Virtual Reality & Physically-Based Simulation

Collision Detection

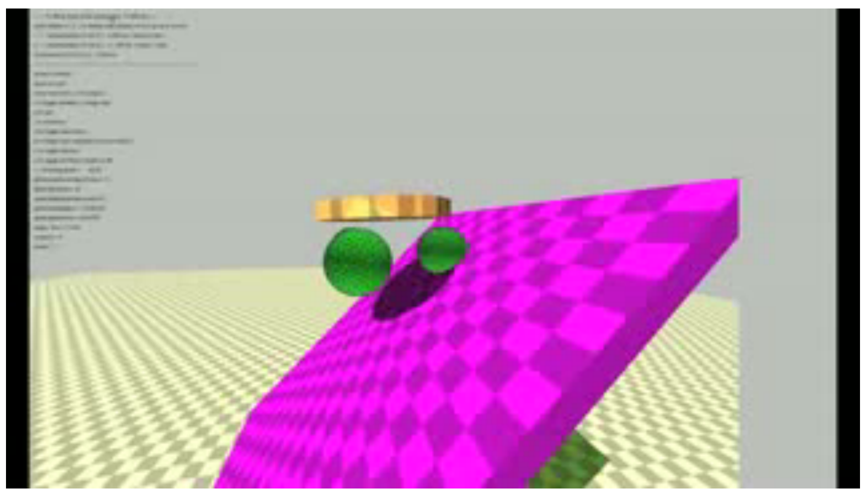
G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

Examples of Applications

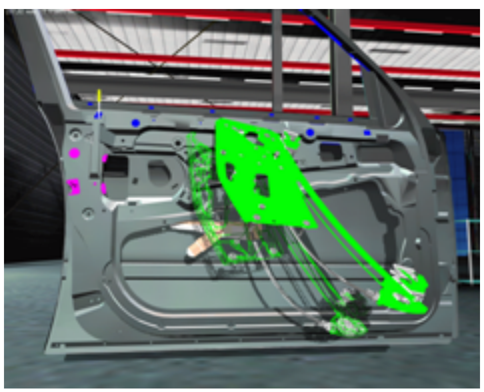
Virtual Prototyping

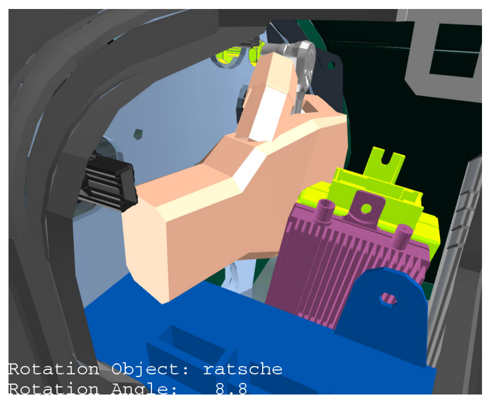


Physically-based simulation

Application Areas for Collision Detection

- Collision detection is an enabling technology for:
  - Physically-based simulation
  - Interaction in VR
  - Haptic rendering
- Application areas:
  - Games, animation, surgery, virtual prototyping, path planning, online robot collision avoidance





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

- In this chapter: only collision detection

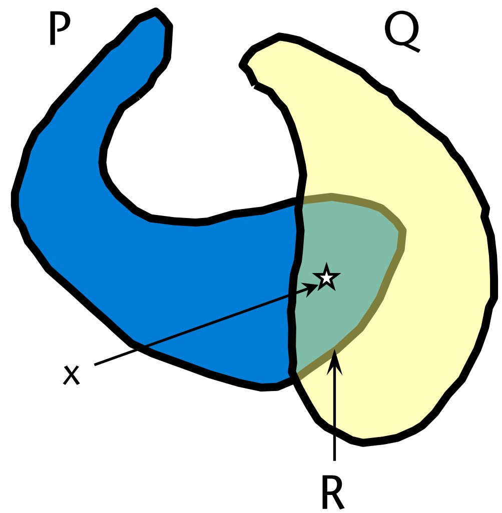
Definitions

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

  "P and Q collide" $:\Leftrightarrow$

  $P \cap Q \neq \varnothing \Leftrightarrow$

  $\exists x \in^3 : x \in P \wedge x \in Q$



- The construction problem:

  compute $R := P \cap Q$

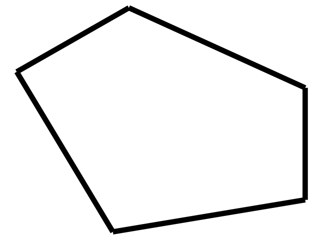
- For polygonal objects we define collisions as follows:

  P,Q collide $\Leftrightarrow \quad \exists f \in F^P \exists f' \in F^Q : f \cap f' \neq \varnothing$

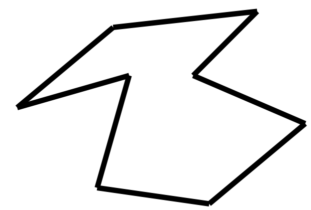
- The games community often has a different definition of "collision"

Objekt-Klassen

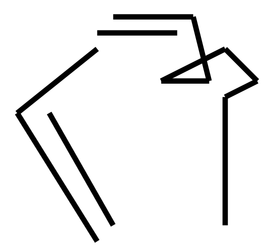
- Convex
- Closed and simple
  (no self-penetrations)
- Polygon soups
  - Not necessarily closed
  - Duplicate polygons
  - Coplanar polygons
  - Self-penetrations
  - Degenerate cardigans
  - Holes
- Deformable



konvex

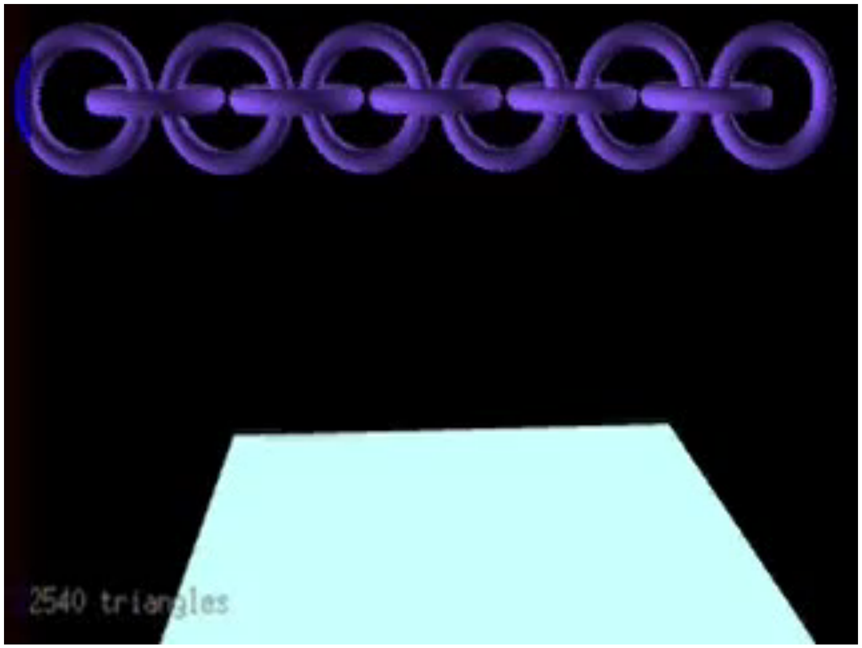


einfach & geschlossen

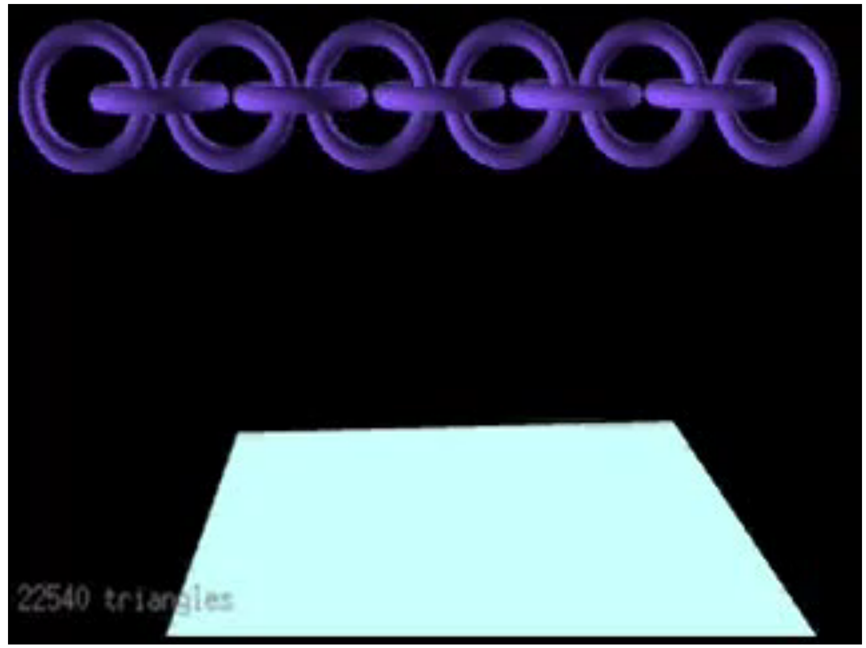


polygon soup

Importance of the Performance of Collision Detection



naïve algorithm
(test all pairs of polygons)



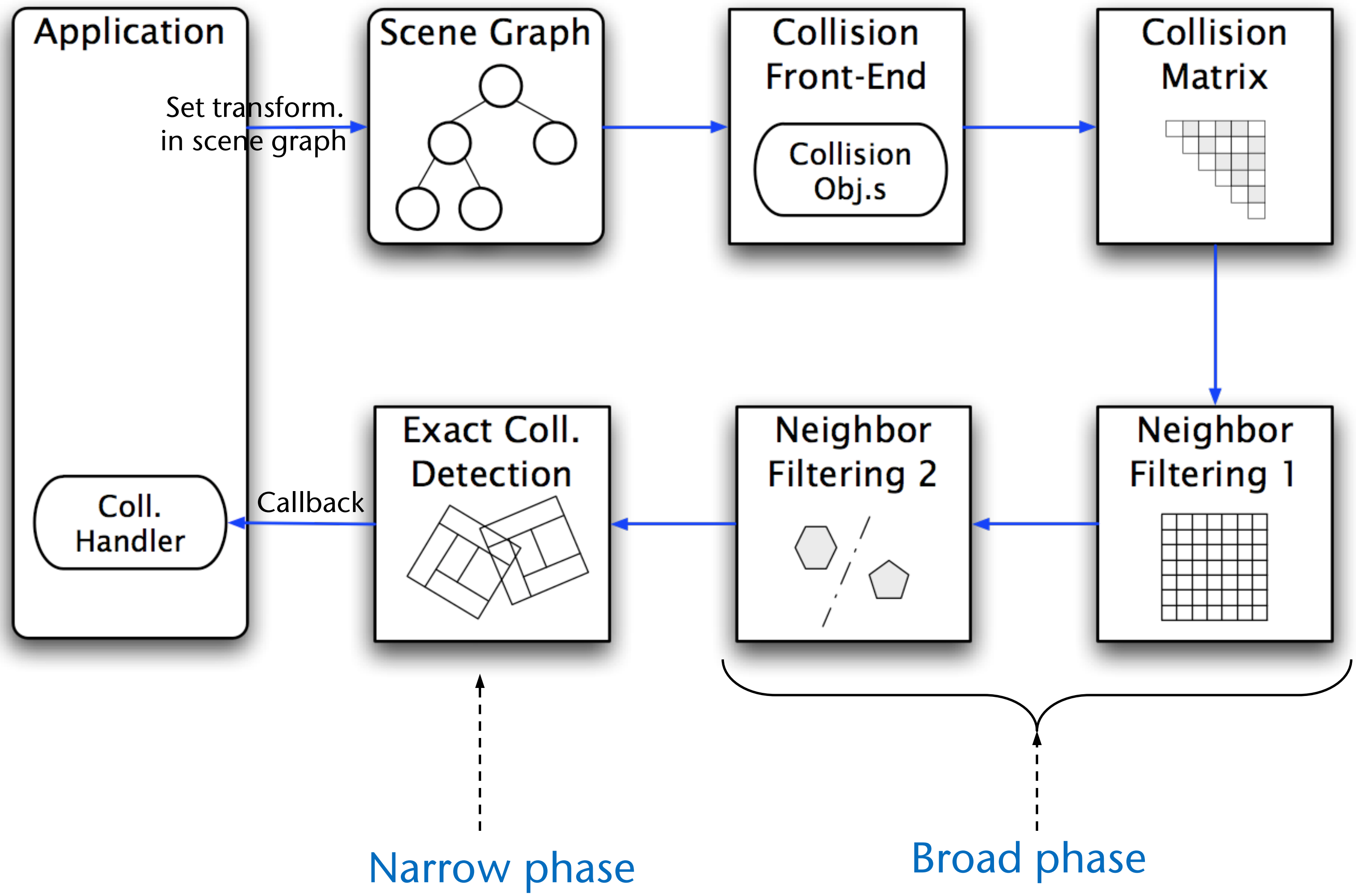
clever algorithm
(use bbox hierarchy)

Conclusion: the performance of the algorithm for collision detection determines (often) the overall performance of the simulation!

Requirements on Collision Detection

- Handle a large class of objects
- Lots of moving objects (some 1000)
- 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 to a large zu große zusätzliche Datenstrukturen (<2x);
  - 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 Interest 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

  ⇒ Collision Interest Matrix

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								

- 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

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
- 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
- Idea: try to determine neighbors (i.e., close objects) very quickly
- ➢ 3D grid, sweep plane, etc.

