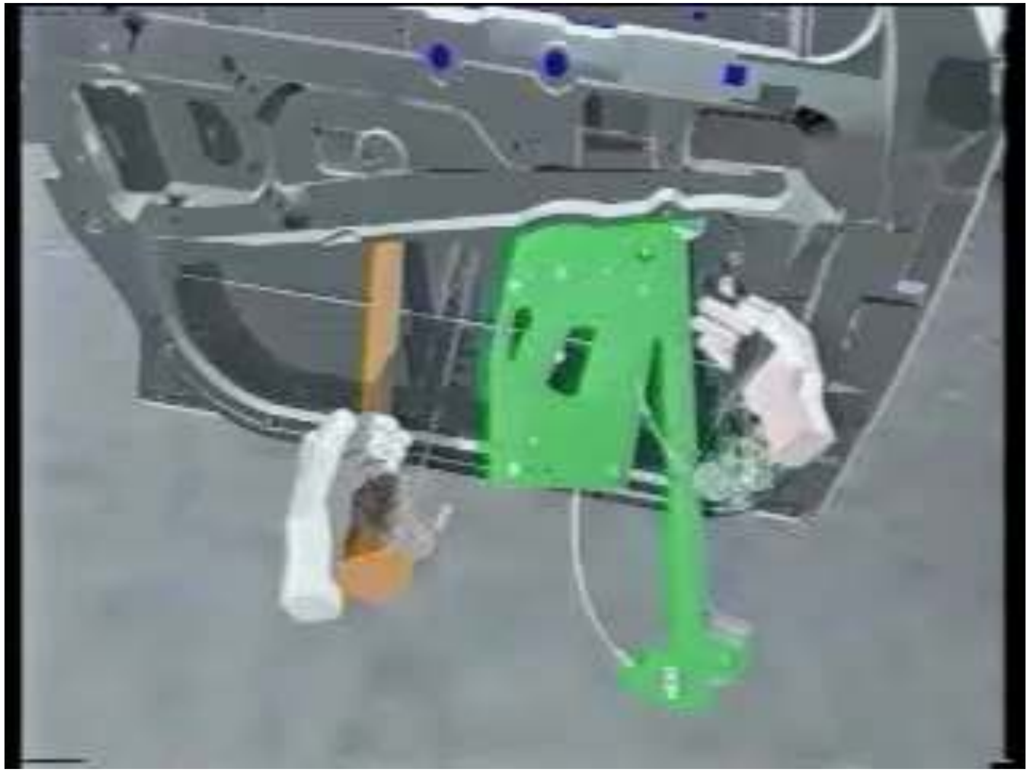
Virtual Reality & Physically-Based Simulation
Collision Detection

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

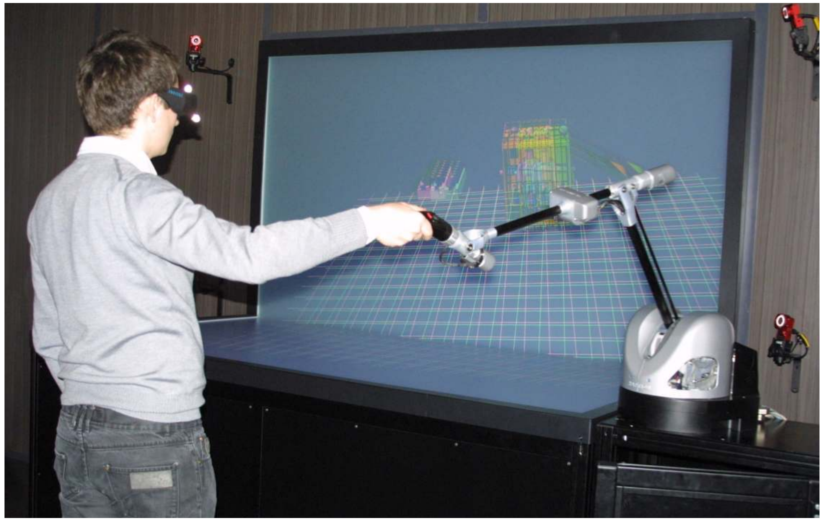
Examples of Applications

Virtual Assembly Simulation

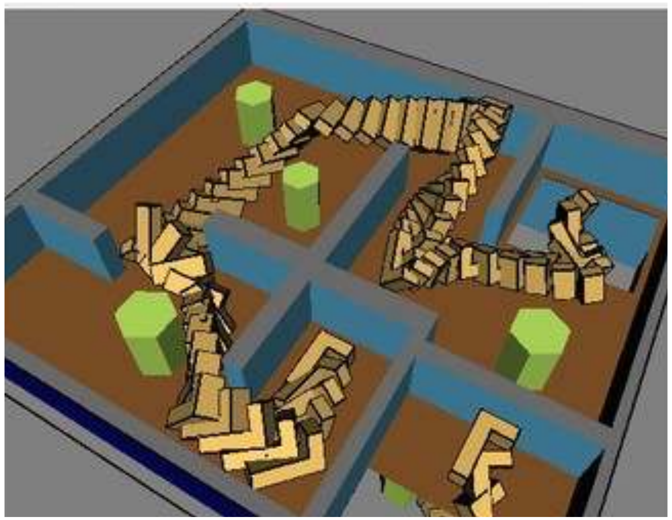


Virtual Ergonomics Investigation

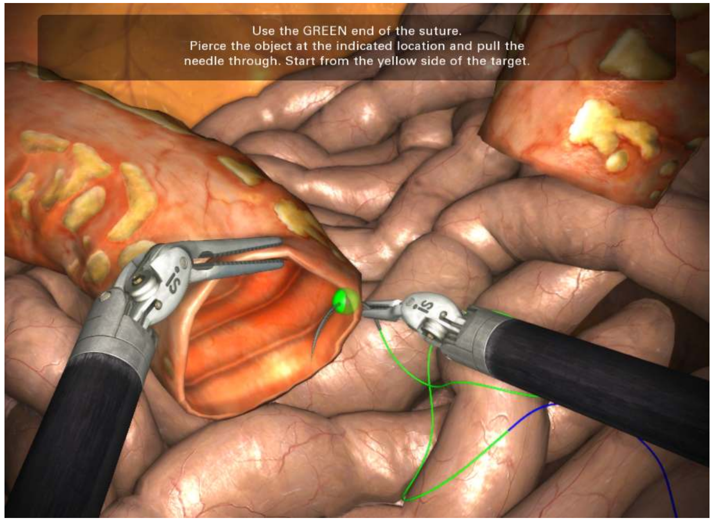
Other Uses of Collision Detection



Rendering of force feedback



Robotics: path planning
(piano mover's problem)



Medical training simulators

Games

How Would You Approach the Problem of Coll.Det.?

https://www.menti.com/f1b5t74e21

Definitions

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

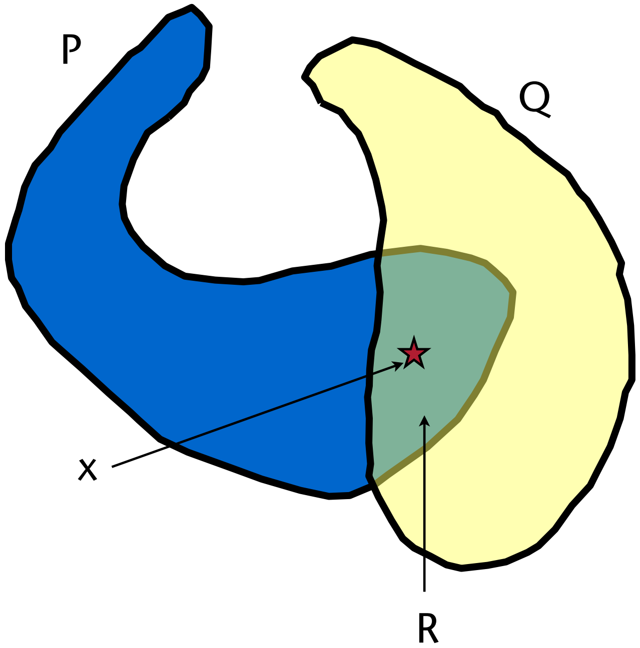
  "P and Q collide" $\Leftrightarrow$

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

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

- The construction problem:

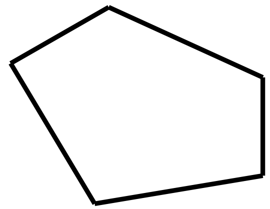
  compute $R := P \cap Q$



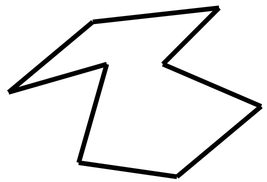
- For polygonal objects we define collisions as follows: *P* and *Q* collide iff there is (at least) one face of *P* and one of *Q* that intersect each other
- The games community often has a different definition of "collision"

Classes of Objects

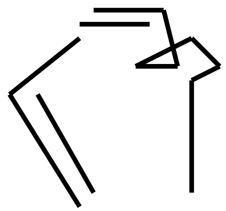
- Convex
- Closed and simple
  (no self-penetrations)
- Polygon soups
  - Not necessarily closed
  - Duplicate polygons
  - Coplanar polygons
  - Self-penetrations
  - Degenerate cardigans
  - Holes
- Deformable



Convex

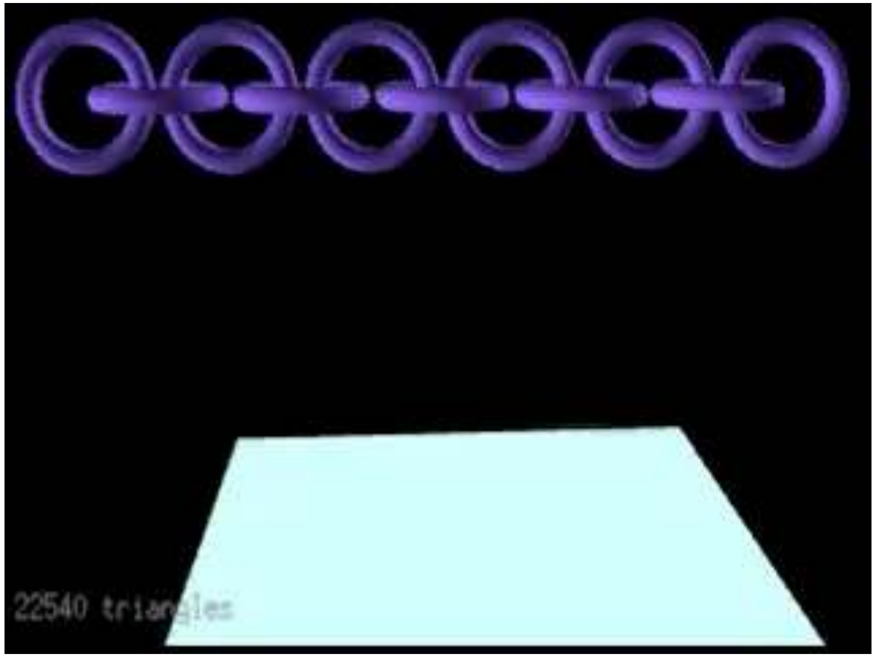


Simple & closed

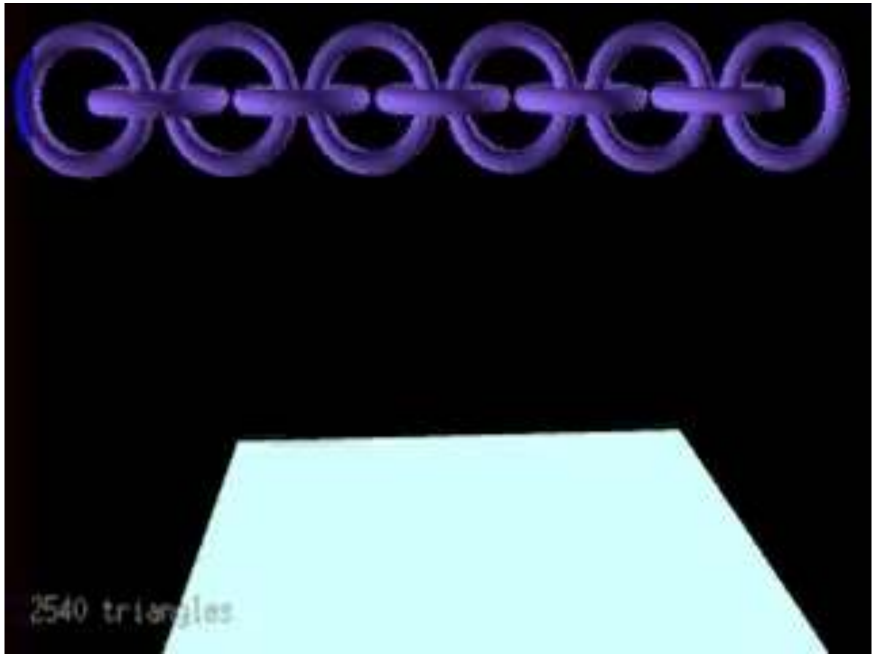


Polygon soup

Importance of the Performance of Collision Detection



Clever algorithm (use bbox hierarchy)



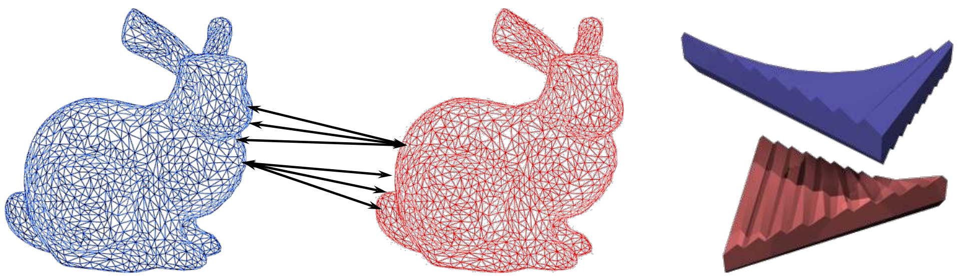
Naïve algorithm (test all pairs of polygons)

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

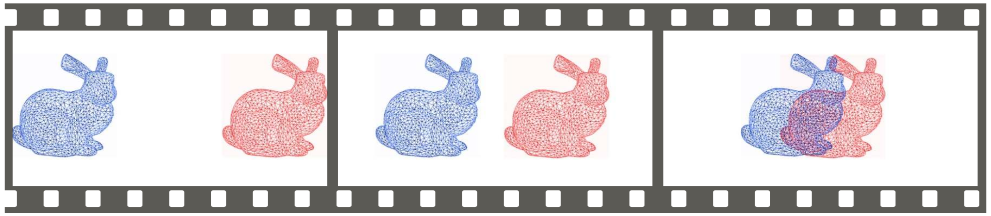
In many simulations, the coll.det. part takes 60-90 % of the overall time

Why is Collision Detection so Hard?

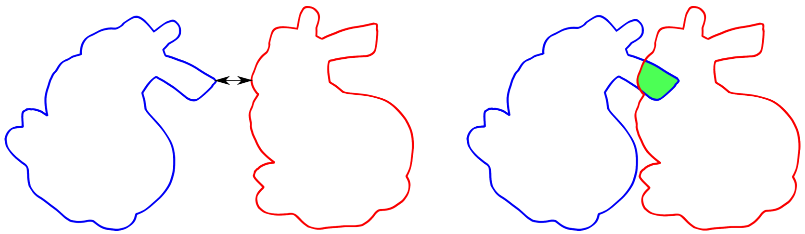
1. All-pairs weakness:



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 original data)
  - Preprocessing for these auxiliary data structures should not take too long, so that it can be done at startup time (< 5sec / object)

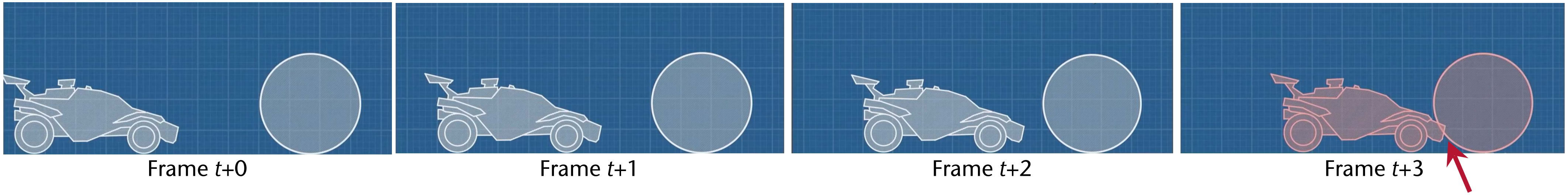
Another Problem Related to Collision Detection

- Physics consistency (or inconsistency): *small* changes in the starting conditions can result in *big* changes in the outcomes

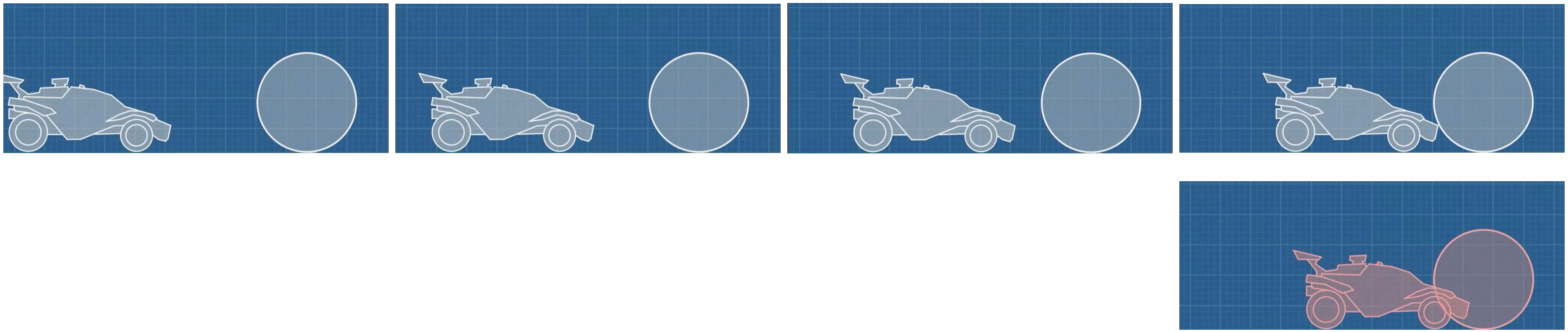
2nd time, the ball has been moved slightly

Explanation by Way of Example

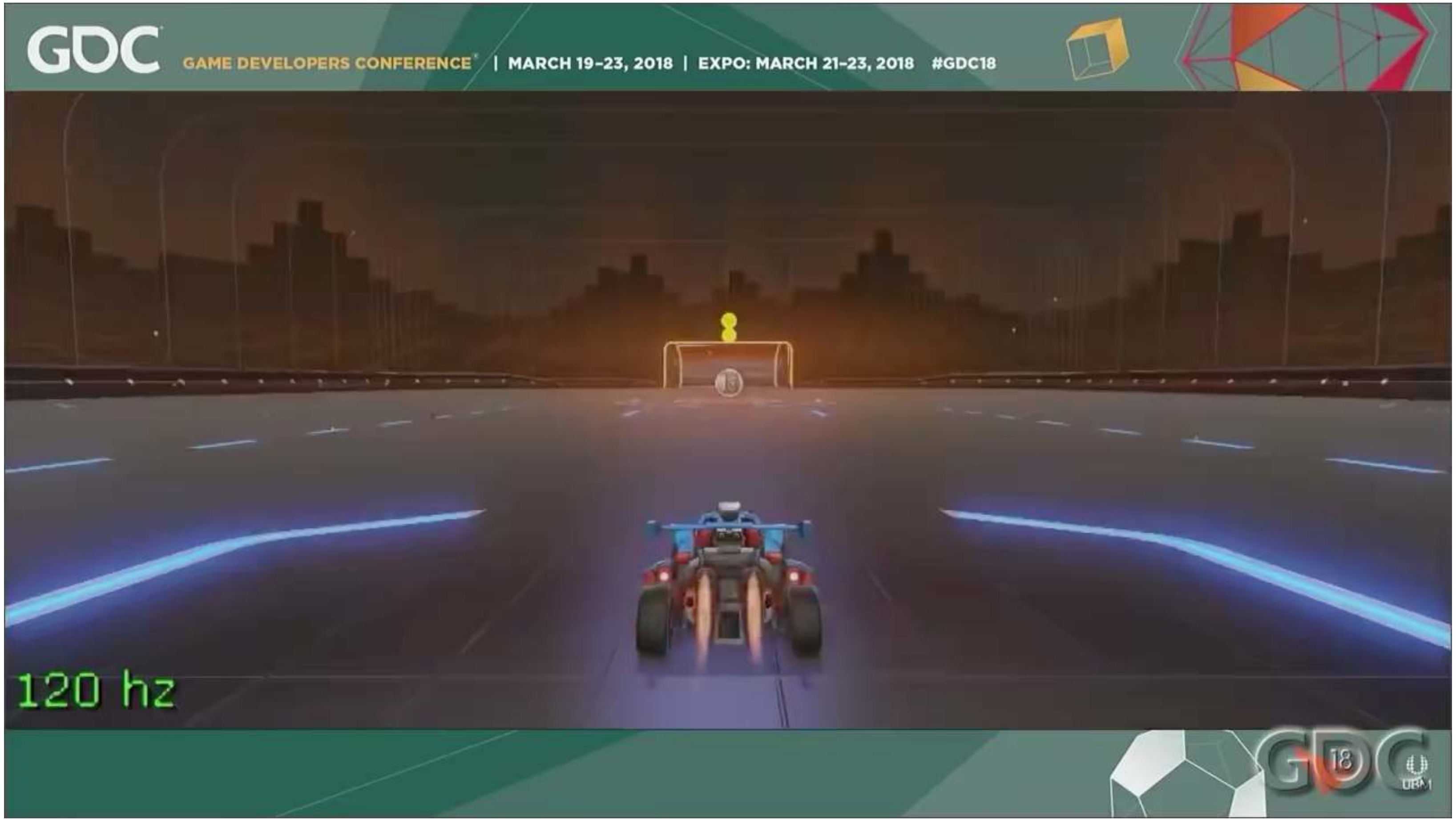
Run 1



Run 2 (ball has been moved slightly)



