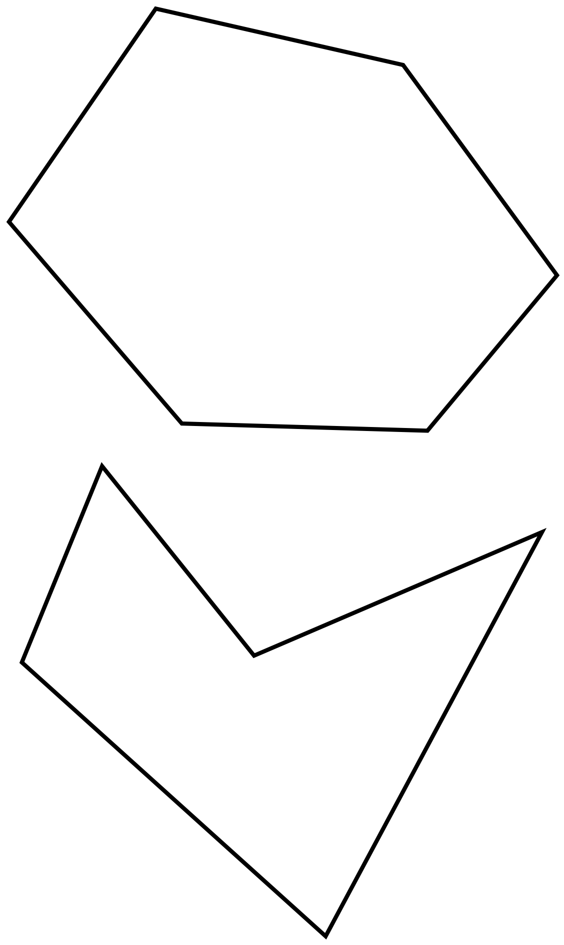
3D convex hulls

Computational Geometry [csci 3250]
Laura Toma
Bowdoin College

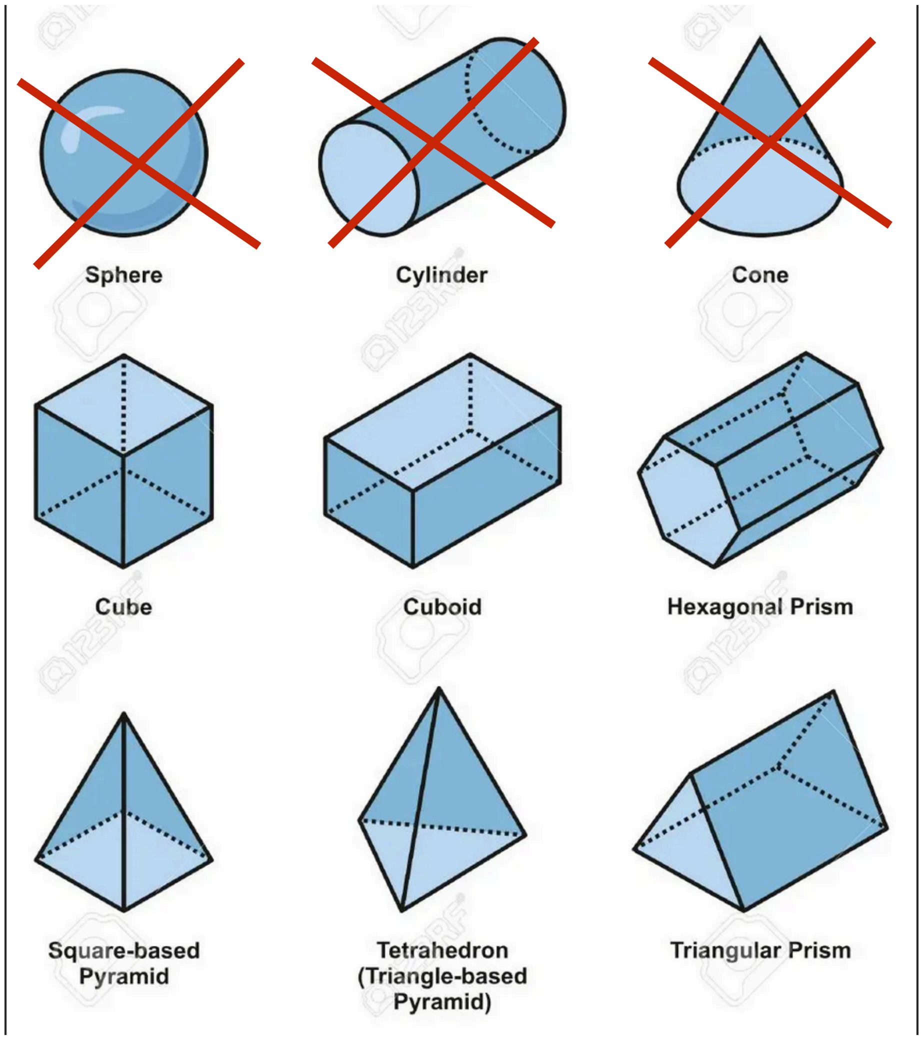
2D



polygon

flat, plane, two-dimensional closed shape bounded by line segments

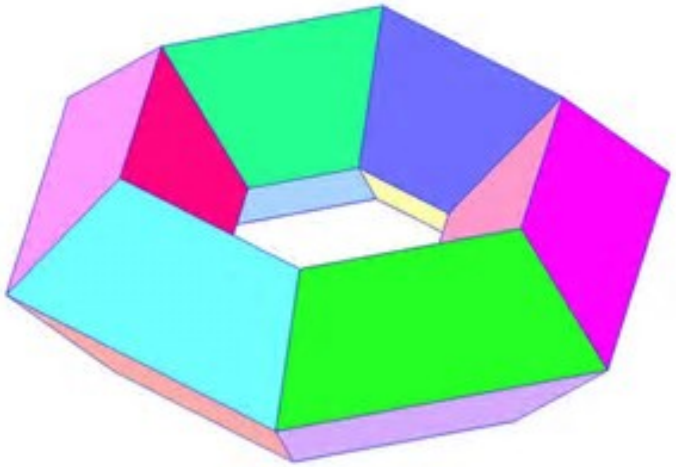
3D

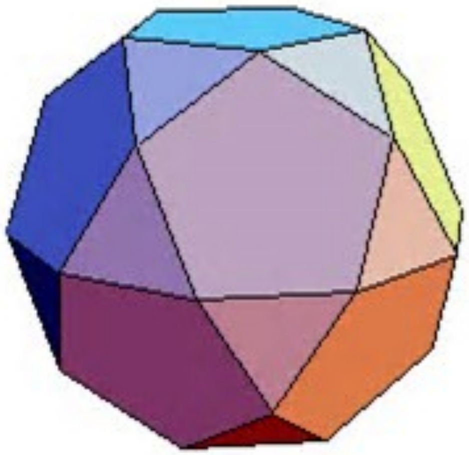


polyhedron

A three-dimensional shape with flat polygonal faces

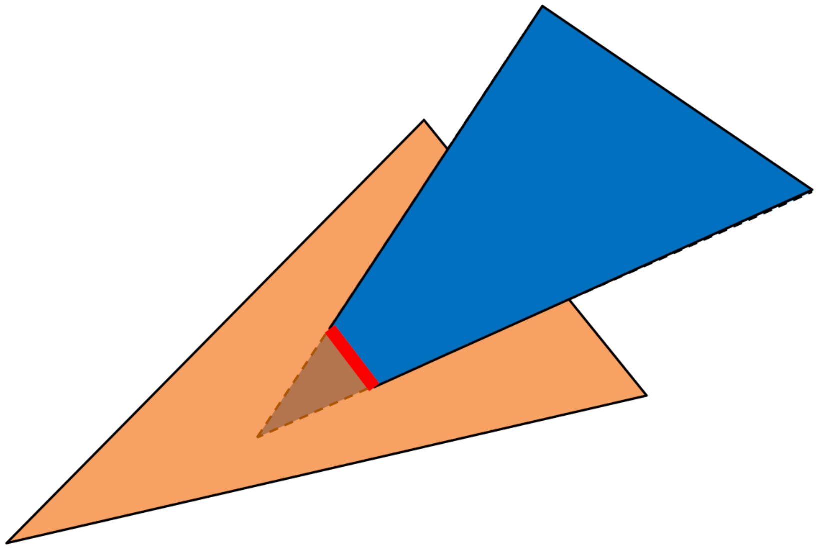
Polyhedron





A **polyhedron** is a region of space whose boundary consists of vertices, edges and (flat) faces, such that:

- Faces intersect properly
    - two faces are either disjoint; or
    - have a single vertex in common; or
    - have two vertices and the edge between them in common



A **polyhedron** is a region of space whose boundary consists of vertices, edges and (flat) faces, such that:

- Faces intersect properly
- The local topology must be proper (a neighborhood of every point must be homeomorphic with an open disk)

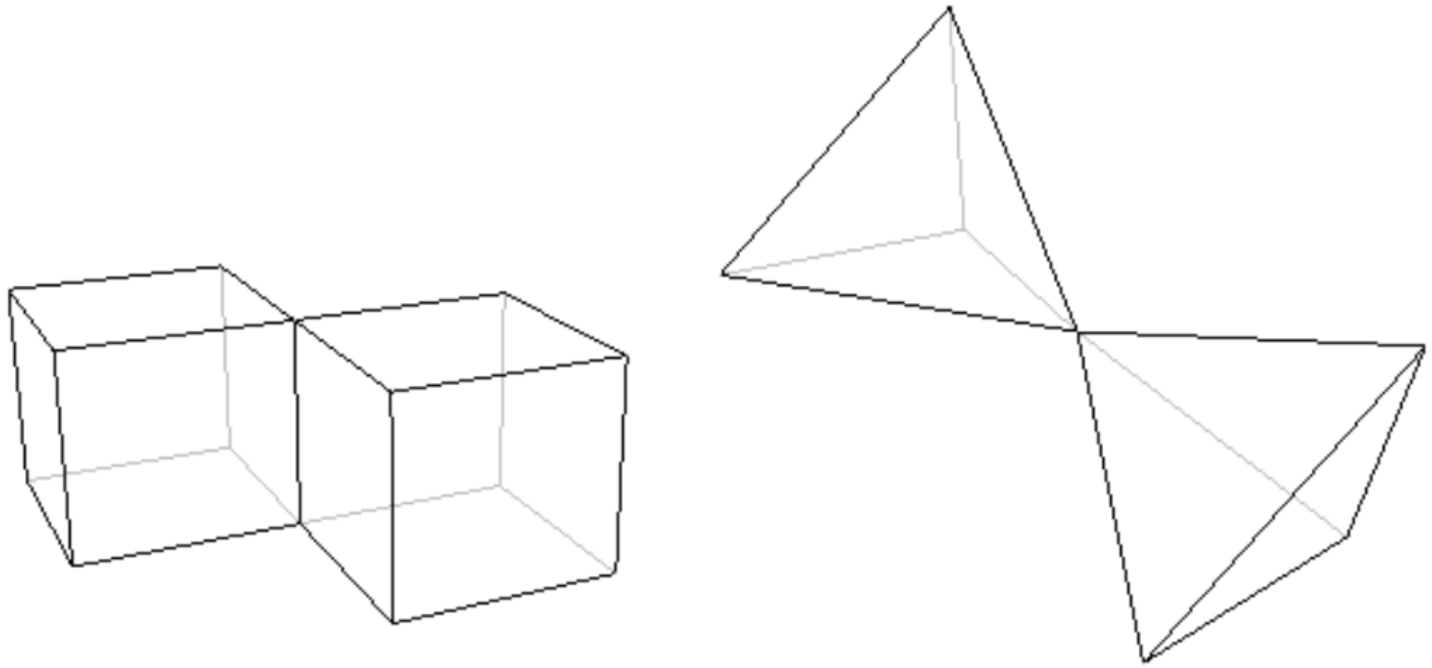
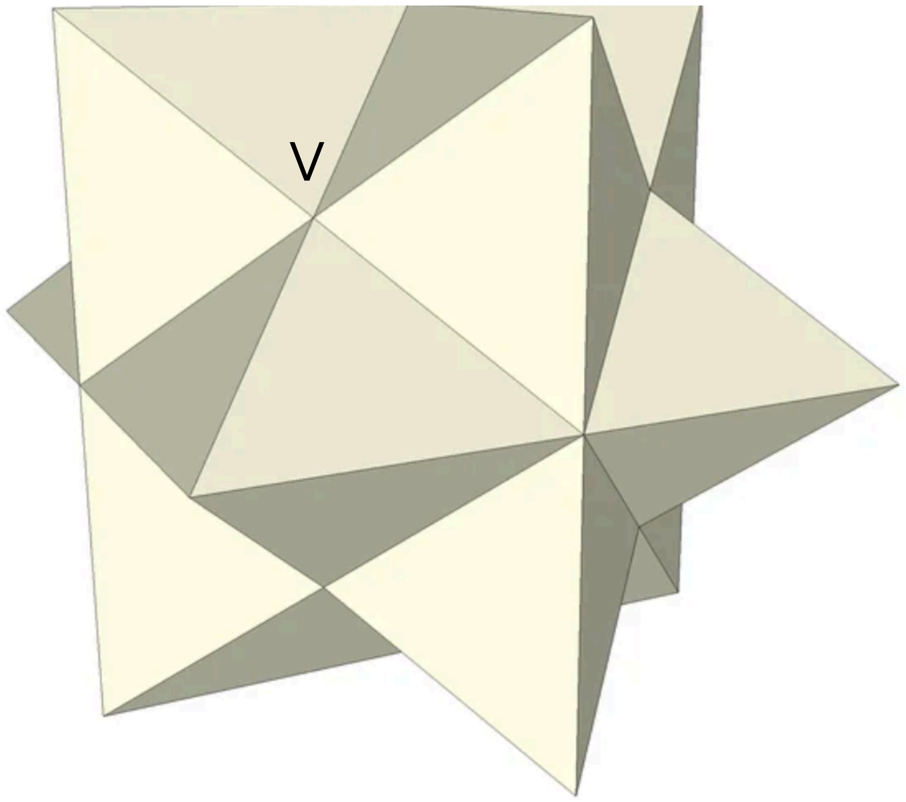
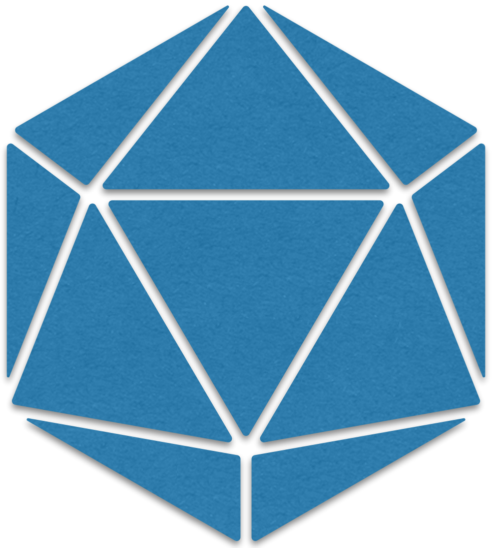


Figure 4: These objects are not polyhedra because they are made up of two separate parts meeting only in an edge (on the left) or a vertex (on the right).