The 3D Grid

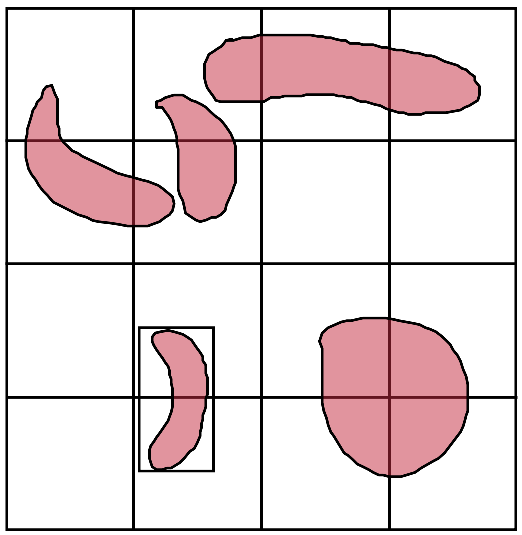
Idea:

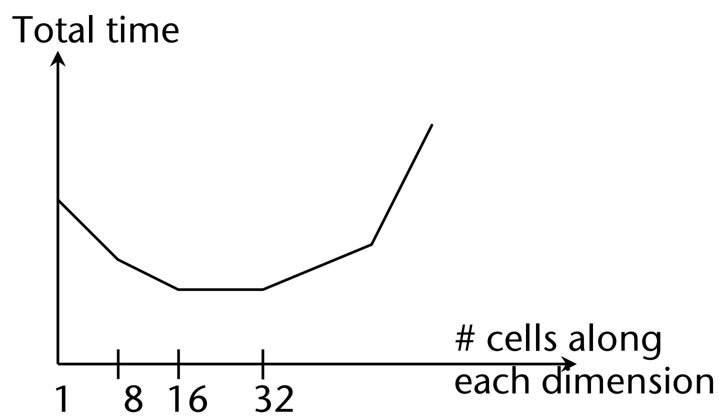
1. Partition the "universe" by a grid
2. Objects are considered neighbors, if they occupy the same cell
3. Determine cell occupancy by bbox
4. When objects move → update grid

- Neighbor-finding = find all cells that contain more than one bbox
  - Data structure here: hash table (!)
  - Collision in hash table → probably neighbor

The trade-off:

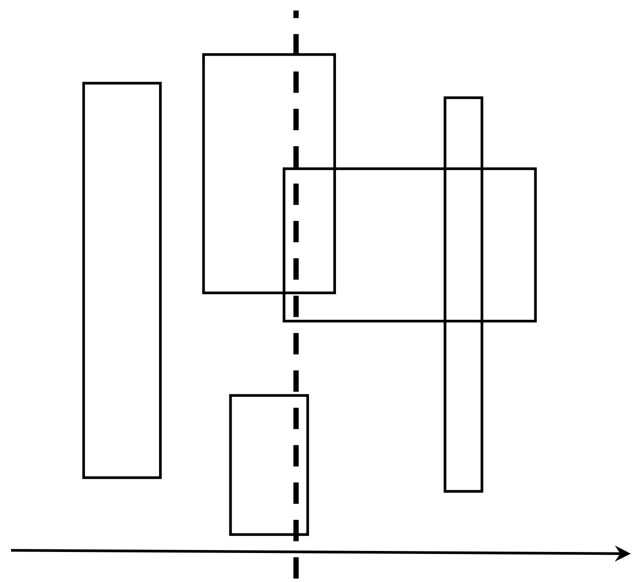
- Fewer cells = larger cells
  - ➢ Distant objects are still "neighbors"
- More cells = smaller cells
  - ➢ Objects occupy more cells
  - ➢ Effort for updating increases





The Plane Sweep Technique (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
  jump from box border to box border:
    if current box border is the left side (= "opening"):
      add this box to the list of active boxes
      check the current box against all others in the active list
    else (= "closing"):
      remove this box from the list of active boxes



Frame-to-Frame Coherence

- Observation:

  Two consecutive images in a sequence differ only by very little (usually).

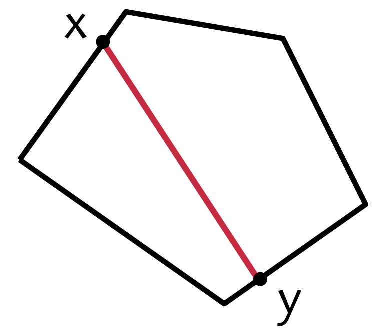
➢ Terminology: frame-to-frame or temporal coherence

- Examples:
  - Motion of a camera
  - Motion of objects in a film / animation
- Applications:
  - Computer Vision (e.g. tracking of markers)
  - MPEG
  - Collision detection
  - Ray-tracing of animations (e.g. using kinetic data structures)
- Algorithms based on frame-to-frame coherence are called "incremental", sometimes "dynamic" or "online" (the latter is actually the wrong term)

Convex Objects

- Definition of “convex polyhedron”:

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

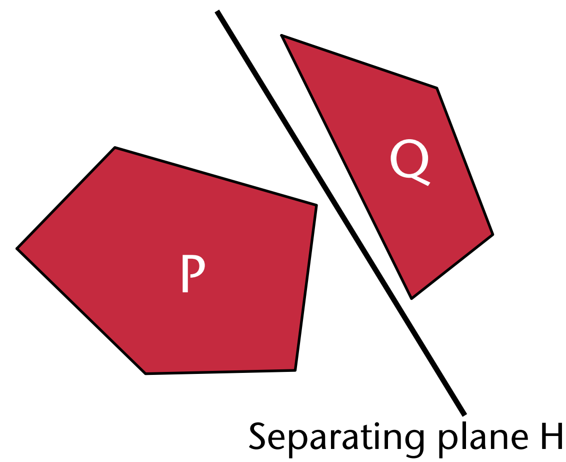


- A condition for "non-collision":

  P and Q are “linearly separable” $\Leftrightarrow$

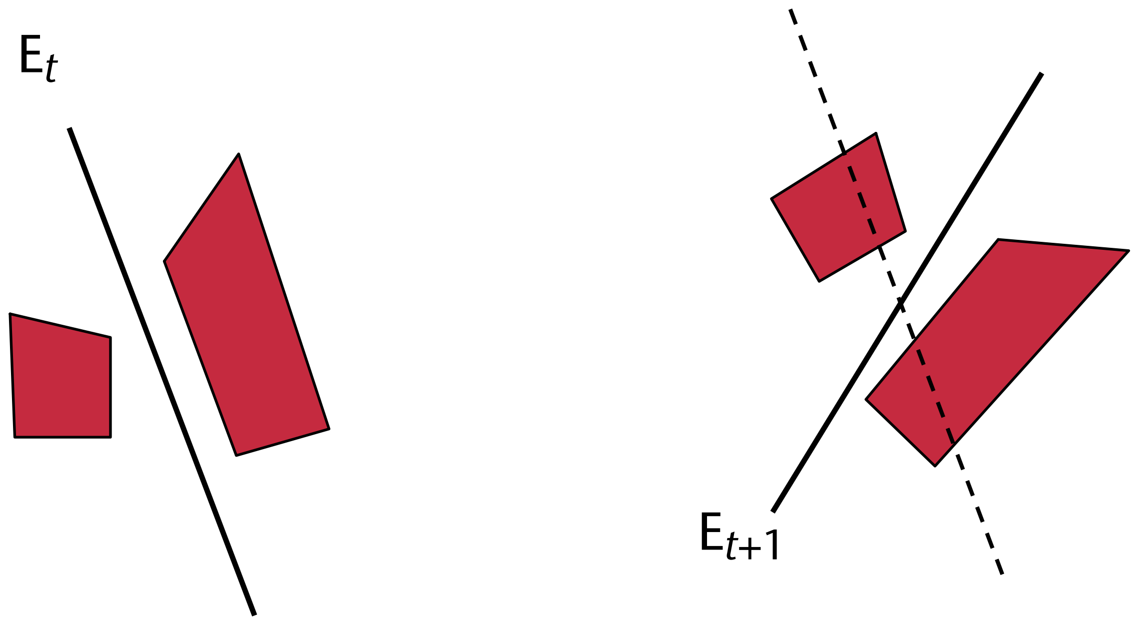
  $$\exists \text{ half-space } H : P \subseteq H \wedge Q \subseteq H^c$$

  (“P is completely to one side of H,
  Q completely on the other side”)



The Algorithm “Separating Planes”

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



load E_t = separating plane between P & Q at time t

$E := E_t$

repeat max n times

    if exists $v \in \text{vertices}(P)$ on the back side of E:

        rot./transl. E such that v is now on the front side of E

    if exists $v \in \text{vertices}(Q)$ on the front side of E:

        rot./transl. E such that v is now on the back side of E

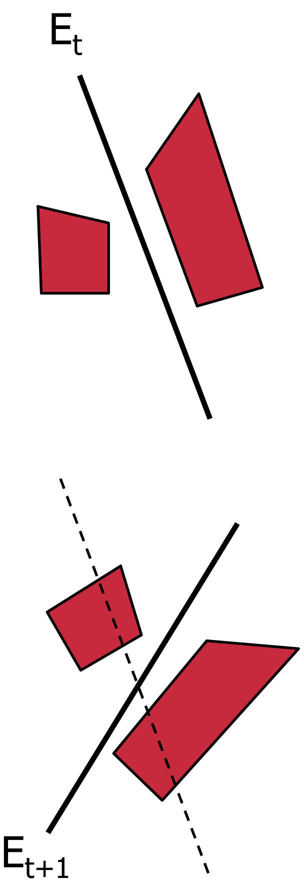
    if there are no vertices on the "wrong" side of E, resp.:

        return "no collision"

if there are still vertices on the "wrong" side of E:

    return "collision" {could be wrong}

save E_{t+1} := E for the next frame



For details on the "rot./transl. E" step → see perceptron learning algorithm

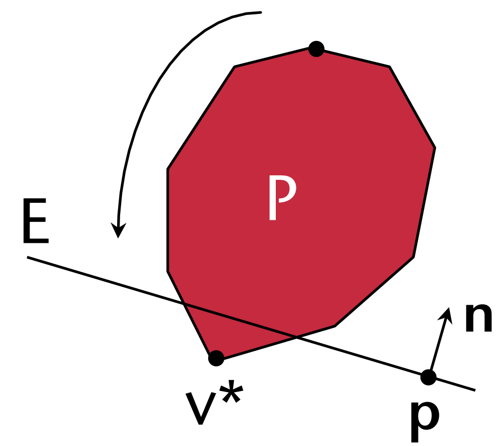
How to Find a Vertex on the "Wrong" Side *Quickly*

- The brute-force method:

  test all **v** whether $f(\mathbf{v}) = (\mathbf{v} - \mathbf{p})\cdot\mathbf{n} > 0$

- Observation:

  1. *f* is linear,
  2. P is convex ⇒ *f*(*x*) has (usually) exactly one minimum over all points *x* on the surface of P
  3. $\exists^1 \mathbf{v}^* \;:\; f(\mathbf{v}^*) = \min$

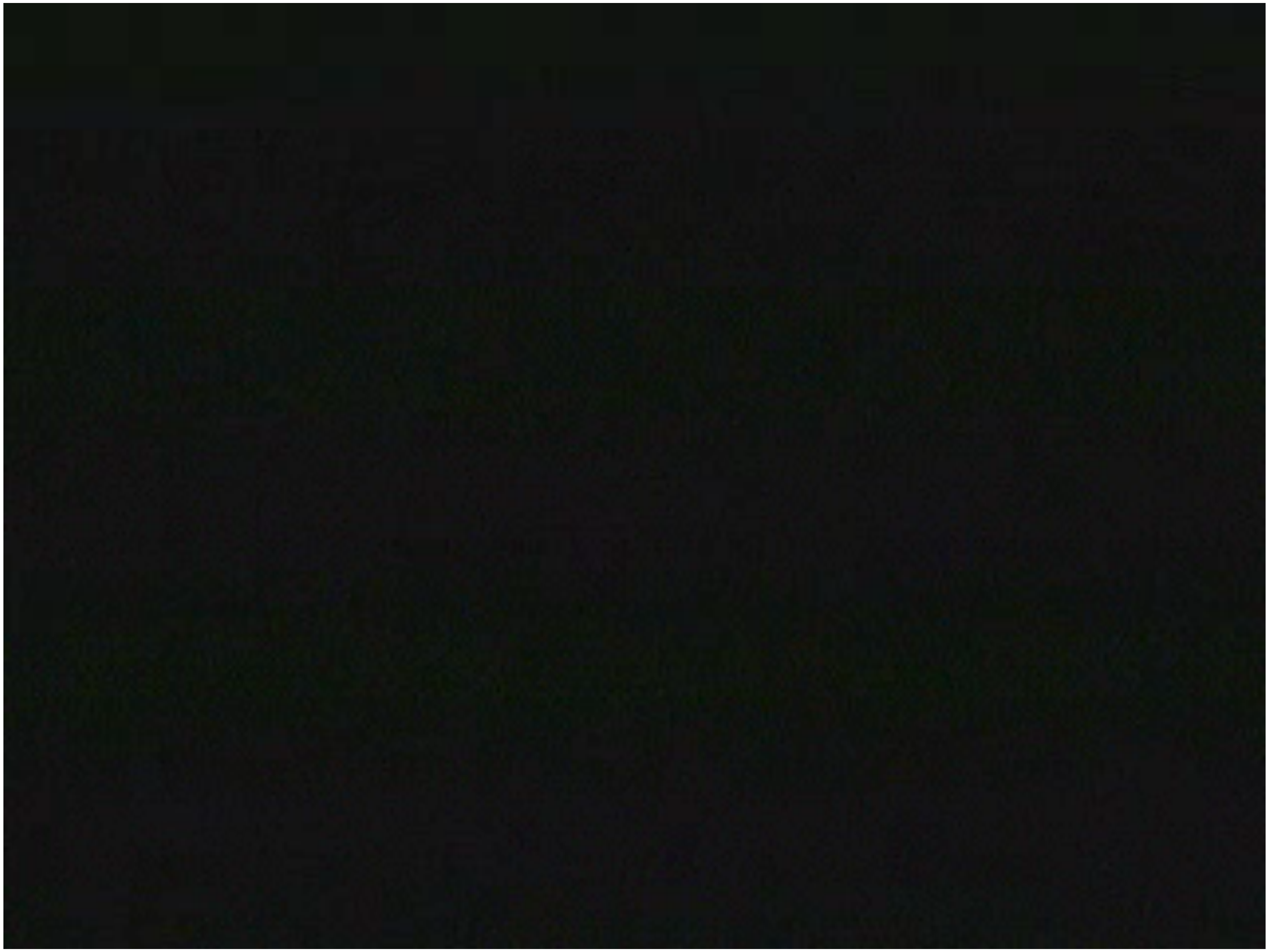


- The algorithm (steepest descent on the surface w.r.t. *f*):
  - Start with an arbitrary vertex **v**
  - Walk to the neighbor **v'** of **v** for which $f(\mathbf{v}') = \min.$ (among all neighbors)
  - Stop if there is no neighbor **v'** of **v** for which $f(\mathbf{v}') < f(\mathbf{v})$

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



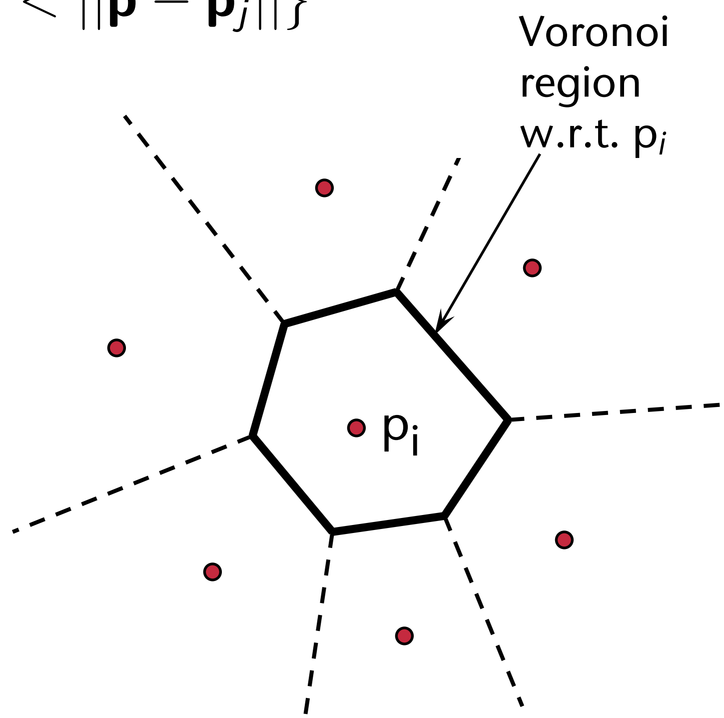
Closest Feature Tracking

- Proposed by Lin & Canny in 1992 (→ "Lin-Canny-Algorithm")
- Idea:
  - Maintain the minimal distance between a pair of objects
  - Which is realized by one point on the surface of each object
  - If the objects move continuously, then those points move continuously on the surface of their objects
- The algorithm is based on the following methods:
  - Voronoi diagrams
  - The “closest features” lemma

Voronoi Diagrams for Point Sets

- Given a set of points $S = \{\mathbf{p}_i\}$, called sites (or generators)
- Definition of a Voronoi region/cell :