One Way of Alleviation: Faster Coll.Det. ⟶ Faster Frame Rate



Same experiment: 2nd time, the ball has been moved slightly, but frame rate is much higher now

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

Achieving a Fixed Framerate for Rendering *and* Simulation

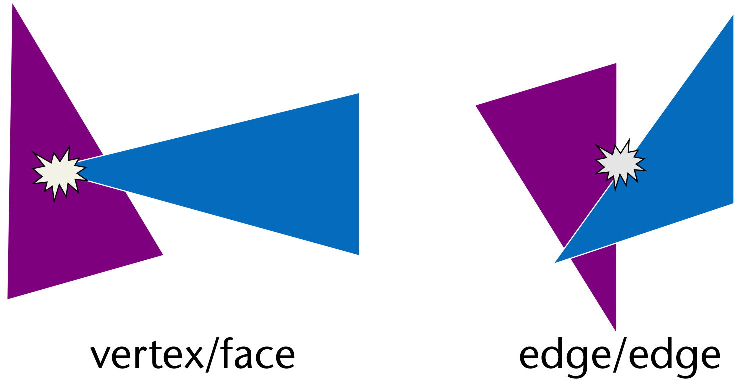
```
t = accumulator = 0;  Δt = 0.001;              // time in seconds
oldTime = currentHighresTimer()
repeat
  render scene with current state              // try to use LOD's etc.
  check collisions with current positions      // large time variability
    ⟶ new forces
  // calc delta-t since last frame
  newTime = currentHighresTimer()
  frameTime = newTime - oldTime
  oldTime = newTime
  // advance physics sim. in small steps to current time
  accumulator += frameTime
  while accumulator >= Δt:
    integrate( state, t, Δt )
    accumulator -= Δt;  t += Δt
until quit
```

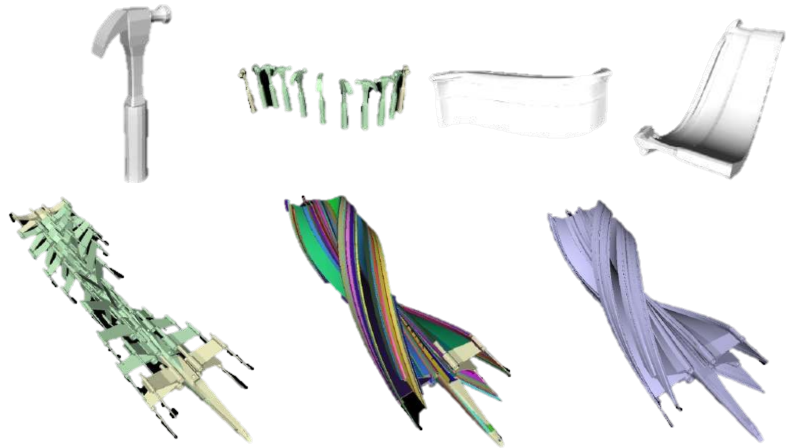
Terminology: Continuous / Discrete Collision Detection

- Discrete coll.det.: compute penetration measure (or just yes/no) for "static" objects at the current point in time
- Continuous coll.det.: find exact point in time where first contact occurs
  - Usually, this assumes that objects between frames move/rotate linearly

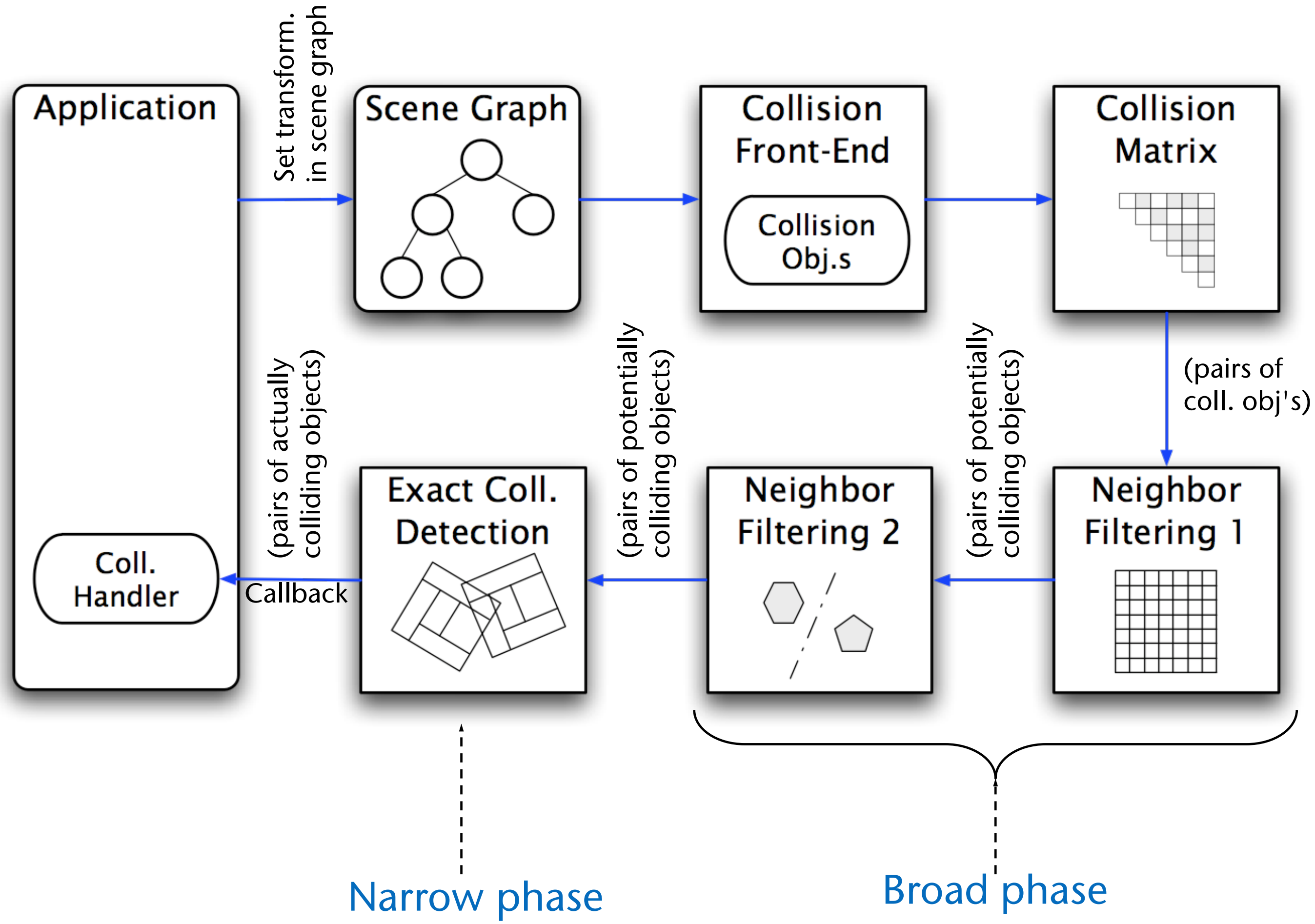
The Difficulties of Continuous Coll.Det.

- Finding the exact, first contact of polygons moving in space amounts to checking several cases
  - Each case needs to consider 4 points
  - Each of those points is a linear function in *t*
  - Necessary condition for hit: all 4 points lie in a plane at some point in time
  - Amounts to solving a polynomial of degree 5!
- Swept volumes (aka. space-time volumes) can help to determine potentially colliding pairs
  - But difficult to calculate
  - Many false positives





The Collision Detection Pipeline



The Collision Interest Matrix

- Interest in collisions is specific to different applications / objects:
  - 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
- Solution: Collision Interest Matrix
- Elements in this matrix comprise:
  - Flag for collision detection
  - Additional info that needs to be stored from frame to frame for each pair for incremental algorithms (e.g., the separating plane)
  - Callbacks to the simulation / coll. handling

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

Methods for the Broad Phase

- Broad phase = one or more filtering steps
  - 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

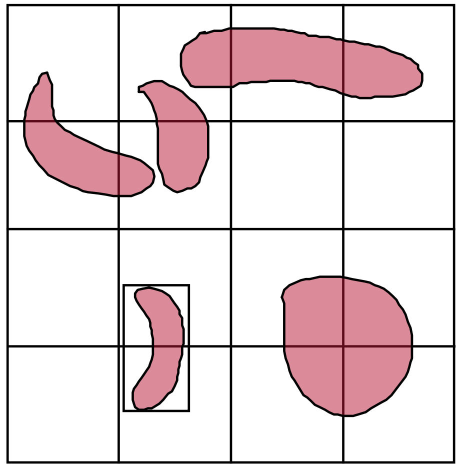
➢ *Brute-force* method needs to compare $O(n^2)$ many pairs of bboxes

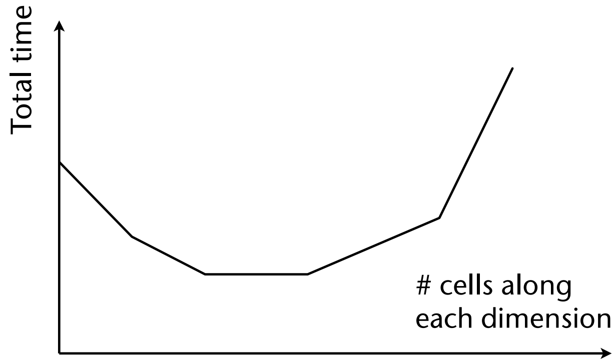
- Goal: determine neighbors more efficiently

The 3D Grid

1. Partition the "universe" by a 3D 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 obj
  - Data structure here: hash table (!)
  - Collision in hash table ⟶ potentially colliding pair
- 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

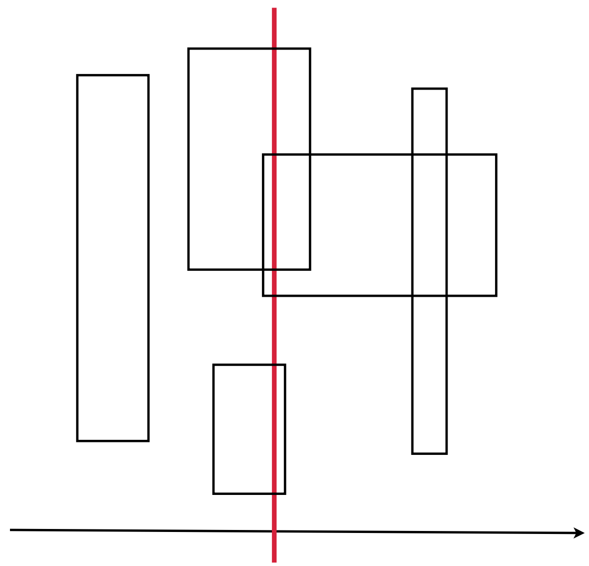




The Plane Sweep Technique (aka Sweep and Prune)

- The idea: sweep a plane through space, perpendicular to the X axis
- Solve the problem on that plane
- 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
```



Temporal Coherence

- Observation:

  Two consecutive images in a sequence differ only by very little (usually).
- Terminology: temporal coherence (a.k.a. frame-to-frame coherence)
- Algorithms based on frame-to-frame coherence are called “incremental”, sometimes “dynamic” or “online” (albeit the latter is the wrong term)
- Examples:
  - Motion of a camera
  - Motion of objects in a film / animation
- Applications:
  - Computer Vision (e.g. tracking of markers)
  - Video compression
  - Collision detection
  - Ray-tracing of animations (e.g. using kinetic data structures)

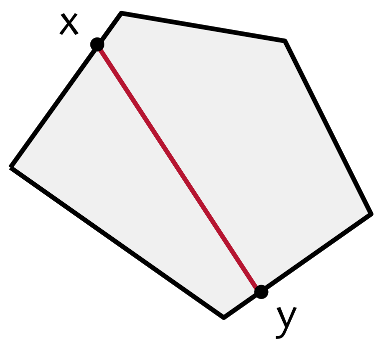
Do You Know Examples/Applications of Frame-to-Frame Coherence?

https://www.menti.com/f1b5t74e21

Collision Detection for Convex Objects

- Definition of "convex polyhedron":

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

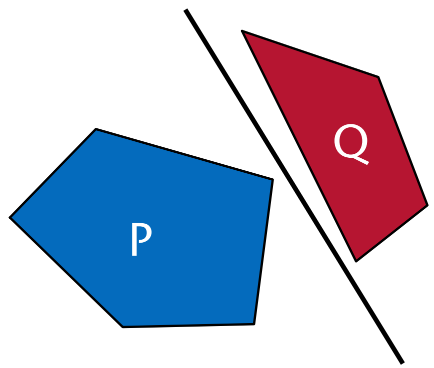


- A condition for "non-collision":

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

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

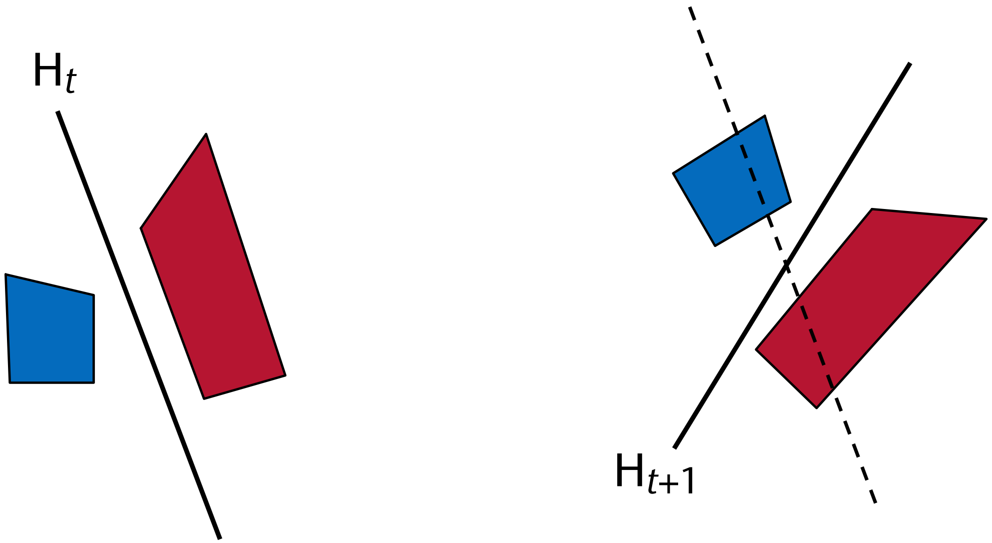
$$\exists \mathbf{h} \in \mathbb{R}^4 \; \forall \mathbf{p} \in P, \mathbf{q} \in Q : \; (\mathbf{p}, 1) \cdot \mathbf{h} > 0 \; \wedge \; (\mathbf{q}, 1) \cdot \mathbf{h} < 0$$



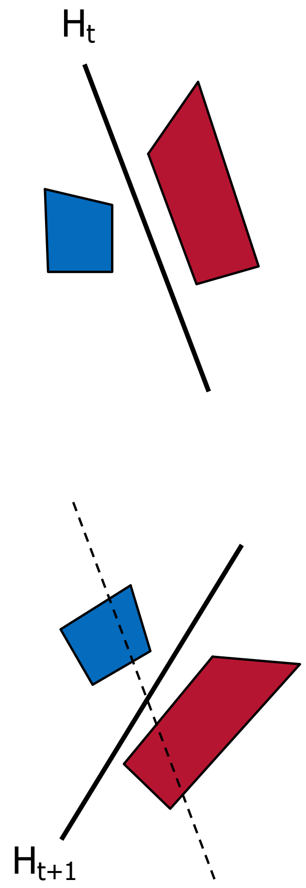
Separating plane H

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)

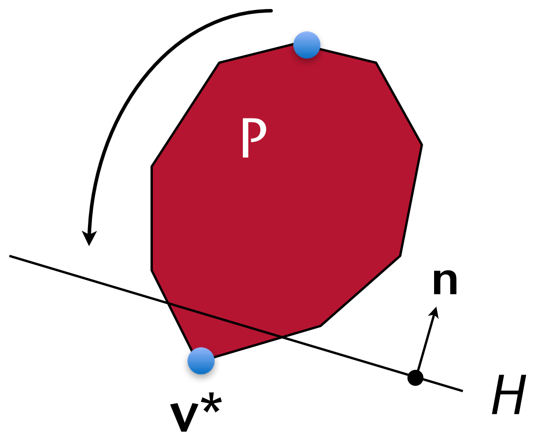


```
load Ht = separating plane between P & Q at time t
H := Ht
repeat max n times
    if exists v ∈ vertices(P)  on the back side of H:
        rot./transl. H such that v is now on the front side of H
    if exists v ∈ vertices(Q)  on the front side of H:
        rot./transl. H such that v is now on 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 **h**, such that $\forall \mathbf{p} \in P : \mathbf{h}\cdot\mathbf{p} > 0$ 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., $\eta' = 0.97\eta$ after each iteration, prevents cycling
- (For object Q, some signs need to be changed)

- Perceptron Learning Rule (has been known in machine learning for a long time):
  whenever we find $\mathbf{p} \in P$ with $\mathbf{h} \cdot \mathbf{p} < 0$, update **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 ⇔

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

- 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 **h*****h**^{k} :

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

- 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*:

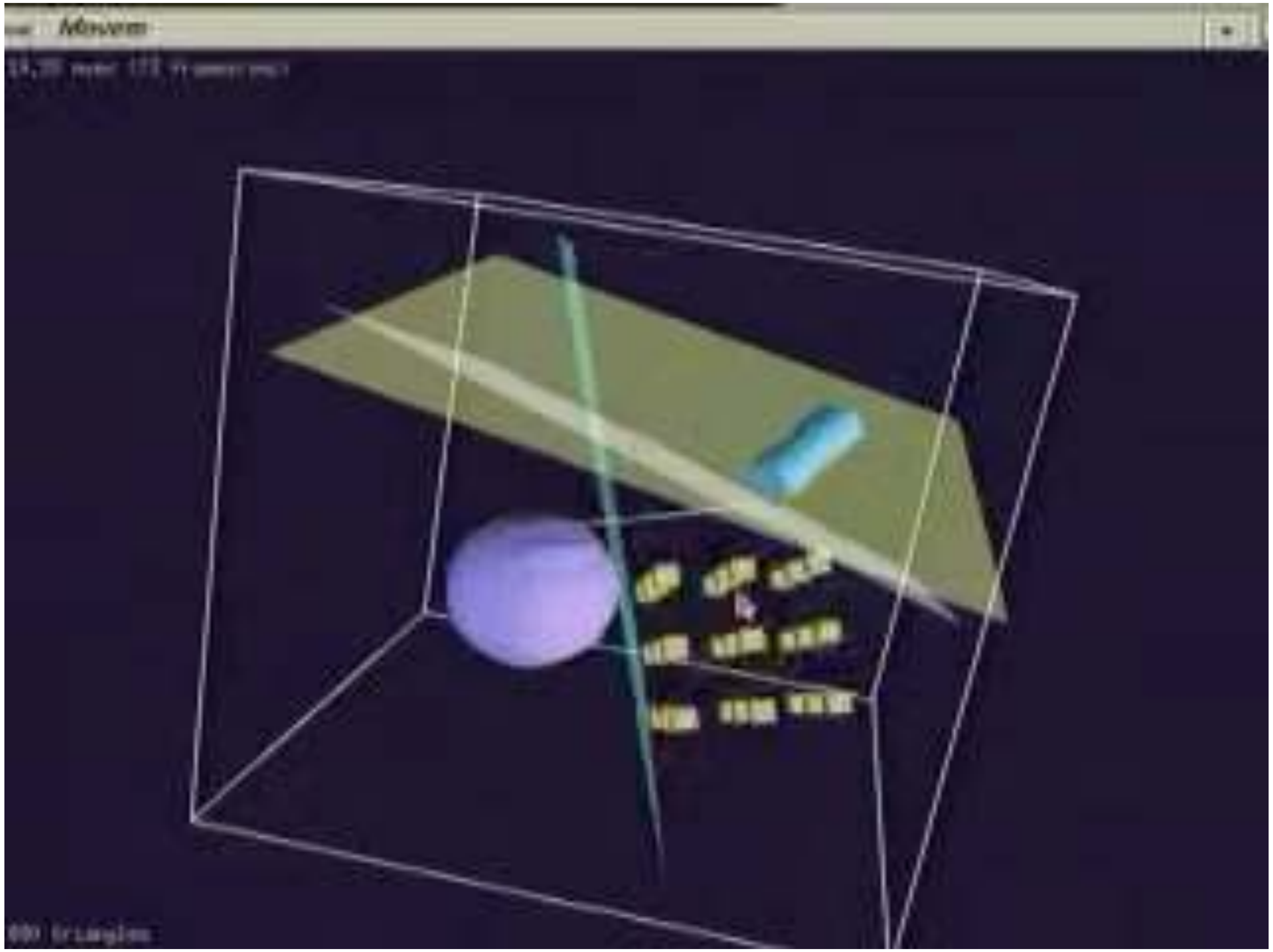
$$k \leq \frac{D^2}{d^2}$$

- In other words, the factor $\frac{D^2}{d^2}$ gives a hint at how difficult the problem is (except, we don't know *d* or *D* in advance)
- To some extent, $\frac{d}{D}$ is measures the "difficulty" of the problem

