Insertion Sort and Closest Features
A Novel Approach to Collision Detection

Alexander Schinner

Otto-von-Guericke Universitt Magdeburg, Germany

September, 1997

The basic idea:

Information is expensive!

MD-Simulation of granular materials

$\Downarrow$

slowly changing system

$\Downarrow$

The last time-step's information is nearly correct.

$\Downarrow$

Reuse this information as good starting conditions.

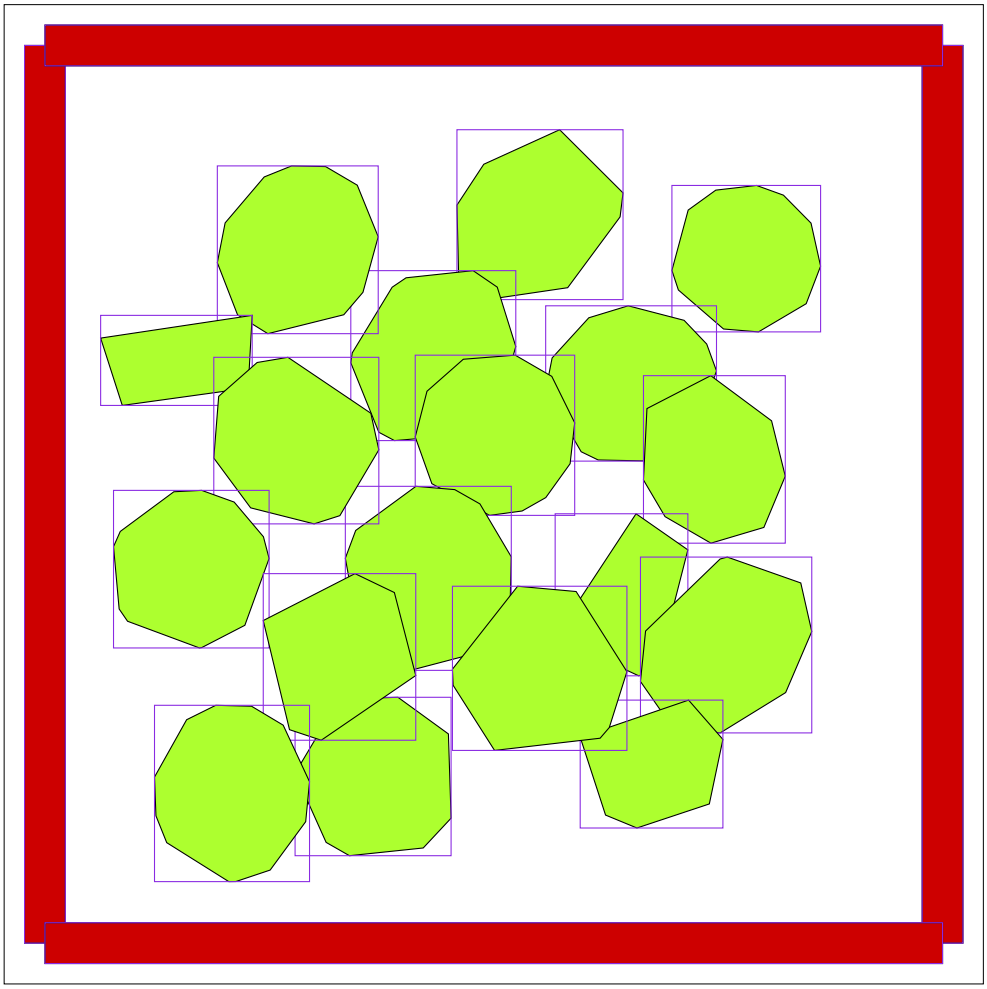
The steps of a MD-simulation

1. Check for bounding box collisions
2. Check for particle collisions
3. Calculate overlap
4. Do the physics (not discussed here)

Increasing need of time

Try to exclude as many particle-pairs as possible from being considered in the next level.

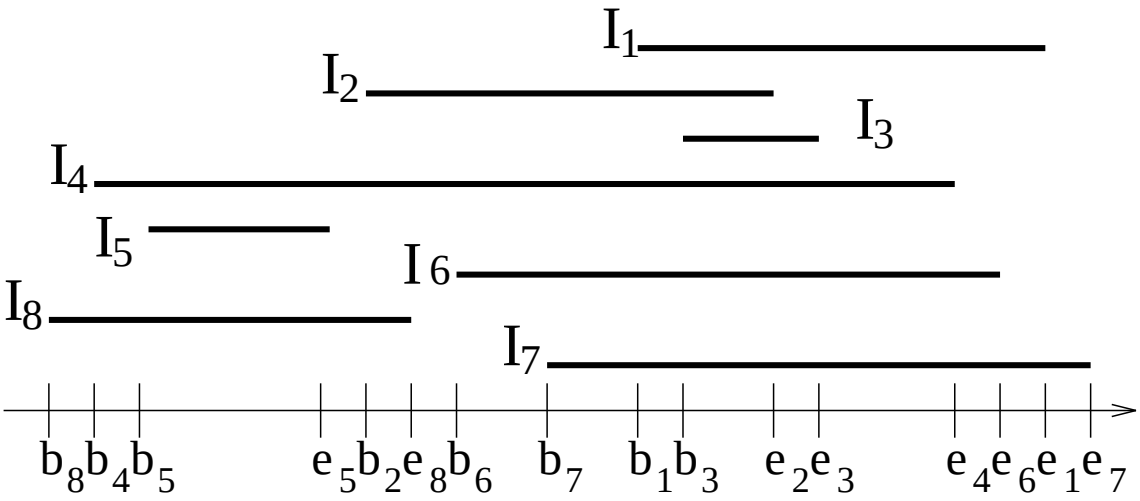
Bounding Boxes



Pairwise check $\Rightarrow N^2$ checks!