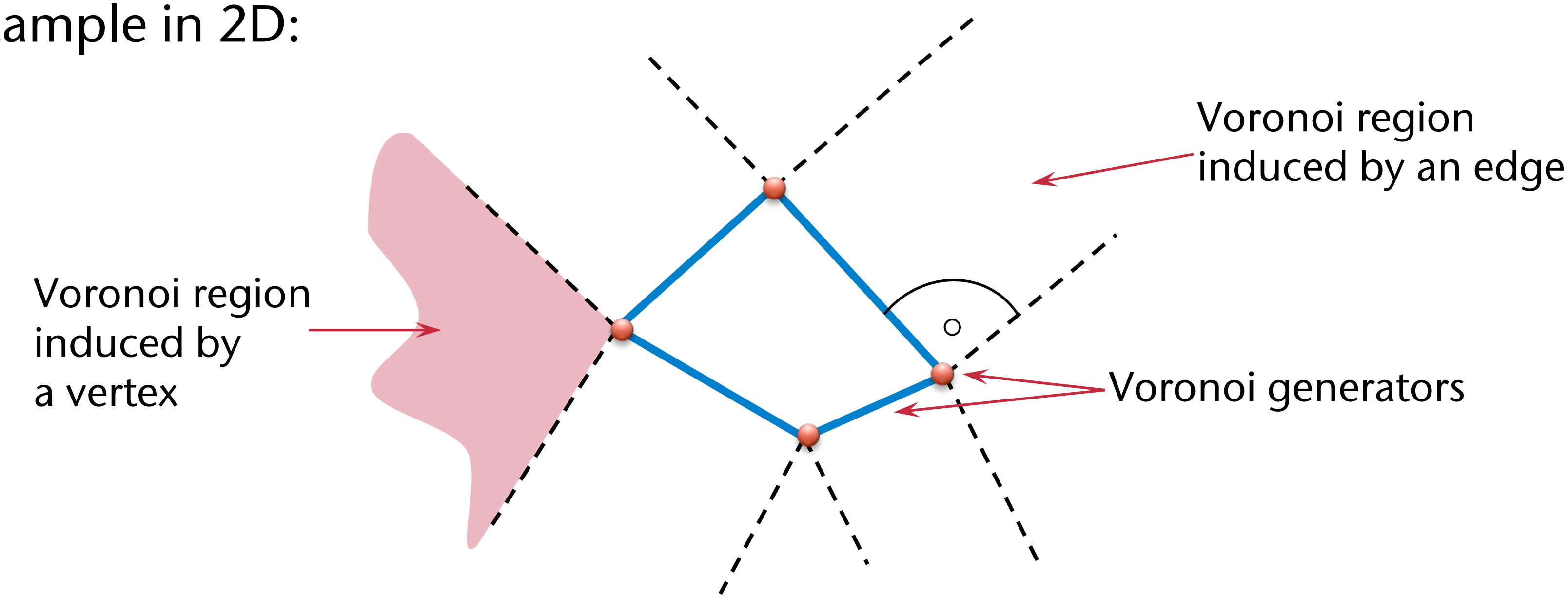
$$V(p_i) := \{\mathbf{p} \in \mathbb{R}^2 \mid \forall j \neq i : ||\mathbf{p} - \mathbf{p}_i|| < ||\mathbf{p} - \mathbf{p}_j||\}$$



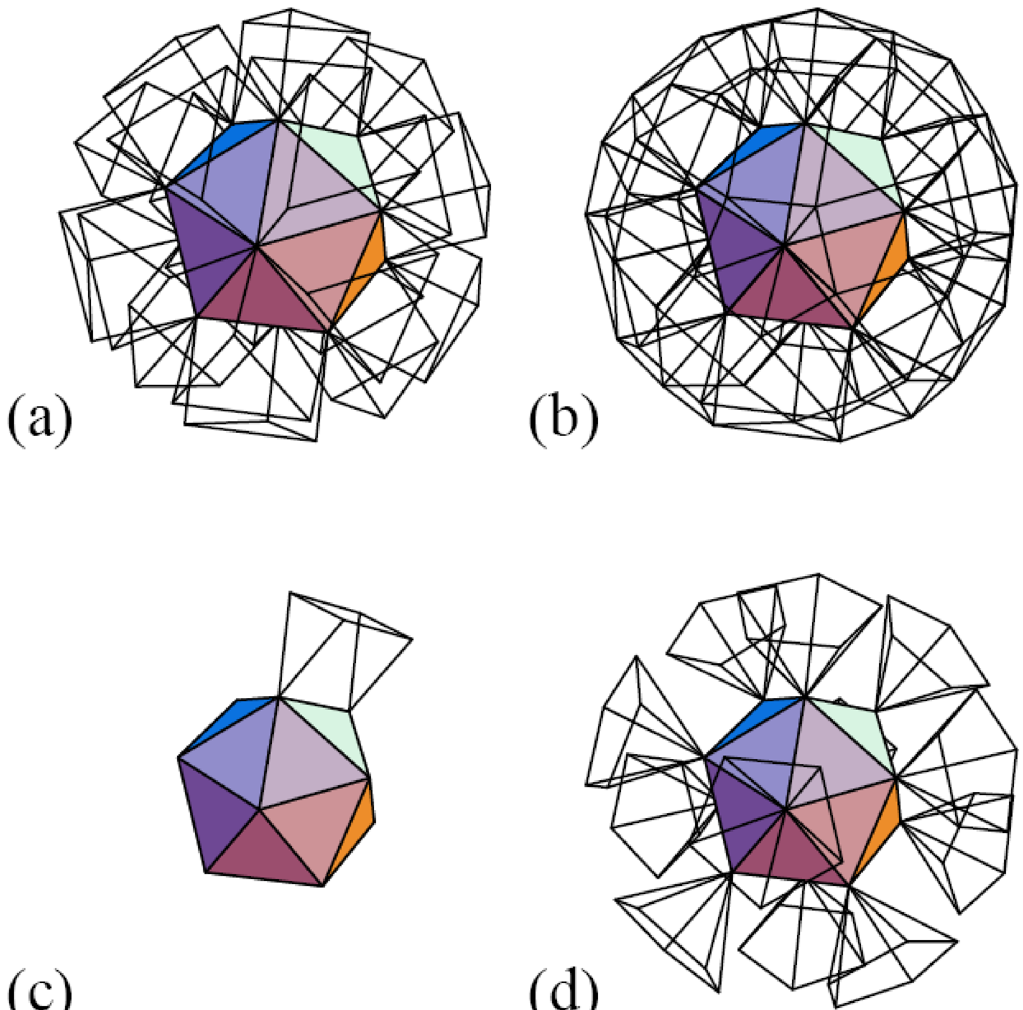
- Definition of Voronoi diagrams:
  The Voronoi diagram $\mathcal{VD}(S)$ over a set of points S is the union of all Voronoi regions over the points in S.
- $\mathcal{VD}(S)$ induces a partition of the plane into Voronoi edges, Voronoi nodes, and Voronoi regions
- Interaktive Demo: http://web.cs.uni-bonn.de/I/GeomLab/VoroGlide/

Voronoi Diagrams over Sets of Points, Edges, Polygons

- Voronoi diagrams can be defined analogously in 3D (and higher dimensions)
- What if the generators are not points but edges / polygons?
- Definition of a Voronoi cell is still the same:
  The Voronoi region of an edge/polygon := all points in space that are closer to "their" generator than to any other
- Example in 2D:



Outer Voronoi Regions Generated by a Polyhedron



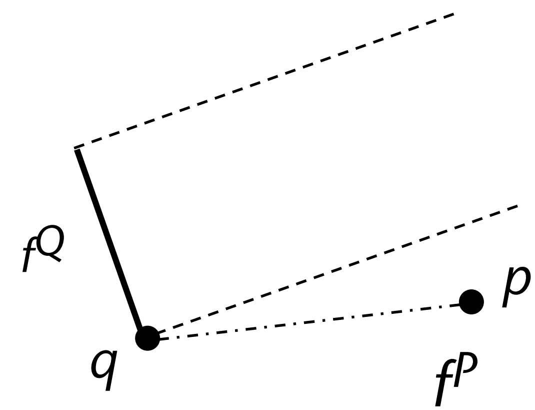
The external Voronoi regions of ...

(a) faces
(b) edges
(c) a single edge
(d) vertices

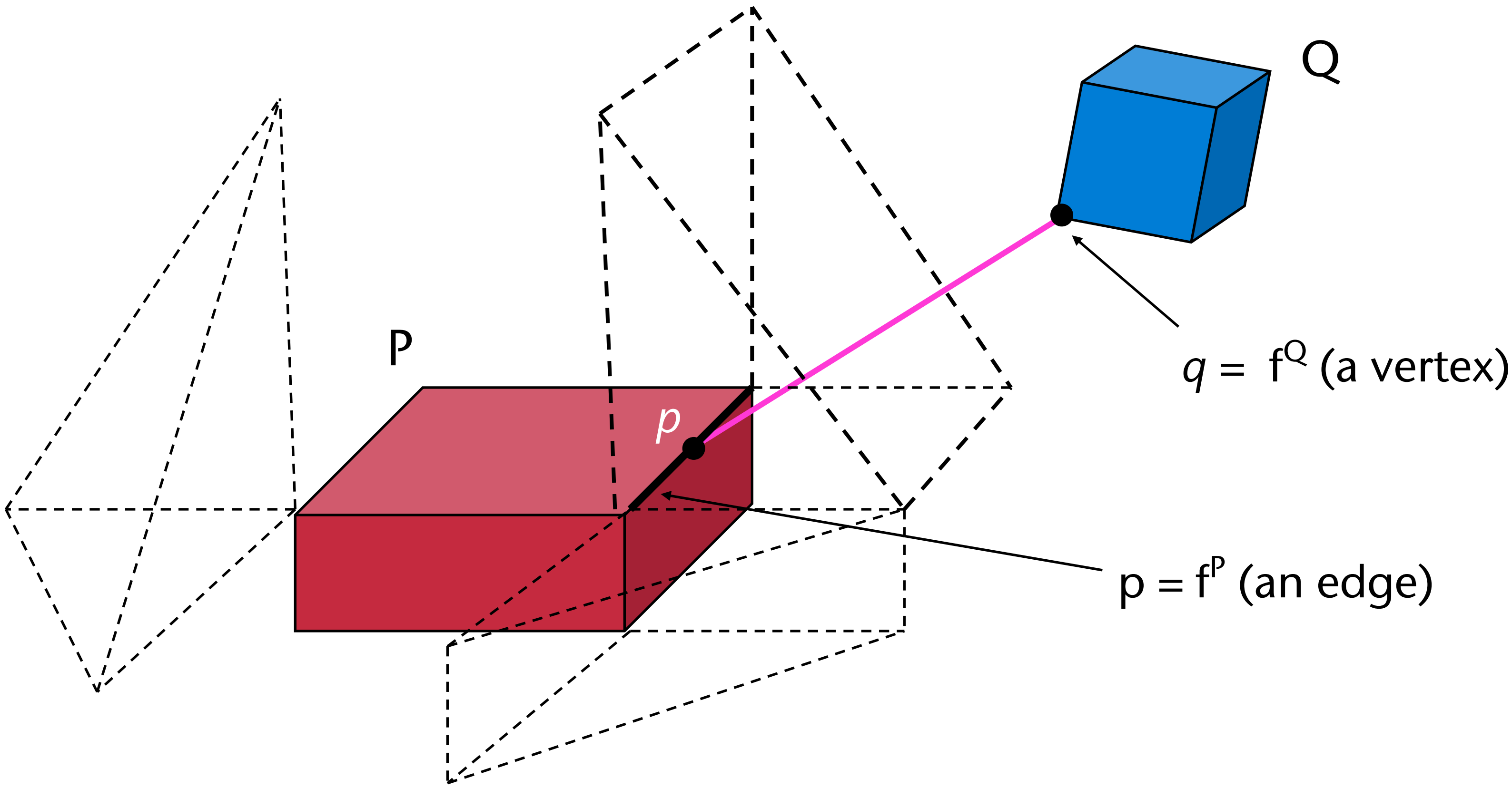
Outer Voronoi regions for convex polyhedra can be constructed very easily!
(We won't need inner Voronoi regions.)

Closest Features

- Definition *Feature* f^P := a vertex, edge, polygon of polyhedron P.
- Definition "Closest Feature":
  Let f^P and f^Q be two features on polyhedra P and Q, resp., and let p, q be points on f^P and f^Q , resp., that realize the minimal distance between P and Q, i.e.
  $$d(P,Q) = d\left(f^P, f^Q\right) = ||p - q||$$
  Then f^P and f^Q are called "closest features".
- The "closest feature" lemma:
  Let $V(f)$ denote the Voronoi region generated by feature f; let p and q be points on the surface of P and Q realizing the minimal distance. Then
  f^P, f^Q are closest features $\Leftrightarrow$ p is in $V(f^Q)$, q is in $V(f^P)$.



Example



The Algorithm (Another Kind of a Steepest Descent)

Start with two arbitrary features f^P, f^Q on P and Q, resp.

while (f^P, f^Q) are not (yet) closest features and dist(f^P, f^Q) > 0 :

  if (f^P,f^Q) has been considered already:

    return “collision” (b/c we've hit a cycle)

  compute p and q that realize the distance between f^P and f^Q

  if $p \in V(q)$ und $q \in V(p)$:

    return “no collision”, (f^P,f^Q) are the closest features

  if p lies on the "wrong" side of $V(q)$:

    f^P := the feature on that "other side" of $V(q)$

  do the same for q, if $q \notin V(p)$

if dist(f^P, f^Q) > 0 :

