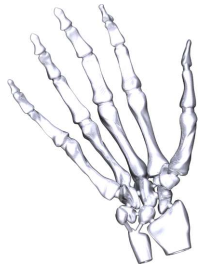
Complexity

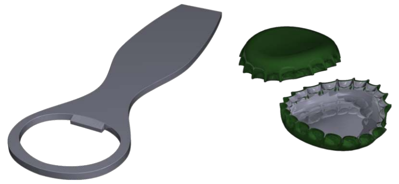
- Number of levels in the grid hierarchy: $O\left(\log \frac{d_{max}}{d_{min}}\right)$
  where d_{max} = biggest triangle (circumcircle, cell size)
- If A is k-free, then for each polygon in B, the upsweep is $O\left(\log \frac{d_{max}}{d_{min}}\right)$
- Same for the second phase
- In total, worst-case (sequential) complexity is $O\left(n \cdot \log \frac{d_{max}}{d_{min}}\right)$
- Assuming the ratio $d_{max}:d_{min}$ is bounded and we have $O(n)$ many concurrent threads available, then the parallel complexity is $O(1)$!
- We can use the algo even if we don't know k, or even if A,B are not k-free (just the complexity is not guaranteed any more)

Most Objects Are *K*-Free

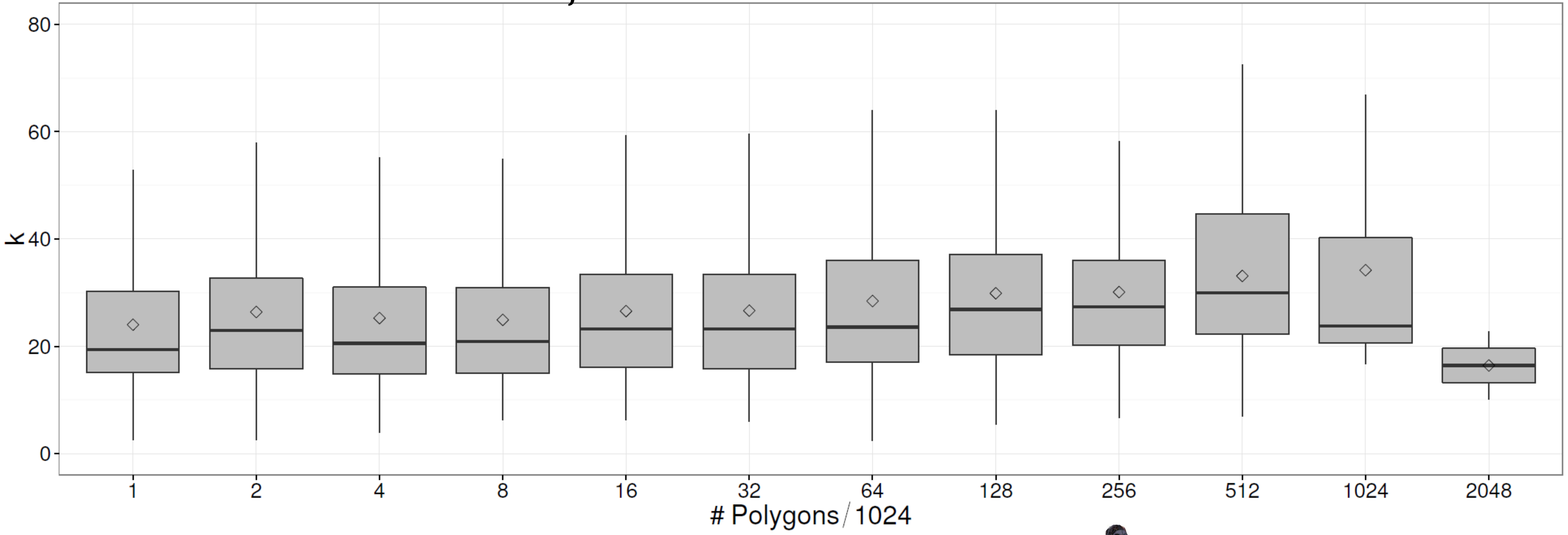








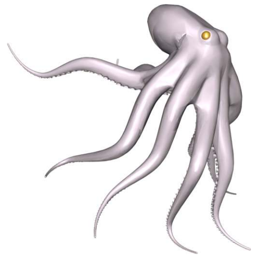
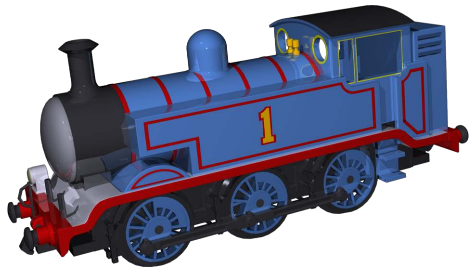
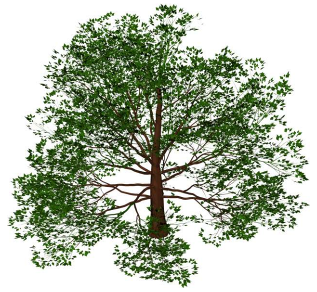
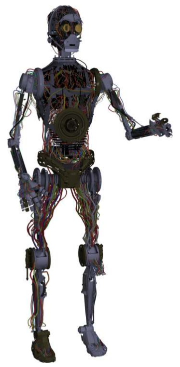

The *k* factors for 8000 objects from the 3D Meshes Research Database of INRIA





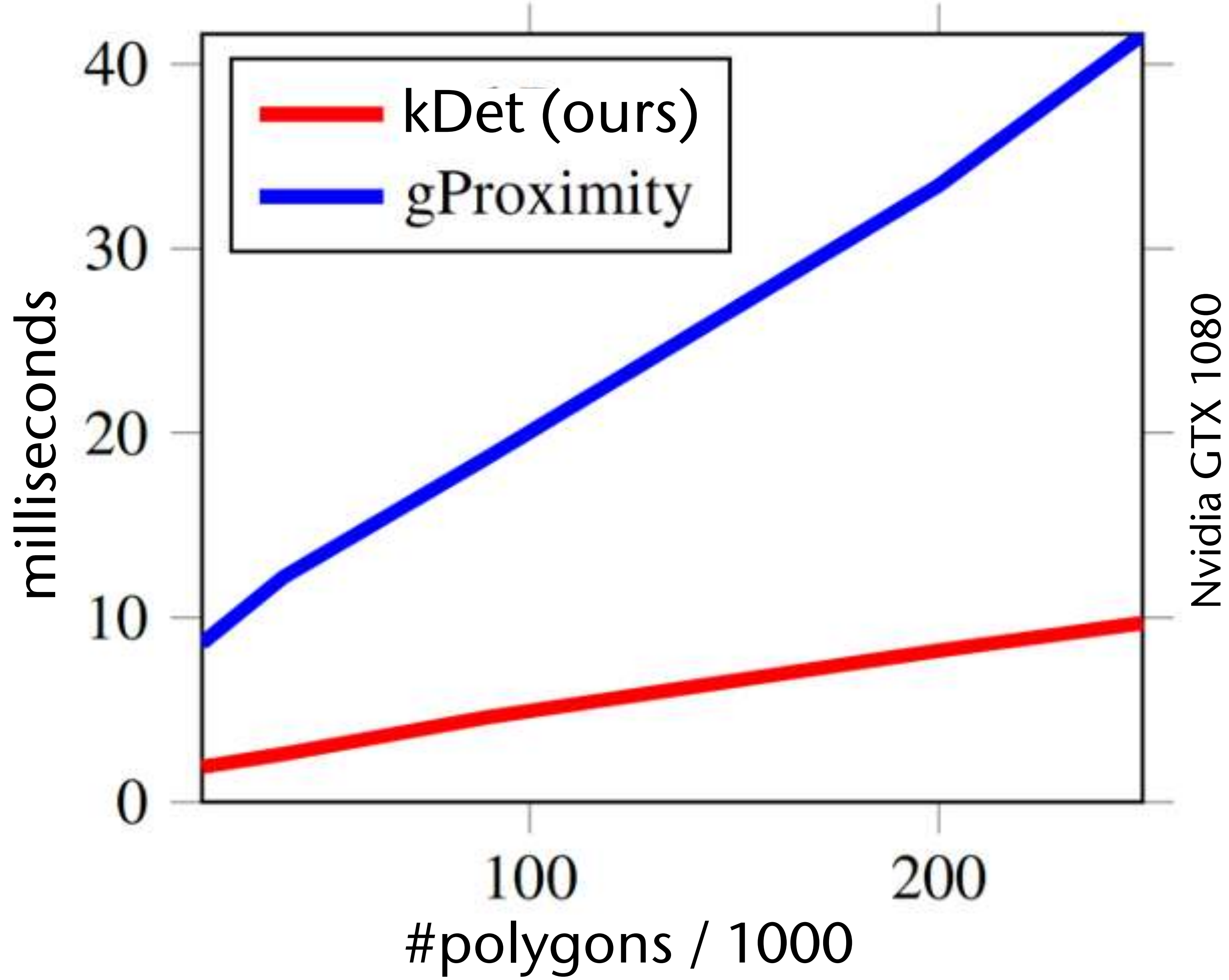




Actual Running Times

- Parallel time complexity: $O\left(\frac{n}{p}\right)$, where p = #processors / #threads