Global test needed!

First consider the one dimensional case:

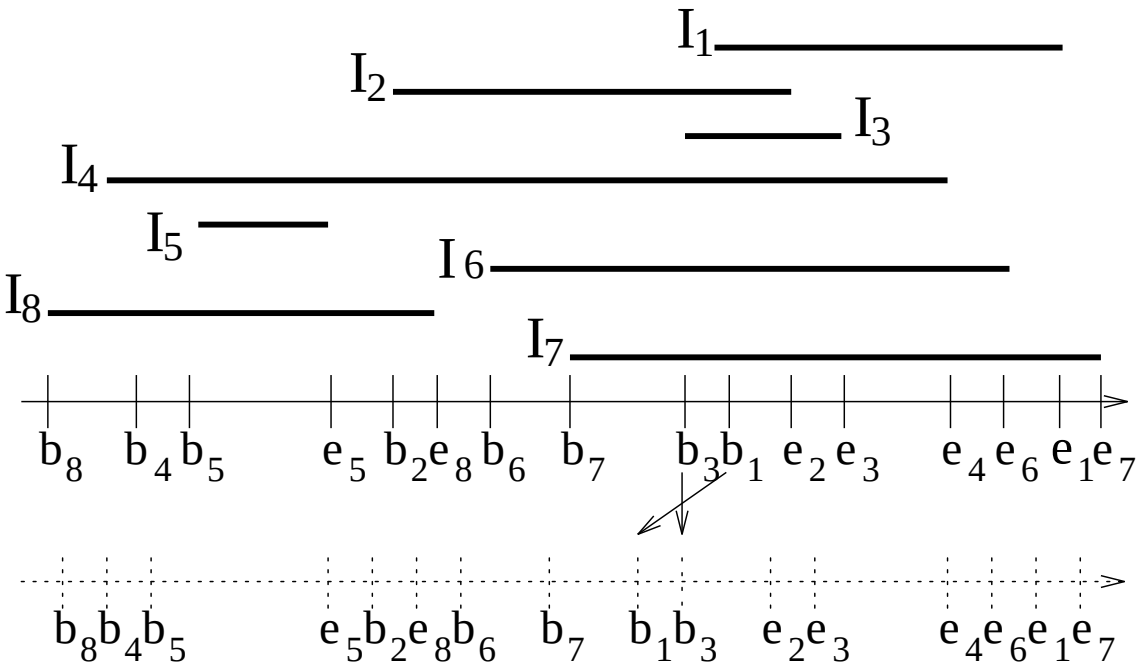


- Sorted List of all boundaries
- Extract collisions from this list with the sweep algorithm $\mathcal{O}(N)$.
- The best sorting algorithm works with $\mathcal{O}(N \log N)$.

$\Rightarrow$ Total cost of $\mathcal{O}(N \log N)$.

Really?

- $\mathcal{O}(N \log N)$ is the worst case
- There are often cases, algorithms are much faster



Reuse the list of the bounding box boundaries sorted in the last time-step.

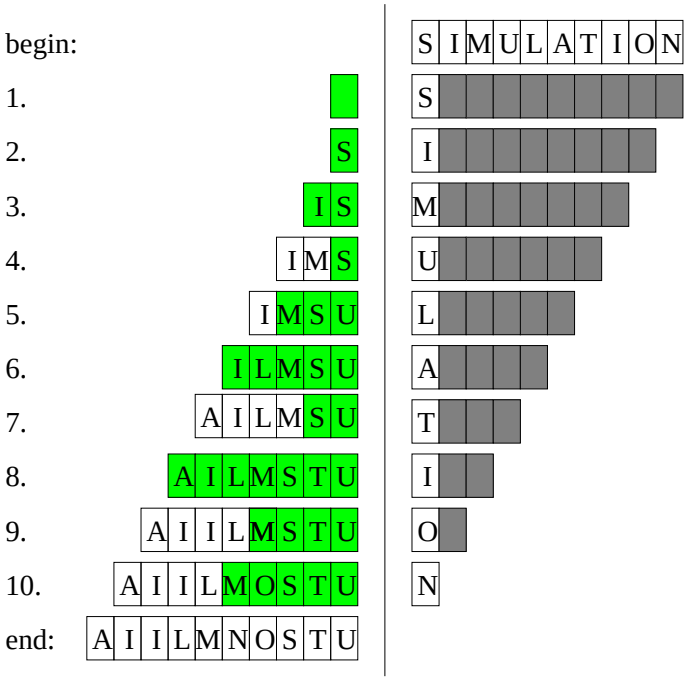
$\Downarrow$

Sort a nearly sorted list.