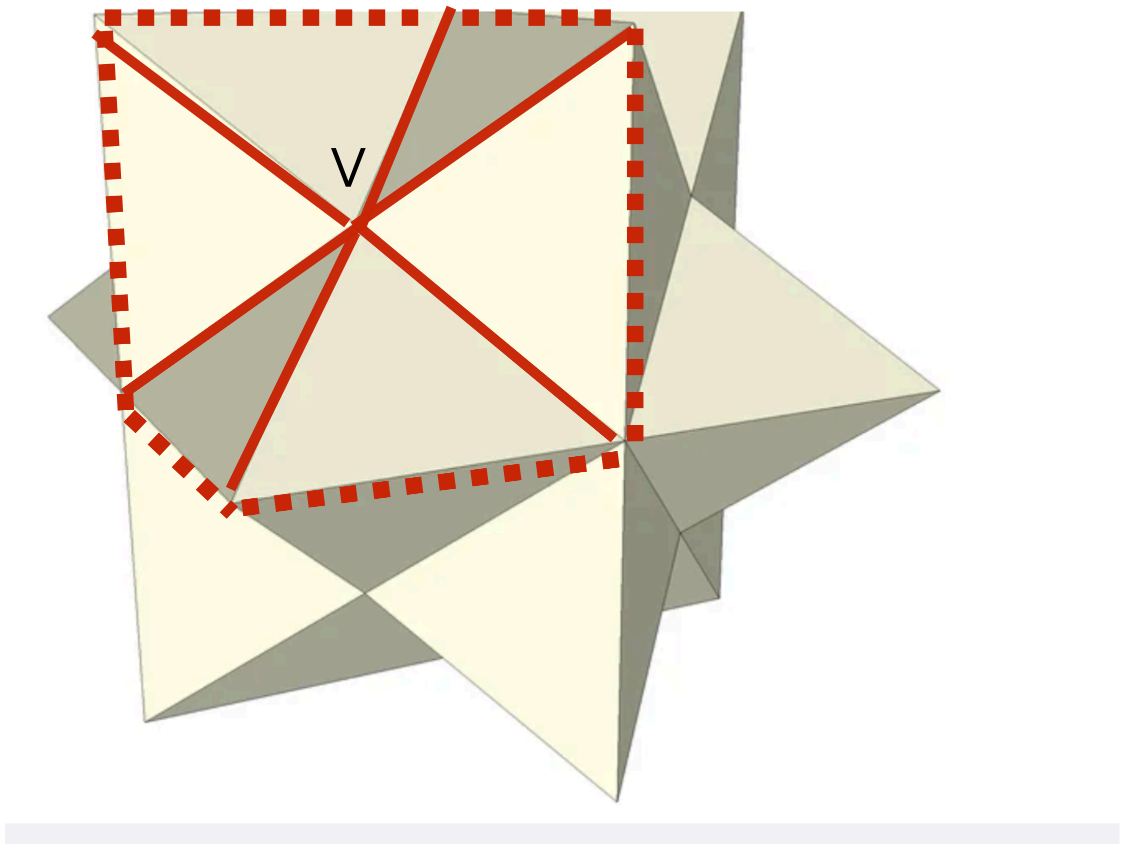
https://plus.maths.org/content/eulers-polyhedron-formula

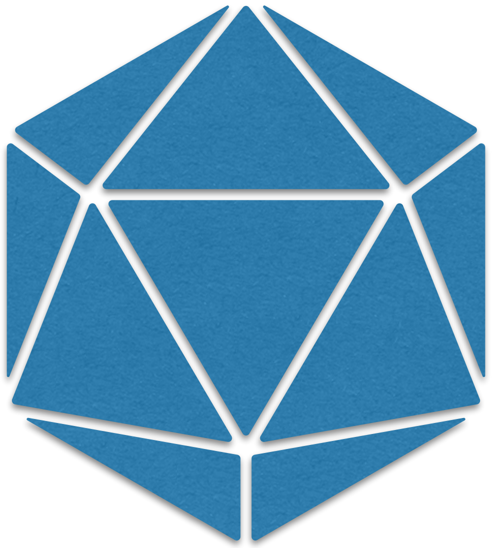
Proper topology: The link of any vertex be is a simple, closet polygonal path.



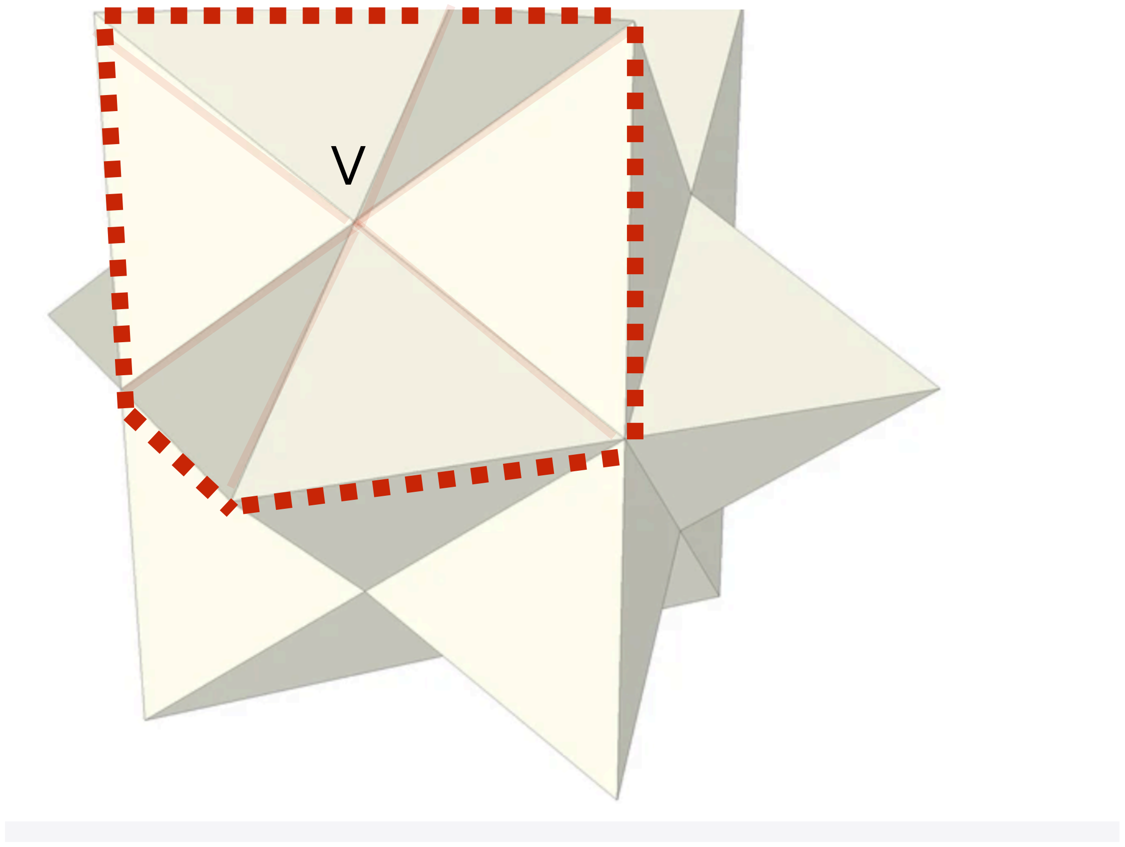


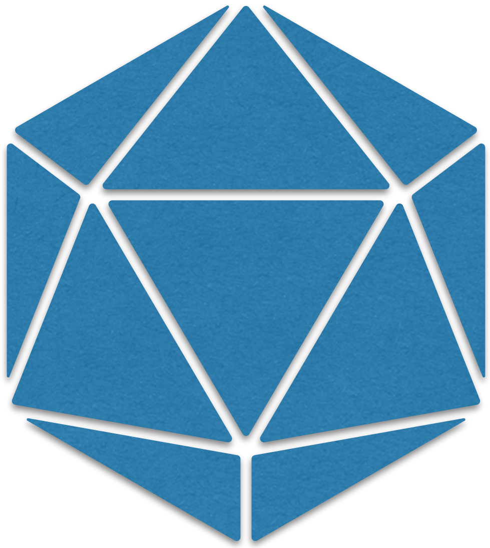
Proper topology: The link of any vertex be is a simple, closet polygonal path.





Proper topology: The link of any vertex be is a simple, closet polygonal path.





Proper topology: The link of any vertex be is a simple, closet polygonal path.

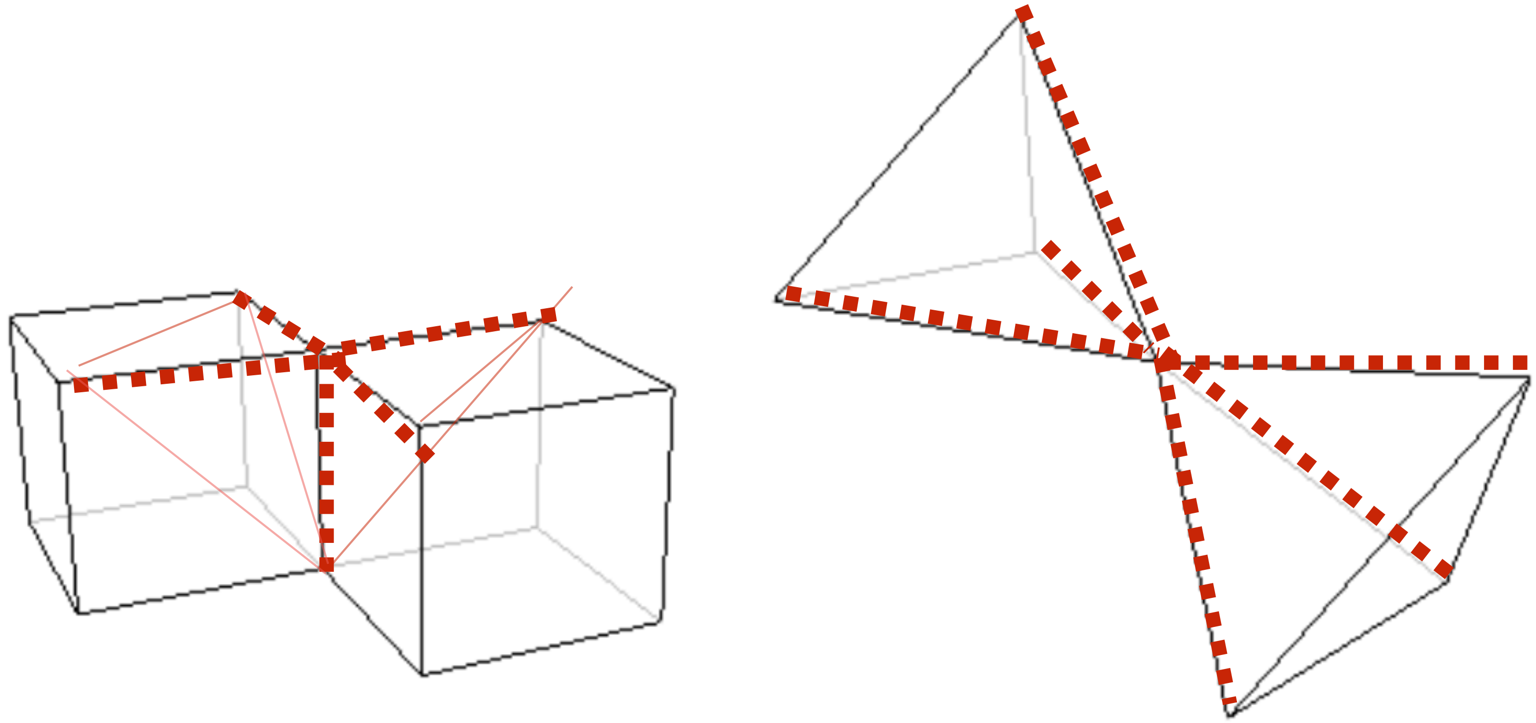


Figure 4: These objects are not polyhedra because they are made up of two separate parts meeting only in an edge (on the left) or a vertex (on the right).

Proper topology: The link of any vertex be is a simple, closet polygonal path.

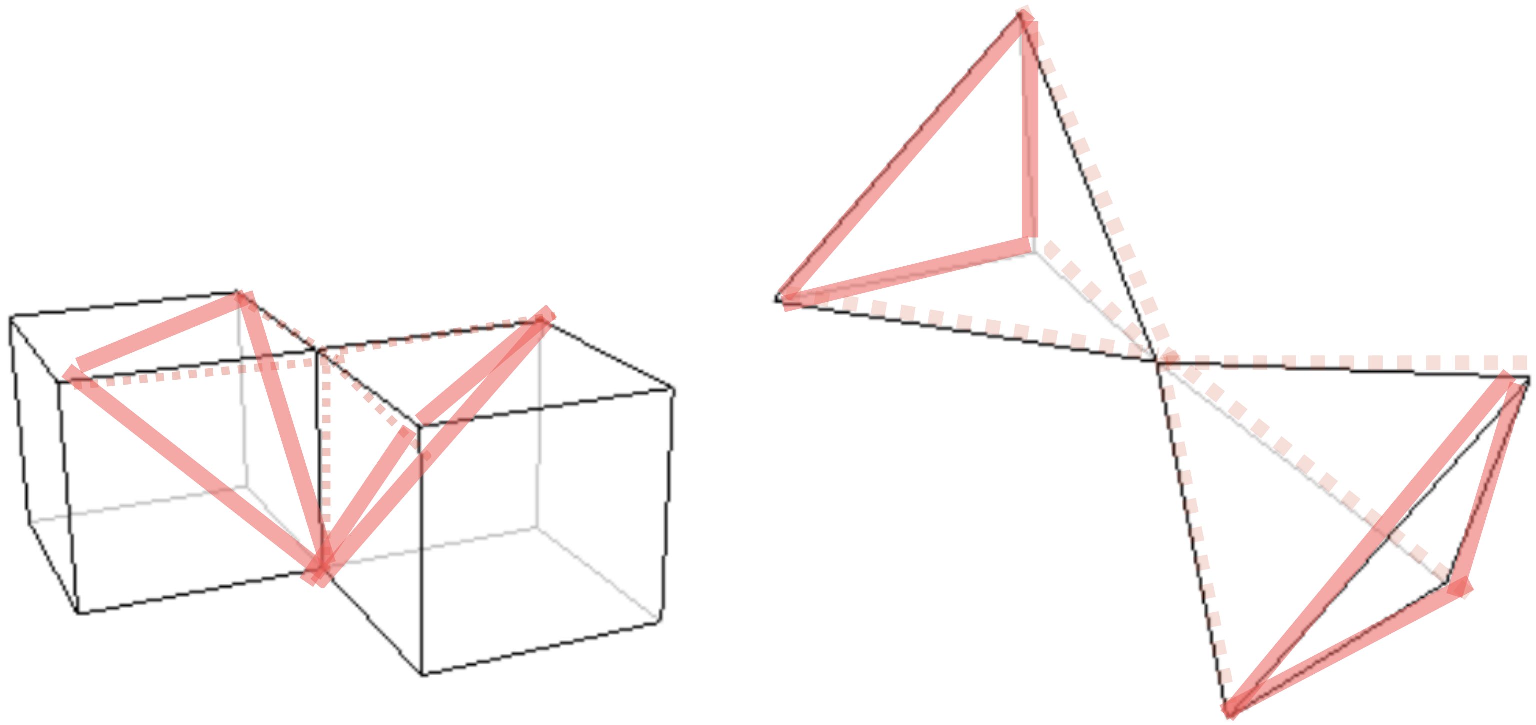
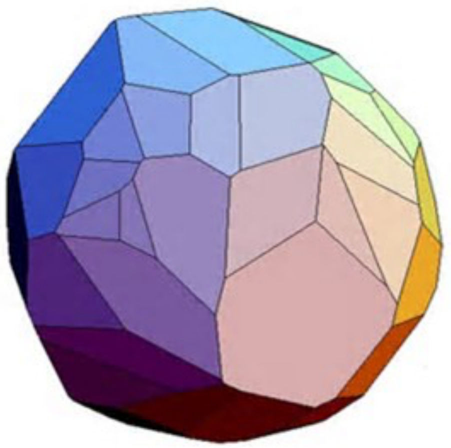
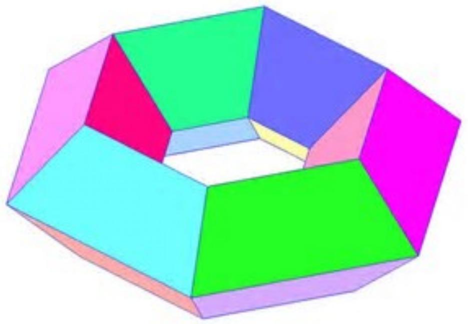


Figure 4: These objects are not polyhedra because they are made up of two separate parts meeting only in an edge (on the left) or a vertex (on the right).