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 Optional

- 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

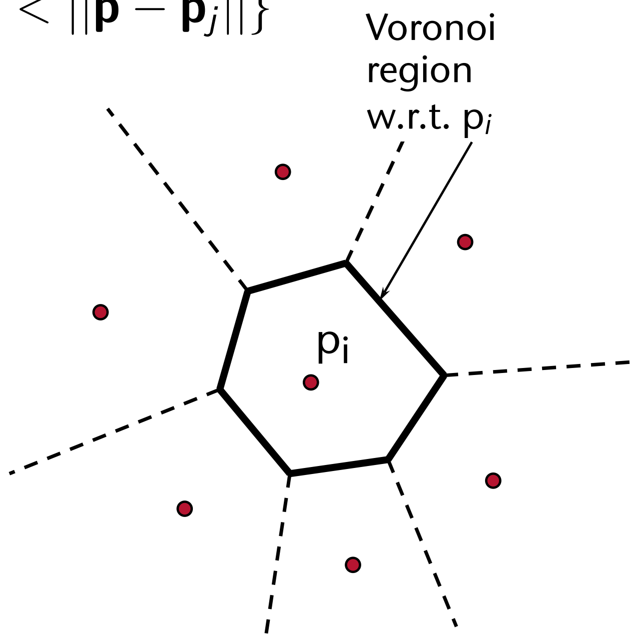
Voronoi Diagrams for Point Sets

Optional

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

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

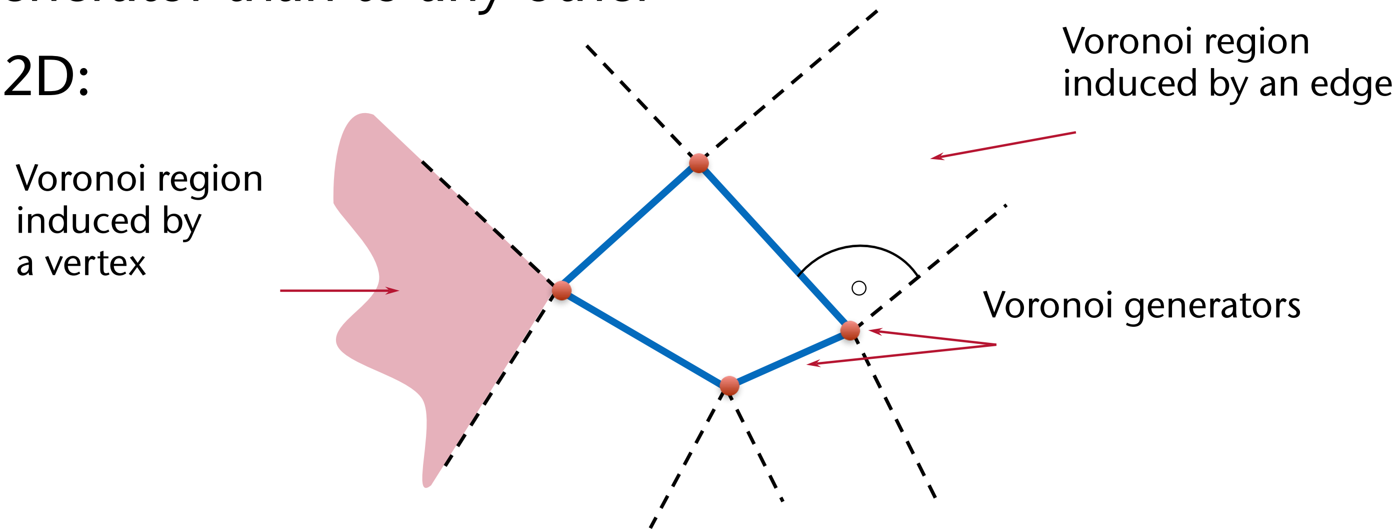
- Definition of Voronoi diagrams:
  The Voronoi diagram $\mathcal{VD}(S)$ over a set of points S is the union of all Voronoi regions over the points in S.
- $\mathcal{VD}(S)$ induces a partition of the plane into Voronoi edges, Voronoi nodes, and Voronoi regions



Optional

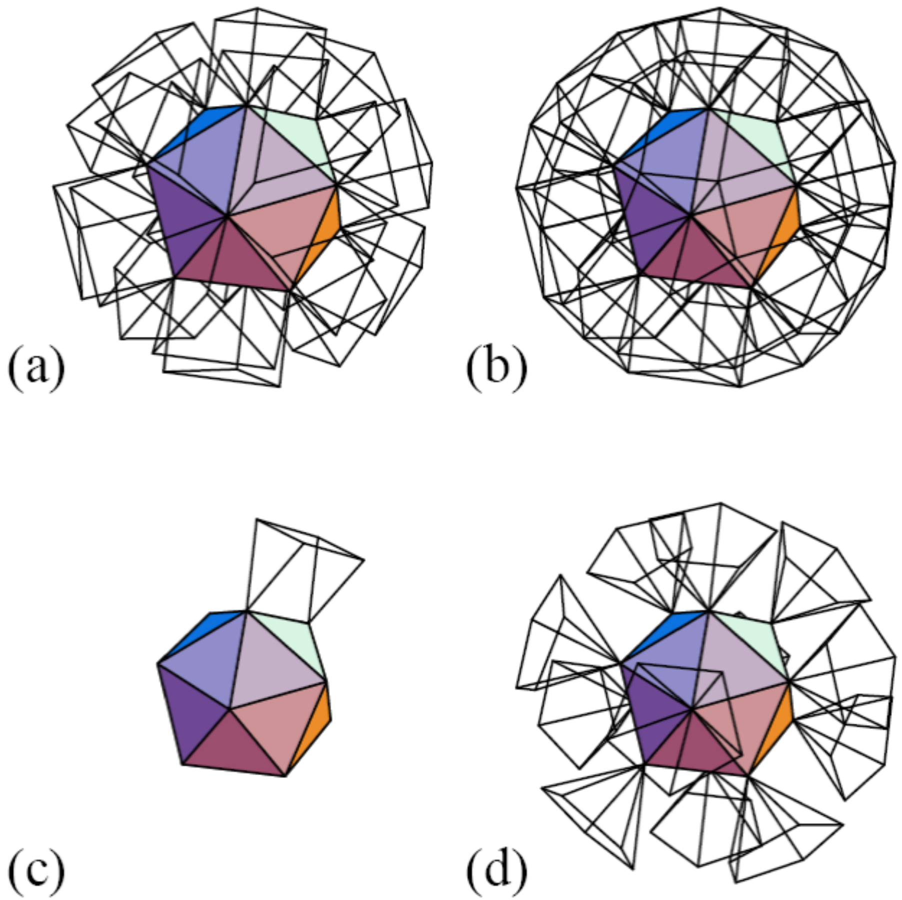
Voronoi Diagrams over Sets of Points, Edges, Polygons

- 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:



Optional

Outer Voronoi Regions Generated by a Polyhedron



The external Voronoi regions of ...

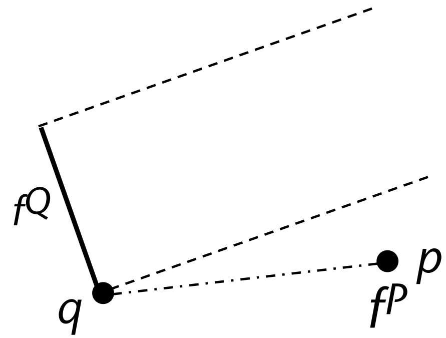
(a) faces
(b) edges
(c) a single edge
(d) vertices

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

Closest Features

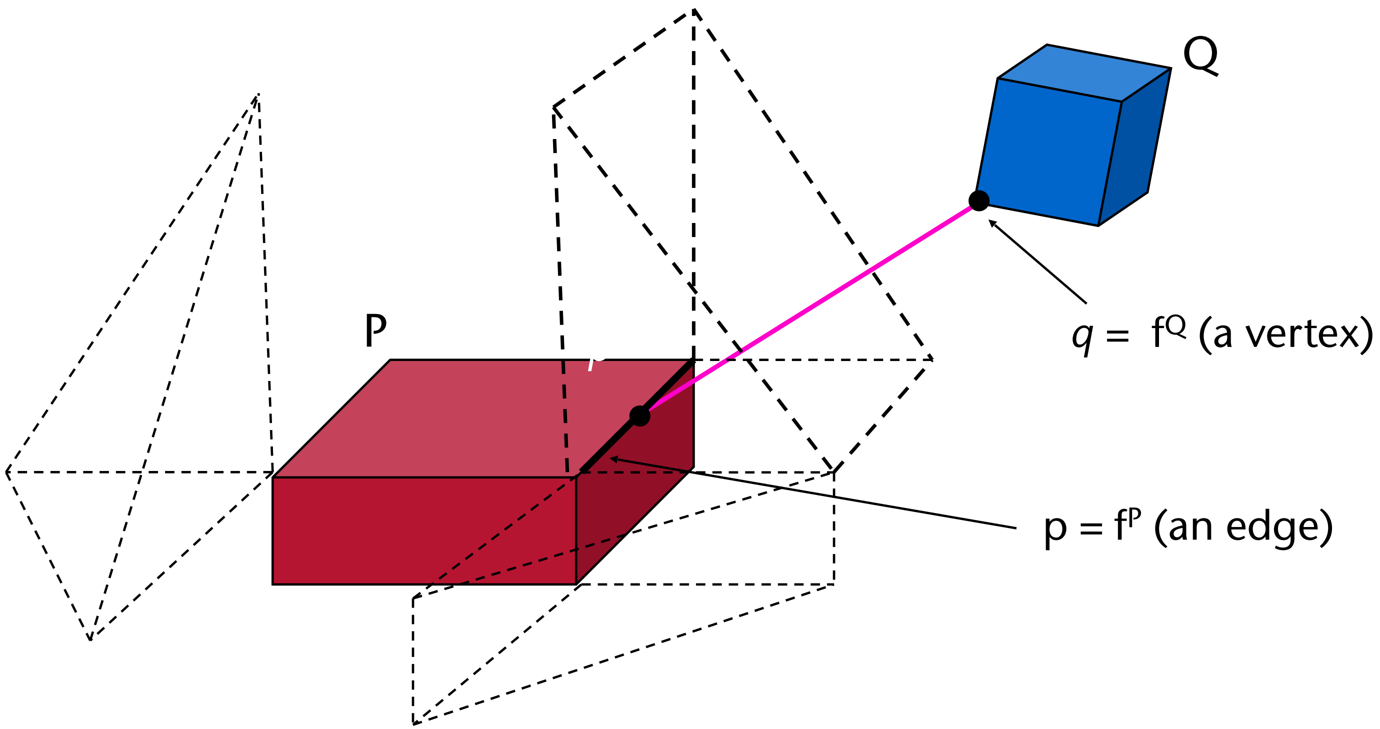
Optional

- 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



Example

Optional



Optional

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 $q \notin V(p)$

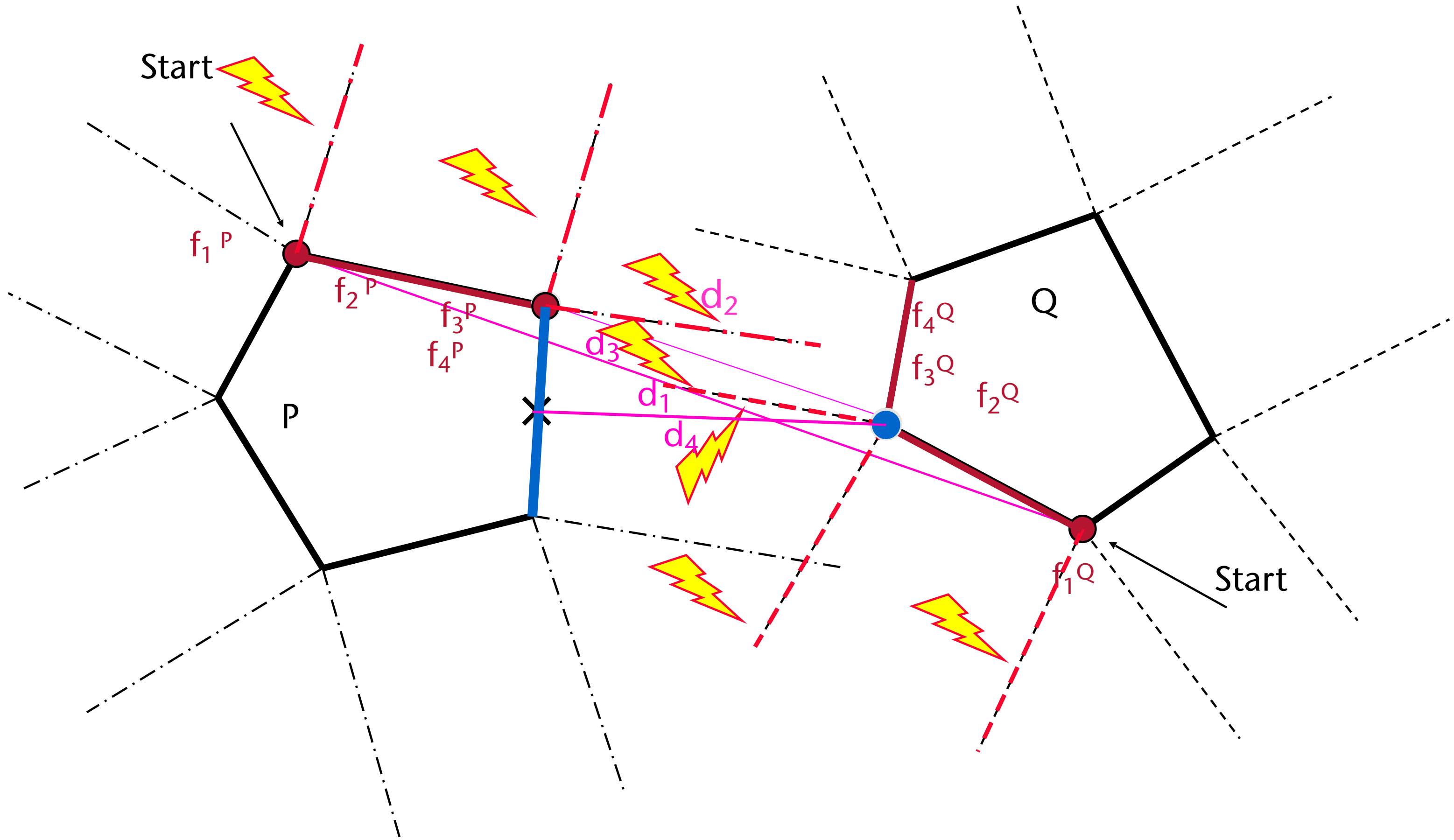
if dist(f^P, f^Q) > 0 :

  return "no collision"

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

Animation of the Algorithm

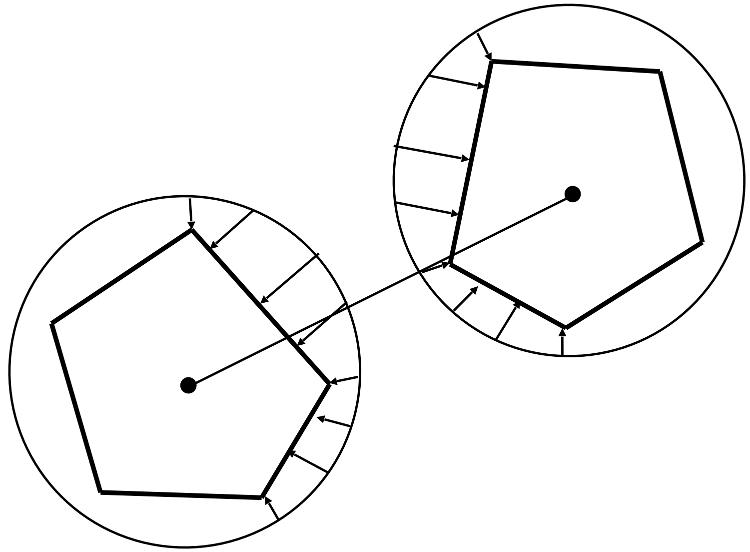
Optional



Some Remarks

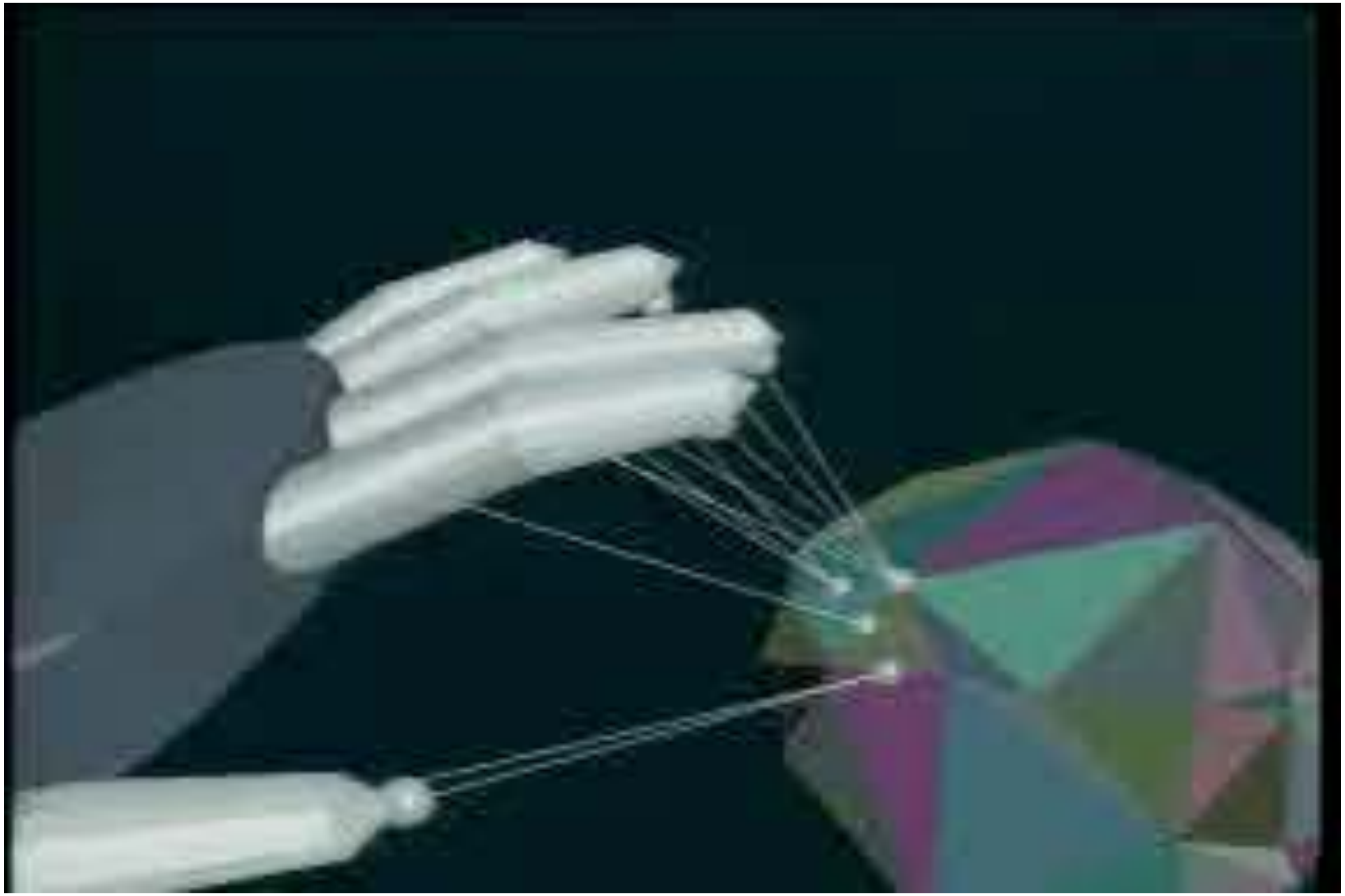
Optional

- 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 projected 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

Optional



UNC-CH

The Minkowski Sum

- Hermann Minkowski (1864 – 1909), German mathematician
- 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:

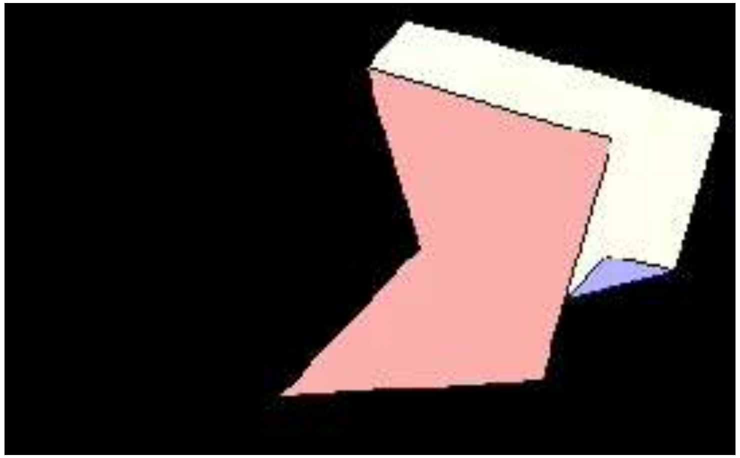
$$A \ominus B = A \oplus (-B)$$

- 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: $A \oplus (B \cup C) = (A \oplus B) \cup (A \oplus C)$
- Invariant against translation: $T(A) \oplus B = T(A \oplus B)$

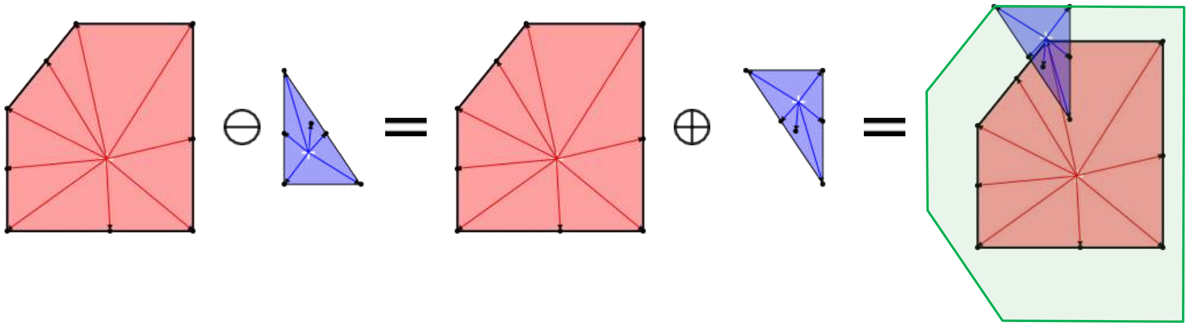
- Intuitive "computation" of the Minkowski sum/difference:



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

- Analogous construction of Minkowski difference:

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

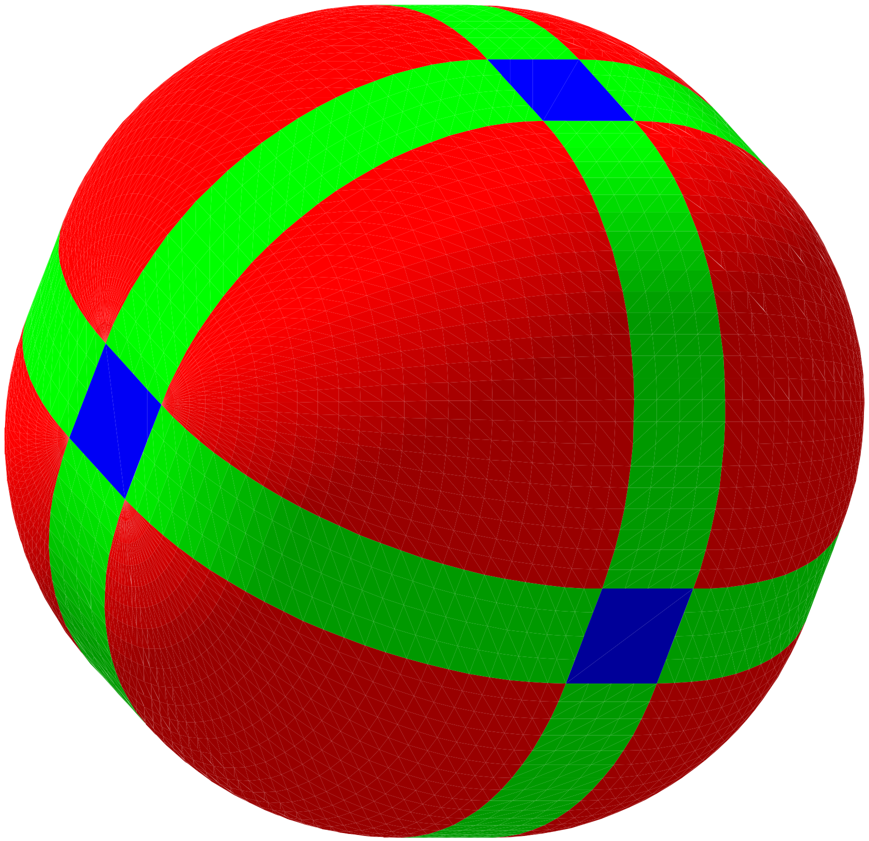


What Objects Were the Original Constituents of this Minkowski Sum?

Don't spoil it by "look-ahead" in the slides!

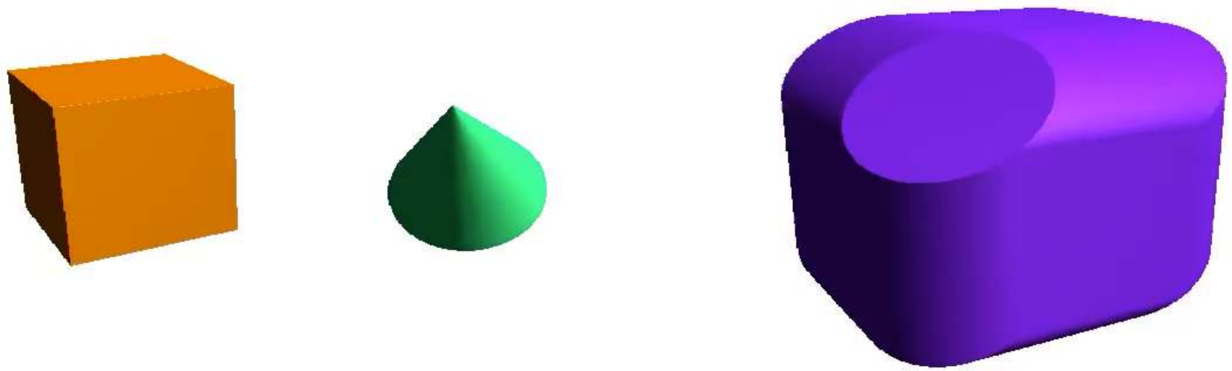
https://www.menti.com/f1b5t74e21

Visualizations of Simple Examples

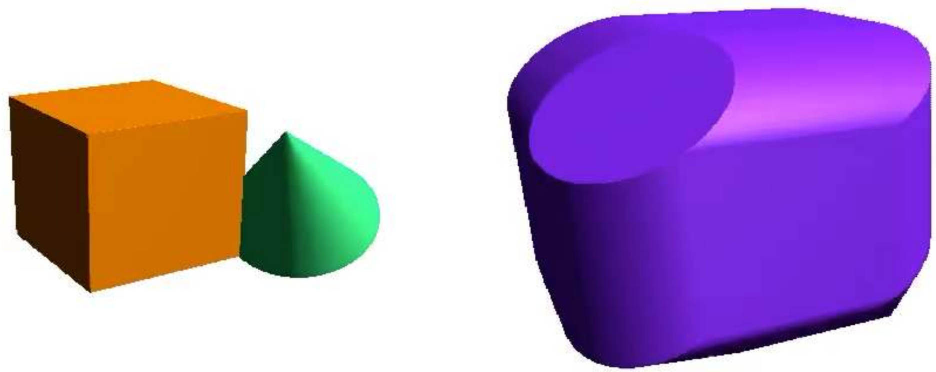


Minkowski sum of a ball and a cube

Minkowski sum of cube and cone, only the cone is rotating



Minkowski sum of cube and cone, both are translating

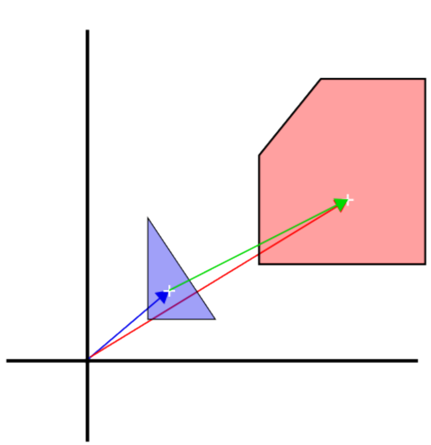


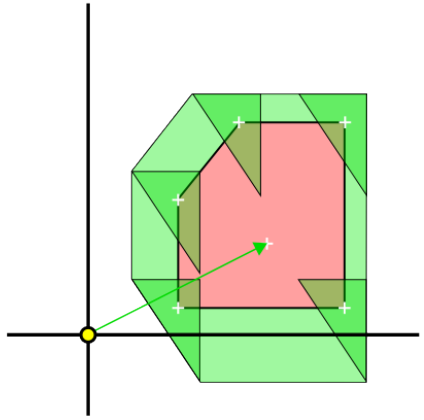
The Complexity of the Minkowski Sum (in 2D, without proofs)

- 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
  - If neither is convex, then $A \oplus B$ has complexity
- Algorithmic complexity of the computation of $A \oplus B$:
  - If A and B are convex, then $A \oplus B$ can be computed in time
  - If only B is convex, then $A \oplus B$ can be computed in randomized time
  - If neither is convex, then $A \oplus B$ can be computed in time

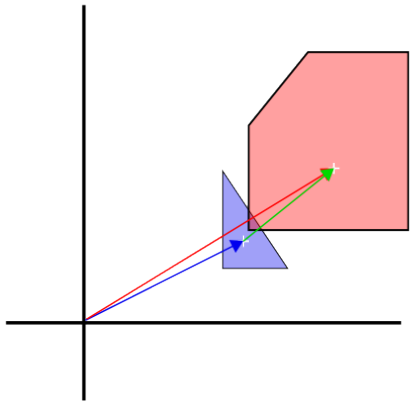
An Intersection Test for Two Convex Objects using Minkowski Sums

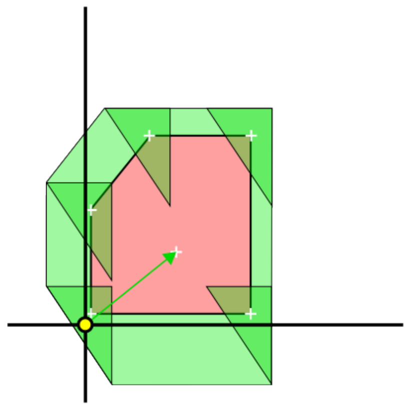
- Compute the Minkowski difference
- A and B intersect ⇔ $0 \in A \ominus B$





- Example where an intersection occurs:

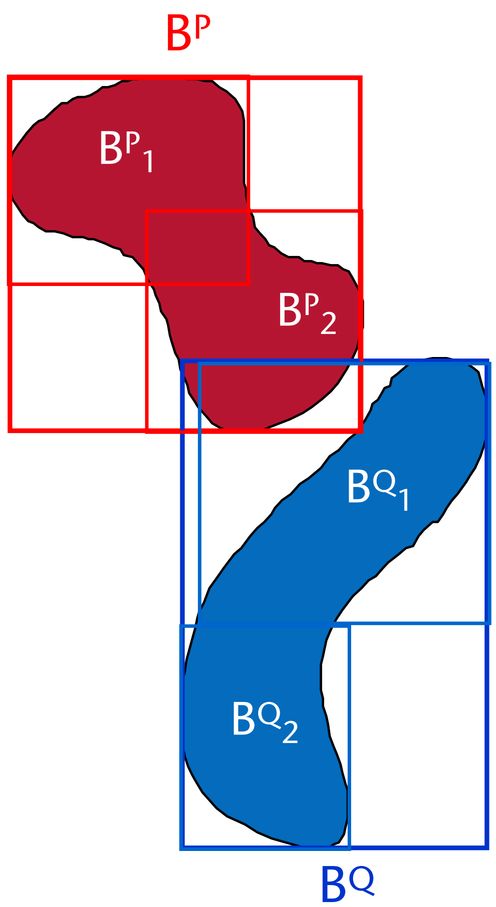




Used in several algorithms, such as
Gilbert-Johnson-Keerthi (GJK)
[see video on the course homepage]

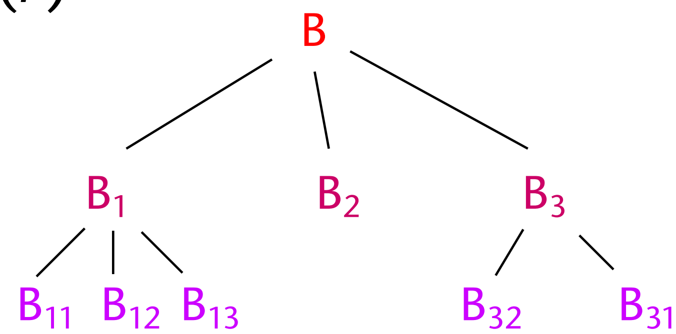
Hierarchical Collision Detection

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

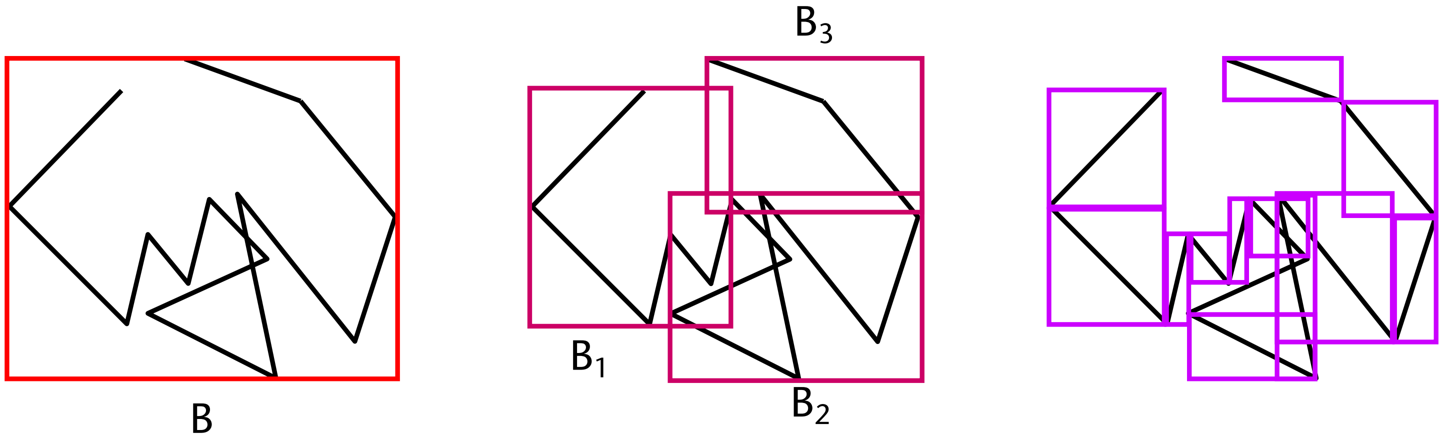


The Bounding Volume Hierarchy (BVH)

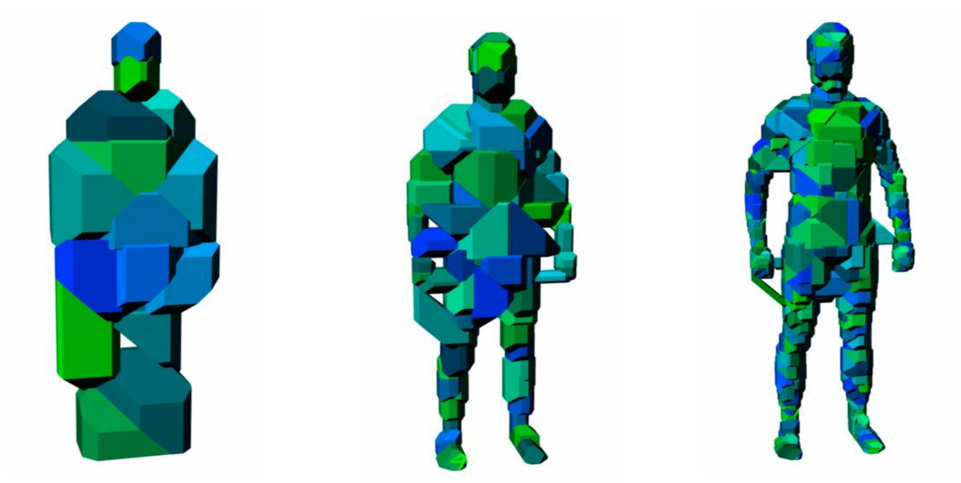
- Constructive definition of a bounding volume hierarchy:
  1. Enclose all polygons, P, in a bounding volume BV(P)
  2. Partition P into subsets $P_1, \ldots, P_n$
  3. Recursively construct a BVH for each P_i and put them as children of P in the tree

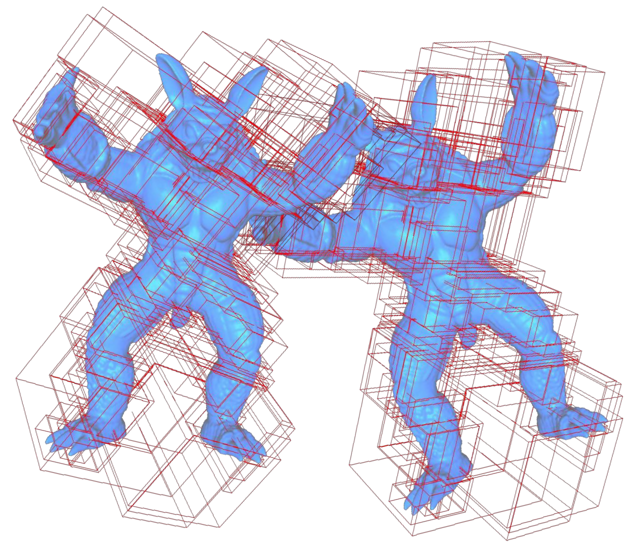


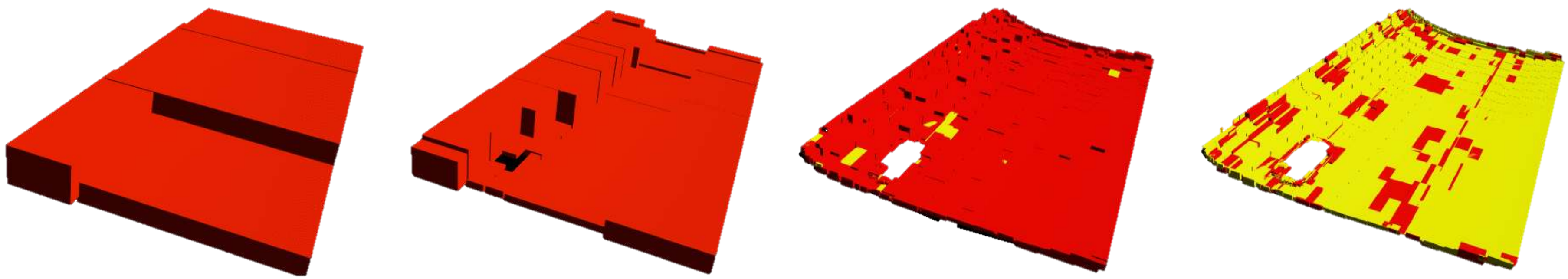
- Typical arity = 2 or 4
- Nodes store BV and pointer to children



Visualizations of Different Levels of Some BVHs



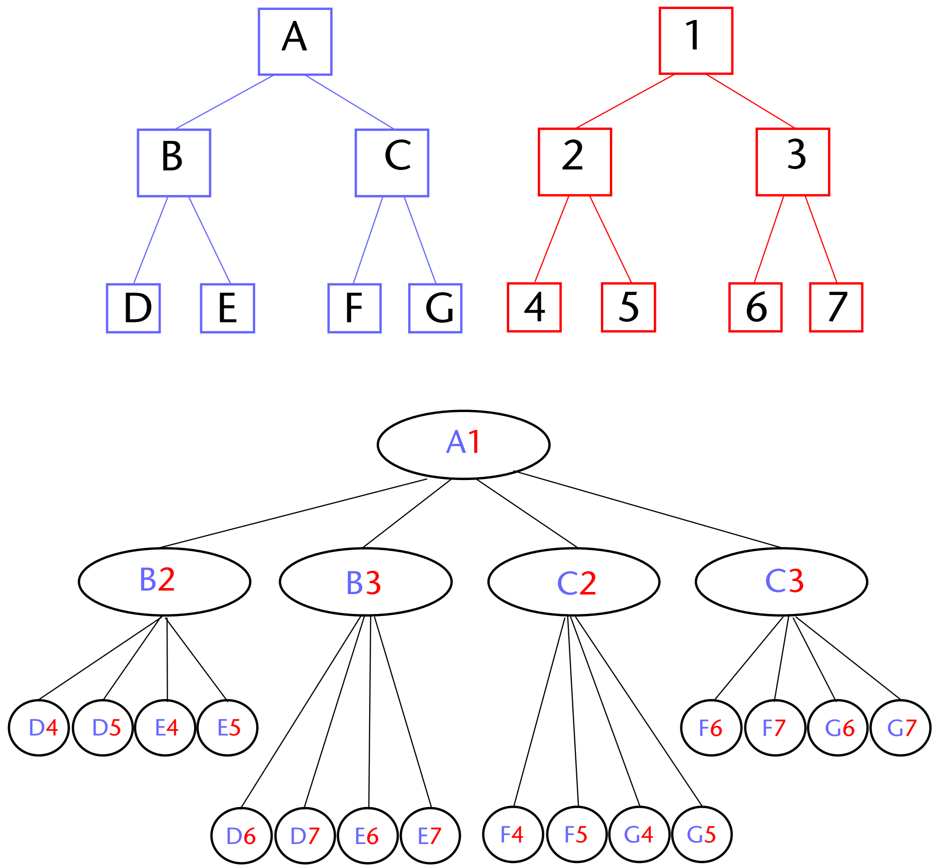




The General Hierarchical Collision Detection Algo

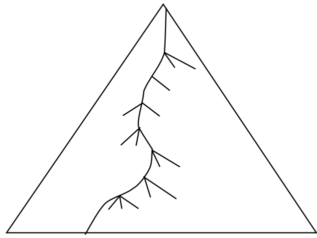
Simultaneous traversal of two BVHs

```
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 )
```



Resulting, conceptual(!) Bounding Volume Test Tree (BVTT)

A Simple Running Time Estimation



- Best-case: $O(\log n)$
- 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, each subtree has ½ of the polygons
  - 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 practice: running time is better/worse depending on degree of overlap

Relationship Between the Type of BV and Running Time

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

Requirements on BV's (for Collision Detection)

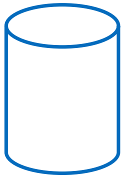
- Very fast overlap test $\longrightarrow$ "simple BVs", even if BV's have been translated/ rotated!
- Little overlap among BVs on the same level in a BVH (i.e., if you want to cover the whole space with the BVs, there should be as little overlap as possible) $\longrightarrow$ "*tight BVs*"

Which Types of BV's Come to Your Mind?