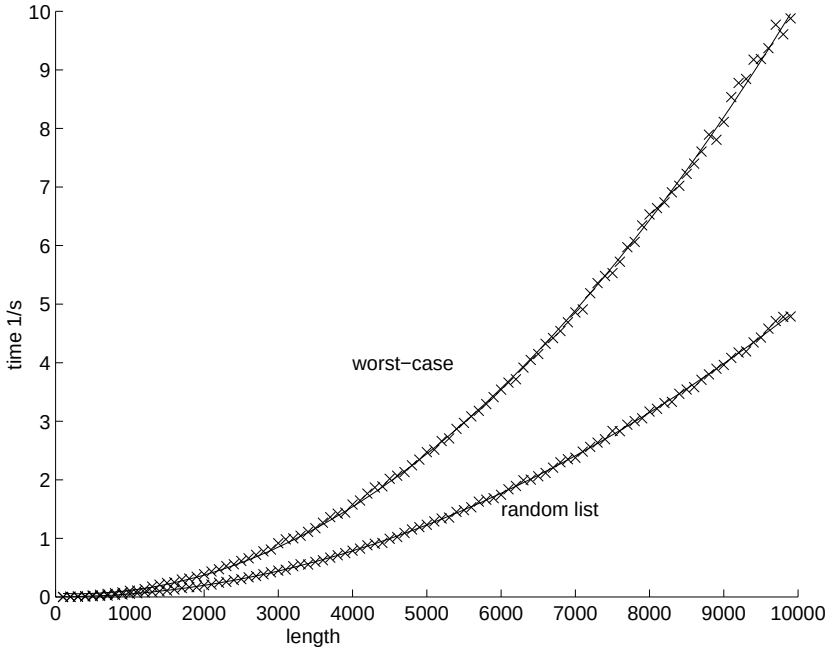
$\Downarrow$

Insertion Sort

Insertion Sort

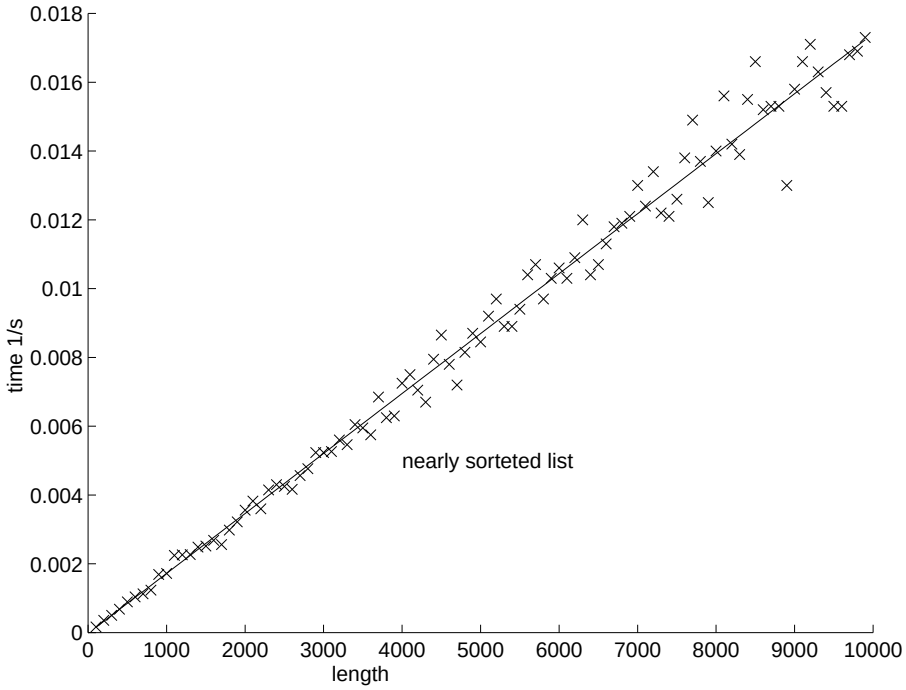


Normally, Insertion sort normally works with $\mathcal{O}(N^2)$.



Insertion Sort

- nearly sorted list
- Compare with 1 neighbor
- $\mathcal{O}(N)$ comparisons and nearly no overhead



Problems

- 2 Steps for finding the overlap on one axis
- in 2 Dimensions get 2 List for X and Y axis
- correlating 2 lists is expensive

Maintain a list of all bounding box collisions:

Want happens, if two boundaries change place?