A **polyhedron** is a region of space whose boundary consists of vertices, edges and (flat) faces, such that:

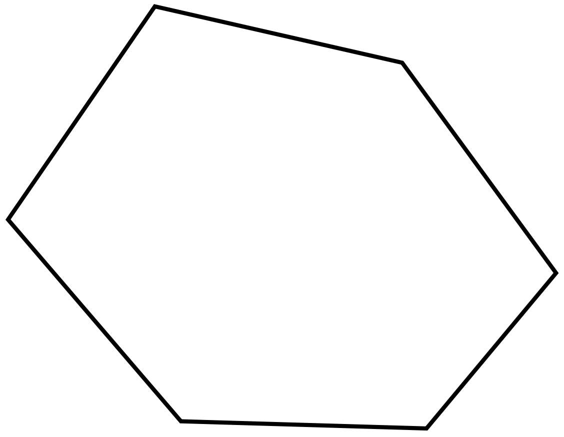
- Faces intersect properly
- The local topology must be proper
- Also the global topology must be proper: surface is connected, closed and bounded.
    - Holes are allowed, as long as they don't disconnect
    - The nb of holes is called the genus of the surface

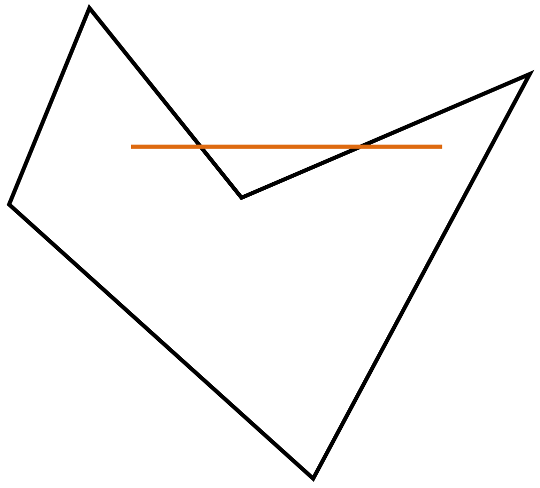

genus=0


genus=1

Convexity

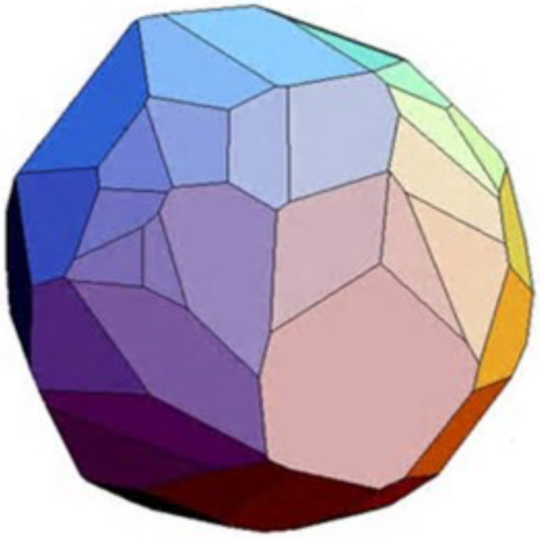
A polygon P is convex if for any p, q in P, the segment pq lies entirely in P.


convex


non-convex

Convexity

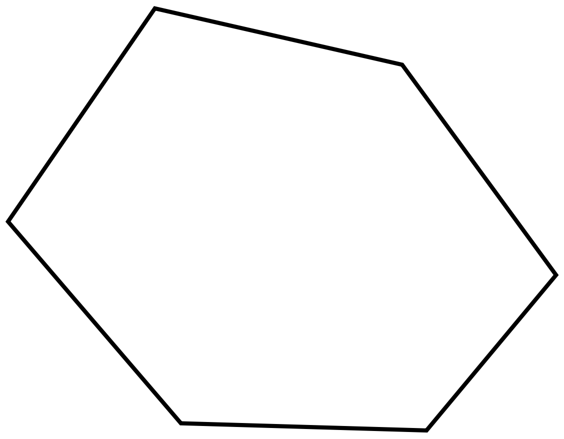
A polyhedron P is convex if for any p, q in P, the segment pq lies entirely in P.

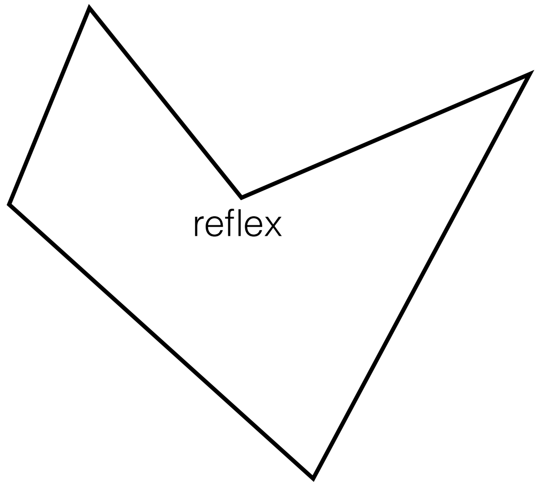

convex

non-convex

Convex polygons: All angles in a convex polygon are $\leq \pi$.

(necessary and sufficient)

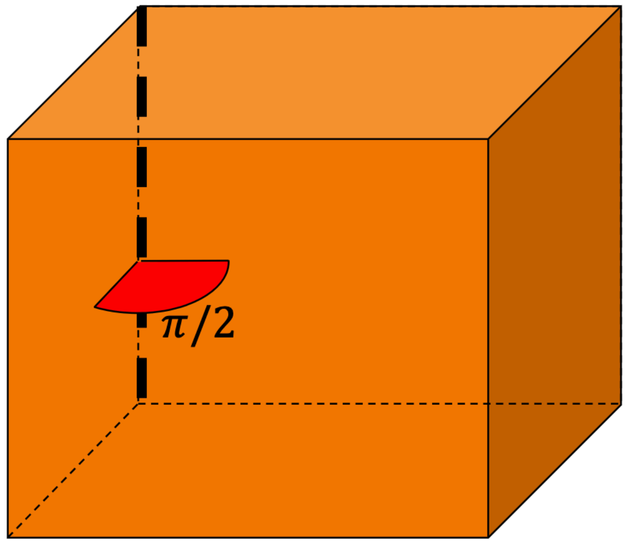

convex



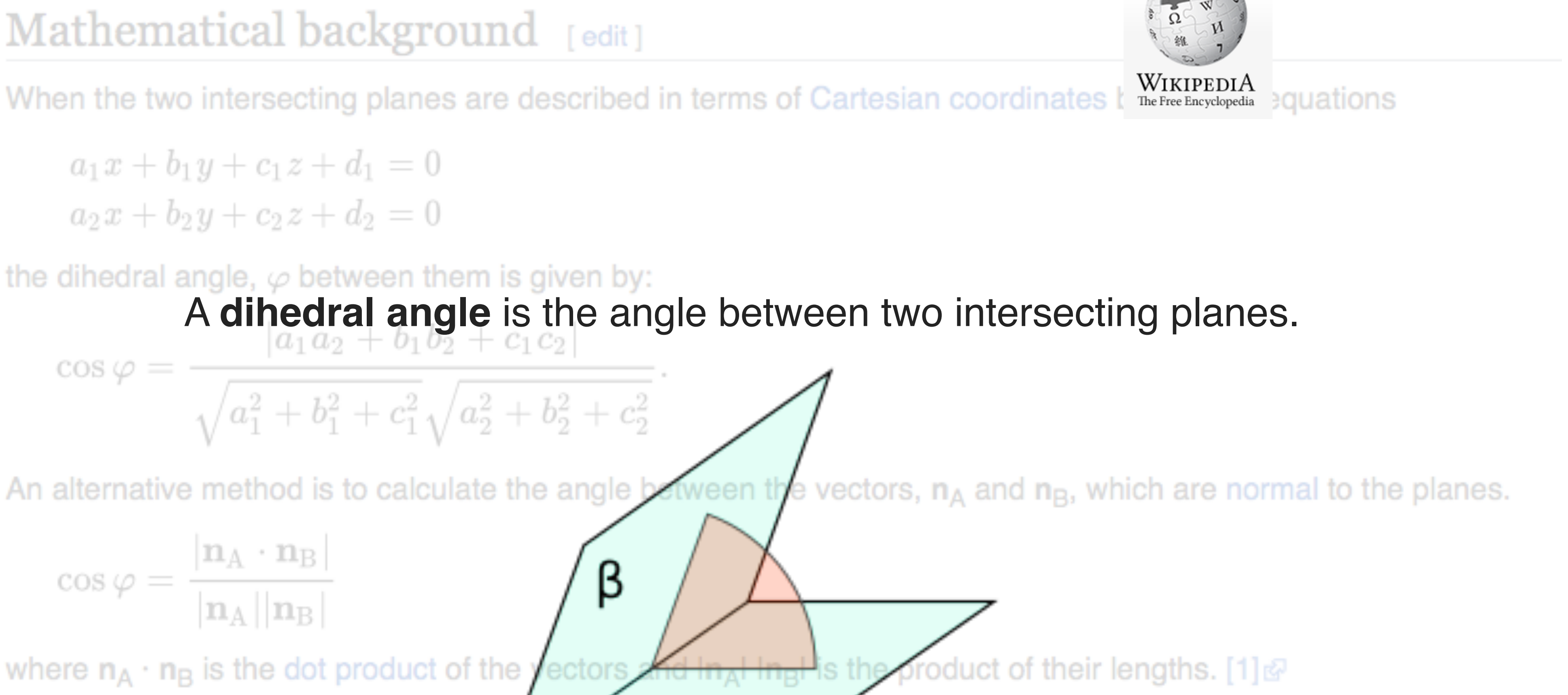
non-convex

Definition:

Given an edge on a polyhedron, the *dihedral angle* of the edge is the internal angle between the two adjacent faces.



Aside:
The dihedral angle is a discrete measure of mean curvature.



Mathematical background [edit]

When the two intersecting planes are described in terms of Cartesian coordinates b[illegible] equations

$$a_1x + b_1y + c_1z + d_1 = 0$$
$$a_2x + b_2y + c_2z + d_2 = 0$$

the dihedral angle, φ between them is given by:

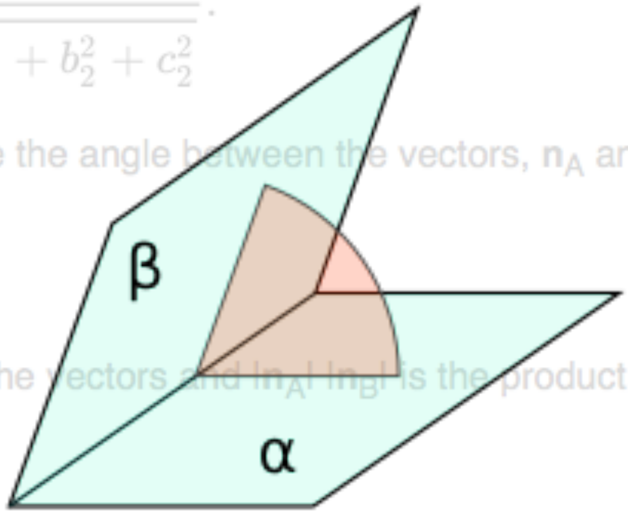
$$\cos\varphi = \frac{|a_1a_2 + b_1b_2 + c_1c_2|}{\sqrt{a_1^2 + b_1^2 + c_1^2}\sqrt{a_2^2 + b_2^2 + c_2^2}}.$$

An alternative method is to calculate the angle between the vectors, $\mathbf{n}_A$ and $\mathbf{n}_B$, which are normal to the planes.

$$\cos\varphi = \frac{|\mathbf{n}_A \cdot \mathbf{n}_B|}{|\mathbf{n}_A||\mathbf{n}_B|}$$

where $\mathbf{n}_A \cdot \mathbf{n}_B$ is the dot product of the vectors and $|\mathbf{n}_A|\,|\mathbf{n}_B|$ is the product of their lengths. [1]

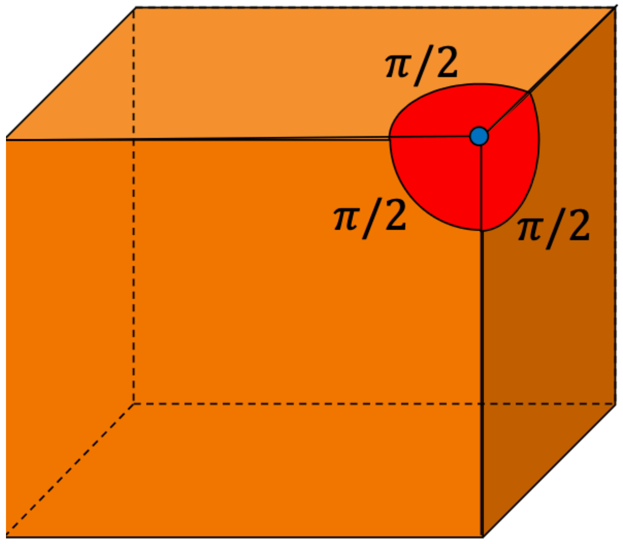
A **dihedral angle** is the angle between two intersecting planes.



Angle between two planes (α, β, green) in a third plane (pink) which cuts the line of intersection at right angles

Definition:

Given a vertex on a polyhedron, the *deficit angle* at the vertex is 2π minus the sum of angles around the vertex.



$\Rightarrow \pi/2$

Aside:
The deficit angle is a discrete measure of Gauss curvature.

Convex polyhedra

- All dihedral angles are $\leq \pi$

  (Necessary and sufficient.)

- Sum of angles around a vertex is $\leq 2\pi$

  (Necessary but not sufficient).

Euler’s formula

- Euler noticed a remarkable regularity in the number of vertices, edges and faces of a convex polyhedron of genus=0.

- **Euler’s formula: $V - E + F = 2$**

- Proof idea:
    - flatten the polygon to a plane
    - prove the formula for a tree
    - prove for any planar graph by induction on E

O(n) edges and faces

- Consider a polyhedron with $V = n$ vertices
- Triangulate its faces. This maximizes E and F.