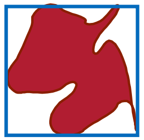
Don't spoil it by "look-ahead" in the slides!

https://www.menti.com/f1b5t74e21

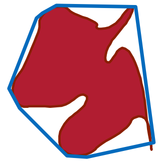
Different Types of Bounding Volumes



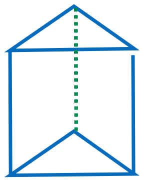
Cylinder
[Weghorst et al., 1985]



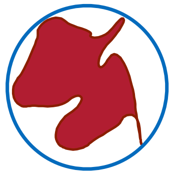
AABB (Axis-aligned bounding box)
(R*-trees) [Beckmann, Kriegel, et al., 1990]



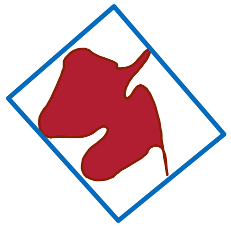
Convex hull
[Lin et. al., 2001]



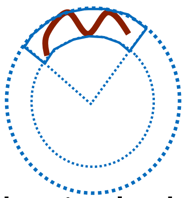
Prism
[Barequet, et al., 1996]



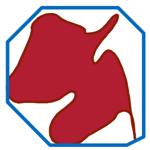
Sphere
[Hubbard, 1996]



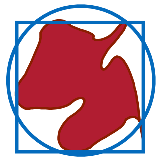
OBB (oriented bounding box)
[Gottschalk, et al., 1996]



Spherical shell
[Manocha, 1997]



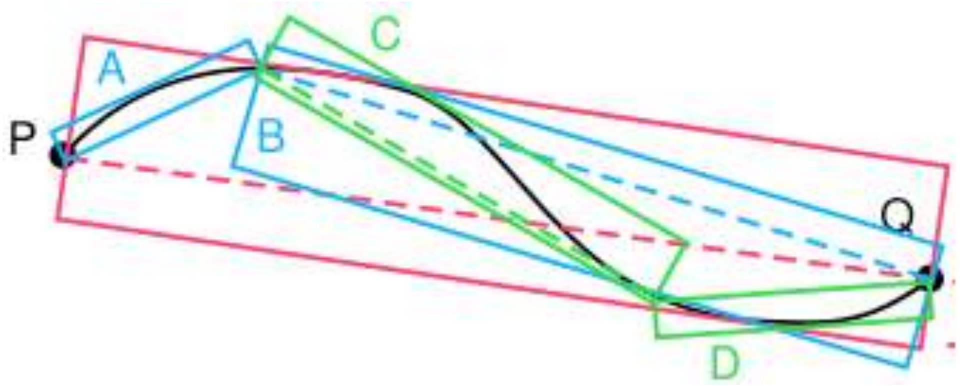
k-DOP / Slabs
[Zachmann, 1998]



Intersection of
several BVs

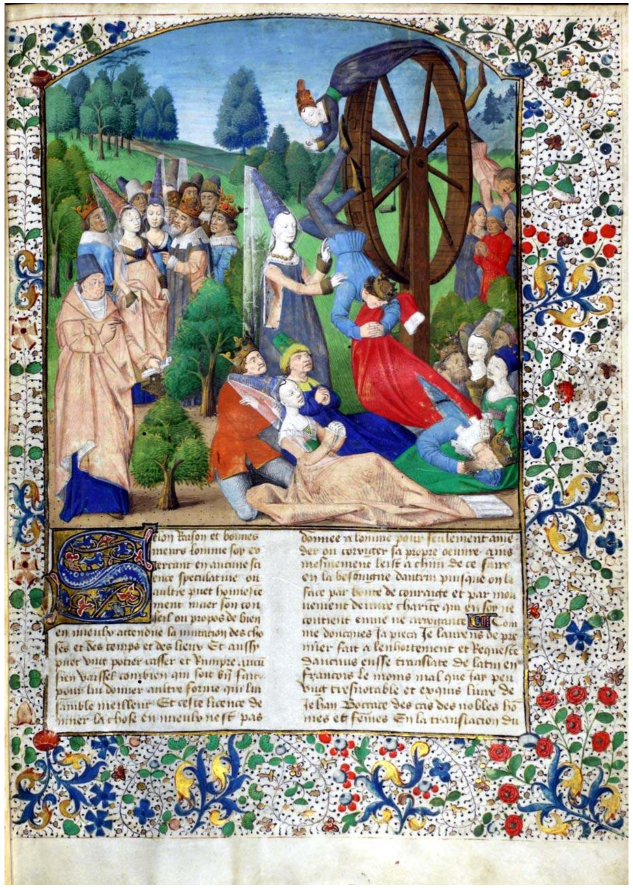
The Wheel of Re-Invention

- 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 (re-invented?) in the 80's 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

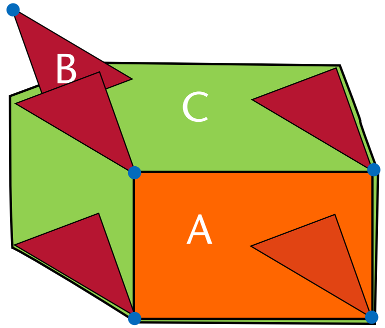
Codex Buranus

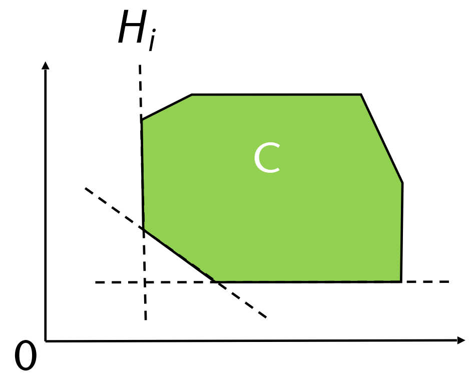
The Intersection Test for Oriented Bounding Boxes (OBB)

- The "separating plane" lemma (aka. "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.
- 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 separating 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.

Proof of the SAT Lemma

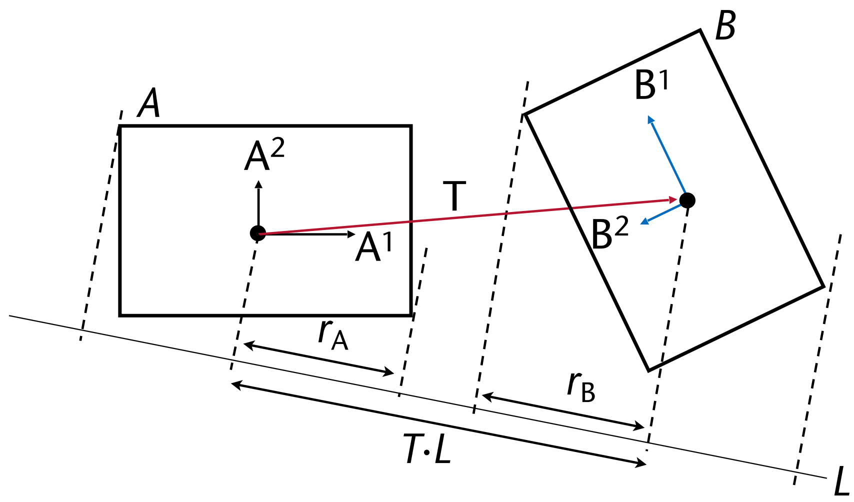
1. Assumption: A and B are disjoint
2. Consider the Minkowski sum $C = A \ominus B$
3. All faces of C are either parallel to one face of A, or to one face of B, or to one edge of A *and* one of B (the latter cannot be seen in 2D)
4. C is convex
5. Therefore: $C = \bigcap_{i=1}^{m} H_i^+$
6. We know: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





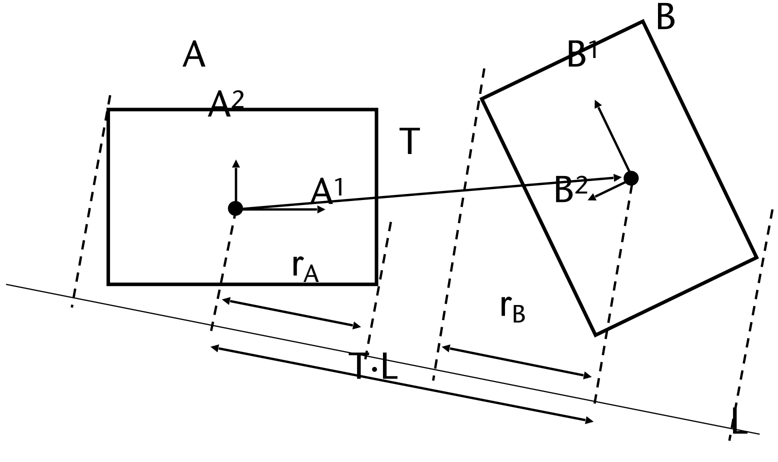
Computing the SAT for OBBs

- Compute everything in the coordinate frame of OBB *A* (wlog.)
- *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 matrix *R*
- Let *L* be a line in space; then *A* and *B* overlap, if $|T \cdot L| < r_A + r_B$
  - Reminder: *L* = normal to the separating plane
- SAT lemma $\rightarrow$ we need to check only a few special lines (15 in case of OBB's)



FYI (not relevant for exam)

- 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:

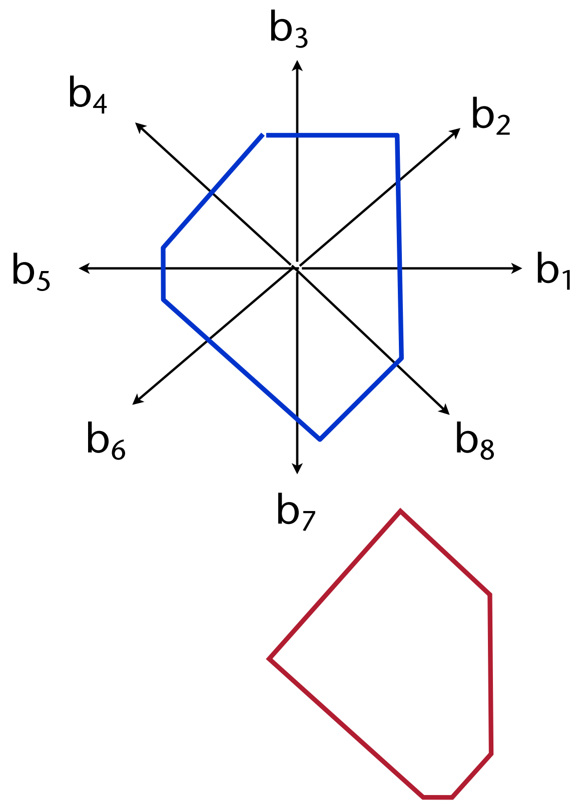
$$|T \cdot L| < a_2|R_{32}| + a_3|R_{22}| + b_1|R_{13}| + b_3|R_{11}|$$

Discretely Oriented Polytopes (k-DOPs)

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

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

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



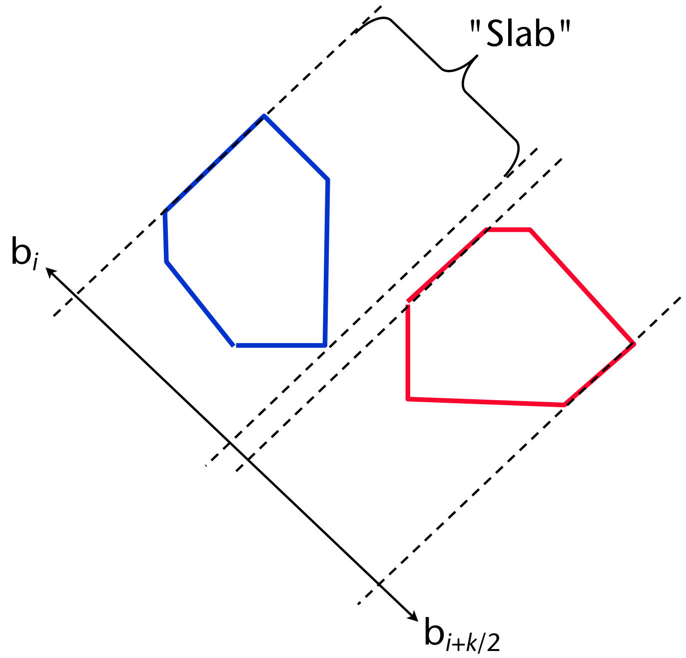
- A *k*-DOP is completely described by $\mathbf{d} = (d_1, \ldots, d_k) \in \mathbb{R}^k$

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

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

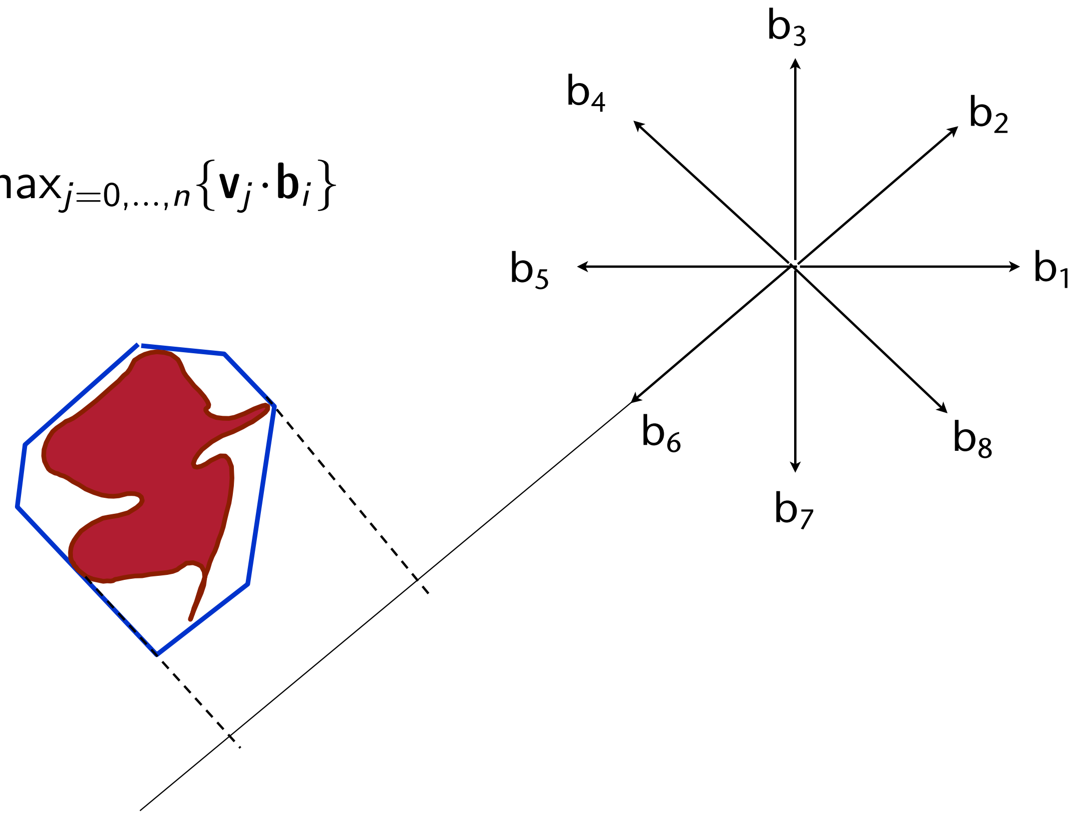
$$\exists i = 1, \ldots, \frac{k}{2} : \left[d_i^1, d_{i+\frac{k}{2}}^1 \right] \cap \left[d_i^2, d_{i+\frac{k}{2}}^2 \right] = \varnothing$$

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



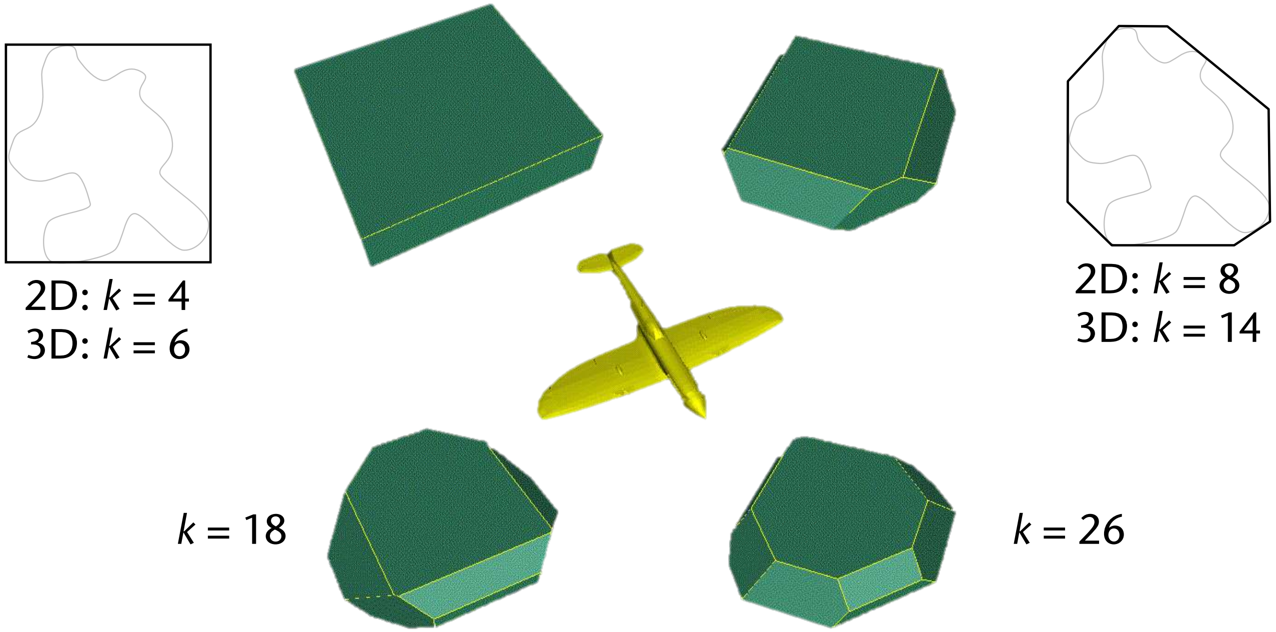
- Note: this is just a generalization of the simple AABB overlap test

- Computation of a *k*-DOP, given a polygon soup with vertices $\mathcal{V}$:
  - $\mathcal{V} = \{\mathbf{v}_0, \ldots, \mathbf{v}_n\}$
  - $D = (d_1 \ldots d_k) \in \mathbb{R}^k$
  - For each *i* = 1, .., *k*, compute $d_i = \max_{j=0,\ldots,n}\{\mathbf{v}_j \cdot \mathbf{b}_i\}$



Some Properties of k-DOPs

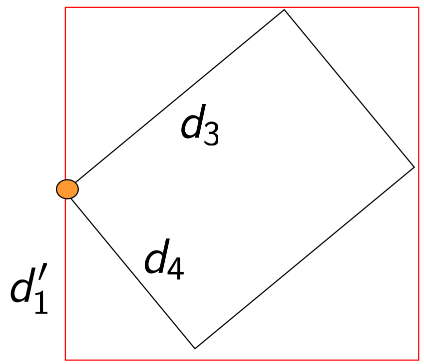
- AABBs are special 6-DOPs
- The overlap test takes time $\in O(k)$, k = number of orientations
- With growing k, the convex hull can be approximated arbitrarily precise



The Overlap Test for Rotated k-DOPs FYI (not relevant for exam)