$\Rightarrow$ Overlap status may change

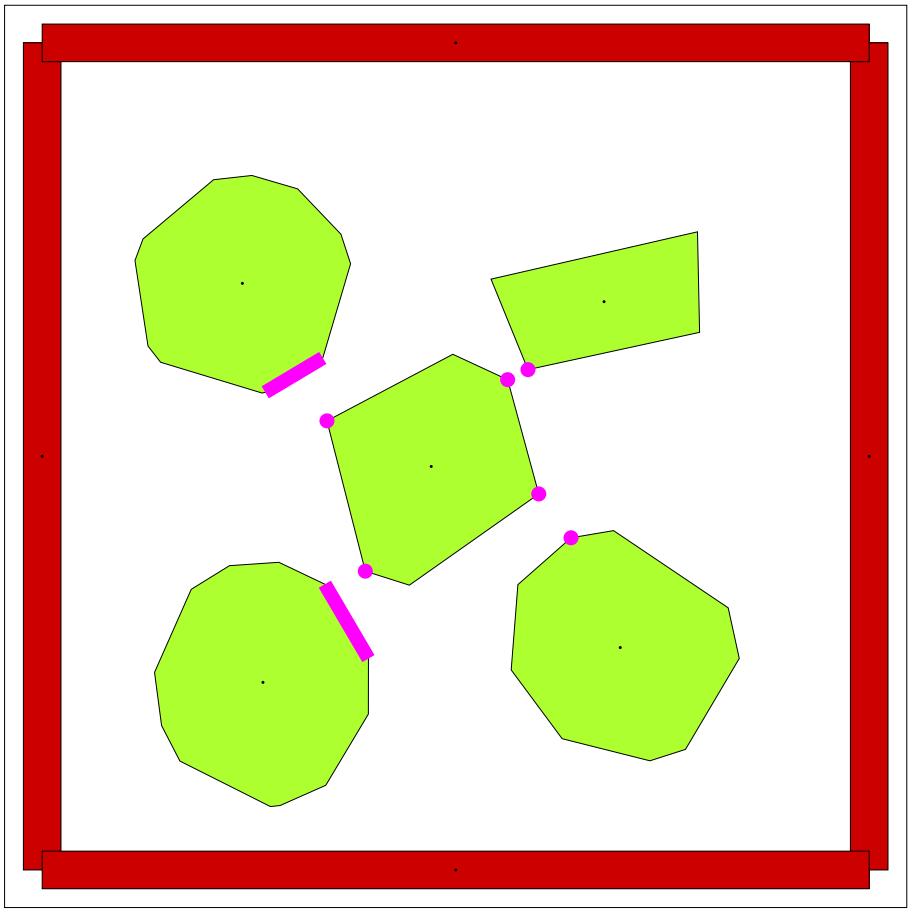
	Y-Axis No Collision	Y-Axis Collision
$bb \to bb$	$\overline{C} \to \overline{C}$	$C \to C$
$ee \to ee$	$\overline{C} \to \overline{C}$	$C \to C$
$be \to eb$	$\overline{C} \to \overline{C}$	$C \to \overline{C}$
$eb \to be$	$\overline{C} \to \overline{C}$	$\overline{C} \to C$

$C =$ collision $\bar{C} =$ no collisions

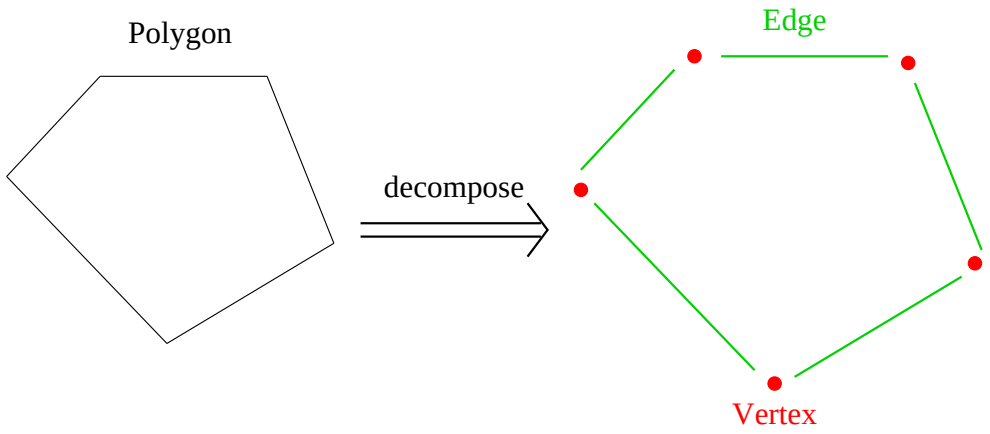
$\Rightarrow$ Update collision list in in one step with $\mathcal{O}(N)$

Closest Feature Algorithm

- Calculate distance of particles
- keep track of the closest features
- local test to confirm the distance