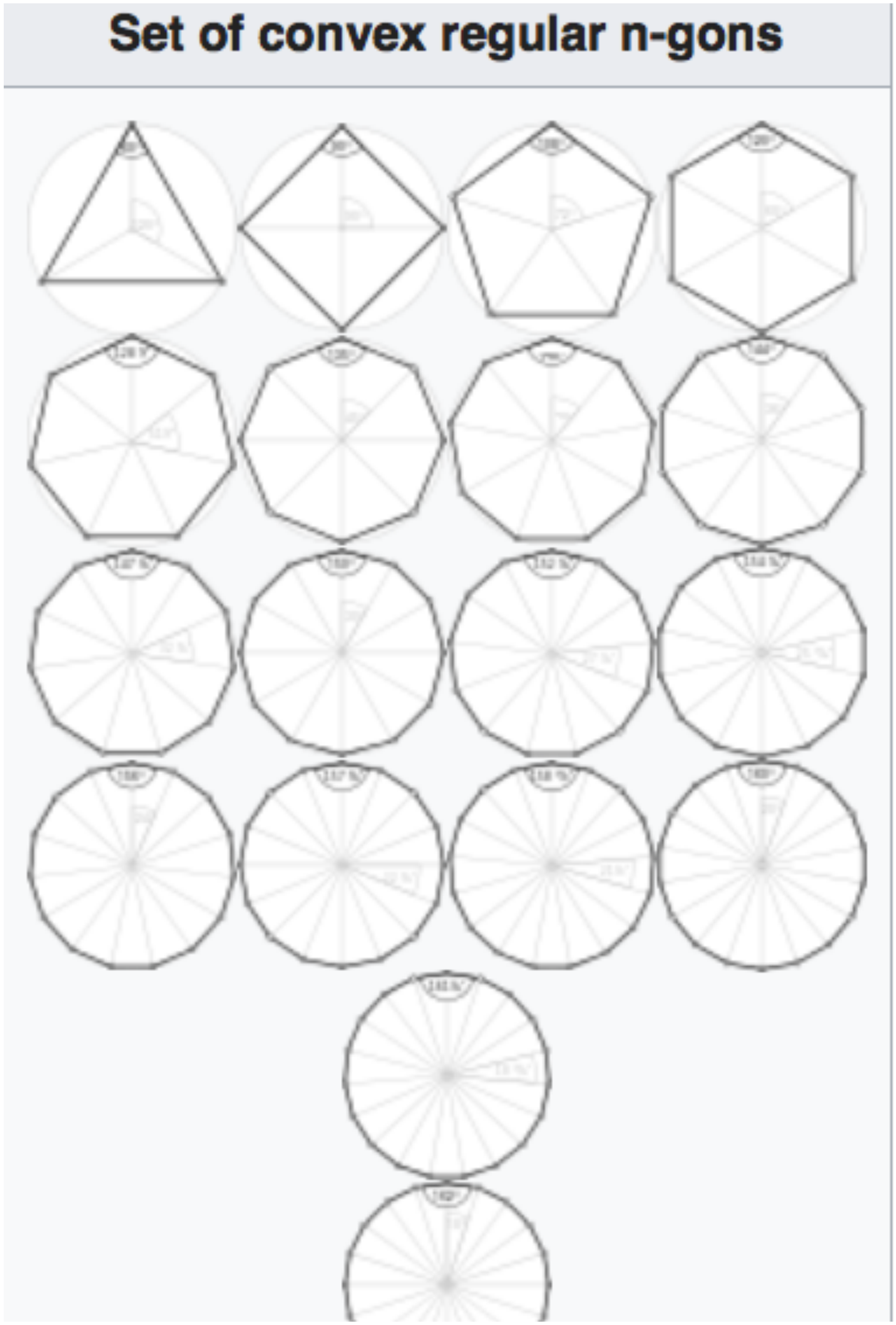
- We have that:
    - $V - E + F = 2$
    - $3F = 2E$
- and from here it follows that $E = 3V - 6 = O(n)$ and $F = 2V - 4 = O(n)$

- Result: The number of vertices, edges and faces in a polyhedron are linearly related.
- We'll say that a polyhedron with n vertices has $O(n)$ edges and $O(n)$ faces

digression start

Regular polygons in 2D

- A regular polygon has equal sides and angles



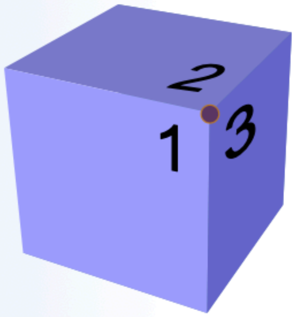
Regular polyhedra in 3D

- Regular polyhedra:
  - faces are congruent regular polygons
  - the number of faces incident to each vertex is the same (and equal angles)

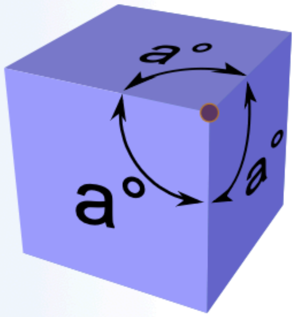
Surprisingly, there exist only 5 regular polytops

Simplest Reason: Angles at a Vertex

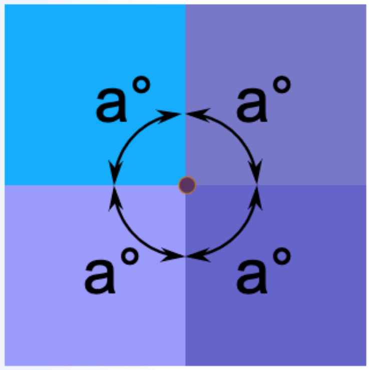
The simplest reason there are only 5 Platonic Solids is this:



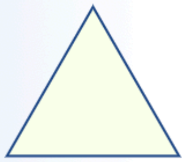
At each vertex **at least 3 faces** meet (maybe more).



When we add up the internal angles that meet at a vertex, it must be **less than 360 degrees**.

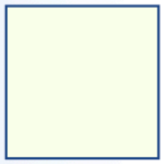


Because at 360° the shape flattens out!



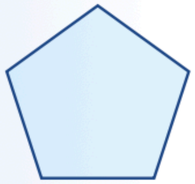
A regular triangle has internal angles of 60°, so we can have:

- 3 triangles (3×60°=180°) meet
- 4 triangles (4×60°=240°) meet
- or 5 triangles (5×60°=300°) meet



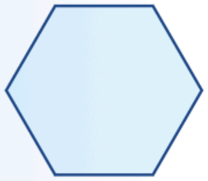
A square has internal angles of 90°, so there is only:

- 3 squares (3×90°=270°) meet



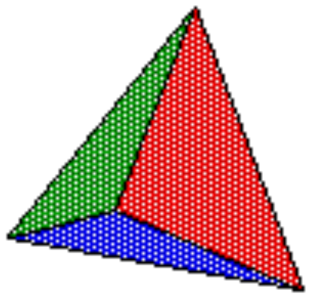
A regular pentagon has internal angles of 108°, so there is only:

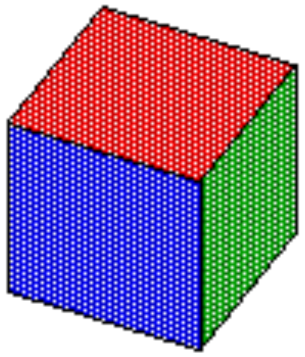
- 3 pentagons (3×108°=324°) meet

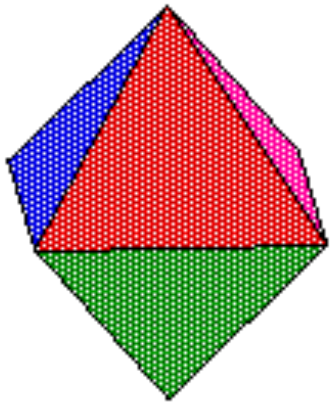


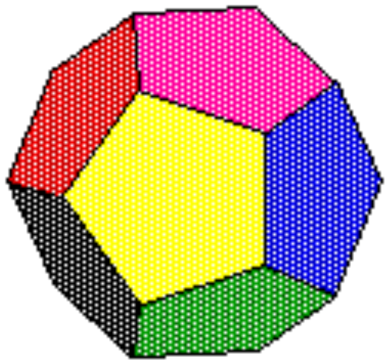
A regular hexagon has internal angles of 120°, but 3×120°=360° which **won't work** because at 360° the shape flattens out.

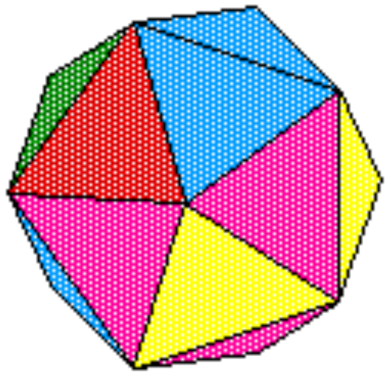
The five Platonic solids


The Tetrahedron


The Cube


The Octahedron


The Dodecahedron


The Icosahedron

The five regular solids discovered by the Ancient Greek mathematicians are:

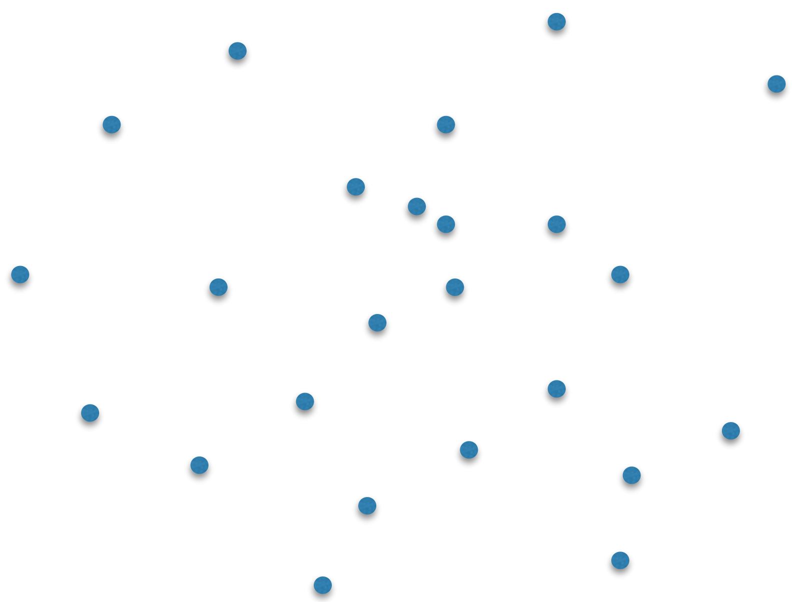
The **Tetrahedron**:	4 vertices	6 edges	4 faces	each with 3 sides
The **Cube**:	8 vertices	12 edges	6 faces	each with 4 sides
The **Octahedron**:	6 vertices	12 edges	8 faces	each with 3 sides
The **Dodecahedron**:	20 vertices	30 edges	12 faces	each with 5 sides
The **Icosahedron**:	12 vertices	30 edges	20 faces	each with 3 sides

The solids are regular because the same number of sides meet at the same angles at each vertex and identical polygons meet at the same angles at each edge.
These five are the only possible regular polyhedra.