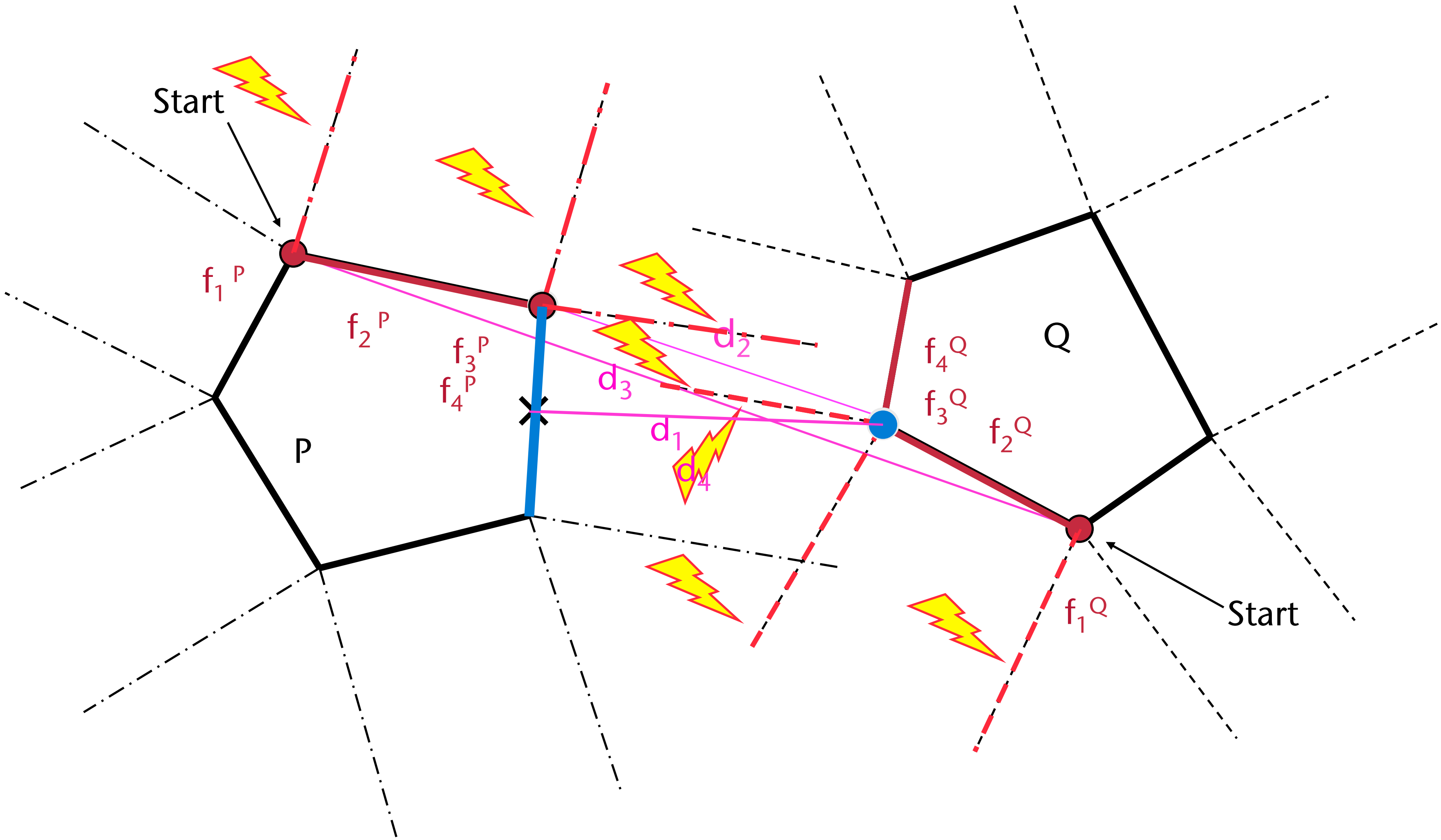
  return "no collision"

else

  return "collision"

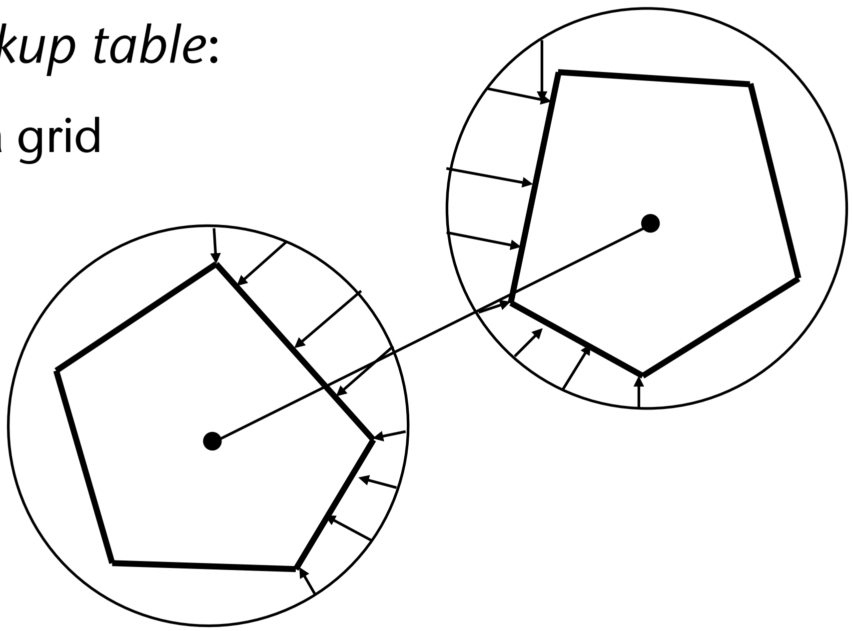
Notice: in case of collision, some features are inside the other object, but we did not compute Voronoi regions inside obnjects!
→ hence the chance for cycles

Animation of the Algorithm

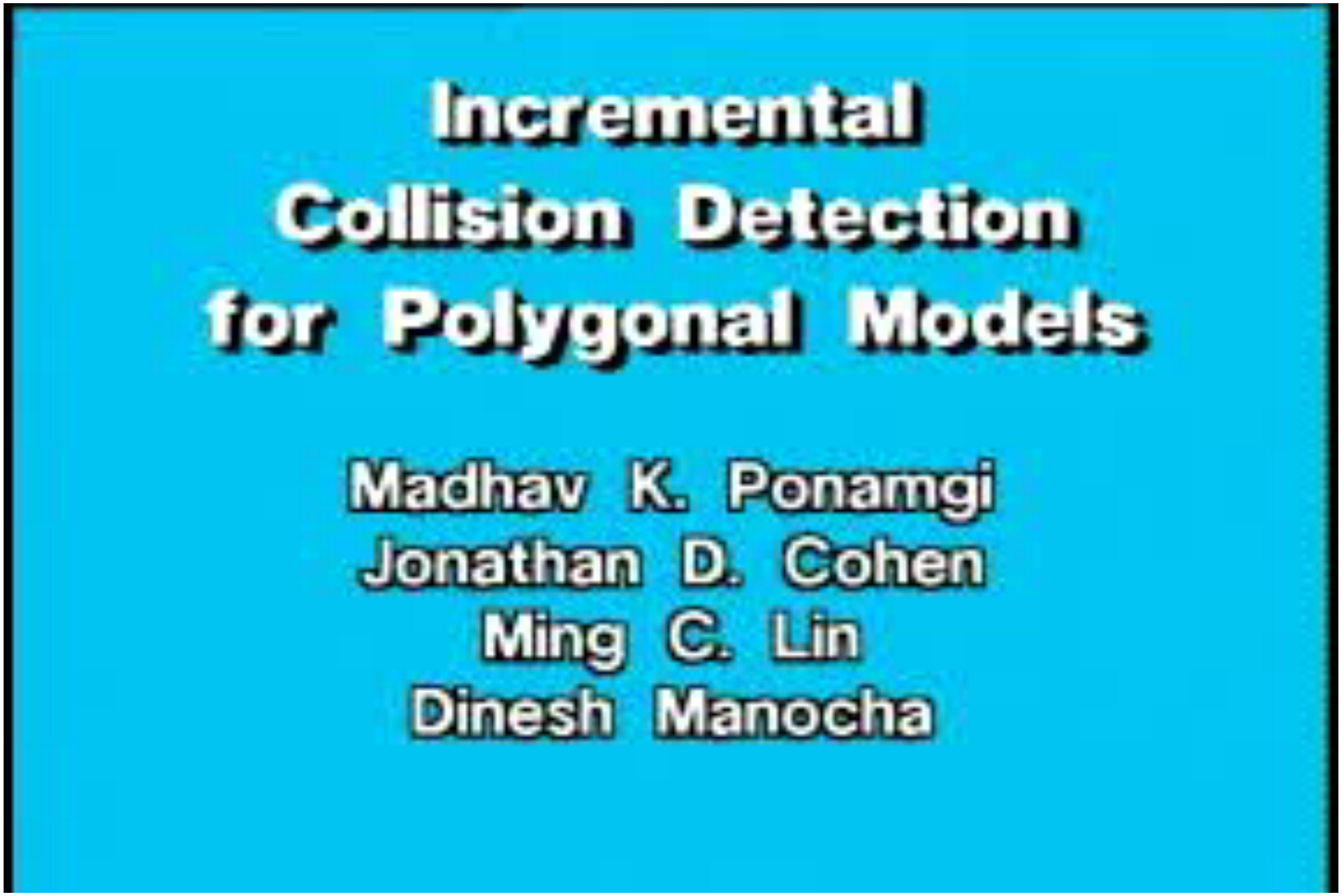


Some Remarks

- A little question to make you think:
  Actually, we don't really need the *Voronoi diagram!*
  (but with a *Voronoi diagram,* the algorithm is faster)
- The running time (in each frame) depends on the "degree" of temporal coherence
- Better initialization by using a *lookup table*:
  - Partition a surrounding sphere by a grid
  - Put each feature in each grid cell that it covers when propjected onto the sphere
  - Connect the two centers of a pair of objets by a line segment
  - Initialize the algorithm by the features hit by that line



Movie


Incremental
Collision Detection
for Polygonal Models
Madhav K. Ponamgi
Jonathan D. Cohen
Ming C. Lin
Dinesh Manocha

The Minkowski Sum

- Hermann Minkowski (1864 – 1909), German mathematician and physicist
- Definition (Minkowski Sum):
  Let A and B be subsets of a vector space;
  the Minkowski sum of A and B is defined as

$$A \oplus B = \{\mathbf{a} + \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}$$

- Analogously, we define the Minkowski difference:

$$A \ominus B = \{\mathbf{a} - \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}$$

- Clearly, the connection between Minkowski sum and difference:

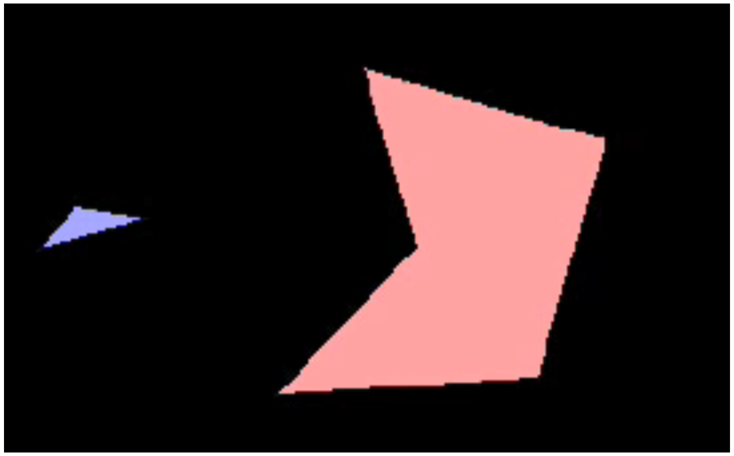
$$A \ominus B = A \oplus (-B)$$

- Applications: computer graphics, computer vision, linear optimization, path planning in robotics, …

Some Simple Properties

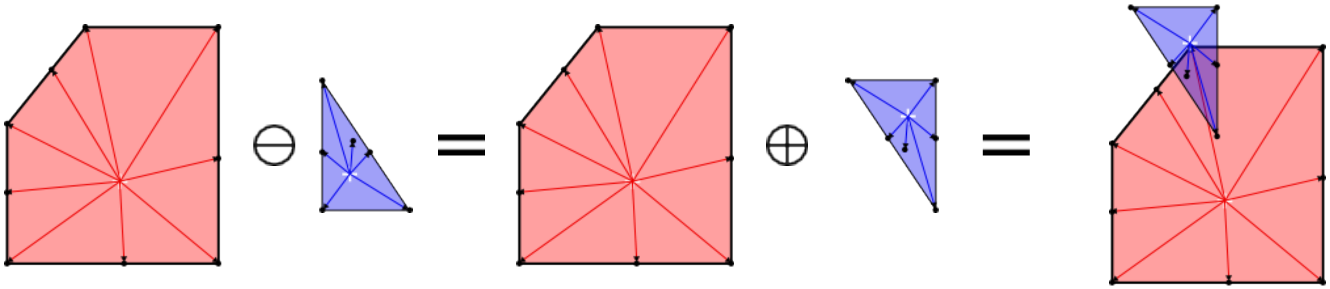
- Commutative: $A \oplus B = B \oplus A$
- Associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- Distributive w.r.t. set union: $A \oplus (B \cup C) = (A \cup B) \oplus (A \cup C)$
- Invariant against translation: $T(A) \oplus B = T(A \oplus B)$

- Intuitive "computation" of the Minkowski sum/difference:

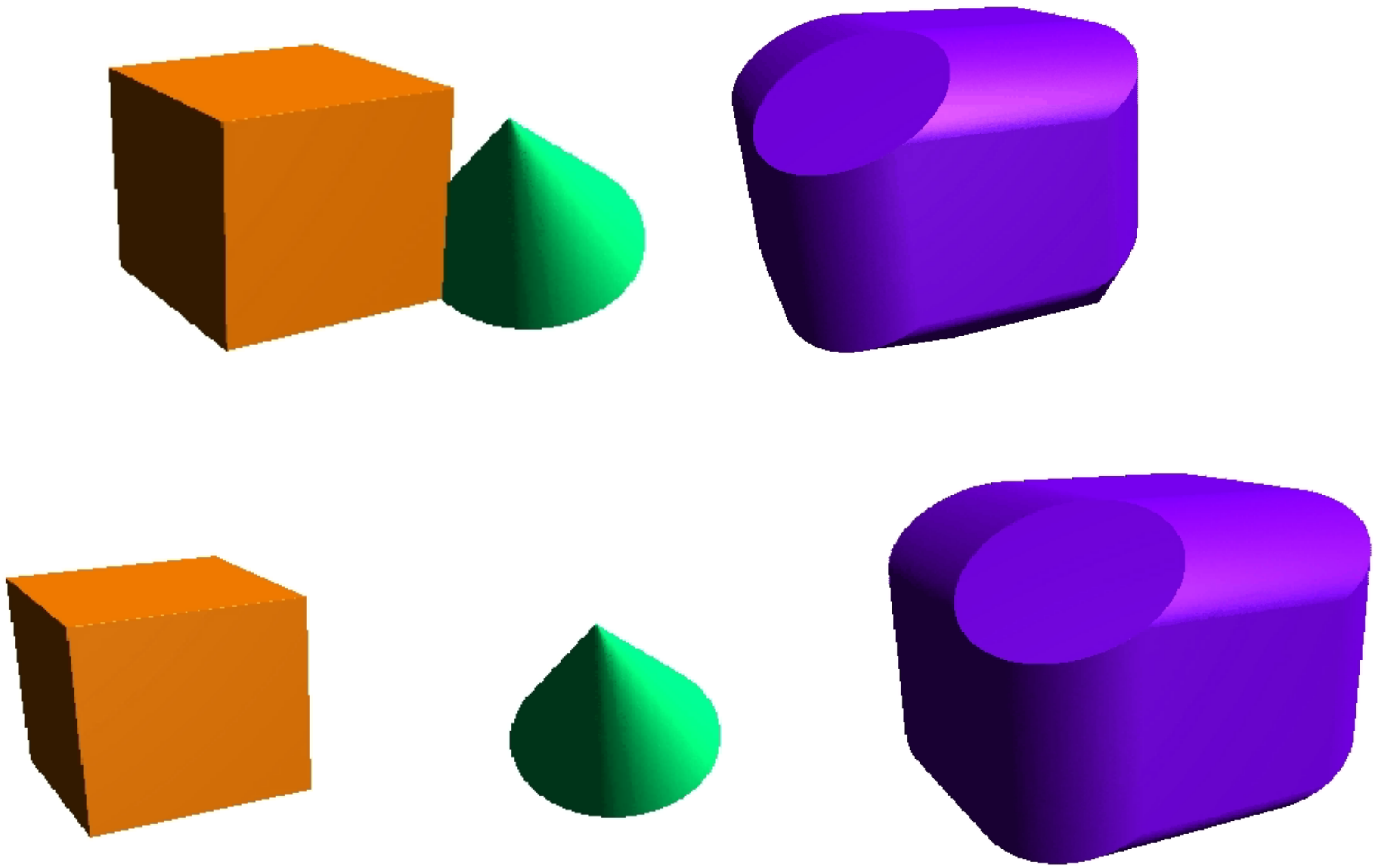


  - Warning: the yellow polygon in the animation shows the Minkowsi sum **modulo**(!) possible translations!