Feature



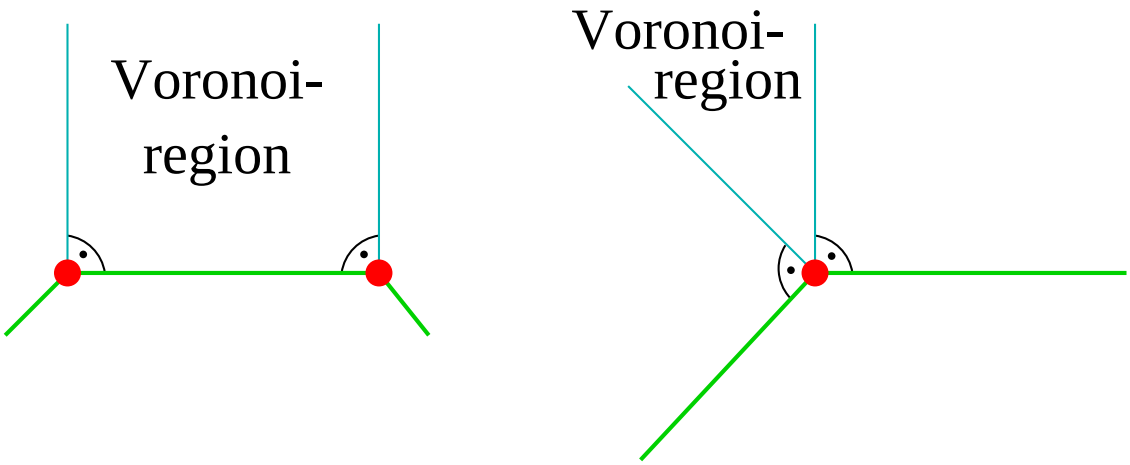
A feature has information on its

- type (edge or vertex)
- geometry
- neighbors
- Voronoi regions
- and more...

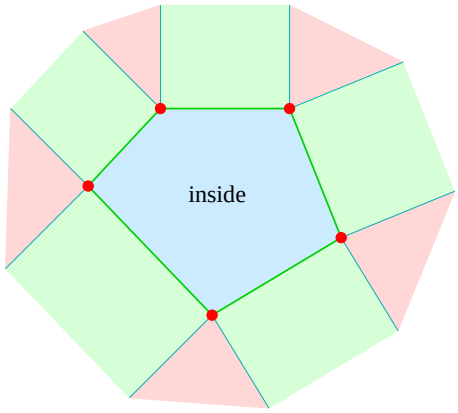
Voronoi Region

If a point p lies inside the Voronoi region of a feature f_a, then

$$\text{dist}(p, f_a) \leq \text{dist}(p, f_b)$$

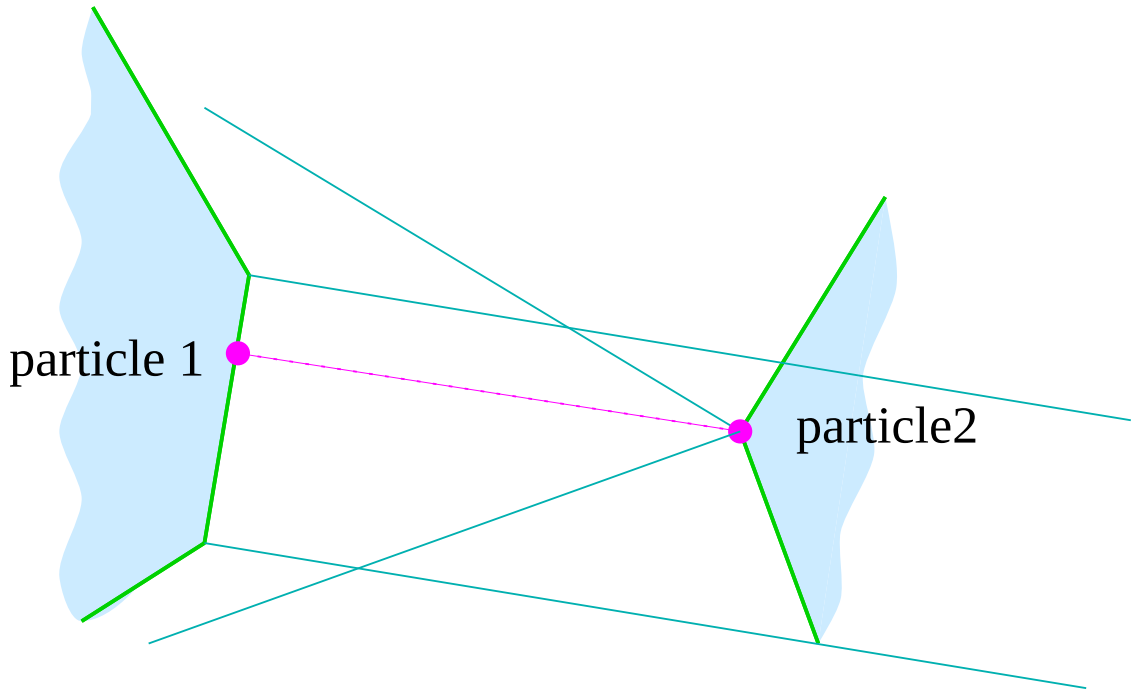


This divides the space outside a particle:



Closest features

If a point P on object P_1 lies inside the Voronoi region of f_2 on object P_2, then f_2 is a closest feature to the point P and vice versa for an Voronoi region of f_1. If we have a pair of features fulfilling the above condition, we have a pair of closest features.



Closest Feature Algorithm

We are looking on two features f_1 and f_b on two polyhedra P_1 and P_2.