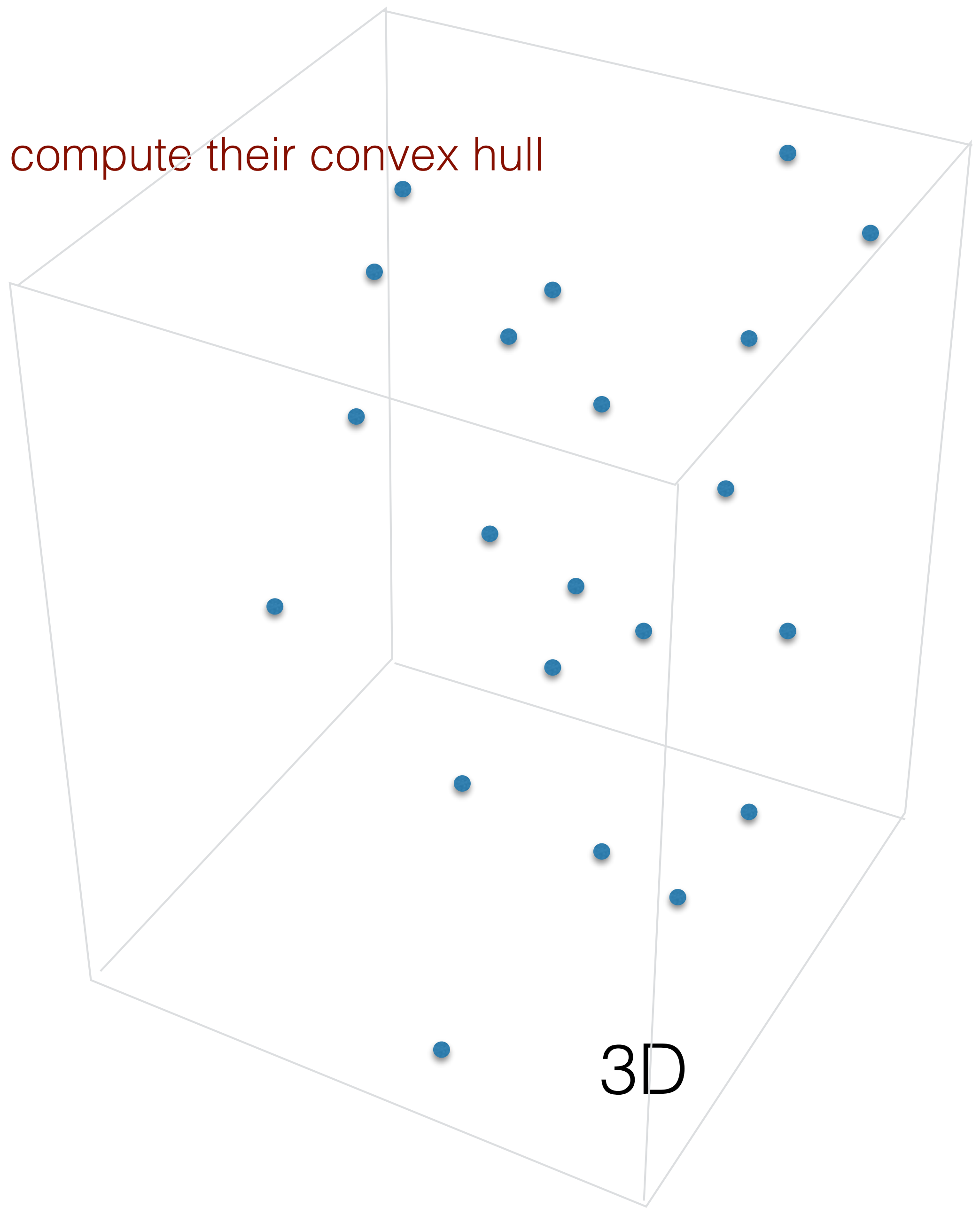
digression end

Convex Hulls

The problem: Given a set P of points, compute their convex hull

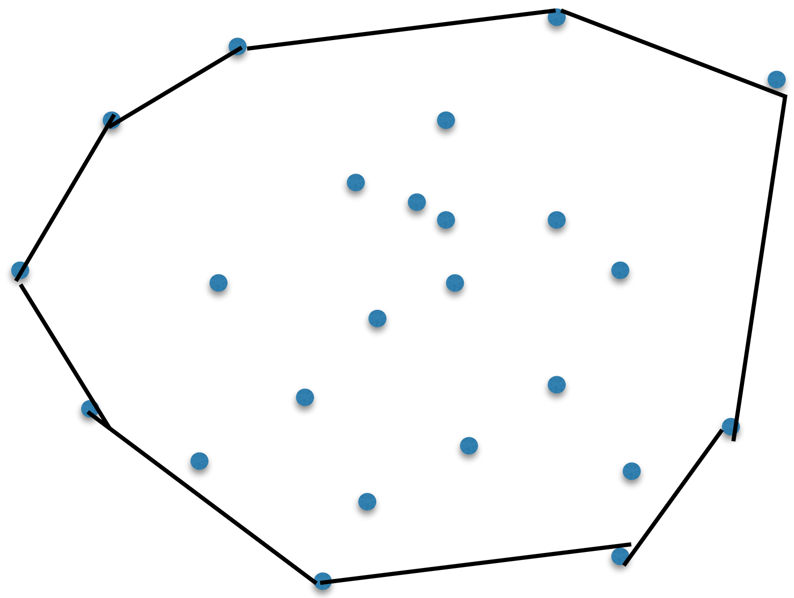


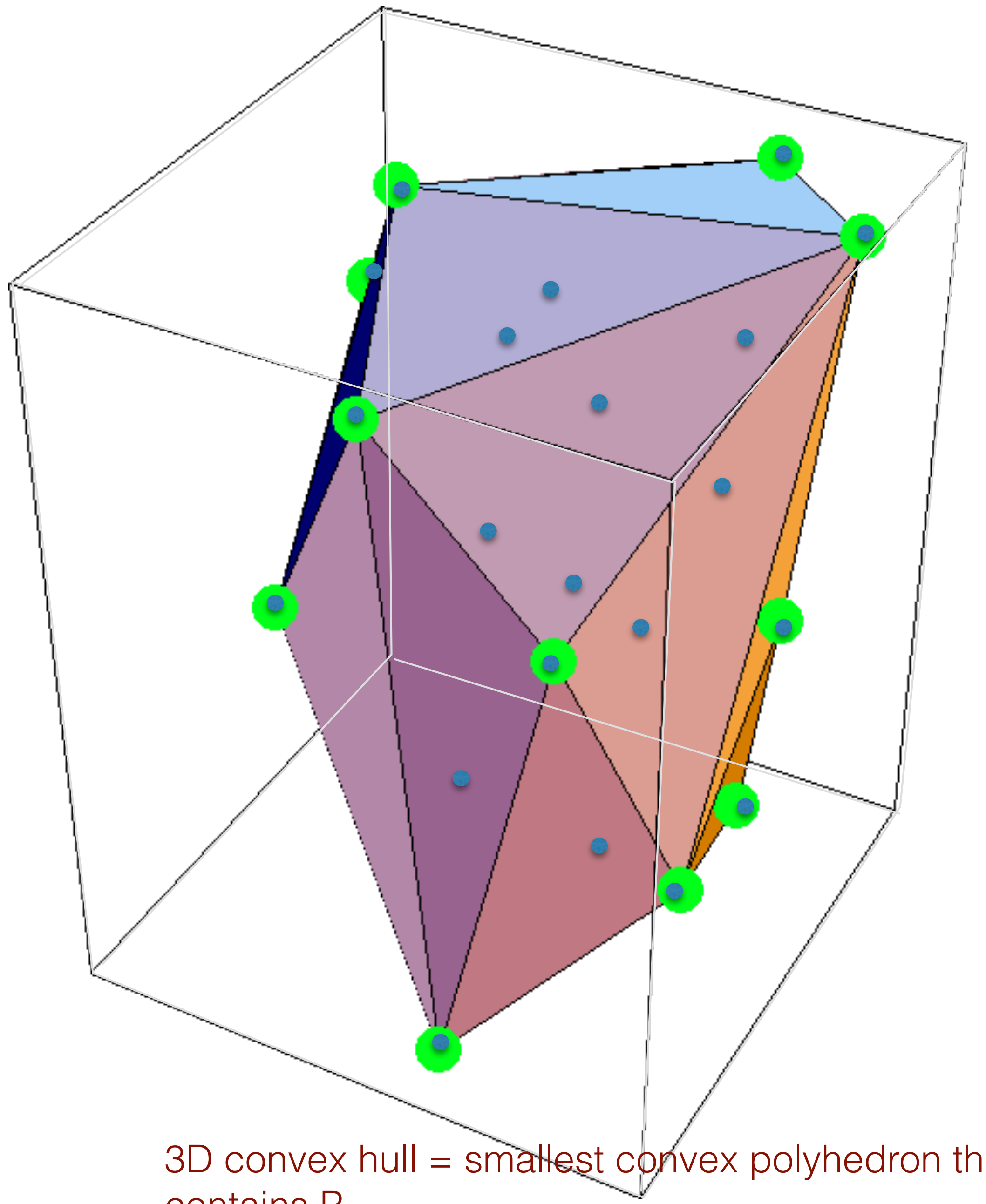
2D



3D

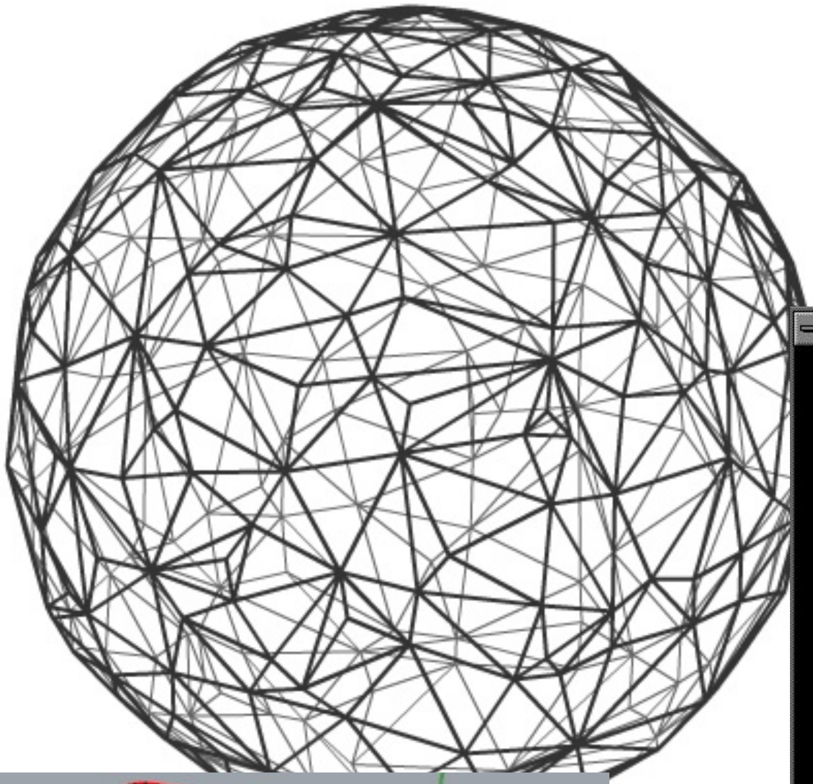
Convex Hulls

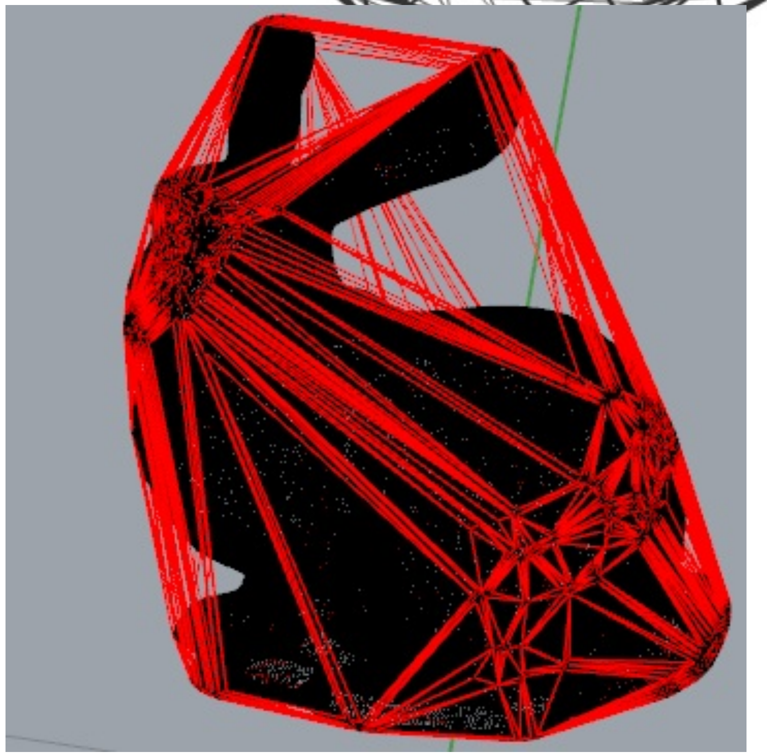


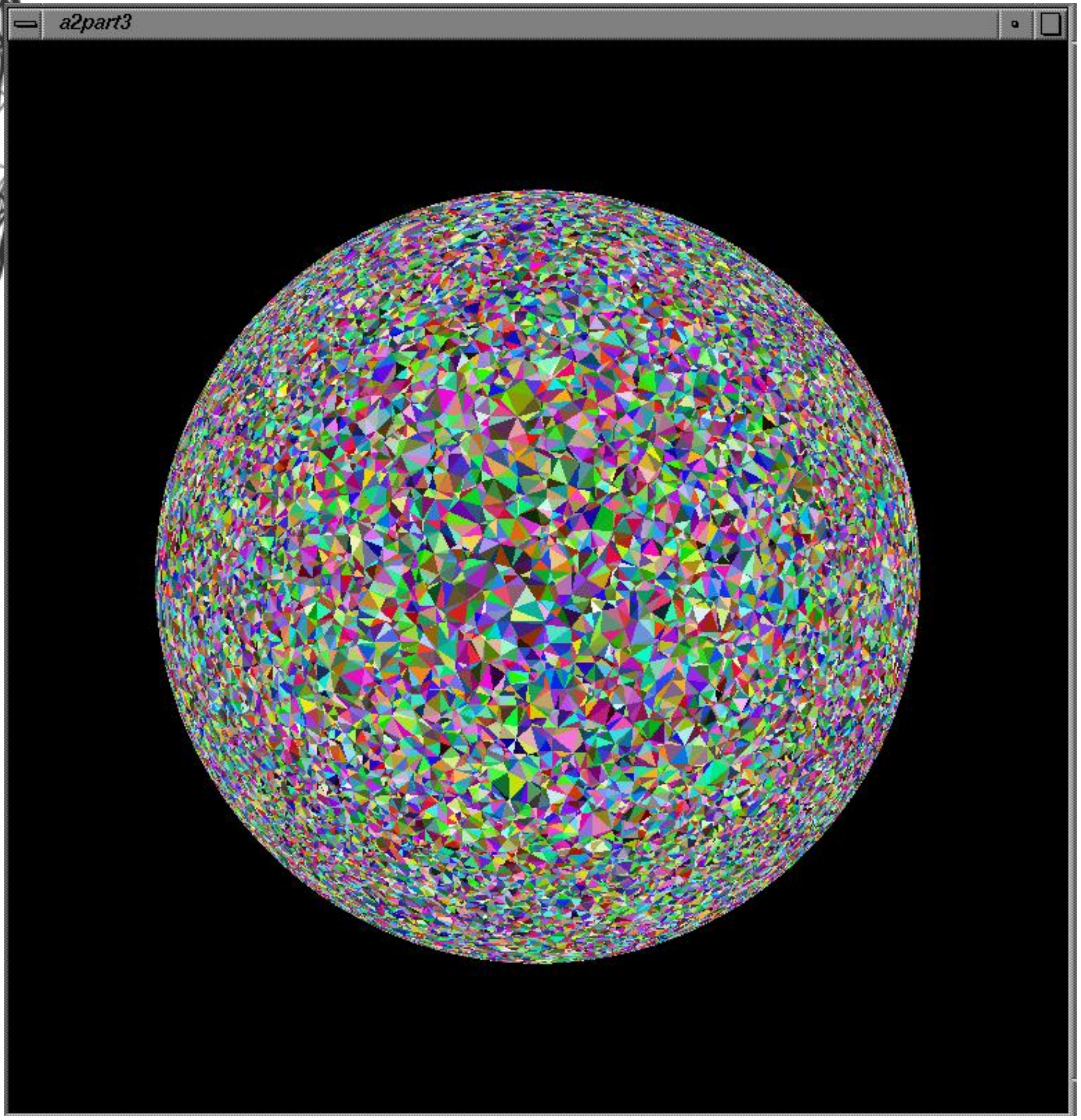


2D convex hull = smallest convex polygon (polytope) that contains P

3D convex hull = smallest convex polyhedron that contains P

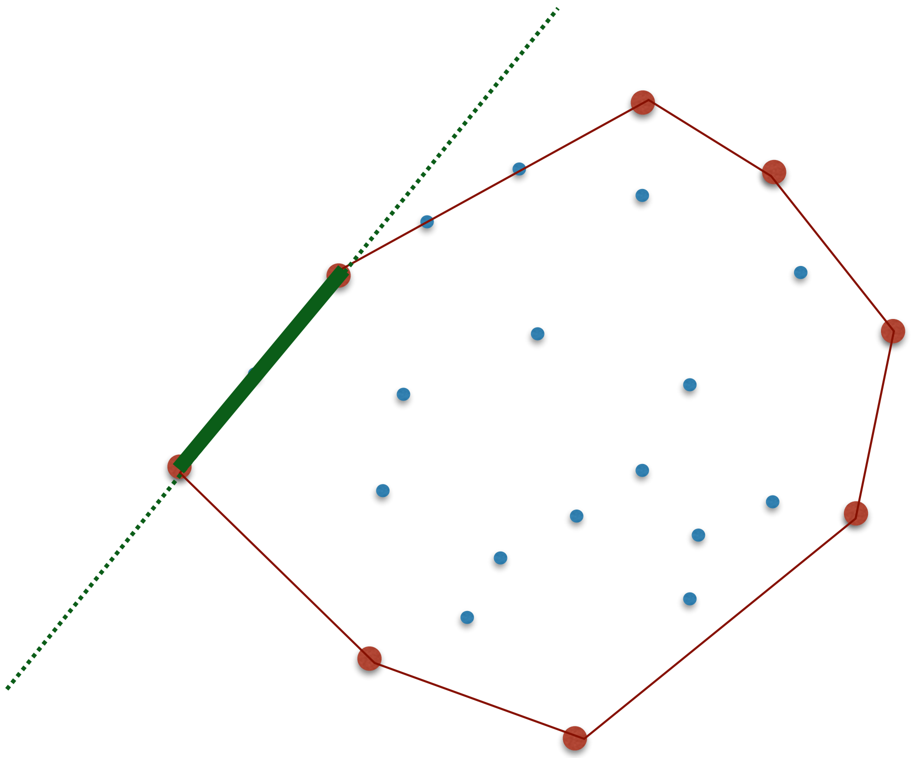


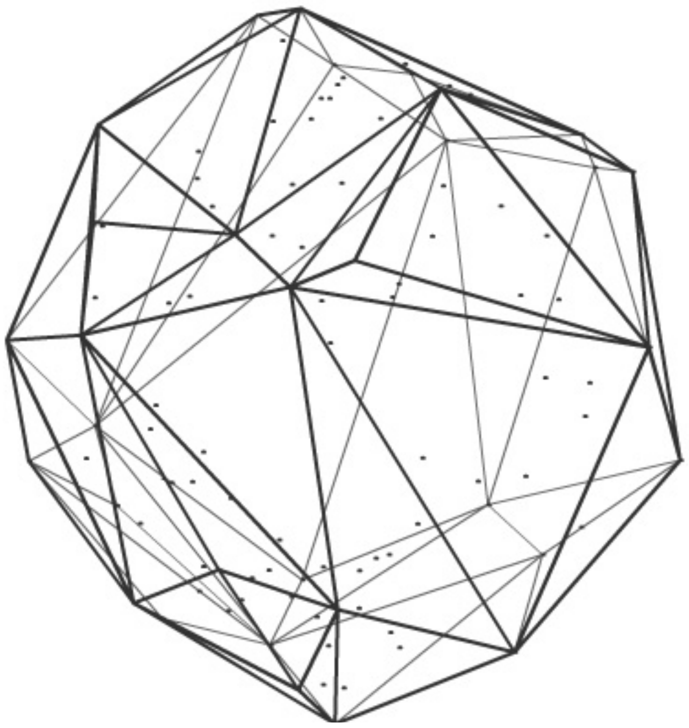



a2part3

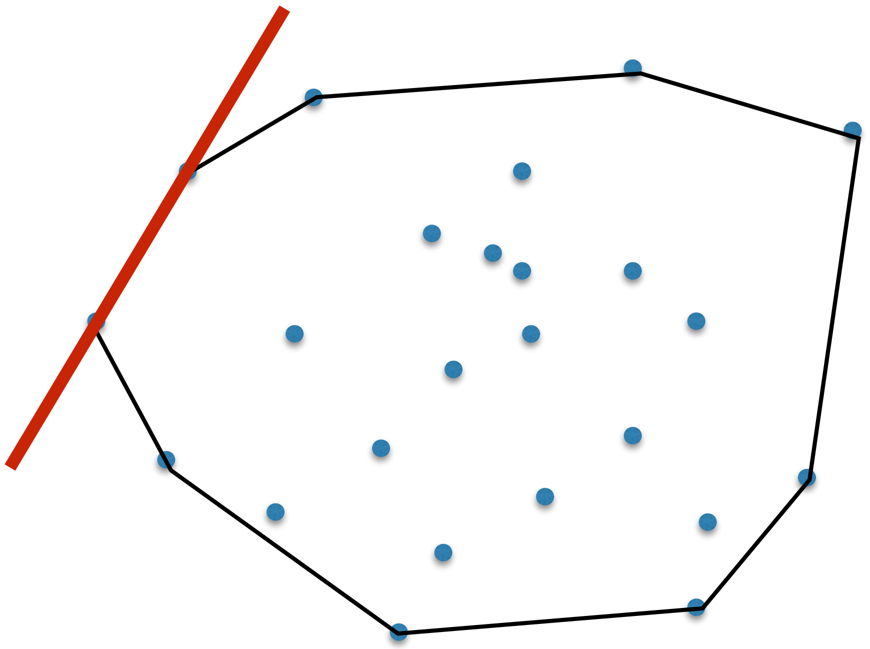
Properties of **3d** hulls

- 3d hull consists of all **extreme** faces, edges and vertices
- All internal angles between faces are < 180
- ~~Walking counterclockwise —> left turns~~
- ~~Points on CH are sorted in radial order wrt a point inside~~