- 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 (w/o proof):

$$d'_i = \mathbf{b}_i \begin{pmatrix} \mathbf{c}_{j^i_1} \\ \mathbf{c}_{j^i_2} \\ \mathbf{c}_{j^i_3} \end{pmatrix}^{-1} \begin{pmatrix} d_{j^i_1} \\ d_{j^i_2} \\ d_{j^i_3} \end{pmatrix} + \mathbf{b}_i T,$$

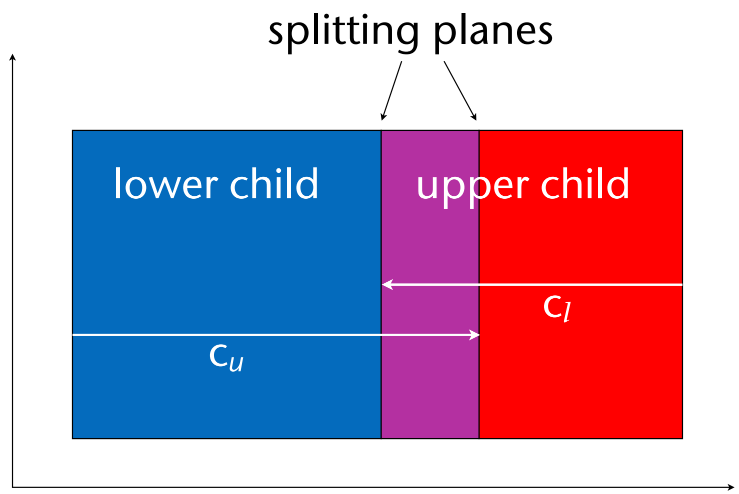


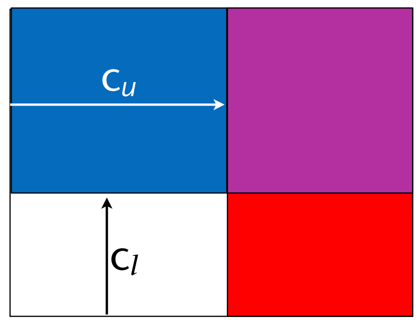
with $\mathbf{c}_j = \mathbf{b}_j R^{-1}$

  - The correspondence $j^i{}_l$ is identical for all DOPs in the same hierarchy (thus, it can be precomputed, and the red terms, too)
  - 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:
  - For defining the children of a node B:
    for the left child, split off a portion of the "right" part of the box B ⟶ "lower child";
    for the right child of B, split off a portion of the left part of B ⟶ "upper child"
- Memory usage: 1 float, 1 axis ID, 1 pointer (= 9 bytes), can fit into 8 bytes
- Other names for the same thing:
  - Bounding Interval Hierarchy (BIH)
  - Spatial kd-tree (SKD-Tree)





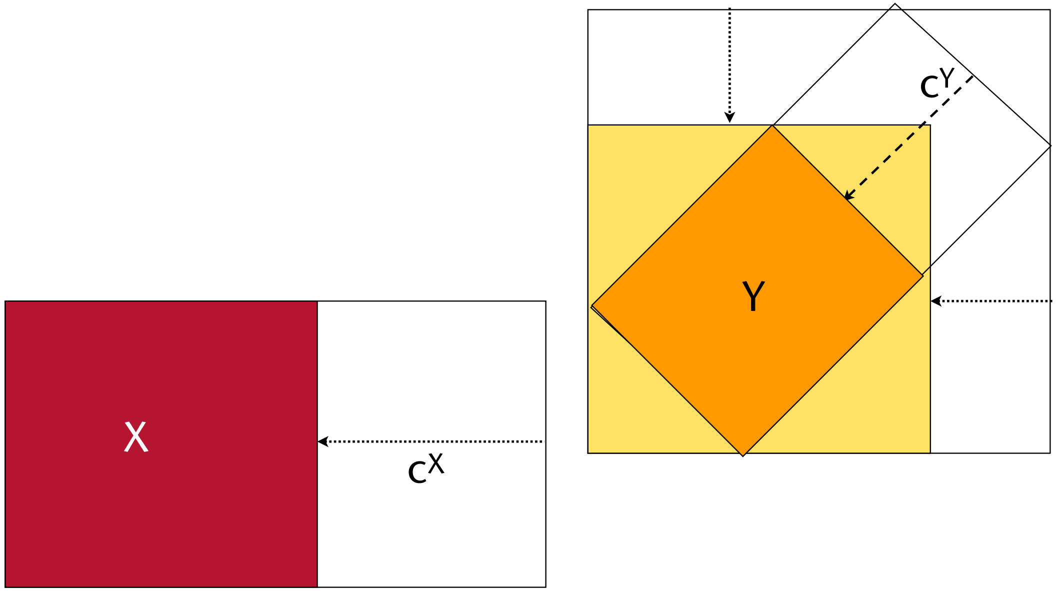
[Zachmann, 2002]

Just FYI

- Overlap tests by "re-alignment" (i.e., enclosing the non-axis-aligned box in an axis-aligned one, exploiting the special structure of restricted boxtrees):

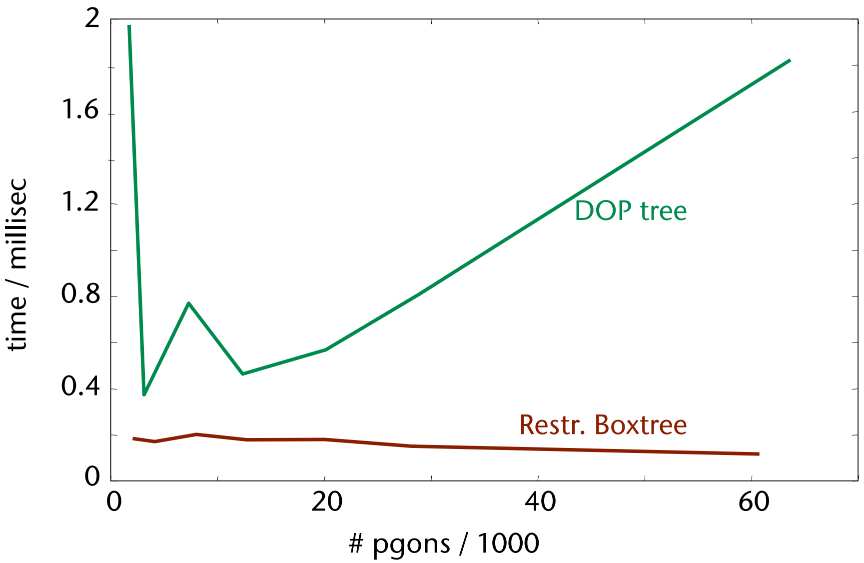
  12 FLOPs (8.5 with a little trick)

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

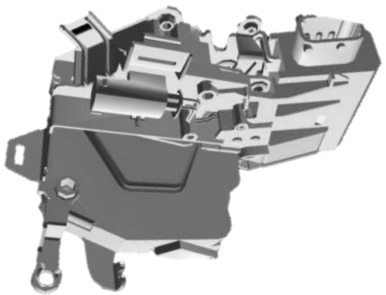


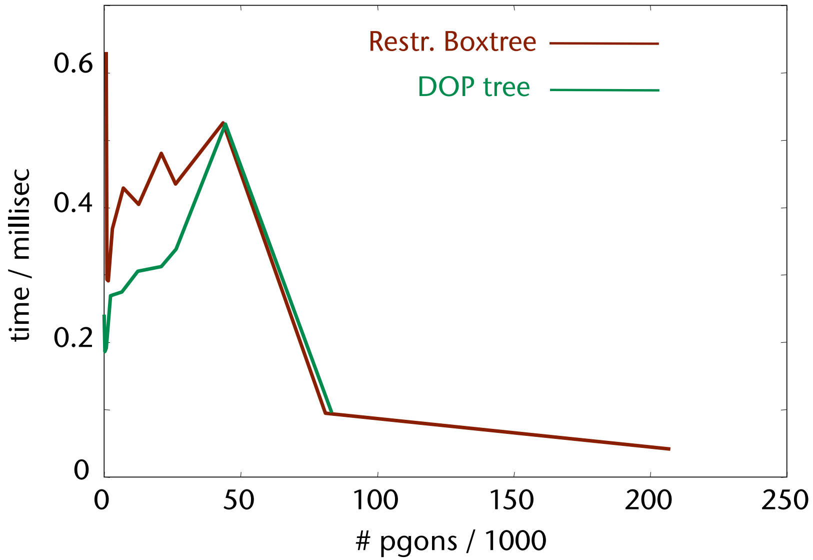
Performance

Car (courtesy VW)



Door lock (BMW)





Master's Thesis Topics

CONFIDENTIAL

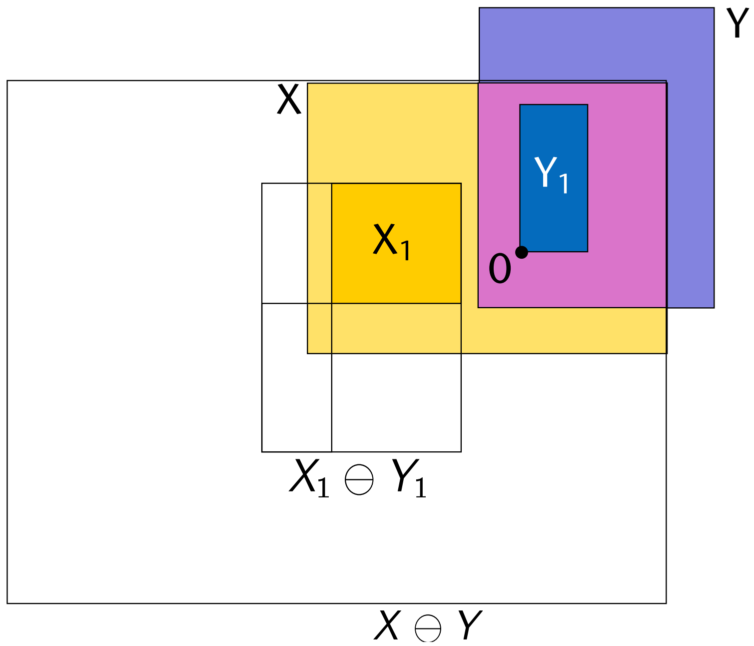
- Investigate the BVH presented in Bauszat et al., "The Minimal Bounding Volume Hierarchy" (2 bits per node!):
  - Can it be used for coll.det.?
  - Would it be faster than my "Minimal Hierarchical Collision Detection" (2002)?
  - How many polygons an the BVH represent and still fit into the L1/L2 cache?
  - Can the BVH be stored such that proximal parts of the obj are contiguous in memory (and thus can be loaded in the cache)?
  - Can it be combined with the SSE/AVX instruction set?

The Construction of BV Hierarchies

- Obviously: if the BVH is bad $\longrightarrow$ 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 criterion?
- Guiding principle: the expected cost for collision detection incurred by a particular split is

$$C(X,Y) = c + \sum_{i,j=1,2} P(X_i,Y_j)\,C(X_i,Y_j) \approx c'\big(P(X_1,Y_1) + \cdots + P(X_2,Y_2)\big)$$

- Given: parent boxes X, Y (intersecting)
- Goal: estimation of $P(X_i,Y_j)$
- Our tool: the Minkowski sum
- Reminder: $X_i \cap Y_j = \emptyset \;\Leftrightarrow\; 0 \notin X_i \ominus Y_j$
- Therefore, the probability is:



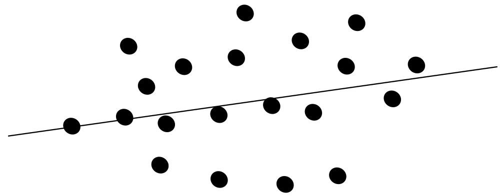
$$P(X_i, Y_j) = \frac{\text{Vol}(\text{ "good" cases})}{\text{Vol(all possible cases)}}$$

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

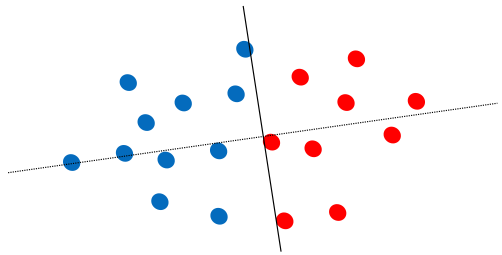
- Conclusion: for a good BVH (in the sense of fast coll.det.), minimize the total volume of the children of each node

The 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 are not too uneven, i.e., a fraction of α and (1-α) polygons goes into the left/right subtree, resp., and is α not "too small"

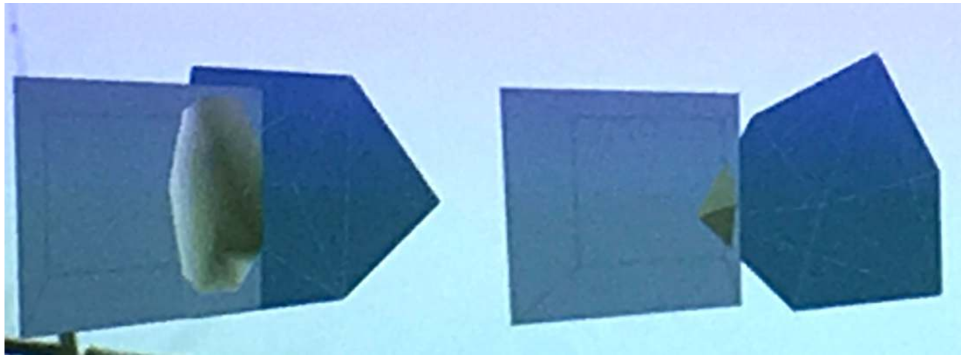
What Could be a Good Measure of Penetration of Virtual Objects?

Don't spoil it by "look-ahead" in the slides!

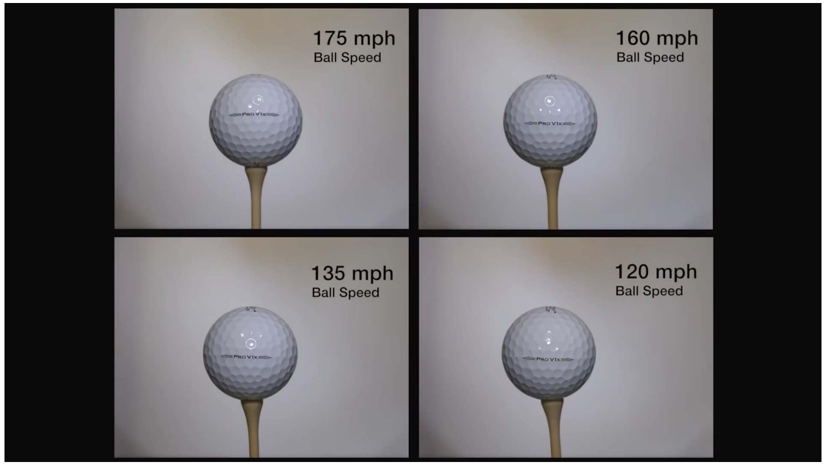
https://www.menti.com/f1b5t74e21

Penetration Measures

- Penetration distance
  - Various forms
  - Suitable for penalty forces generated by ad-hoc "virtual" springs
- Penetration volume
  - Intuitive
  - Physically motivated: buoyancy force of floating objects = vol. of displaced water
  - Continuous
  - Related to deformation energy of colliding objects
  - Requires representation of inner volume of objects



In the configuration on the left, the penetration should be "higher" than in the configuration on the right

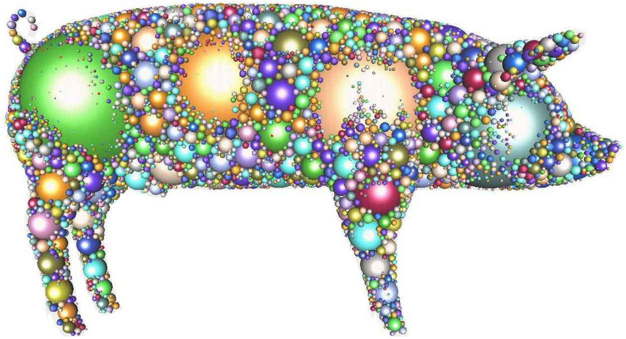


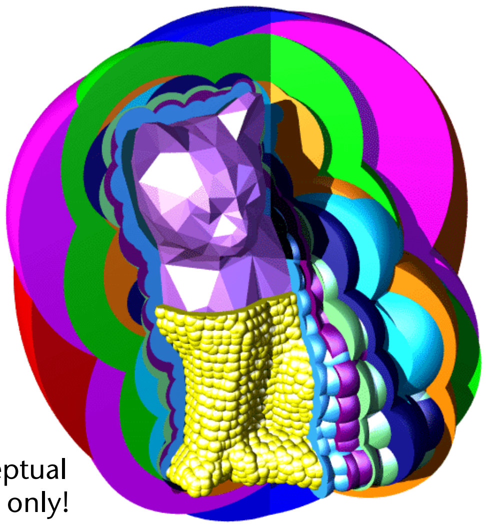
Inner Sphere Trees: the Basic Idea

- Challenge: compute proximity, i.e., distance *or* measure of penetration
- Approach: don't approximate an object from the outside; instead, approximate it
  - from the *inside* ,
  - with *non-overlapping* spheres, and
  - with as little empty volume as possible

➢ Sphere packing

- Build sphere hierarchy on top of *inner* spheres

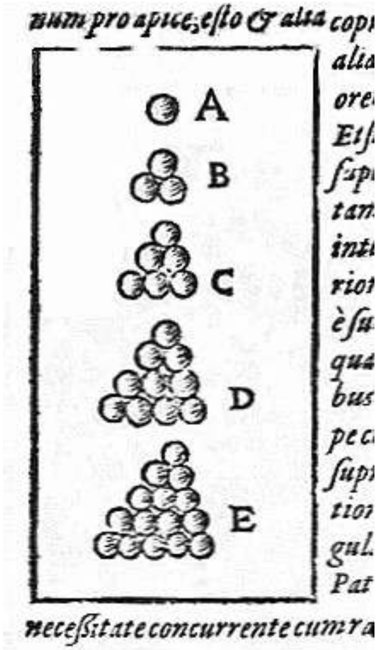




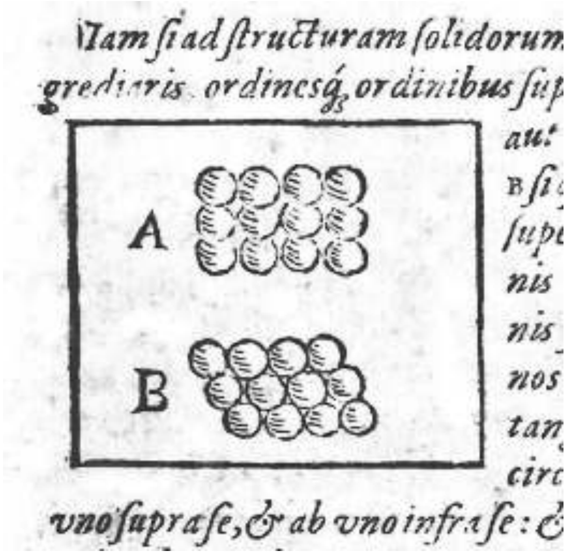
Conceptual image only!