1. Calculate the Voronoi regions V_1 and V_2
2. Calculate a point p_1 on f_1 that is the closest to f_2 and a point p_2 on f_2 that is the closest to f_a
3. Check for $p_1 \in V_2$. If not: choose new f_1 and restart algorithm
4. Check for $p_2 \in V_1$. If not: choose new f_2 and restart algorithm

vertex — vertex

- trivial closest points
- if a Voronoi test fails, continue with neighbor, whose boundary is violated

vertex — edge

- most frequent case
- special case, if vertex lies under the edge
- if a Voronoi test fails, continue with neighbor, whose boundary is violated

edge — edge

- check for intersection
- if no intersection, replace one edge by vertex

function check_vertex_vertex

```
erg = check_voronoi_of_vertex(vertex1,
            *vertex2->x1, *vertex2->y1);
  switch(erg){
  case CF_INSIDE:
    break;
  case CF_VIOLATES_PREVIOUS:
    *in_vertex1=(*in_vertex1)->previous;
    return(CF_NO_MINIMUM);
  case CF_VIOLATES_NEXT:
    *in_vertex1=(*in_vertex1)->next;
    return(CF_NO_MINIMUM);
}
erg = check_voronoi_of_vertex(vertex2,
            *vertex1->x1, *vertex1->y1);
  switch(erg){
  case CF_INSIDE:
    return(CF_MINIMUM);
  case CF_VIOLATES_PREVIOUS:
    *in_vertex2=(*in_vertex2)->previous;
    return(CF_NO_MINIMUM);
  case CF_VIOLATES_NEXT:
    *in_vertex2=(*in_vertex2)->next;
    return(CF_NO_MINIMUM);
}
```

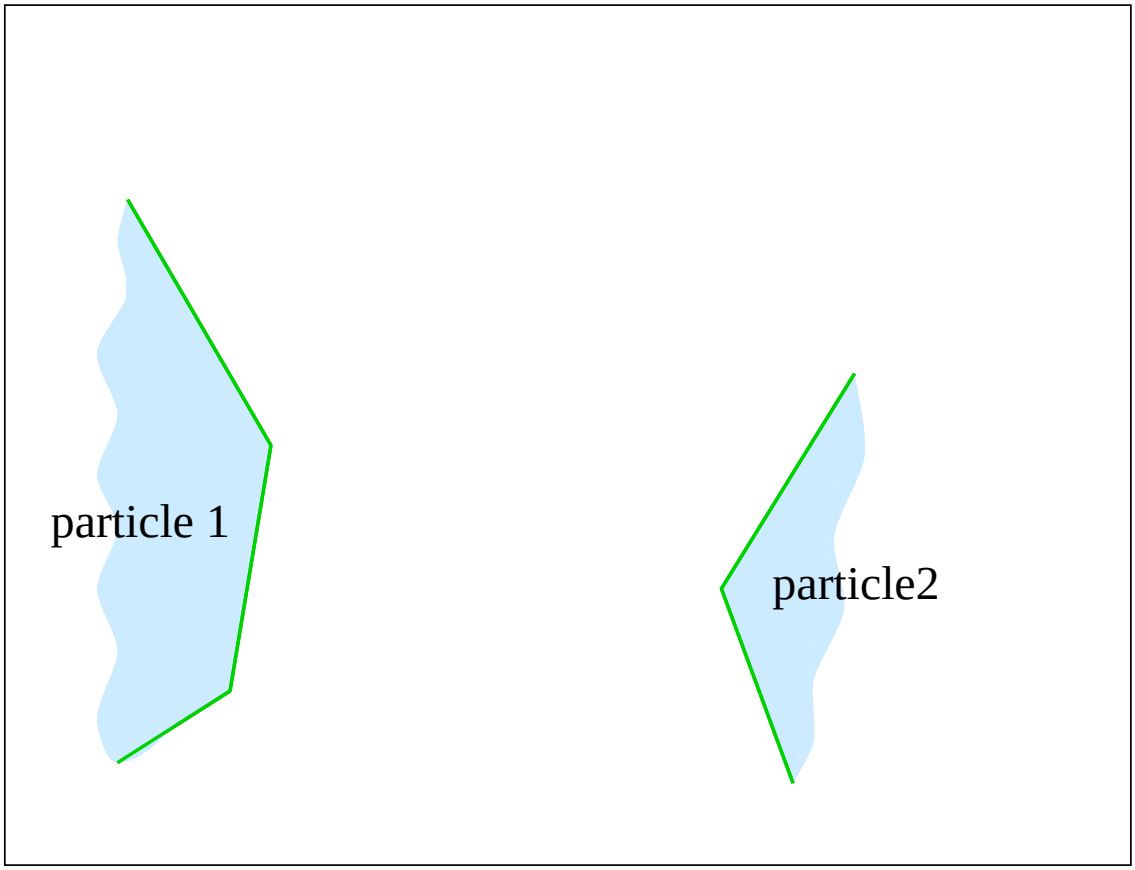
function find_closest_features

```
do{
   type1=coll->feature1->type;
   type2=coll->feature2->type;

   if ((type1==CF_VERTEX)&&(type2==CF_VERTEX)){
     erg=check_vertex_vertex(&(coll->feature1),
         &(coll->feature2),&distance);
   } else if ((type1==CF_VERTEX)&&(type2==CF_EDGE)){
     erg=check_vertex_edge(&(coll->feature1),
         &(coll->feature2),&distance);
   } else if ((type1==CF_EDGE)&&(type2==CF_VERTEX)){
     erg=check_vertex_edge(&(coll->feature2),
         &(coll->feature1),&distance);
   } else {
     erg=check_edge_edge(&(coll->feature2),
         &(coll->feature1),&distance);
   }
 }while(erg==CF_NO_MINIMUM);

 if (distance<=0){
   coll->colliding=TRUE;
 }else{
   coll->colliding=FALSE;
 }
```