Master / Bachelor Thesis Topics

1. Re-implement kDet using modern CUDA, write beautiful code, optimize it
2. Extend to continuous collision detection (with obj motion)
3. Integrate (virtual) re-meshing to lower/achieve a good *k*-factor
4. Can you use the *k*-free property to build better BVH's?

- In case of questions: ask René Weller or me

Master / Bachelor Thesis Topics

CONFIDENTIAL

- Perform collision detection using machine learning
  - Use deep learning, or GLVQ
    - Can it be done in 1 milliseconds ?!
  - For rigid objects first, then deformable, or continuous collision detection

Master Thesis Topic

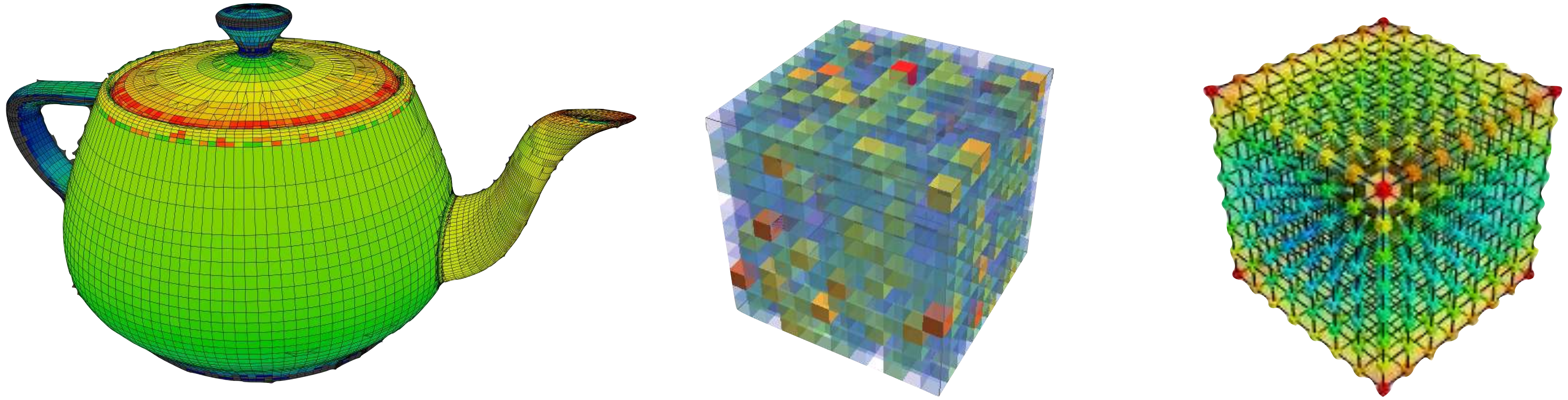
CONFIDENTIAL

- Natural manipulation of virtual objects using the virtual hand
- Use (our) collision detection as a basic building block
- Challenge: no force-feedback
- Approach: non-linear optimization
  - Determine position of dynamic object so as to minimize penetration of the virtual hand
  - Potentially combine with control algorithm (PID, Ricatti) to increase stability
  -

Master / Bachelor Thesis Topics

CONFIDENTIAL

- Client-server system allowing people to check the "coll.det.-readiness" of their geometry
  - Client uploads object via browser
  - Server performs benchmark
  - Gathers statistics and creates heat map
  - Send results back to client
  - Client can view results in browser



Potential ways to visualize the heat map

Master / Bachelor Thesis Topics

CONFIDENTIAL

- Problem: packing arbitrary objects in arbitrary containers
- Applications: fine art, 3D printing
- Special constraints:
  - Various types of objects - should not form clusters
  - Percentage of object types is user-defined
- Especially for the arts application:
  - Increase surface density
  - Make inner / occluded region of container "hollow" (saves material)