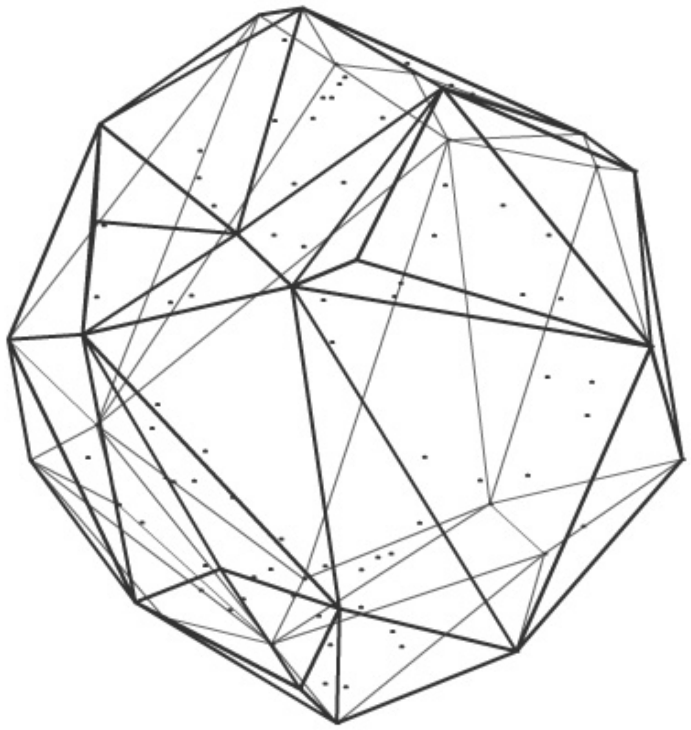




Faces, edges, vertices on the hull are **extreme**.



2D



3D

Naive 3d hull

3d hull: Naive algorithm

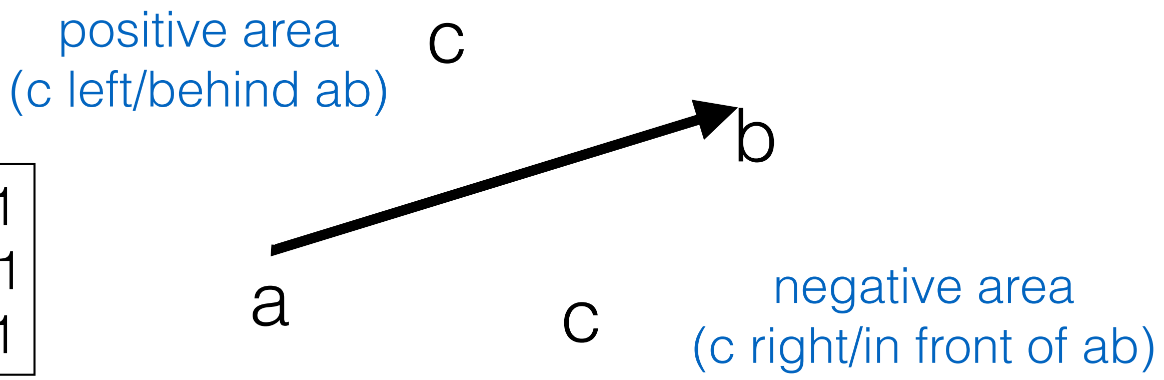
- For every triplet of points (p_i, p_j, p_k):
    - check if face is extreme
    - if True, add (p_i, p_j, p_k) to the list of hull faces

```
// return True if all points  are on the same side of face (a,b,c)
face_is_extreme(point3d a, b, c,  vector<point3d> points)
```

- face_is_extreme() runs in $O(n)$
- Naive() runs in $O(n^4)$

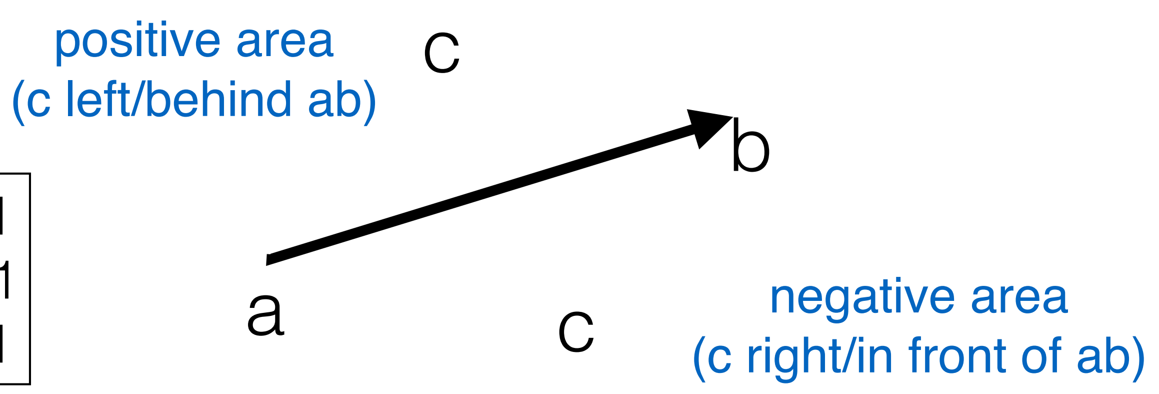
2D

$$2 \text{ signedArea}(a,b,c) = \det \begin{vmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{vmatrix}$$



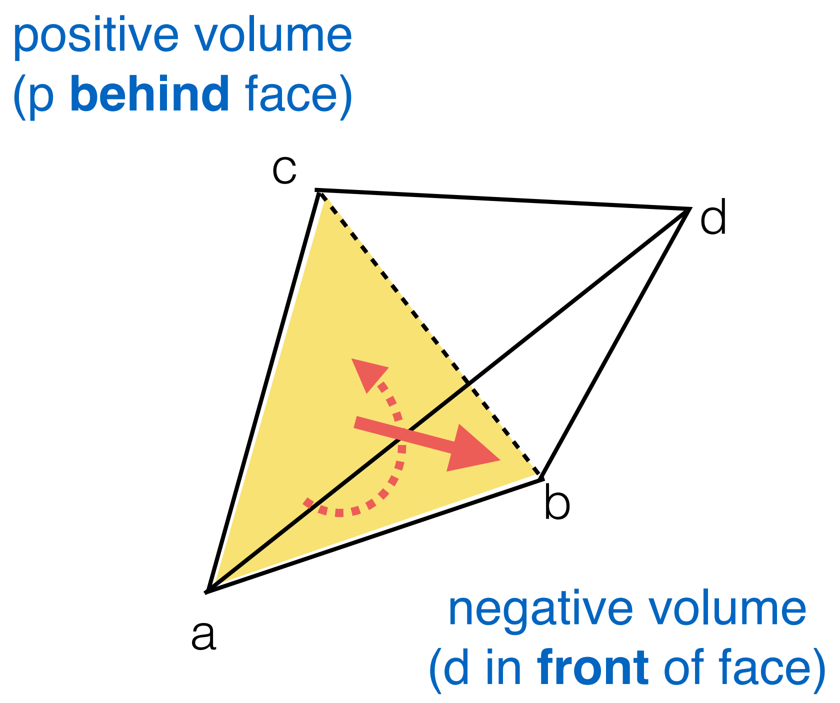
2D

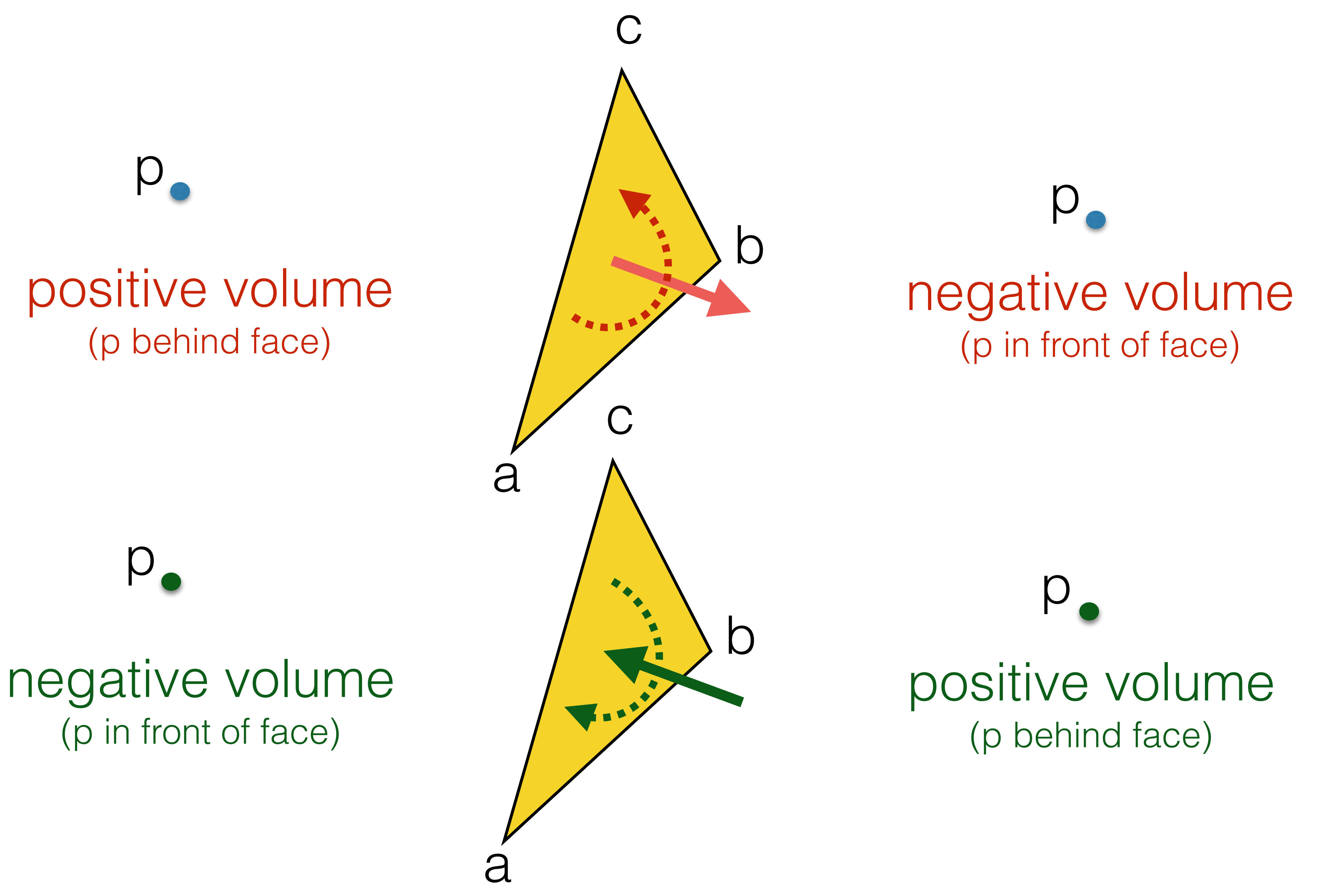
$$2\ \text{signedArea}(a,b,c) = \det \begin{vmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{vmatrix}$$



3D

$$6\ \text{signedVolume}(a,b,c,d) = \det \begin{vmatrix} a.x & a.y & a.z & 1 \\ b.x & b.y & b.z & 1 \\ c.x & c.y & c.z & 1 \\ d.x & d.y & d.z & 1 \end{vmatrix}$$

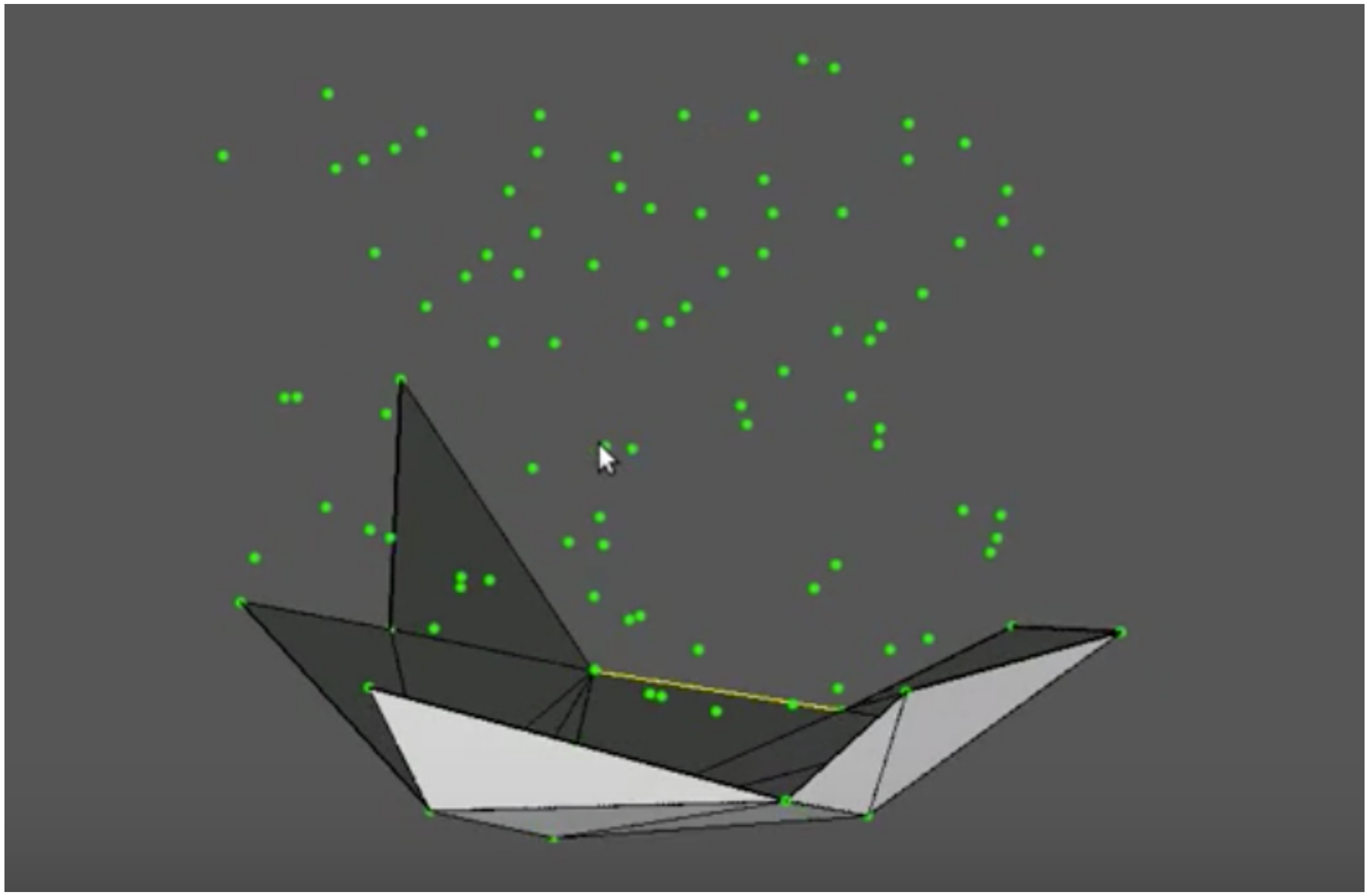




- The convention is that all faces of a polyhedron are oriented so that their normals determined by the right-hand rule point towards the outside.