$$A \ominus B = A \oplus -B = C$$



Visualizations of a Simple Example



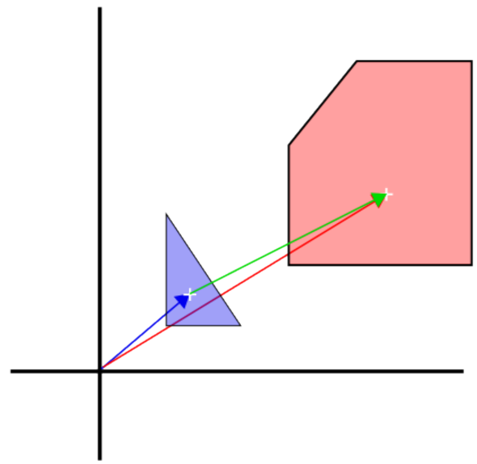
The Complexity of the Minkowski Sum (in 2D)

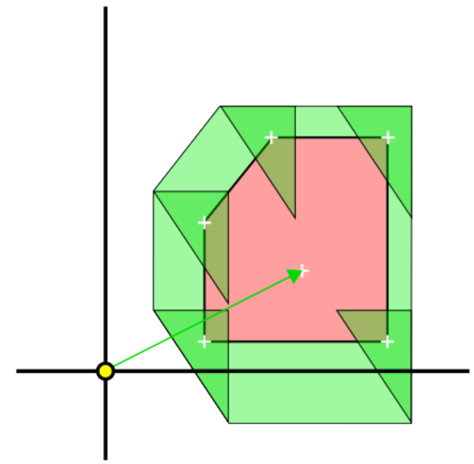
- Let A and B be polygons with n and m vertices, resp.:
  - If both A and B are convex,then $A \oplus B$ is convex, too, and has complexity $O(m+n)$
  - If only B is convex, then $A \oplus B$ has complexity $O(mn)$
  - If neither is convex, then $A \oplus B$ has complexity $O(m^2 n^2)$
- Algorithmic complexity of the computation of $A \oplus B$:
  - If A and B are convex, then $A \oplus B$ can be computed in time $O(m+n)$
  - If only B is convex, then $A \oplus B$ can be computed in randomized time $O(mn\, log^2(mn))$
  - If neither is convex, then $A \oplus B$ can be computed in time $O(mn^2\, log(mn))$

An Intersection Test for Two Convex Objects using Minkowski Sums

- Translate both objects so that the coordinate system's origin 0 is inside *B*
- Compute the Minkowski difference
- A and B intersect ⇔

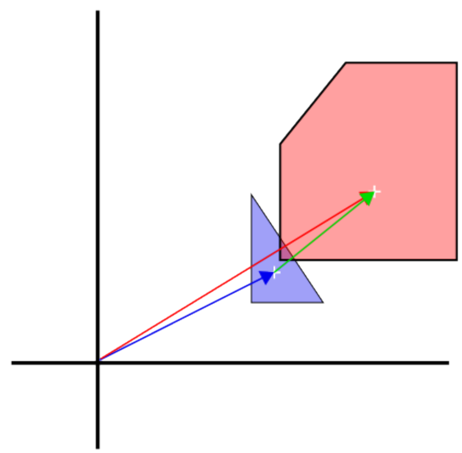
$$0 \in A \ominus B$$

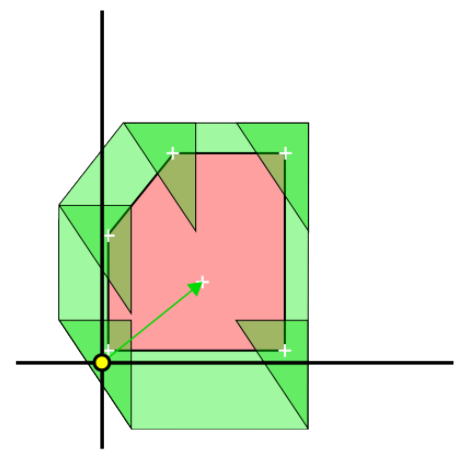




$A \ominus B = A \oplus -B = C$

- Example where an intersection occurs:

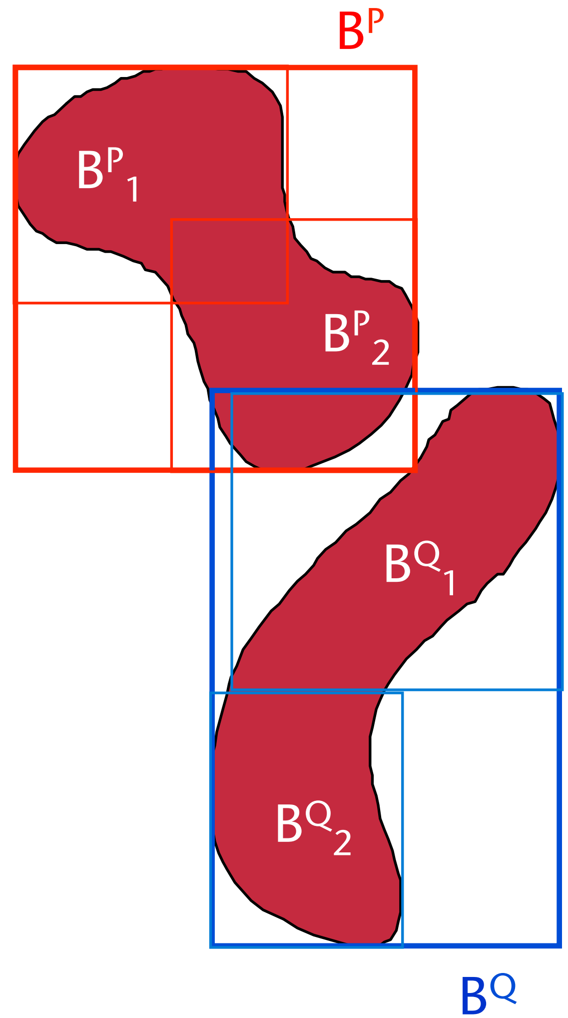




$A \ominus B = A \oplus -B = C$

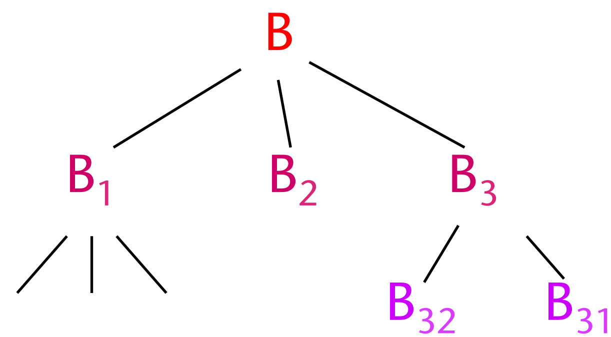
Hierarchical Collision Detection

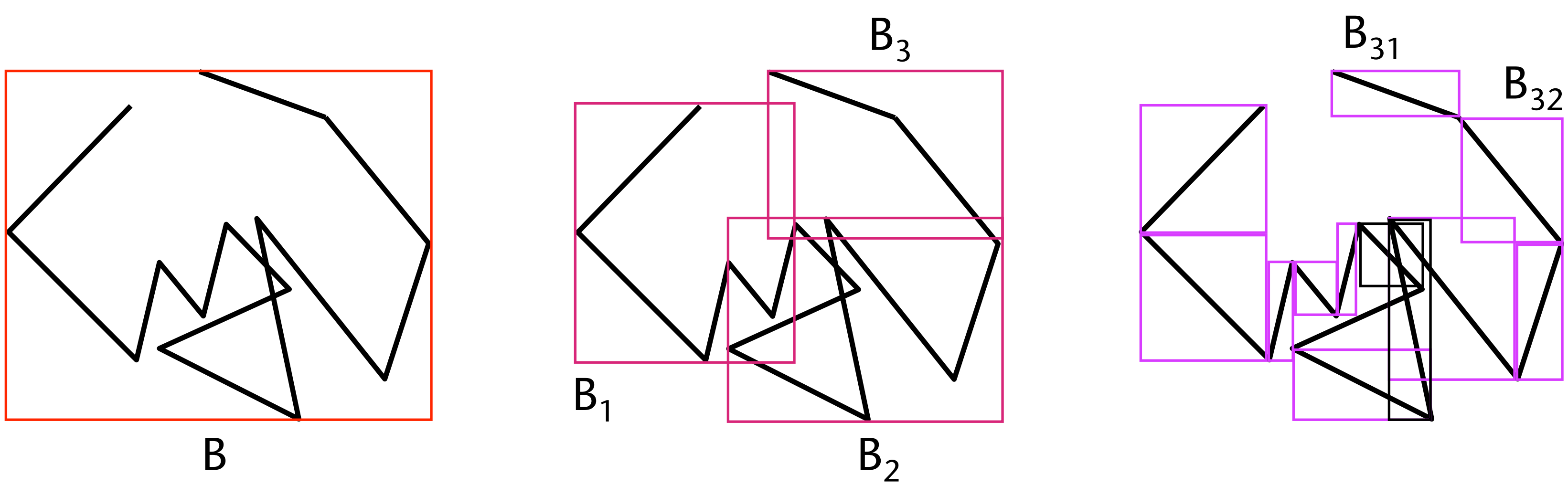
- *The* standard approach for "polygon soups"
- Algorithmic technique: divide & conquer



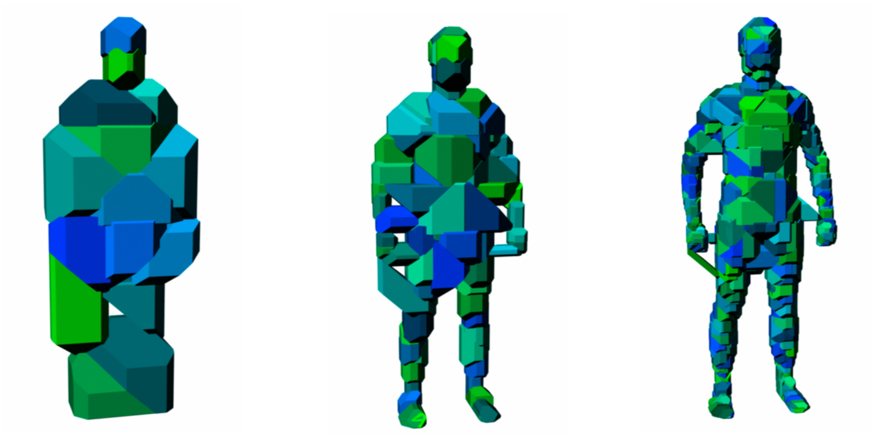
The Bounding Volume Hierarchy (BVH)

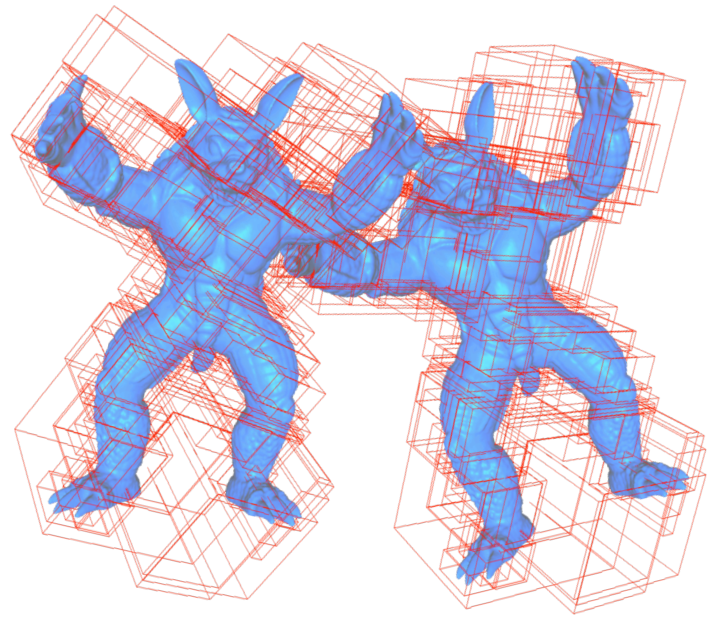
- Constructive defintion of a bounding volume hierarchy:
  1. Enclose all polygons, P, in a bounding volume BV(P)
  2. Partition P into subsets $P_1, \ldots, P_n$
  3. Rekursively construct a BVH for each P_i and put them as children of P in the tree
- Typical arity = 2 or 4

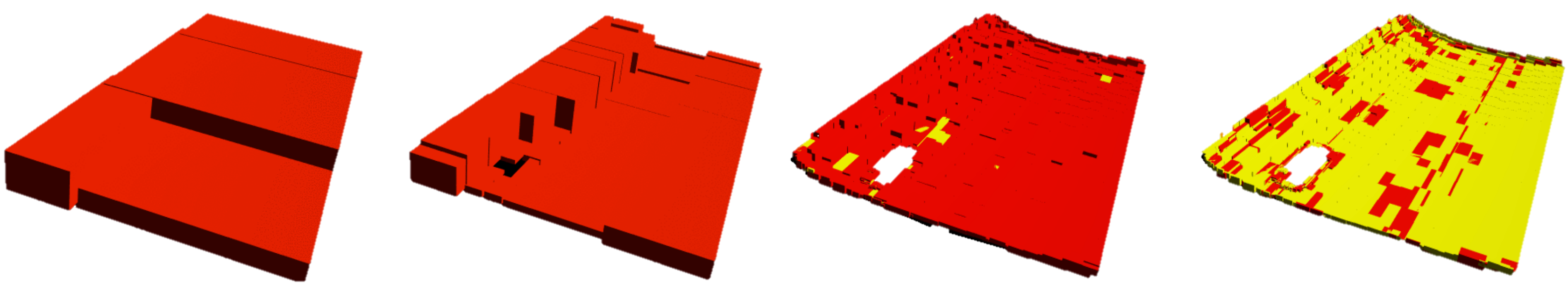




- Visualizations of different levels of some BVHs:





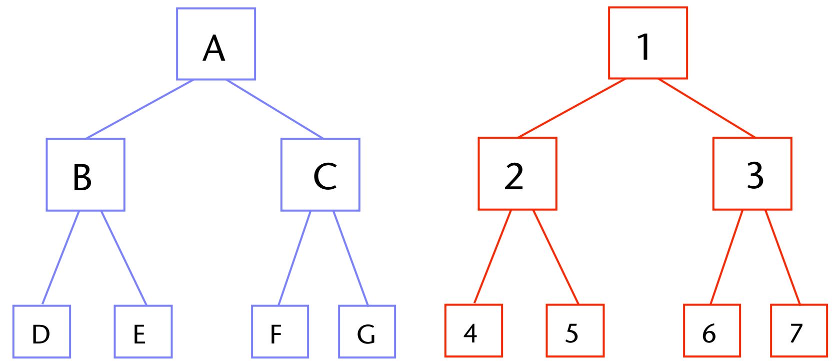


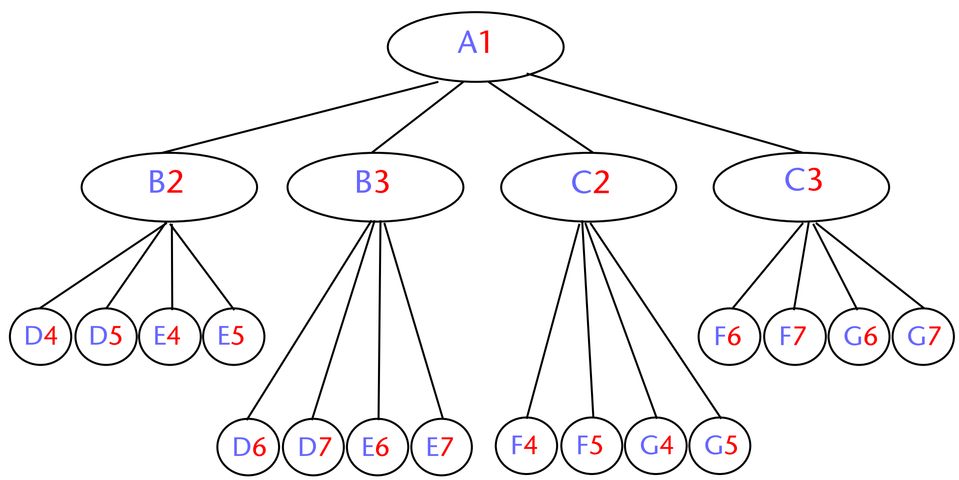
The General Hierarchical Collision Detection Algo