Example

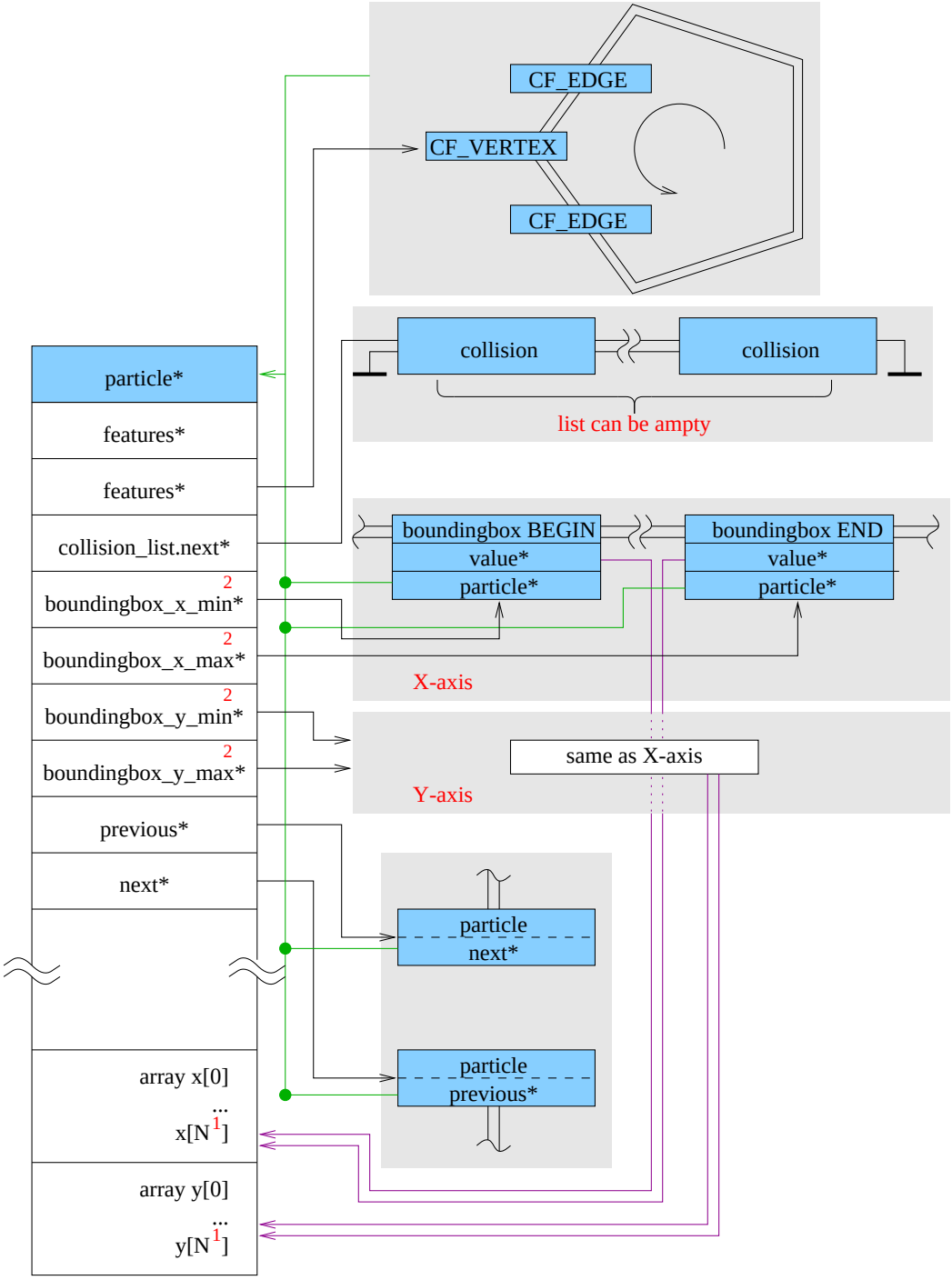


disadvantages

- complicate program structure
- needs a lot of memory

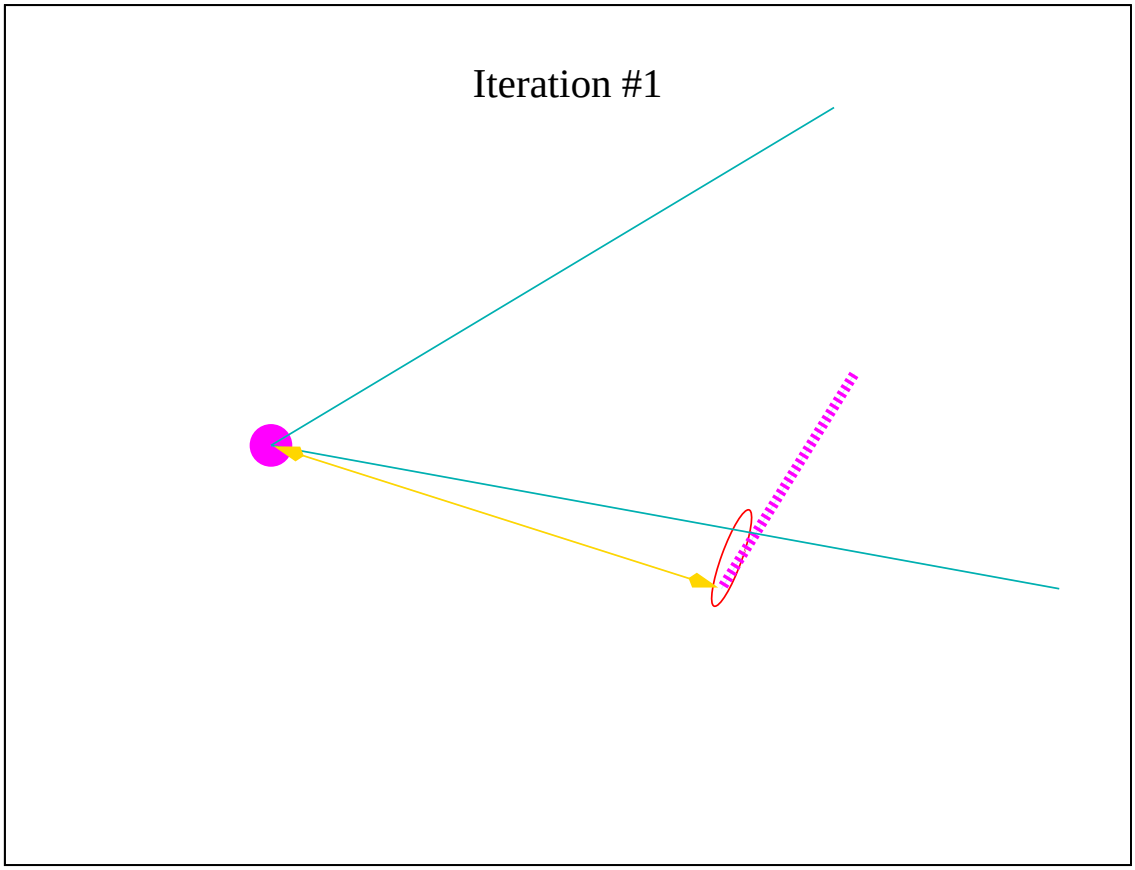
advantages

- the total cost for the bounding box algorithm is $\mathcal{O}(N)$
- the total cost for the closest feature algorithm is $\mathcal{O}(\text{const})$