The Long History of Sphere Packings

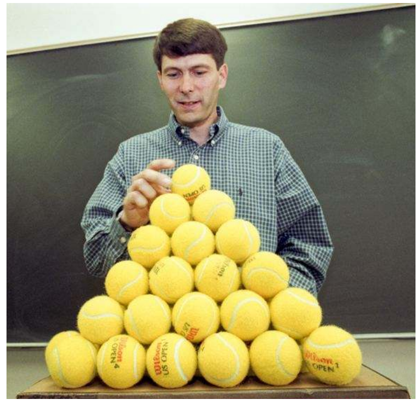
Johannes Kepler
(1571 – 1630)



Kepler's Conjecture
(1611)



$$V = \frac{\pi}{\sqrt{18}} \approx 74\%$$

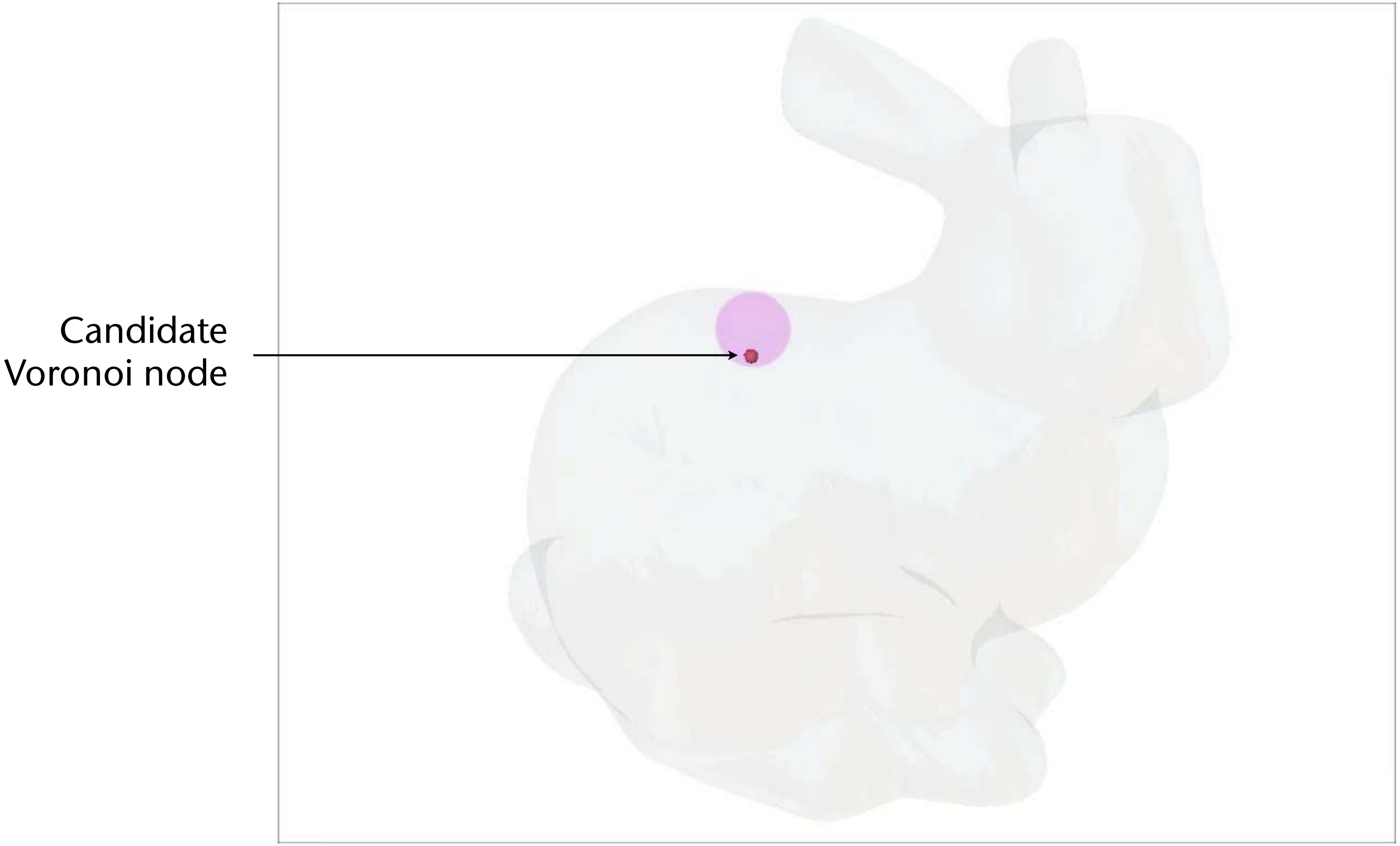


Mathematical proof in 1998 by Thomas Hales and Samuel Ferguson

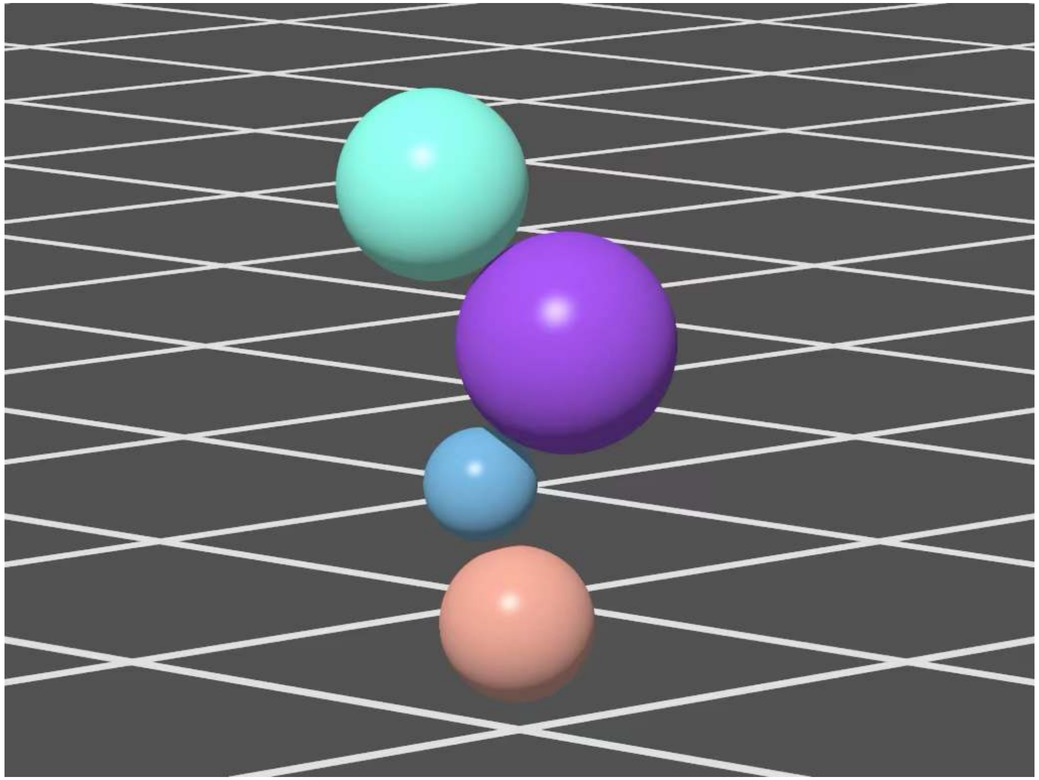
Protosphere

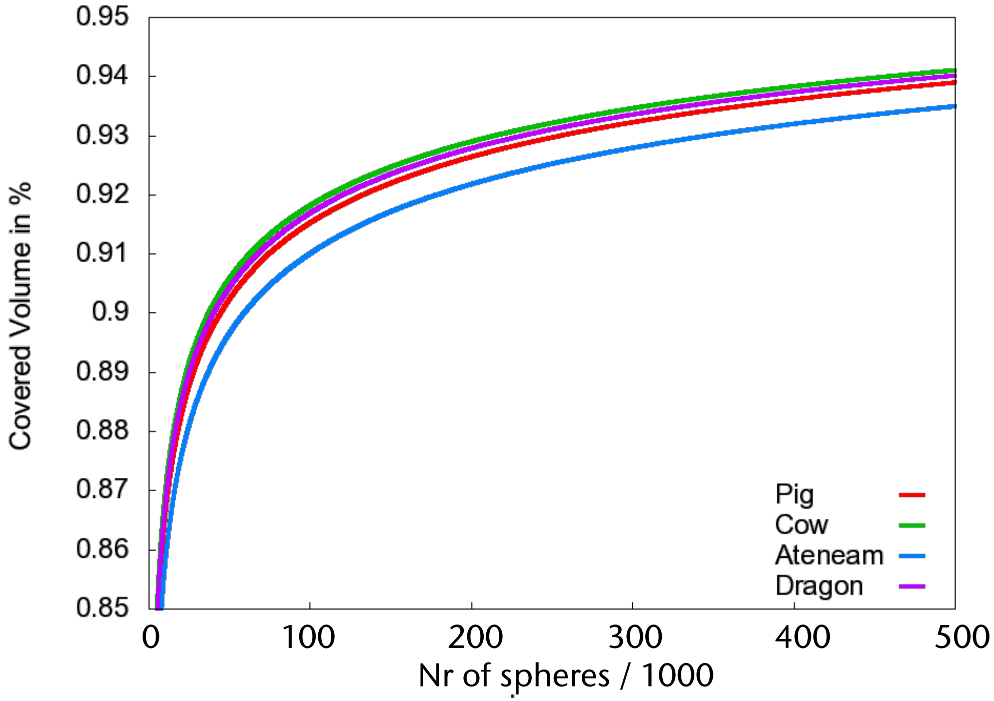
- Our requirements / variety of sphere packings:
  - Non-overlapping
  - Arbitrary radii
  - Must work for any kind of container (not just boxes)
- Optimization according to some criteria, e.g. number of spheres
- Our approach:
  - Find inner Voronoi nodes of container object
    - (See course "Computational Geometry for CG")
    - In our case, use approximation by iterative algorithm
  - Place spheres
  - Compute new Voronoi nodes of object *plus* spheres

Visualization of Our Algorithm



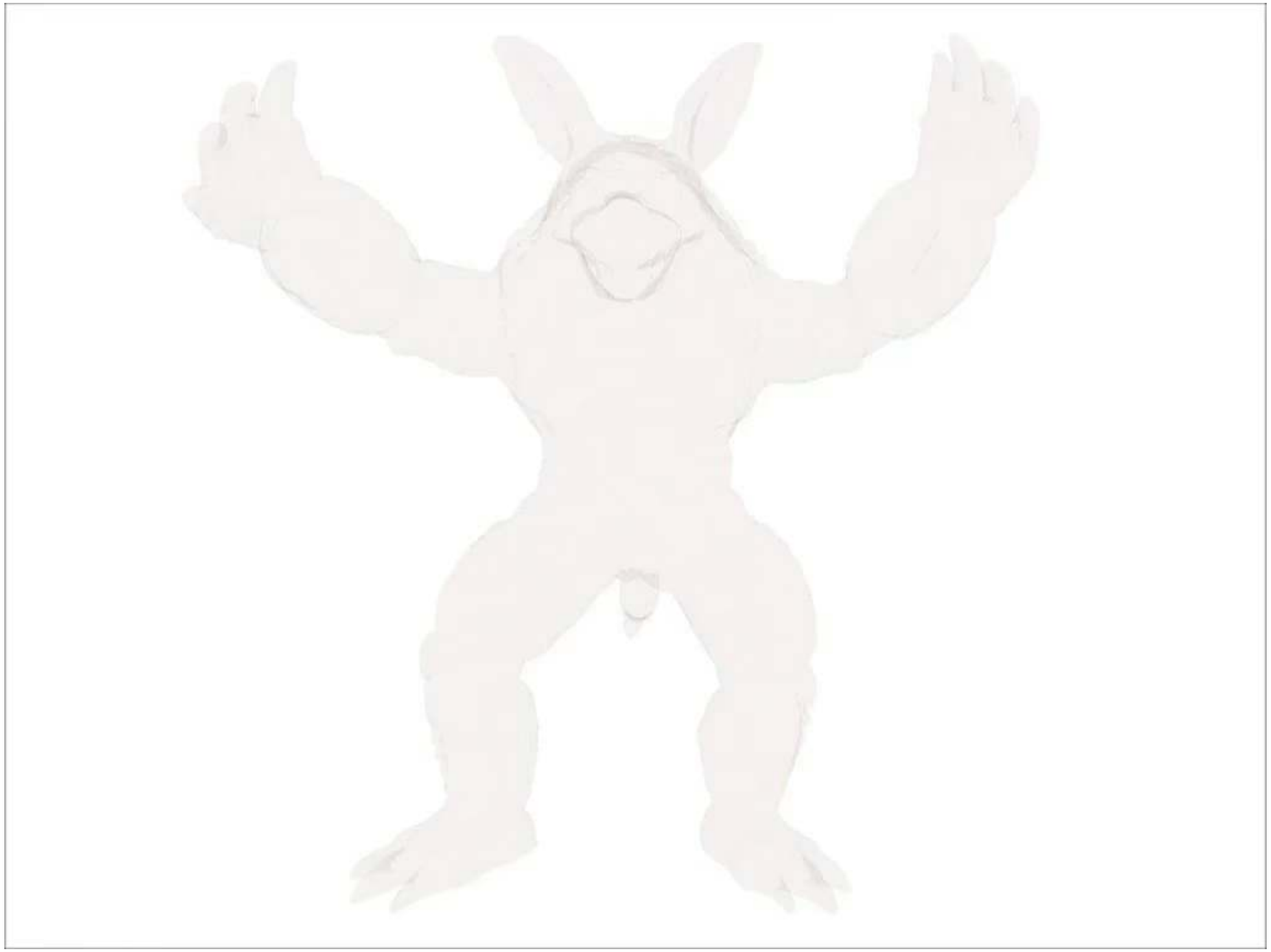
Results



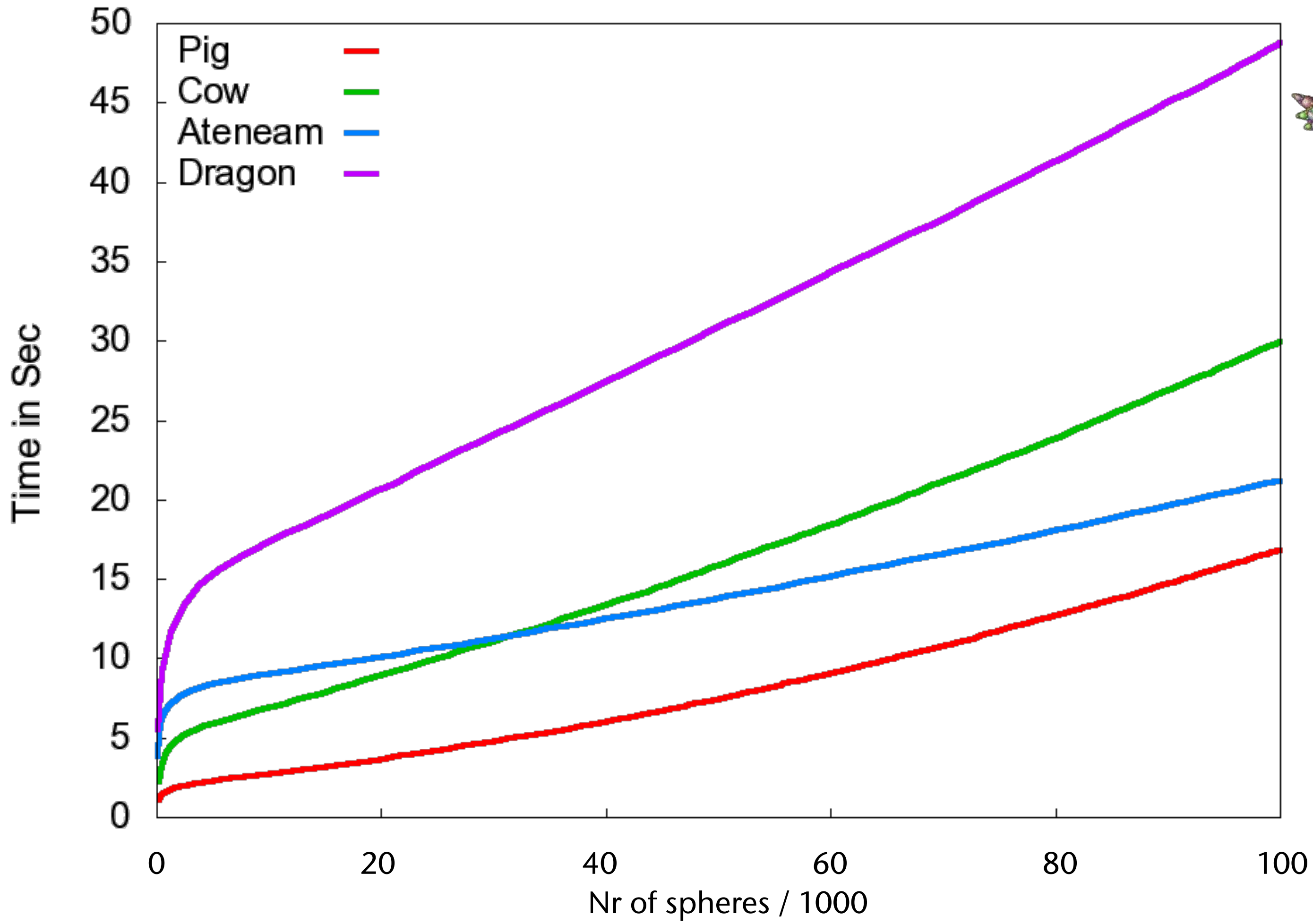


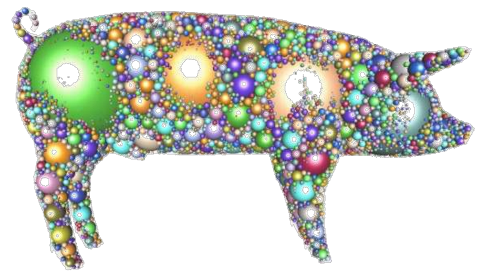
0.95
0.94
0.93
0.92
0.91
0.9
0.89
0.88
0.87
0.86
0.85
Covered Volume in %
0
100
200
300
400
500
Nr of spheres / 1000
Pig
Cow
Ateneam
Dragon

The Algorithm can be Parallelized for the GPU



Performance of Construction of Sphere Packing

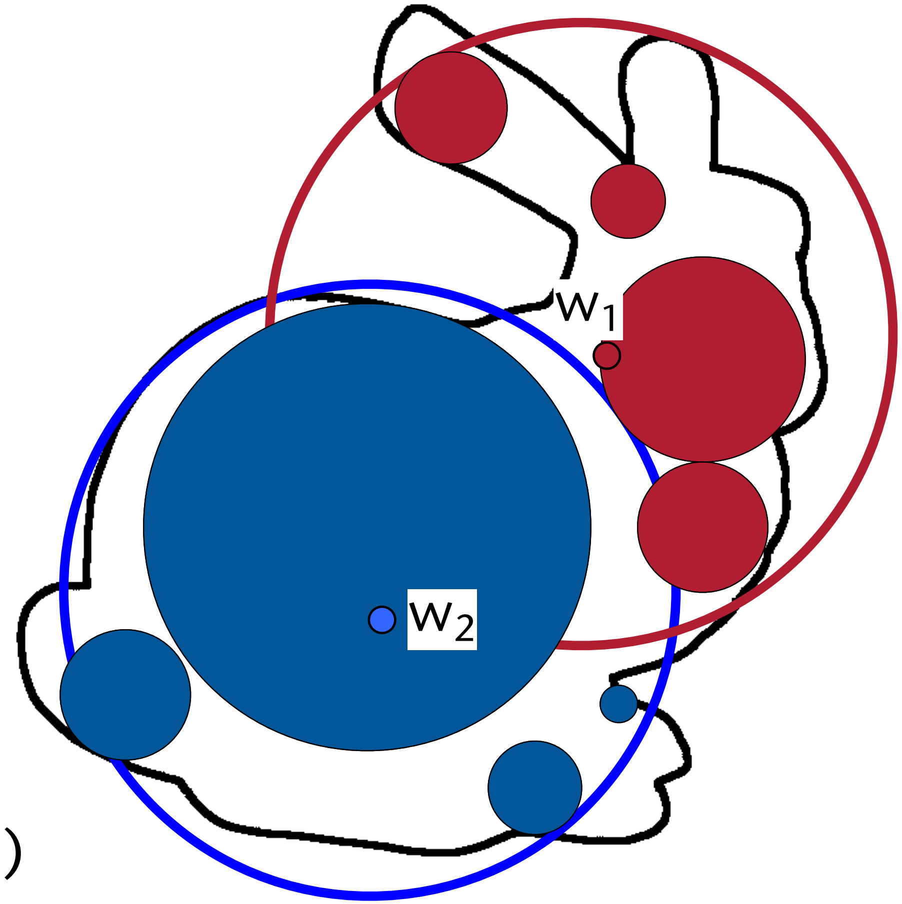




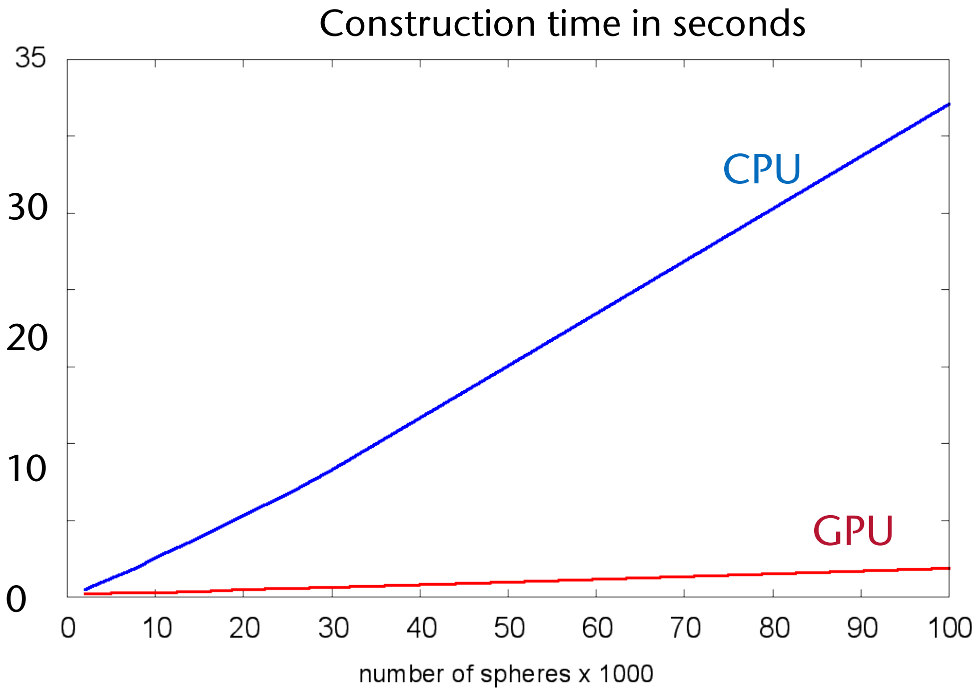
Nvidia Geforce GTX 480

Construction of Hierarchy Over Sphere Packing

- IST = sphere tree over sphere packing
- Constructions is based on a clustering method known from machine learning (*batch neural gas* clustering)
  - Bears some resemblance to k-means, but more robust against outliers and starting configuration
- We can assign "importance" to spheres
- Parallelizable on the GPU
- Naturally generalizes to higher tree degrees (out-degree of 4-8 seems optimal)