- Simultaneous traversal of two BVHs:

<u>traverse(X, Y)</u>

if X,Y do not overlap **then**
  return
if X,Y are leaves **then**
  check polygons
else
  for all children pairs **do**
  traverse(X_i, Y_j)

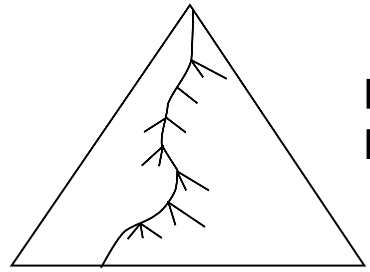




Bounding Volume Test Tree (BVTT)

A Simple Running Time Estimation

- *Best-case*: $O(\log n)$



Path through the
Bounding Volume Test Tree (BVTT)

- Extremely simple *average-case* estimation:
  - Let P[k] = probability that *exactly* k children pairs overlap, $k \in [0,\ldots,4]$

$$P[k] = \binom{4}{k} / 16 \ , \quad P[0] = \frac{1}{16}$$

  - Assumption: all events are equally likely → 16 possible events
  - Expected running time:

$$T(n) = \tfrac{1}{16} \cdot 0 + \tfrac{4}{16} \cdot T(\tfrac{n}{2}) + \tfrac{6}{16} \cdot 2T(\tfrac{n}{2}) + \tfrac{4}{16} \cdot 3T(\tfrac{n}{2}) + \tfrac{1}{16} \cdot 4T(\tfrac{n}{2})$$

$$T(n) = 2T(\tfrac{n}{2}) \in O(n)$$

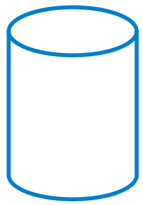
- In praxi: running time is better/worse depending on degree of overlap

Different Kinds of Bounding Volumes

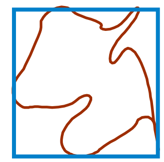
Requirements (for collision detection):

- *Very* fast overlap test → "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) → "*tight BVs*"

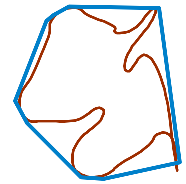
Different Kinds of Bounding Volumes



Cylinder
[Weghorst et al., 1985]

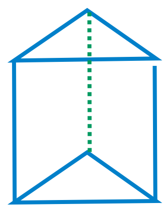


Box, AABB (R*-trees)
[Beckmann, Kriegel, et al., 1990]

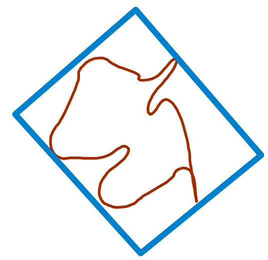


Convex hull
[Lin et. al., 2001]

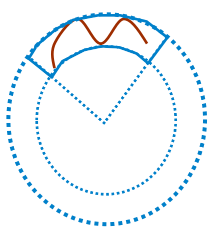
Sphere
[Hubbard, 1996]



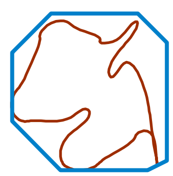
Prism
[Barequet, et al., 1996]



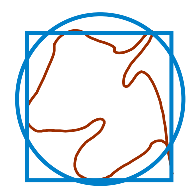
OBB (oriented bounding box)
[Gottschalk, et al., 1996]



Spherical shell
[Manocha, 1997]



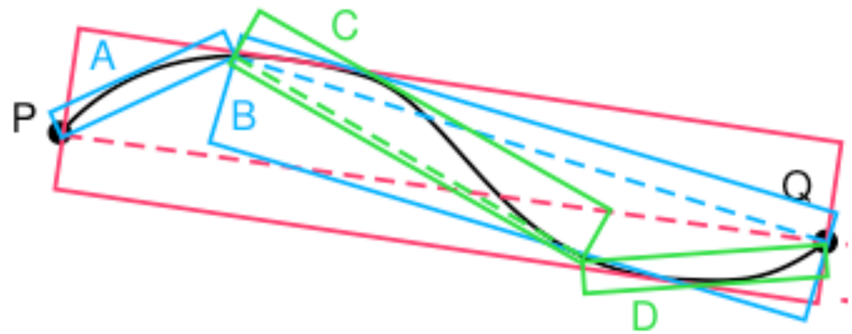
k-DOP / Slabs
[Zachmann, 1998]



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.

Digression: the Wheel of Fortune (Rad der Fortuna)

Boccaccio De Casibus Virorum Illustrium Paris: 1467

Codex Buranus

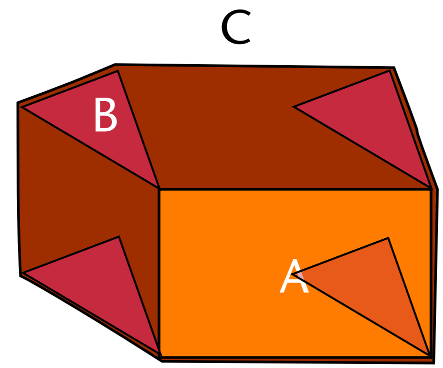
The Intersection Test for Oriented Bounding Boxes (OBB)

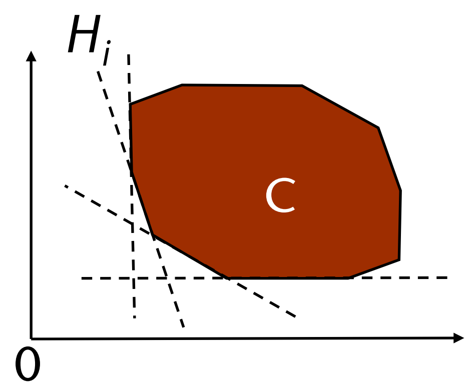
- Lemma "Separating Axis Test" (SAT):
  Let *A* and *B* be two convex 3D polyhedra.
  If there is a separating plane, then there is also a separting plane that is either parallel to one side of *A*, or parallel to one side of *B*, or parallel to one edge of *A* and one edge of *B* simultaneously.
  [Gottschalk, Lin, Manocha; 1996]

- The "separating plane" lemma
  (just a different wording of the "separating axis" lemma):
  Two convex polyhedra *A* and *B* do *not* overlap $\Leftrightarrow$
  there is an axis (line) in space so that the projections of *A* and *B* onto that axis do not overlap.
  This axis is called the separating axis.

Proof of the SAT Lemma

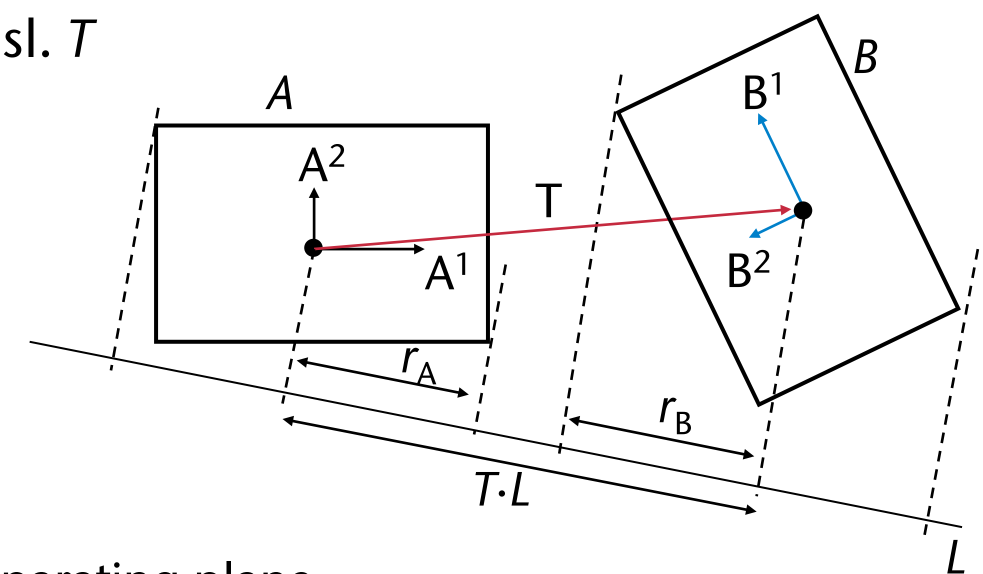
1. Assumption: A and B are disjoint
2. Consider the Minkowski sum $C = A \ominus B$
3. All faces of C are either parallel to one face of A, or to one face of B, or to one edge of A *and* one of B (the latter cannot be seen in 2D)
4. C is convex
5. Therefore: $C = \bigcap_{i=1}^{m} H_i$
6. $A \cap B = \emptyset \Leftrightarrow 0 \notin C$
7. $\exists i : 0 \notin H_i$ (i.e., 0 is outside some H_i)
8. That H_i defines the separating plane; the line perpendicular to H_i is the separating axis.





Actually Computing the SAT for OBBs

- W.l.o.g.: compute everything in the coordinate frame of OBB *A*
- *A* is defined by: center *c*, axes A^1, A^2, A^3 , and extents a^1, a^2, a^3, resp.
- B's position relative to *A* is defined by rot. *R* and transl. *T*
- In the coord. frame of *A*: B^i are the columns of *R*
- Let L be a line in space; then *A* and *B* overlap, if $|T \cdot L| < r_A + r_B$
  - Remark: *L* = normal to the separating plane

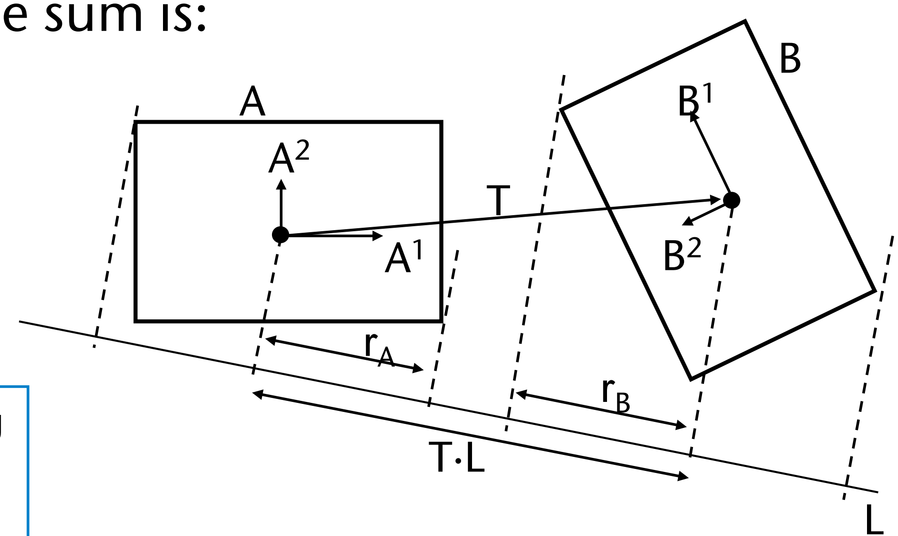


- According to the lemma, we need to check only a few special lines
- With boxes, that number of special lines = 15

- Example: $L = A^1 \times B^2$
- We need to compute: $r_A = \sum_i a_i |A^i \cdot L|$ (and similarly r_B)
- For instance, the 2nd term of the sum is:

$$\begin{aligned} & a_2 A^2 \cdot (A^1 x B^2) \\ & = a_2 B^2 \cdot (A^2 x A^1) \\ & = a_2 B^2 \cdot A^3 \\ & = a_2 R_{32} \end{aligned}$$

Since we compute everything in A's coord. frame
→ A^3 is 3rd unit vector, and
B^2 is 2ns column of R



- In general, we have one test of the following form for each of the 15 axes:

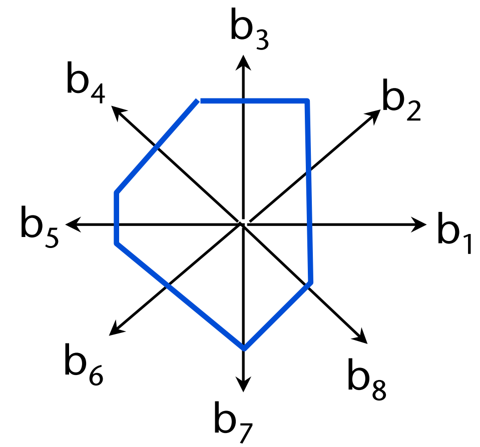
$$|T \cdot L| < a_2|R_{32}| + a_3|R_{22}| + b_1|R_{13}| + b_3|R_{11}|$$

Discretely Oriented Polytopes (k-DOPs)

- Definition of k-DOPs:

  Choose k fixed vectors $\mathbf{b}_i \in \mathbb{R}^3$, with k even,
  and $b_i = - b_{i+k/2}$.
  A k-DOP is a volume defined by

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

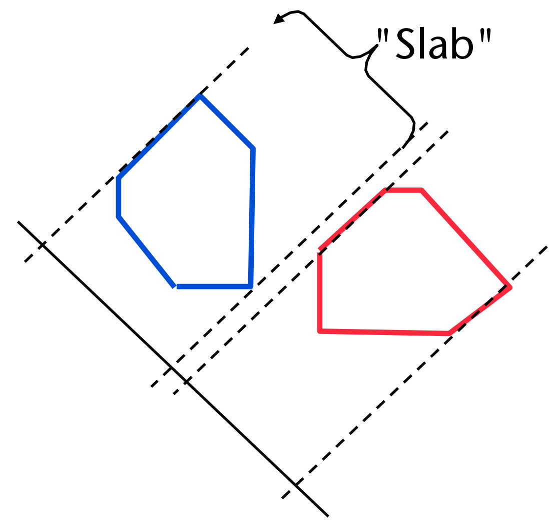


- A k-DOP is completely described by:$D = (d_1 \ldots d_k) \in \mathbb{R}^k$
- The overlap test for two (axis-aligned) k-DOPs:

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

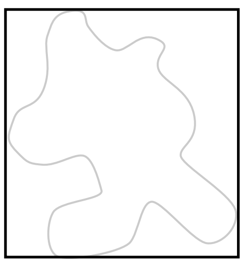
$$\forall i = 1, .., \frac{k}{2} : \left[d_i^1, d_{i+\frac{k}{2}}^1 \right] \cap \left[d_i^2, d_{i+\frac{k}{2}}^2 \right] = \emptyset$$

i.e., it's just $k/2$ interval tests

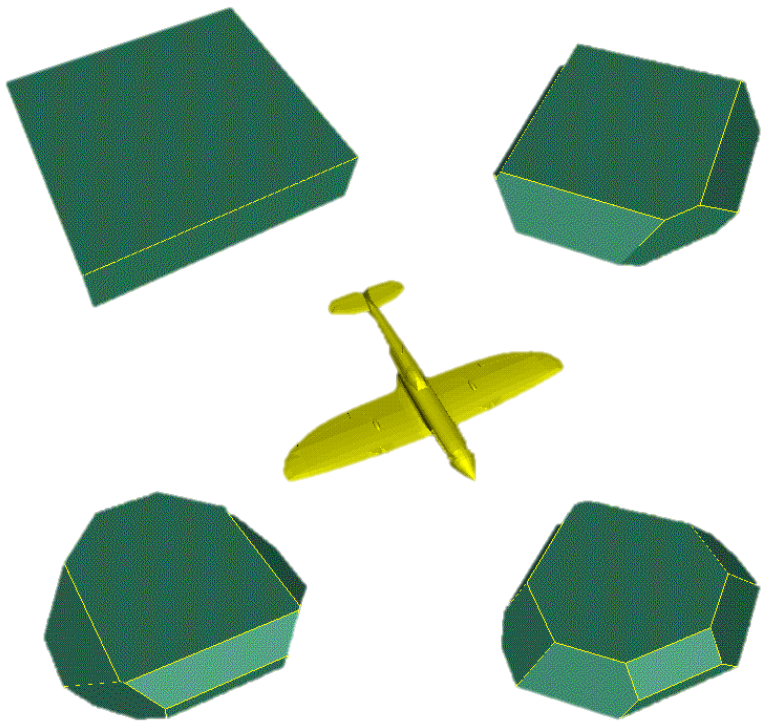


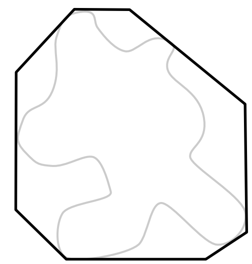
Some Properties of *k*-DOPs

- 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:



2D: $k = 4$
3D: $k = 6$





2D: $k = 8$
3D: $k = 14$

$k = 18$

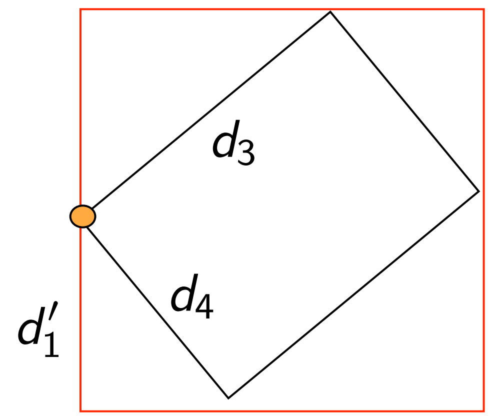
$k = 26$

The Overlap Test for Rotated *k*-DOPs

- The idea: enclose an "oriented" DOP by a new axis-aligned one:
  - The object's orientation is given by rotation R & translation T
  - The axis-aligned DOP D' = (d'_1, …, d'_k) can be computed as follows (without proof):

$$d'_i = \boxed{\mathbf{b}_i \begin{pmatrix} \mathbf{c}_{j_1^i} \\ \mathbf{c}_{j_2^i} \\ \mathbf{c}_{j_3^i} \end{pmatrix}^{-1}} \begin{pmatrix} d_{j_1^i} \\ d_{j_2^i} \\ d_{j_3^i} \end{pmatrix} + \boxed{\mathbf{b}_i T},$$

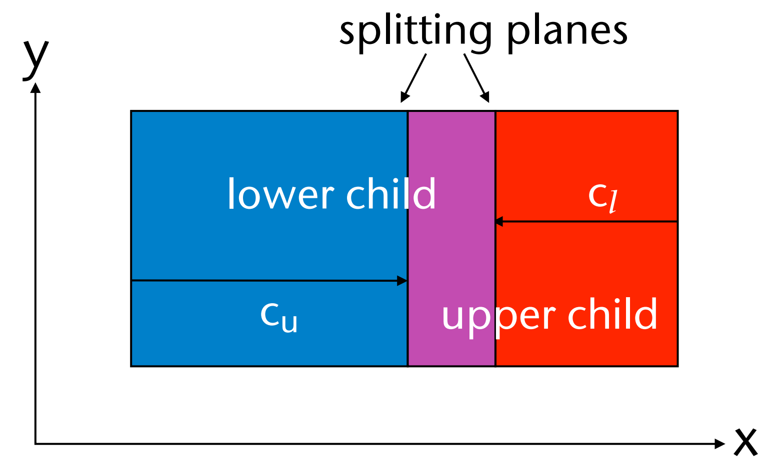
with $\mathbf{c}_j = \mathbf{b}_j R^{-1}$

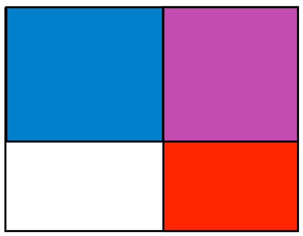


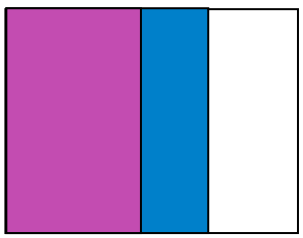
  - The correspondence $j^i{}_l$ is identical for all DOPs in the same hierarchy (thus, it can be precomputed)
  - Complexity: O(k)
    - Compare this to a SAT-based overlap test

Restricted Boxtrees (a Variant of kd-Trees)

- Restricted Boxtrees are a combination of kd-trees and AABB trees:
  - The idea: for the left child of a node B, split off a portion of the "right" part of the box B; for the right child of B, split off a portion of the left part of B

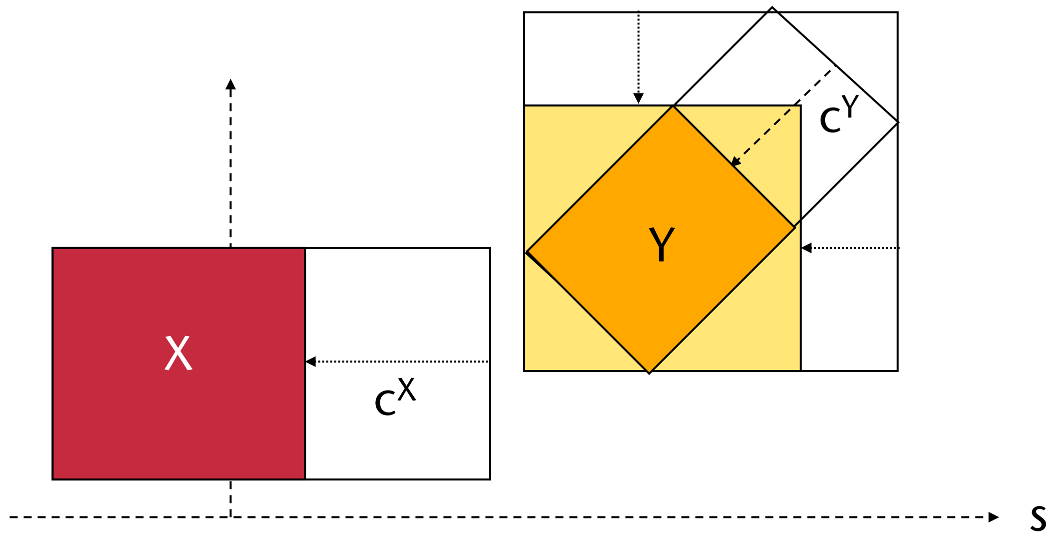






- Memory usage: 1 float, 1 axis ID, 1 pointer (= 9 bytes)
- Other names for the same DS:
  - Bounding Interval Hierarchy (BIH)
  - Spatial kd-tree (SKD-Tree)

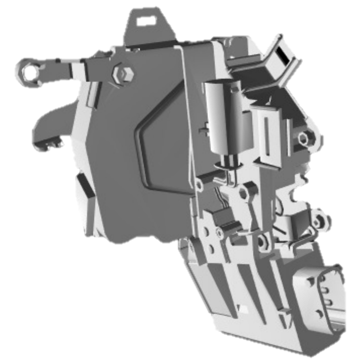
- Overlap Tests by "re-alinment" (i.e., enclosing the non-axis-aligned box in an axis-aligned one, exploiting the special structure of restricted boxtrees):
  12 FLOPs (8.5 with a little trick)



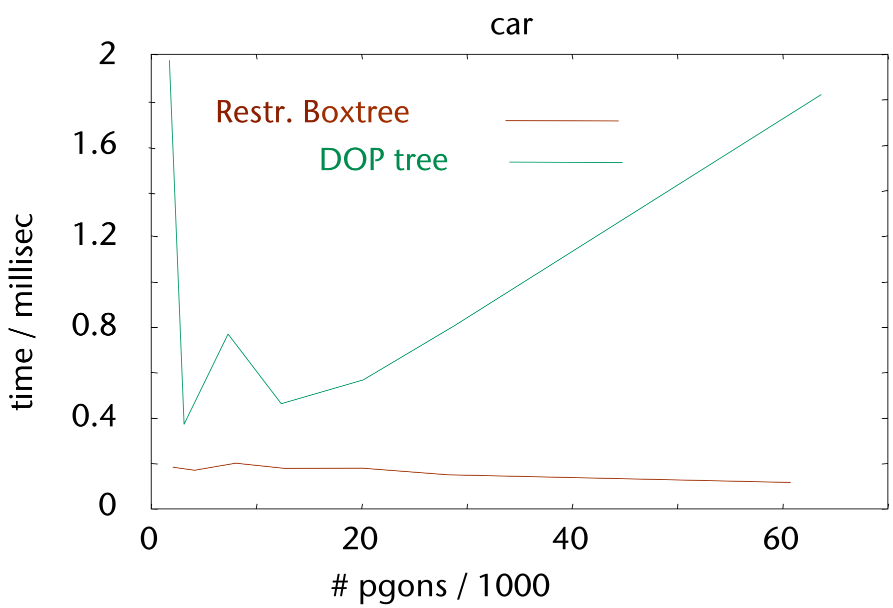
- Compare this to
  - SAT: 82 FLOPs
  - SAT lite: 24 FLOPs
  - Sphere test: 29 FLOPs

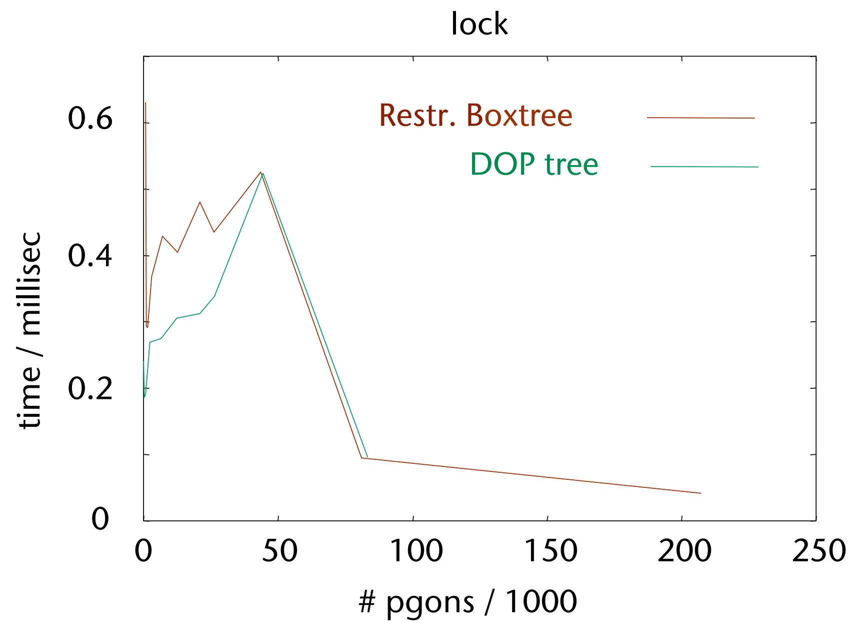
Performance

Car (courtesy VW)



Door lock (BMW)





The Construction of BV Hierarchies

- Obviously:
  if the BVH is bad → collision detection has a bad performance
- The general algorithm for BVH construction: *top-down*
  1. Compute the BV enclosing the set of polygons
  2. Partition the set of polygons
  3. Recursively compute a BVH for each subset
- The essential question: the splitting criteria?
- Guiding principle: the expected cost of collision detection incured by a particular split

$$C(X,Y) = 4 + \sum_{i,j=1,2} P(X_i,Y_j)\,C(X_i,Y_j)$$
$$\approx 4\left(1 + P(X_1,Y_1) + \dots + P(X_2,Y_2)\right)$$

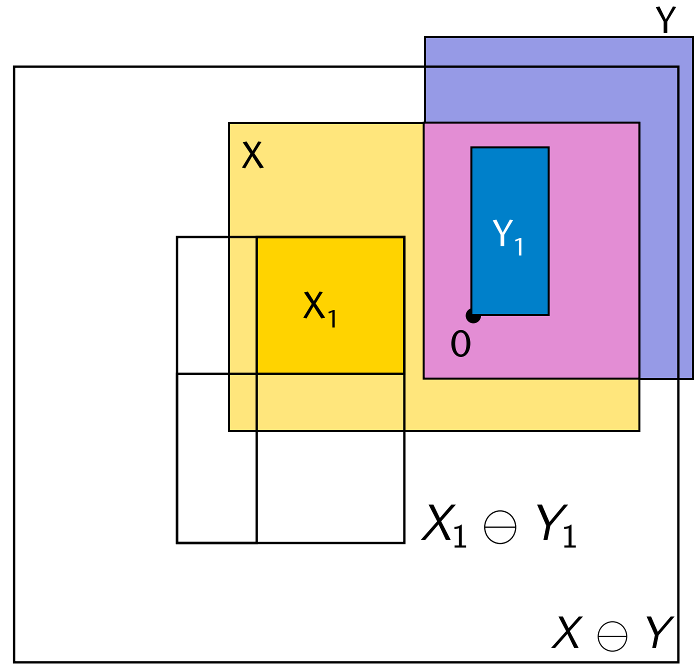
- Goal: estimation of $P(X_i,Y_j)$
- Our tool: the Minkowski sum
- Reminder:

$$X_i \cap Y_j = \varnothing \quad \Leftrightarrow \quad 0 \notin X_i \ominus Y_j$$

- Therefore, the probability is:

$$P(X_i, Y_j) = \frac{\#\ \text{"good" cases}}{\#\ \text{all possible cases}}$$

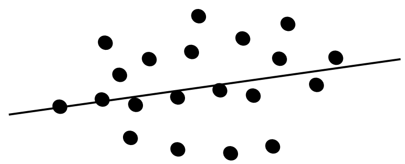
$$= \frac{\text{vol}(X_i \ominus Y_j)}{\text{vol}(X \ominus Y)} = \frac{\text{vol}(X_i \oplus Y_j)}{\text{vol}(X \oplus Y)} \approx \frac{\text{vol}(X_i) + \text{vol}(Y_j)}{\text{vol}(X) + \text{vol}(Y)}$$



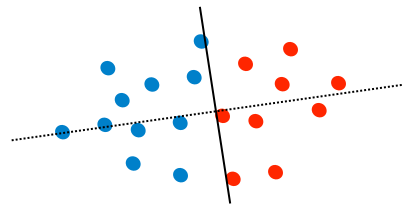
- Conclusion: for a good BVH (for coll.det.) minimize the total volume of the children of each node

Usual Algorithm for Constructing a BVH

1. Find good orientation for a "good" splitting plane using PCA



2. Find the minimum of the total volume by a sweep of the splitting plane along that axis



- Complexity of that *plane-sweep* algorithm:

$$T(n) = n \log n + T(\alpha n) + T((1-\alpha)n) \in O\left(n \log^2 n\right)$$

  - Assumption: splits (α) are not too uneven

Collision Detection among Morphing Objects

- Definition of *Morphing*:
  Given n objects O^i (called morph targets)
  with vertices v^i_j and weights w_i , $\sum_i w_i = 1$.
  Then the morphed object is given by the vertices:

$$\bar{v}_j = \sum_{i=1}^{n} w_i v^i_j \quad , \; j = 1, \ldots, N$$

- Alternative representation:
  - Represent objects O^i as a single, long "vertex vector": $\mathbf{v}^i = \begin{pmatrix} v^i_{1,x} \\ v^i_{1,y} \\ v^i_{1,z} \\ v^i_{2,x} \\ \vdots \\ v^i_{N,z} \end{pmatrix}$
  - Then, the morphed object is:

$$\bar{\mathbf{v}} = \sum_{i=1}^{n} w_i \mathbf{v}^i$$

- Note: all meshes must have the same "topolgy" (i.e., connectivity)!