- also available in 3 dimensions
- only local test, parallel model for shared memory systems possible
- FAST!

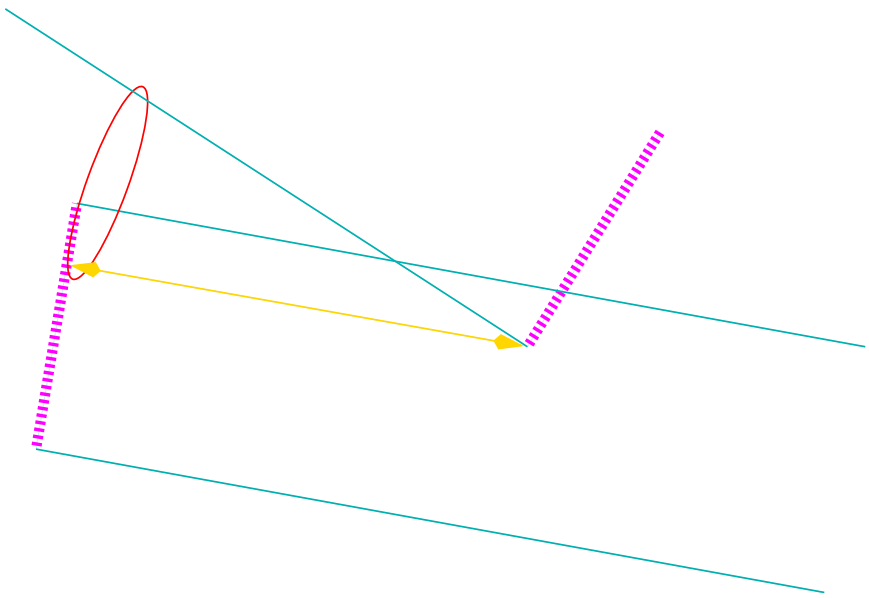


[1] N:=MAX_NUM_CORNER

[2] boundingbox_*_* are type of SENTINEL


Iteration #1

Iteration #2



Iteration #3