Gift wrapping

Gift wrapping in 3D



- YouTube
  - Video of CH in 3D (by Lucas Benevides)

Gift wrapping

- First face: find a face guaranteed to be on the CH
- REPEAT
  - $O(n)$ find an edge e of a face f that's on the CH, and such that the face on the other side of e has not been found.
  - for all remaining points p_i, find the angle of (e, p_i) with f
  - $O(n)$ find point p_i with the max angle; add face (e, p_i) to CH

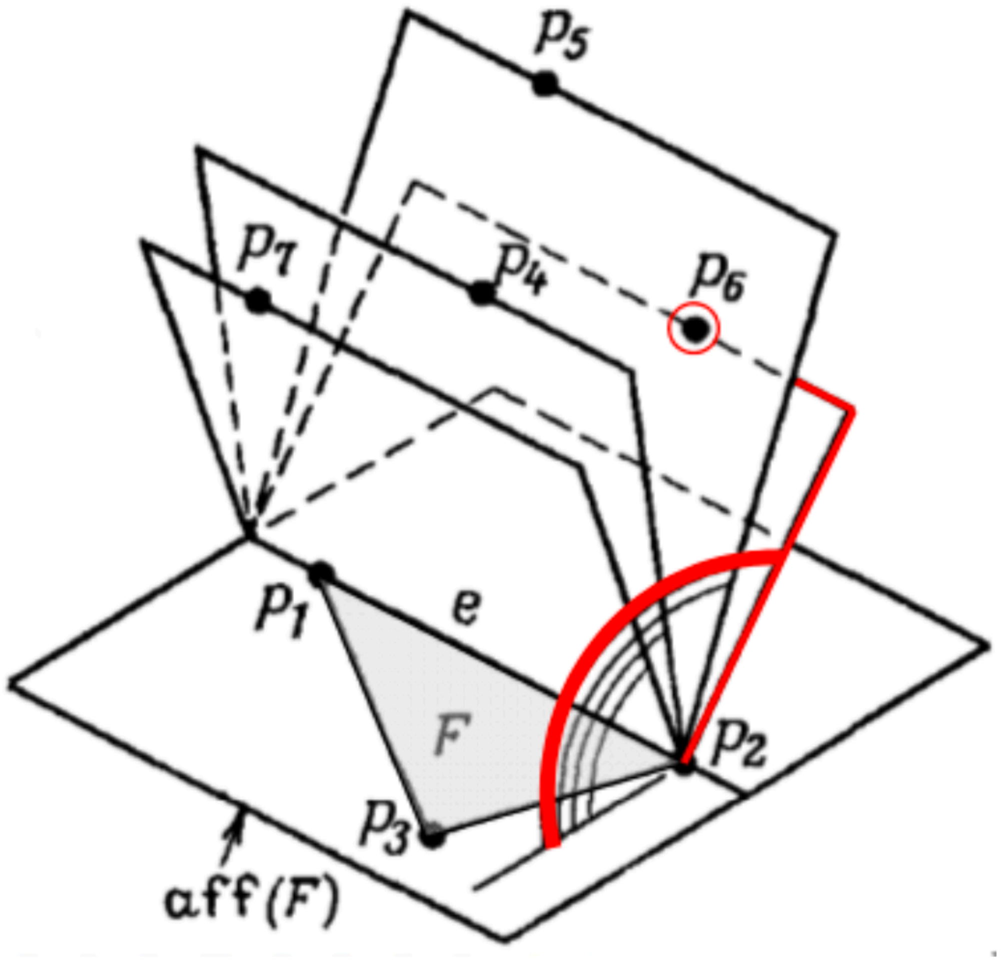
- This runs in time $O(n \cdot F)$ where F is the number of faces on CH

Gift wrapping

- First face: find a face guaranteed to be on the CH
- REPEAT
  - find an edge e of a face f that's on the CH, and such that the face on the other side of e has not been found.
  - for all remaining points p_i, find the angle of (e, p_i) with f
  - find point p_i with the max angle; add face (e, p_i) to CH

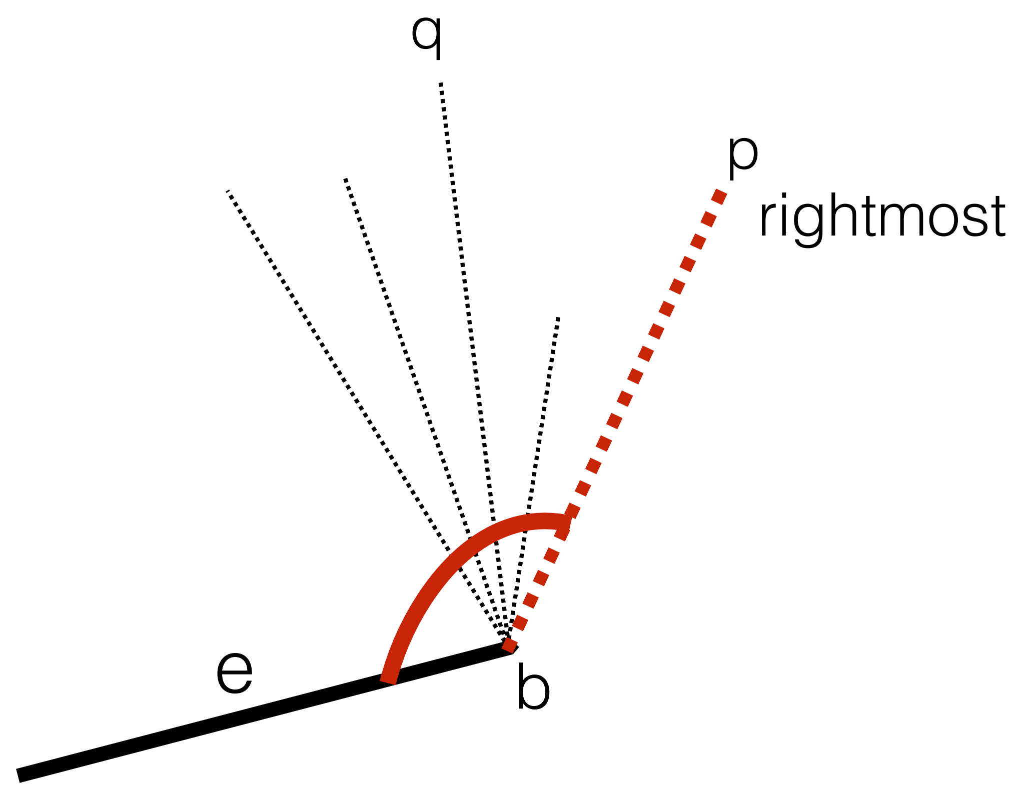
- How to find a first face?
- How to keep track of the boundary of the hull (the edges that have only one face discovered)?
- How to keep track of the hull?   Ideally we would need to store the connectivity (what faces are adjacent, for an edge which faces its adjacent to, etc)
- How to find the angle between two faces, and the face with largest angle?

3D: Given a face on the hull, find an adjacent face



It is the face with largest angle.
Put differently, it is the **front-most** face looking from e.

2D: Given an edge on the hull, find an adjacent vertex

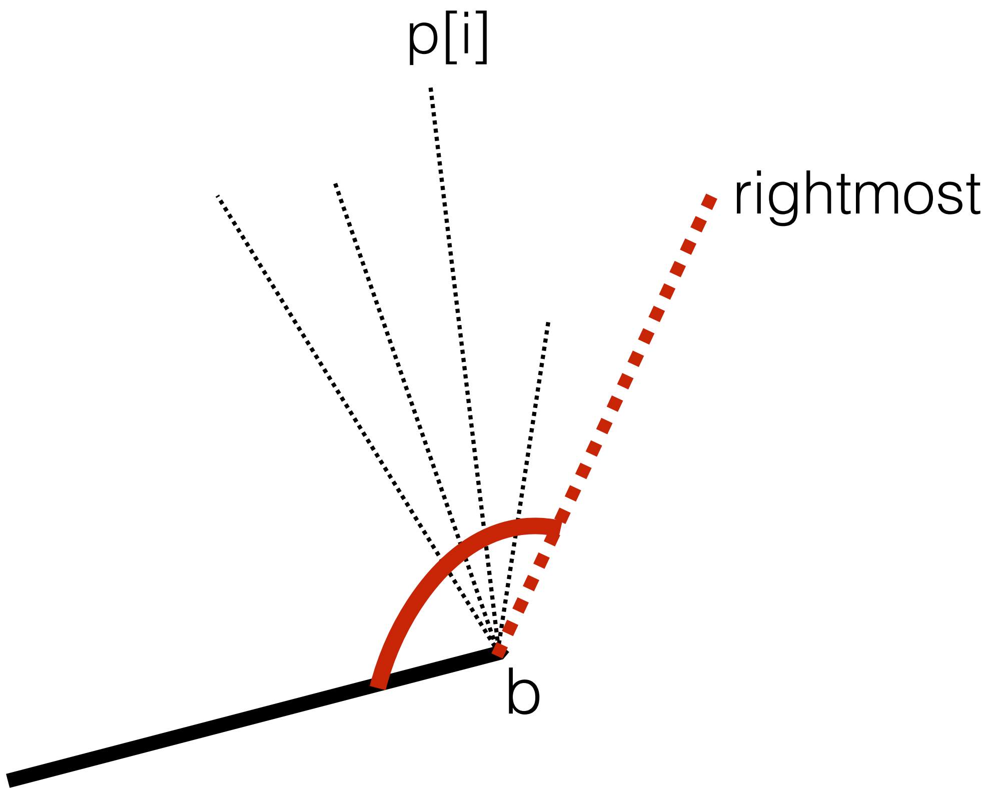


Given an edge e on the convex hull:

Next point p maximizes the angle with e.

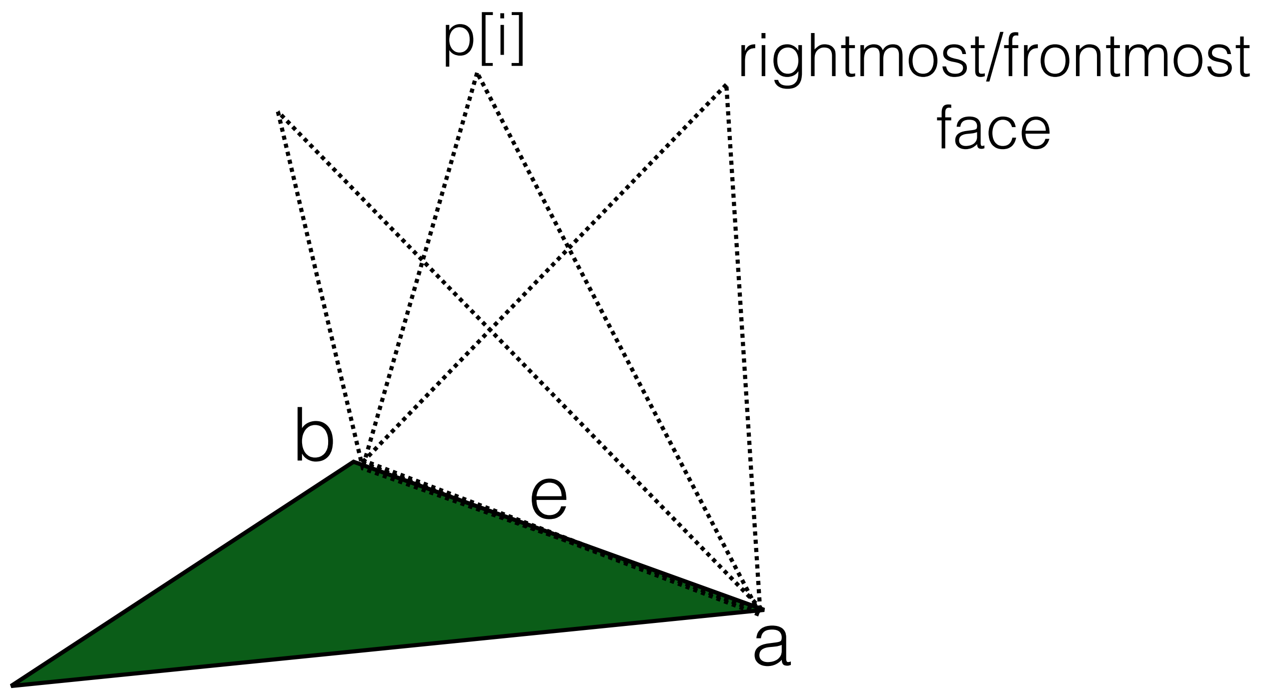
Put differently, b is to the right of all points q.

2D: Given an edge on the hull, find an adjacent vertex



- rightmost = p[0]
- for (i = 0; i < points.size(); i++)
    - if p[i]==b continue;
    - if rightOf(b, rightmost, points[i]): rightmost=points[i]

3D: Given a face on the hull, find an adjacent face



- frontmost = p[0]
- for (i = 0; i < points.size(); i++)
    - if p[i]== a or p[i]==b: continue;
    - if inFront(a, b, frontmost, points[i]): frontmost=points[i]

How to find the first face?