- *Morphing* of k-DOP's:

  Given n DOPs $D^i = \left(s^i_1, \ldots, s^i_{\frac{k}{2}}, e^i_1, \ldots, e^i_{\frac{k}{2}}\right)$.

  We define the morphed DOP

$$\overline{D} = (\overline{s}_1, \ldots, \overline{s}_{\frac{k}{2}}, \overline{e}_1, \ldots, \overline{e}_{\frac{k}{2}}) \; , \; (\overline{s}_j, \overline{e}_j) = \left(\sum w_i s^i_j \, , \, \sum w_i e^i_j\right)$$

- Conjecture:

  If the morph targets O^i are bounded by the D^i, then the morphed object is bounded by the morphed DOP, i.e.

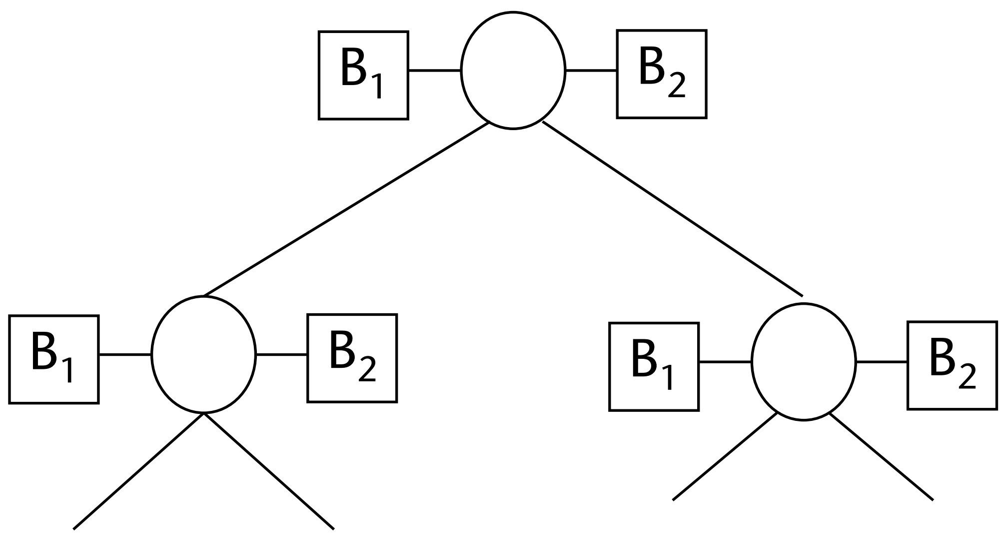
$$\forall l \; : \; \mathbf{v}^i_l \in D^i \quad \text{then} \quad \overline{\mathbf{v}}_j \in \overline{D}$$

- Proof:

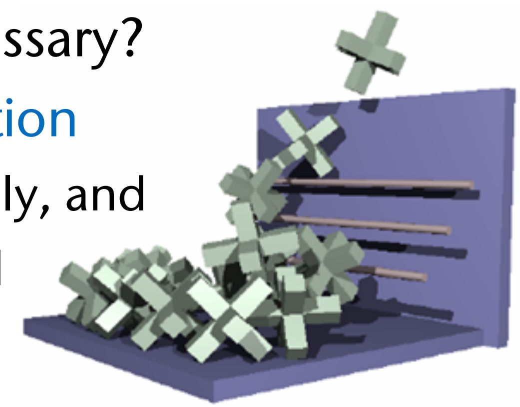
$$\forall l \; : \; \overline{s}_j = \sum_{i=1}^{n} w_i s^i_j \leq \sum_{i=1}^{n} w_i \left(\mathbf{v}^i_l \cdot \mathbf{b}^j\right) \leq \sum_{i=1}^{n} w_i e^i_j = \overline{e}_j$$

- This is also true analogously for spheres (doesn't work for OBBs)

- Data structure of a BVH for morphed objects:
  - At each node of the "morphed BVH", store a BV for each of the morph targets
  - Each of these BV's of the morph targets must enclose the same subset of polygons!



Time-Critical Collision Detection



- Is 100% exact collision detection really necessary?
- Consequence: approximate collision detection
  - Try to perform collision detection approximately, and
  - Try to take advantage of that → increase speed
- Problems of classical BVH traversal:
  - Early exit does not yield *any* information at all
  - There is no level of detail (unless specifically crafted)
- Goal: continuous and and controlled balance between running time and accuracy
- Idea: utilize a remaining degree of freedom in the simultaneous traversal algorithm
- New algorithm:
  - For a given pair of BV's, estimate the probbility of collision within
  - First "visit" those subtrees with high probability
  - No stack any more, instead use priority queue (p-queue)

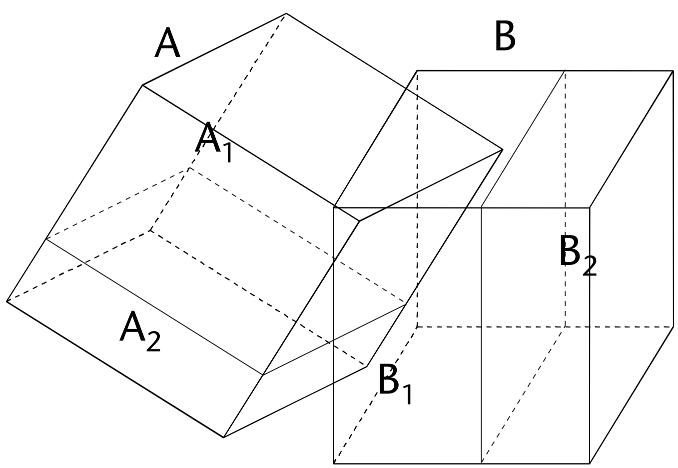
Overview of the New Probability-Driven Algorithm

priority queue q;

(A,B) ←

(A,B)

(A_1,B_1), p=0.9
(A_1,B_2), p=0
(A_2,B_1), p=0.5
(A_2,B_2), p=0

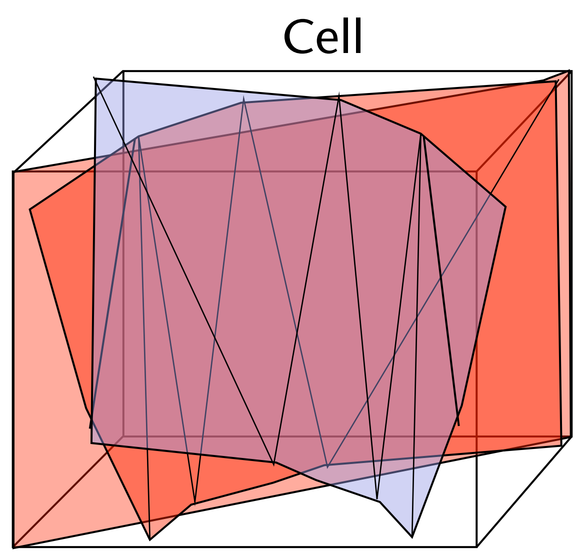


<u>Traverse(A,B)</u>

P-queue q

q.insert(A,B,1)
while q not empty
  A,B ← q.pop
  forall A_i, B_j
    p ← Pr[collision in A_i, B_j]
    if p ≥ pmin
      return "collision"
    if p ≥ 0
      q.insert(A_i, B_j, p)
return "no collision"

Thought Experiment ("Gedankenexperiment")

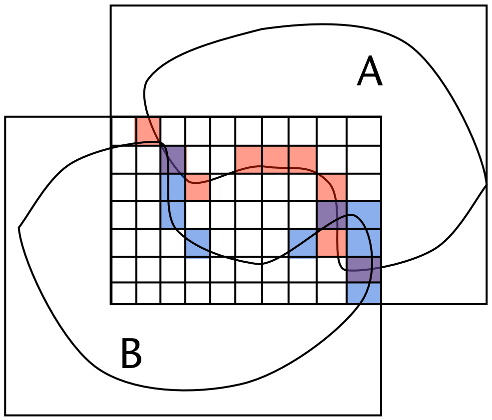


"Well-Filled" Cell

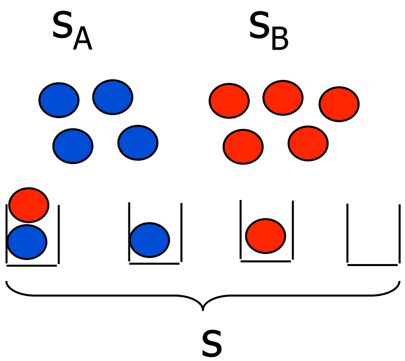
→ Collision Cell

Estimation of the Probability of a Collision (Idea only)

- "Well-filled" = surface area in a cell is larger than a specific threshold
- Idea:
  - Partition $A \cap B$ by grid
  - Compute probability of cell that is *well-filled* by *A* and *B*

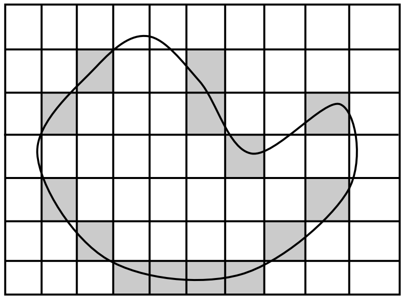


- During runtime: estimate following param's
  - s = number of grid cells in $A \cap B$
  - s_A , s_B = number of cells well-filled by surface of *A* or *B*, resp.
- Estimate probability for intersection by probability that one (or more) cell is well-filled by *A* and *B*:
  - Purely combinatoric "balls into bins" model
  - Probability $$Pr = 1 - \frac{\binom{s-s_B}{s_A}}{\binom{s}{s_A}}$$



Efficient Implementation

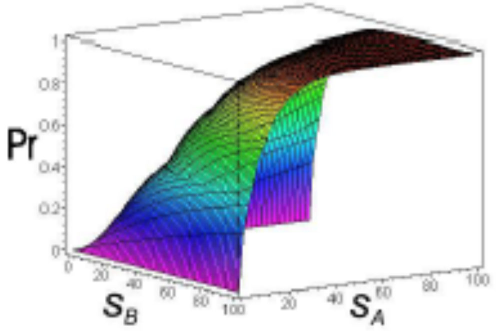
- Partitioning $A \cap B$ and counting number of well-filled cells at runtime is too expensive
- Solution: preprocessing and further estimations
- Augmented BVH (ADB-tree):
  - For each BV, partition BV by grid (e.g., 8^3)
  - Store number of well-filled grid cells with node
    - Just one additional integer per node!



- At runtime, estimate s_A and s_B by

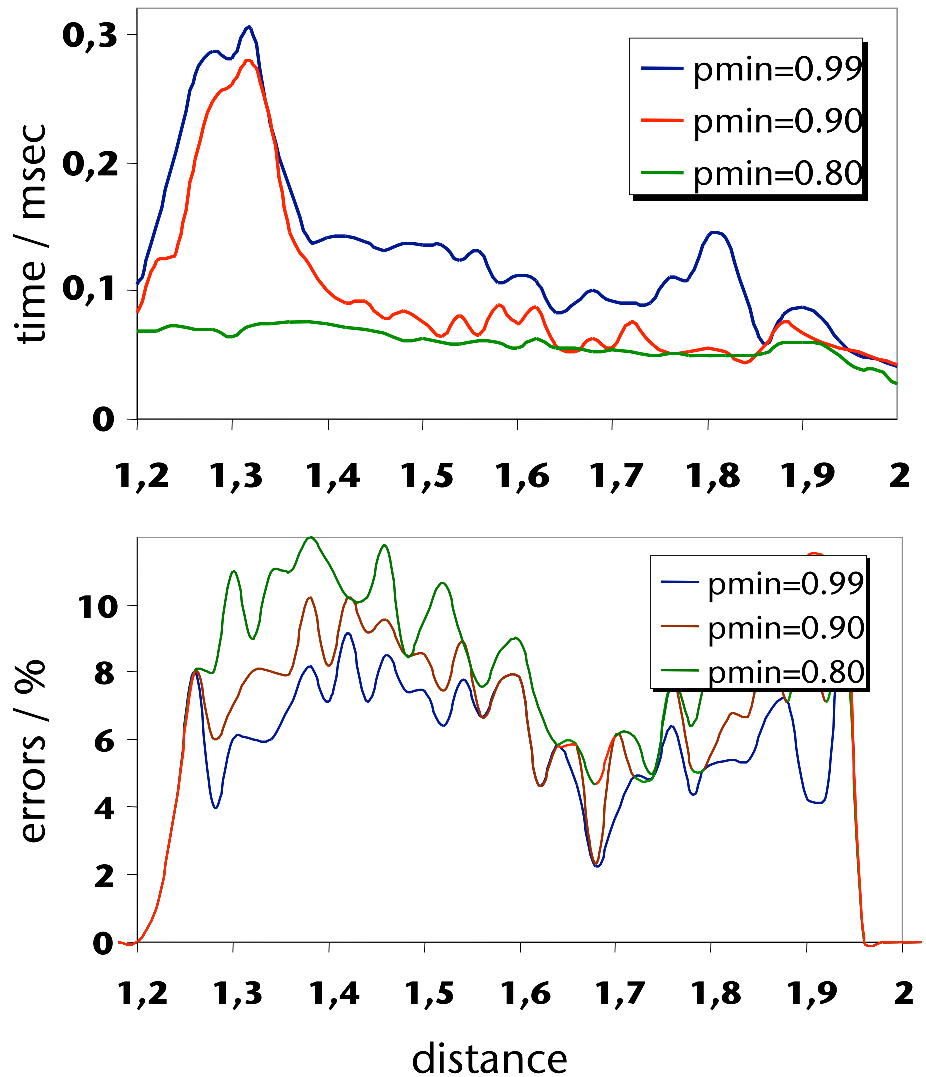
$$s'_A = s_A \frac{\mathrm{Vol}(A)}{\mathrm{Vol}(A \cap B)}$$

- Precompute function *Pr* and store in a Lookup Table



Result

- Time vs. erro:



Open Problems

- Can we estimate collision normals that way, too?
- Utilize orientation of polygons, in order to improve the estimation of an intersection!
- What about deformable geometry?!