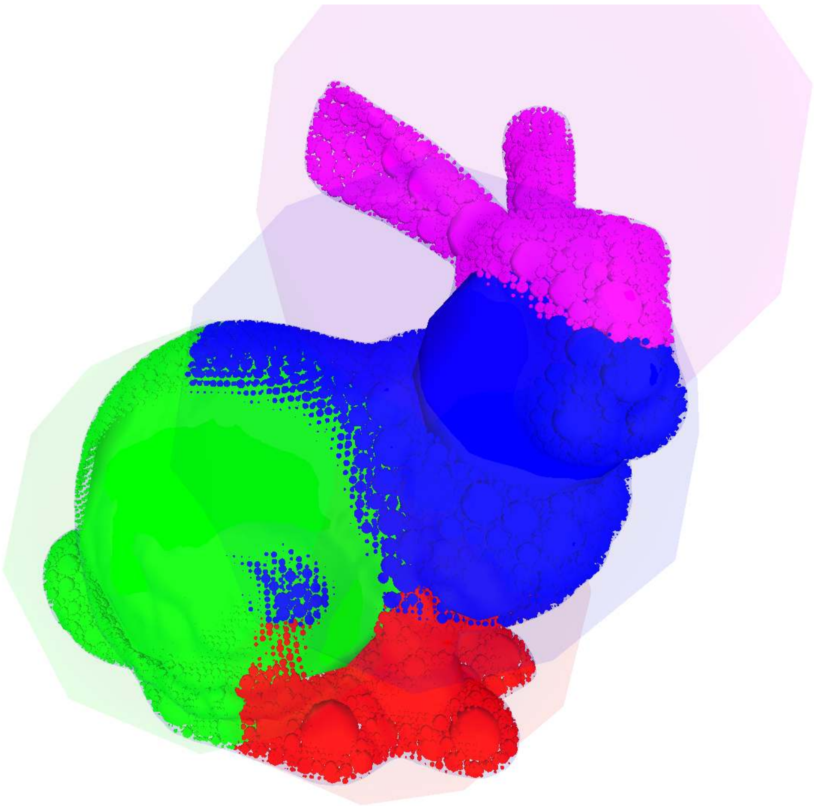


- BNG hierarchy construction on CPU has complexity of $O(n \log n)$
- Parallelization of BNG reduces complexity to $O(\log^2 n)$

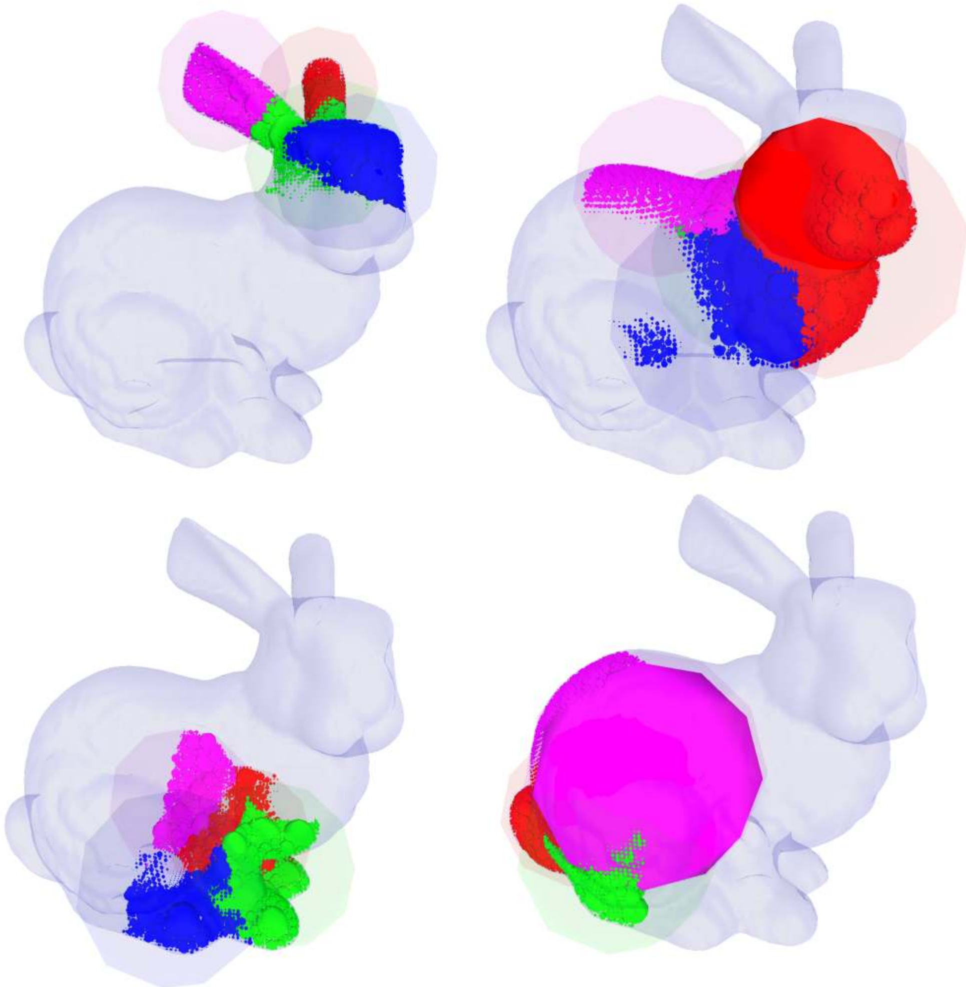


Geforce GTX 780

Examples



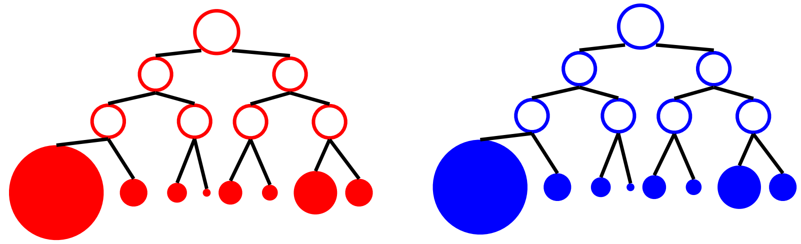
Clustering underneath root



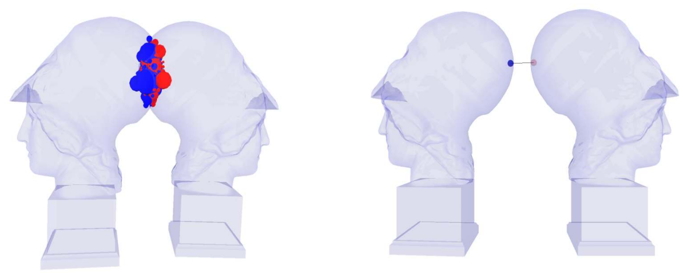
Clustering underneath level 1 nodes

Proximity / Penetration Query Using ISTs

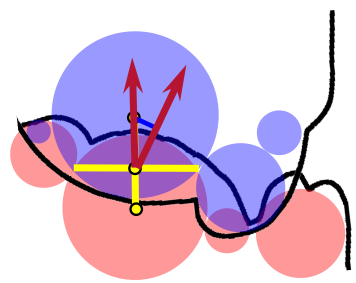
- Works by the standard simultaneous traversal of BVHs



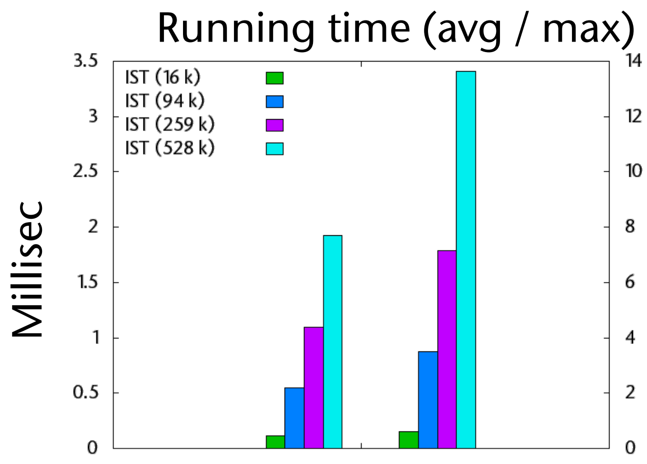
- First algo that can compute both *minimal distance* or *intersection volume* with one *unified* algorithm

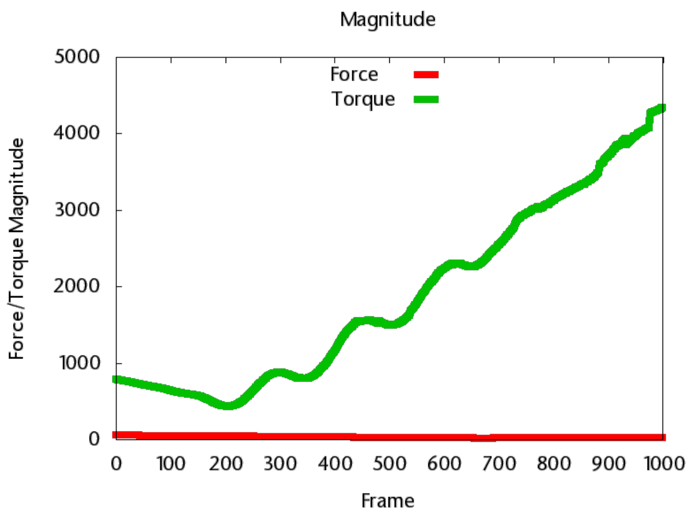


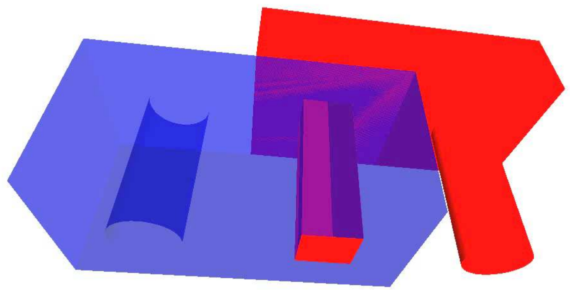
- Can compute forces and torques
  - Which can be proven to be continuous



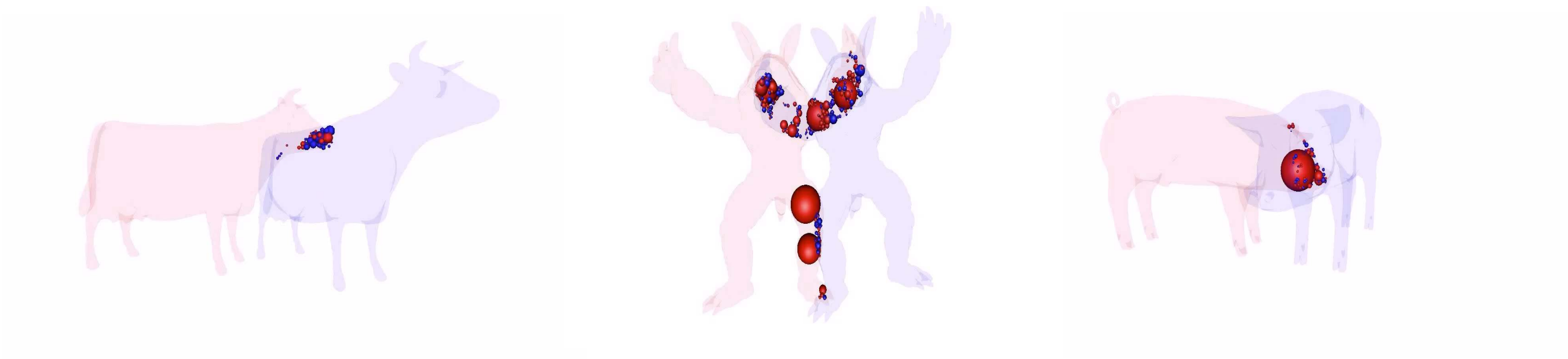
Computation Timings for the Intersection Volume

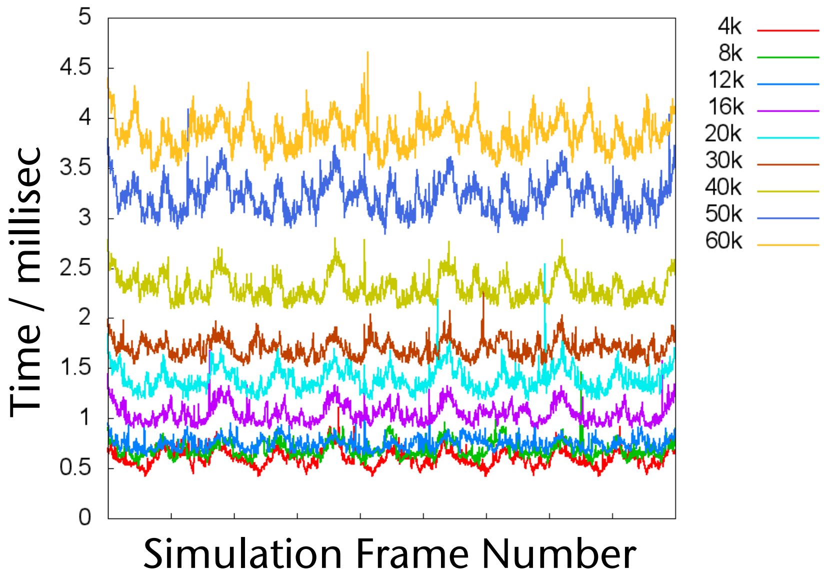


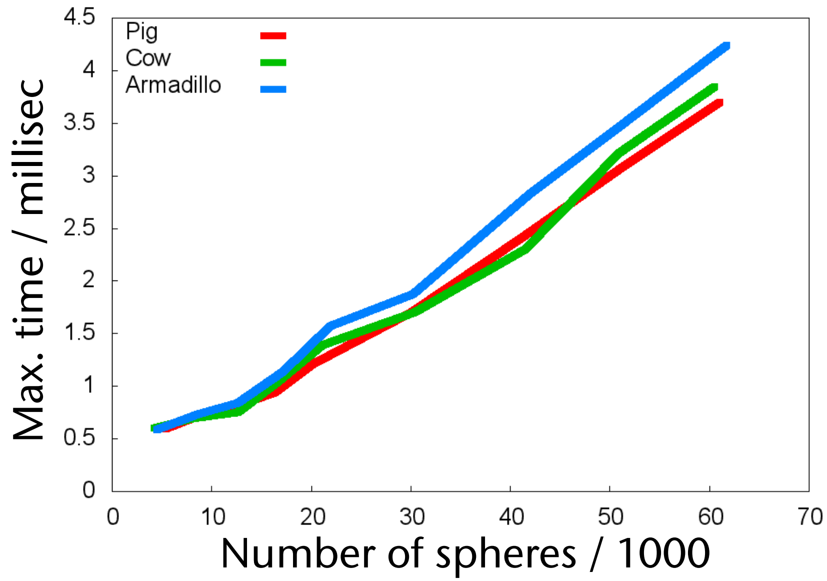




Parallel Computation Times for Intersection on GPU







Penalty Forces for Simulation/Force-Feedback

- Accumulate sphere-sphere interaction forces:
  - Linear force:

$$\mathbf{f}_{ij}^{\text{blue}} = \text{Vol}(s_j^{\text{red}} \cap s_i^{\text{blue}}) \cdot \mathbf{n}_i^{\text{blue}}$$

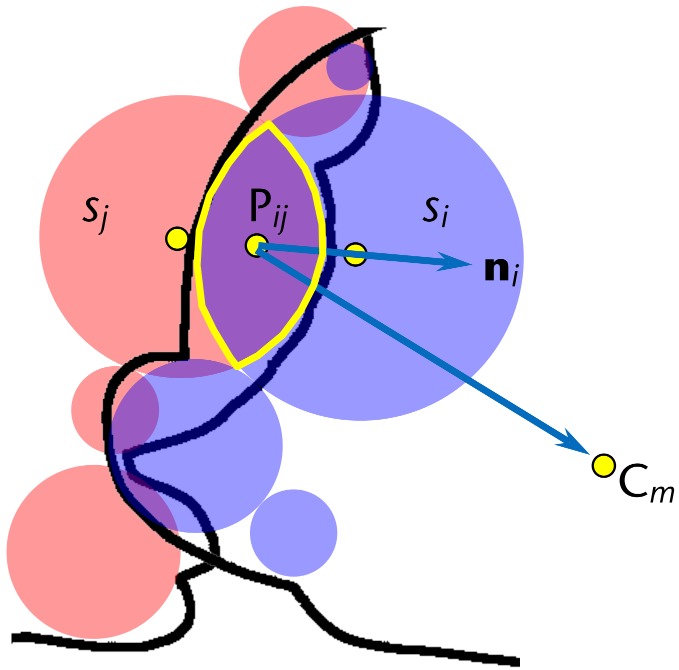
$$\mathbf{f}^{\text{blue}} = \sum \mathbf{f}_{ij}^{\text{blue}}$$

  - Torque:

$$\tau_{ij}^{\text{blue}} = (P_{\text{ij}} - C_m) \times \mathbf{f}_{\text{ij}}$$

$$\tau^{\text{blue}} = \sum \tau_{ij}^{\text{blue}}$$

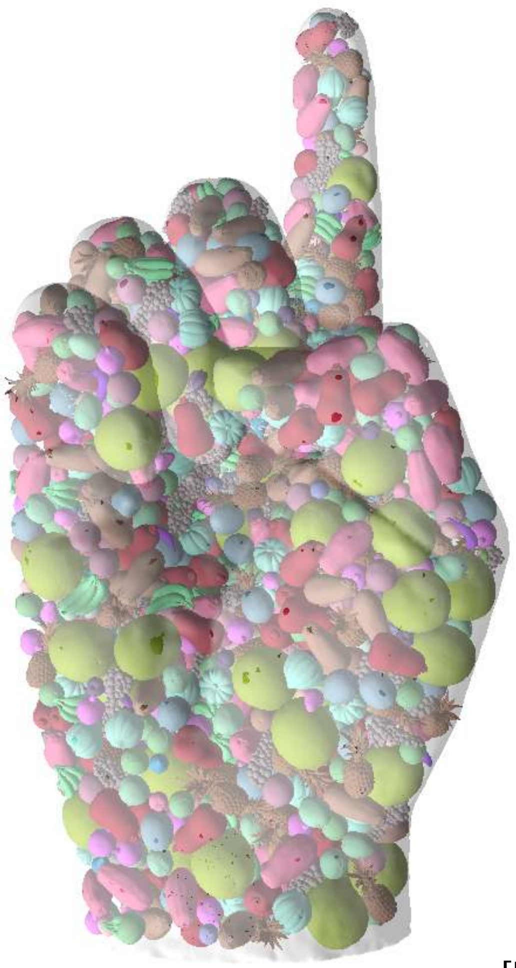
- Forces/torques an be proven to be continuous

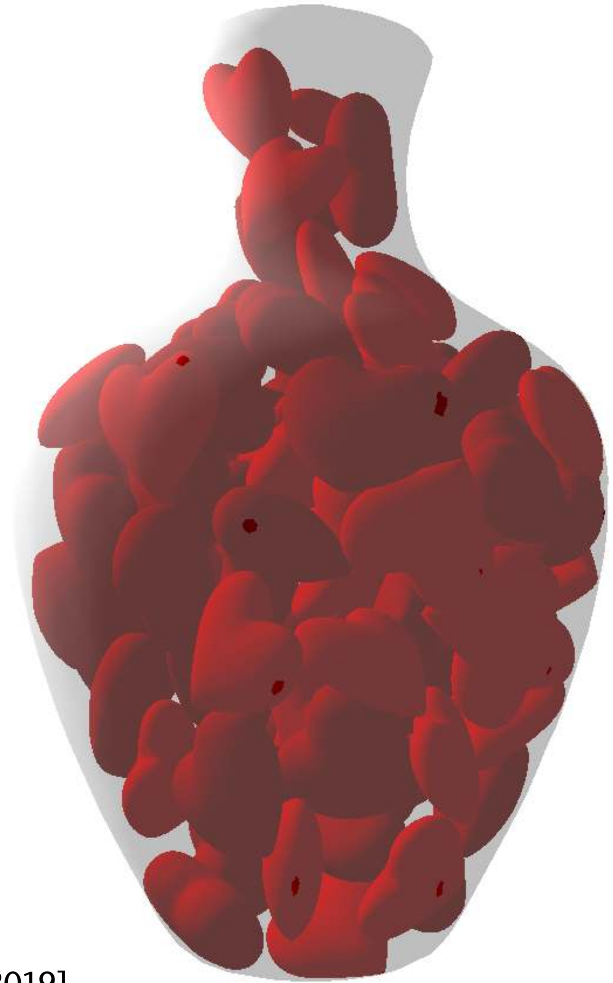


Application: Multi-User Haptic Workspace

12 moving objects ; 3.5M triangles ; 1 kHz simulation rate ; intersection volume ≈ 1-3 msec

Application: Bin Packing





[Meißenhelter et al. 2019]

Master / Bachelor Thesis Topics

CONFIDENTIAL

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