IDEA 1

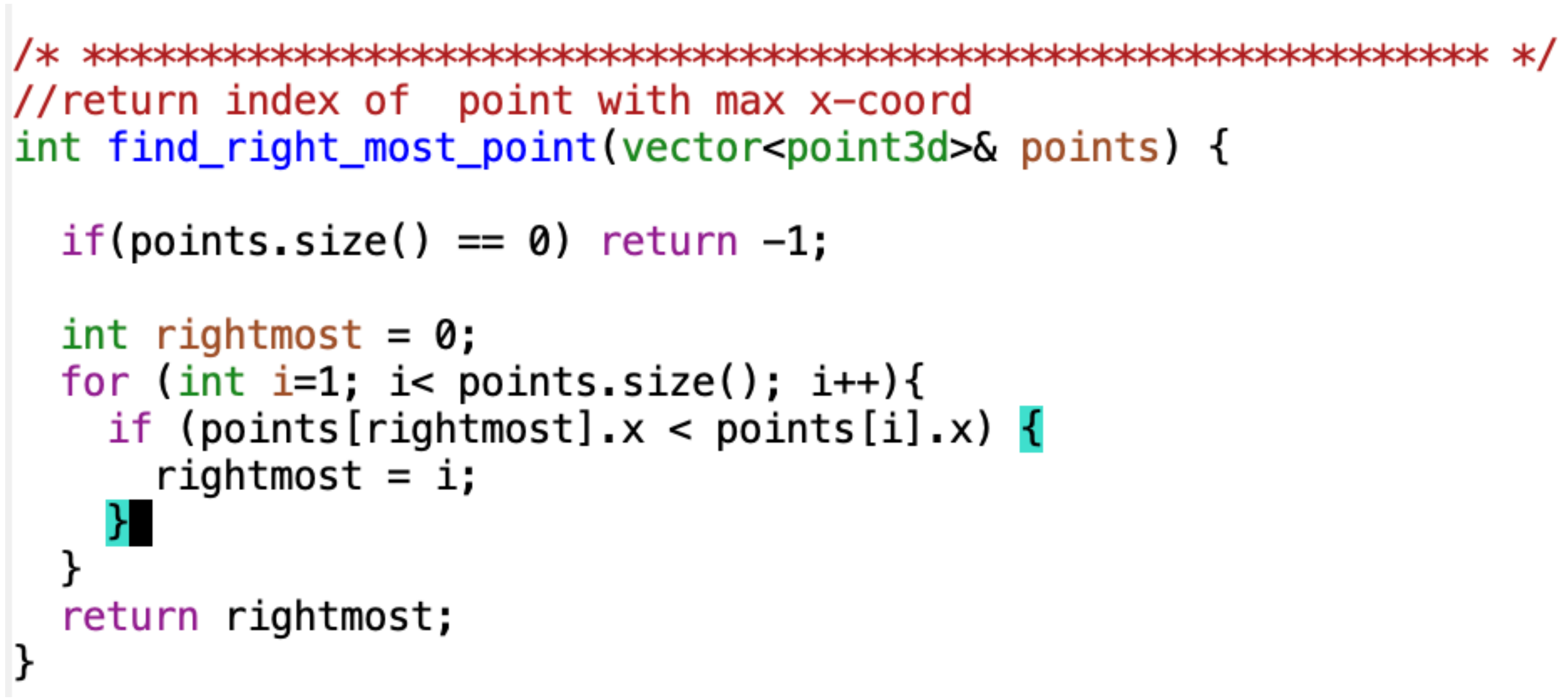
- Find an extreme point (e.g. max x-coord) p = (p.x, p.y. p.z)
- Temporarily create a (fake) point q = (p.x, p.y+1, p.z)
- Iterate through remaining points and find the front-most point r with respect to (p, q)
- => (p,q,r) is an extreme face, so point r is guaranteed to be on the hull
- => edge (p,r) is on the hull
- Iterate through the remaining points, and find the vertex s that is front-most with respect to (p, r)
- => face (p, r, s) is a face on the hull

How to find the first face?

IDEA 2

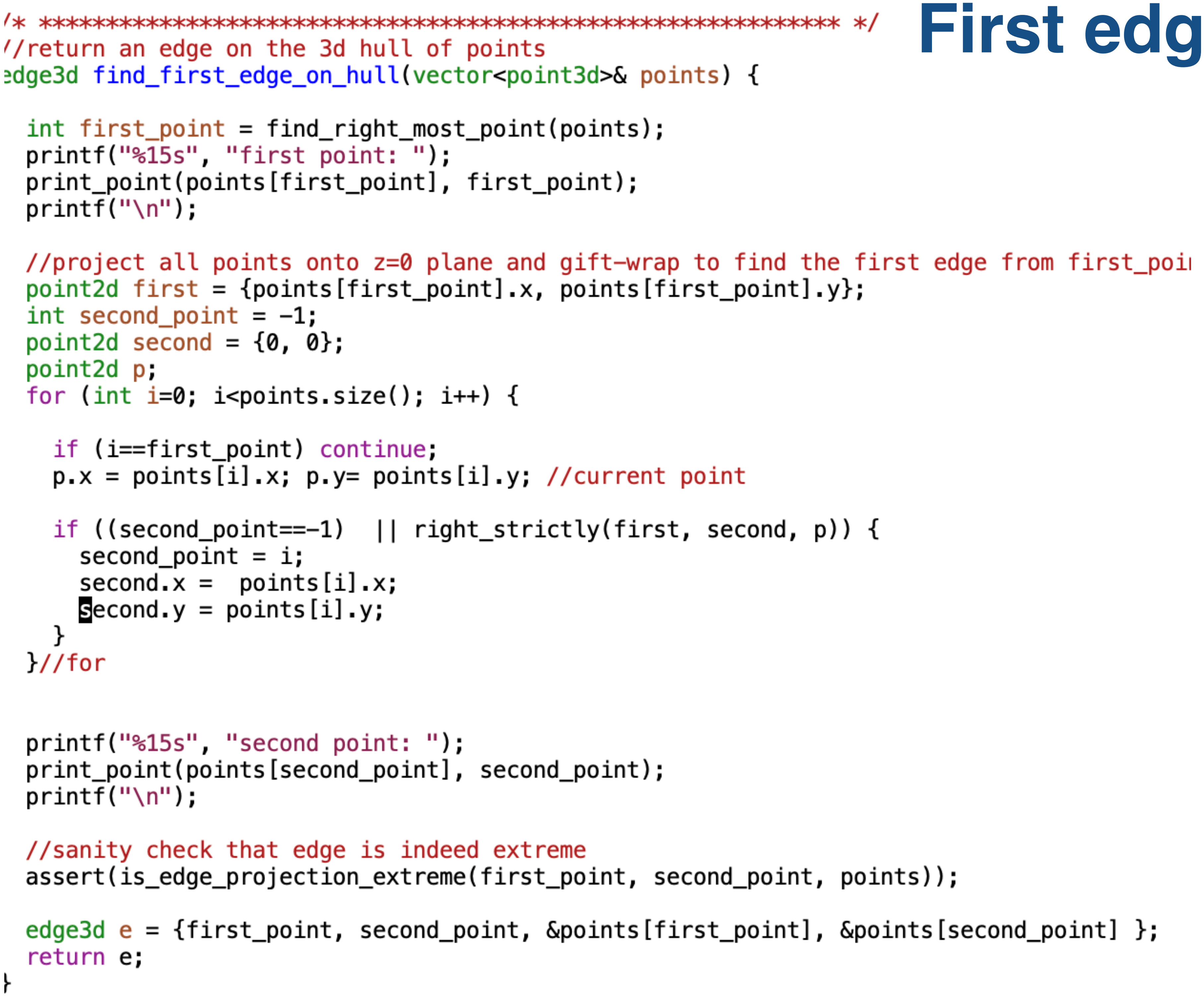
- Find a point on 3d-hull: ind an extreme point, say point with max x-coord. Call this $first$.
- Find an edge on the 3d-hull: Project all points on the xy-plane (ignore their z-coord), and imagine computing the 2D-convex hull of the projection. These edges will be on the 3D-hull. We only need one edge, and to find it we can 2d-gift-wrap around $first$ to find the point $second$ that is right-most when looking from $first$. Thus edge $(first, second)$ is on the 3d-hull.
- Find a face on the 3d-hull: find a point, call it $third$, so that third is right of all faces $(first, second, p)$ for all other points p.

First point



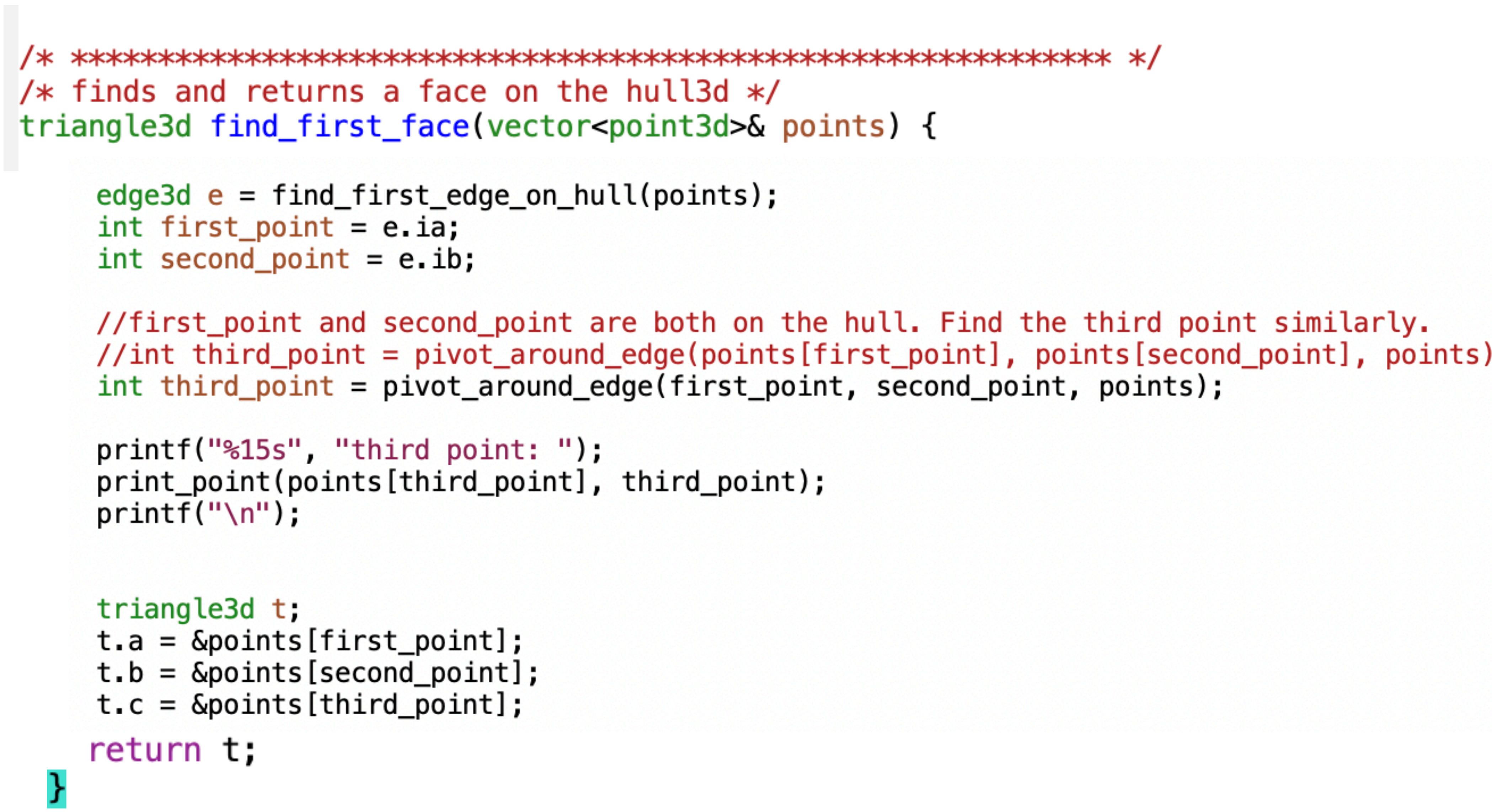
```
/* ************************************************************ */
//return index of  point with max x-coord
int find_right_most_point(vector<point3d>& points) {

  if(points.size() == 0) return -1;

  int rightmost = 0;
  for (int i=1; i< points.size(); i++){
    if (points[rightmost].x < points[i].x) {
      rightmost = i;
    }
  }
  return rightmost;
}
```

First edge



```
/* ************************************************************* */
//return an edge on the 3d hull of points
edge3d find_first_edge_on_hull(vector<point3d>& points) {

  int first_point = find_right_most_point(points);
  printf("%15s", "first point: ");
  print_point(points[first_point], first_point);
  printf("\n");

  //project all points onto z=0 plane and gift-wrap to find the first edge from first_poi
  point2d first = {points[first_point].x, points[first_point].y};
  int second_point = -1;
  point2d second = {0, 0};
  point2d p;
  for (int i=0; i<points.size(); i++) {

    if (i==first_point) continue;
    p.x = points[i].x; p.y= points[i].y; //current point

    if ((second_point==-1)  || right_strictly(first, second, p)) {
      second_point = i;
      second.x =  points[i].x;
      second.y = points[i].y;
    }
  }//for


  printf("%15s", "second point: ");
  print_point(points[second_point], second_point);
  printf("\n");

  //sanity check that edge is indeed extreme
  assert(is_edge_projection_extreme(first_point, second_point, points));

  edge3d e = {first_point, second_point, &points[first_point], &points[second_point] };
  return e;
}
```

First face

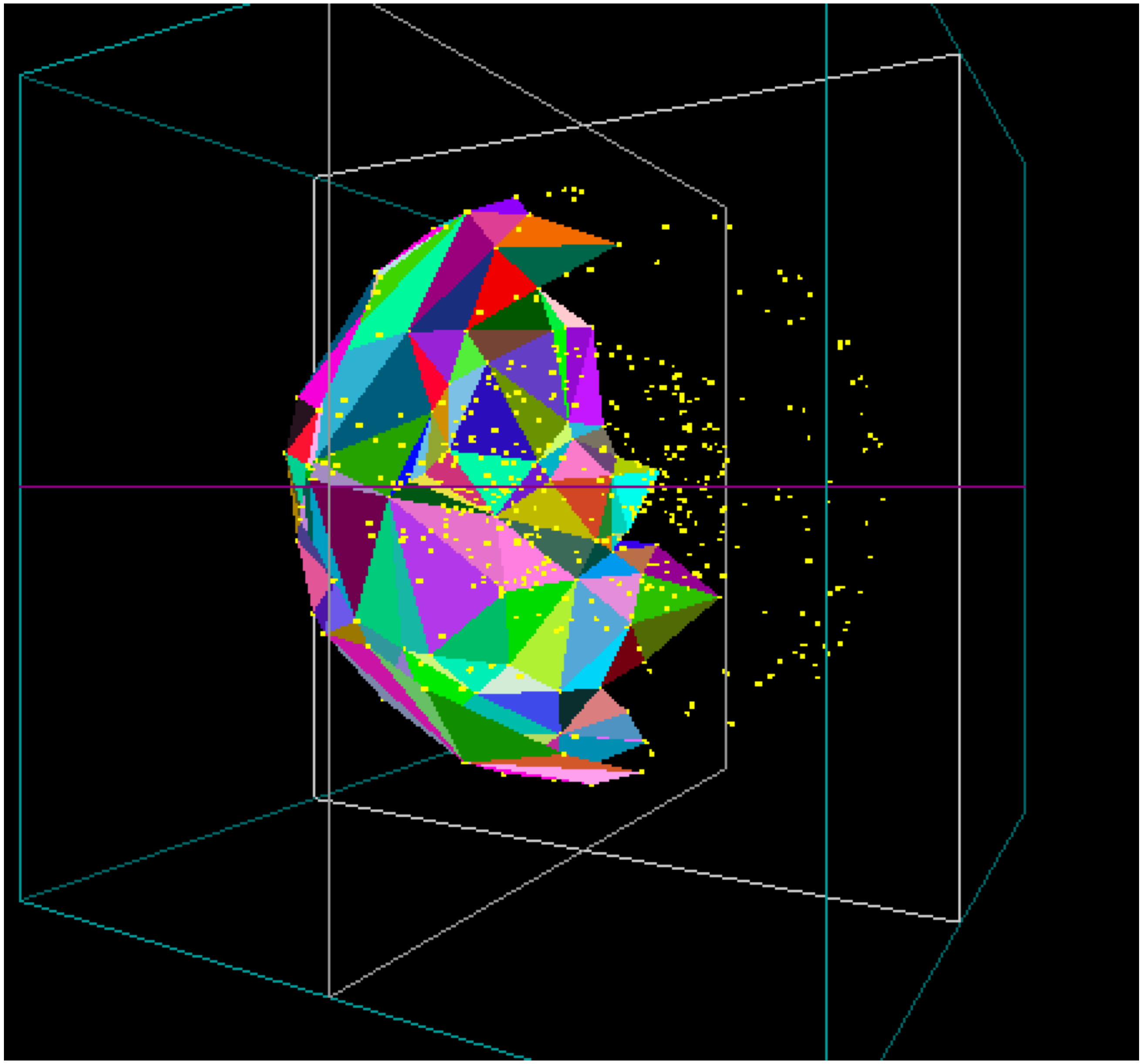


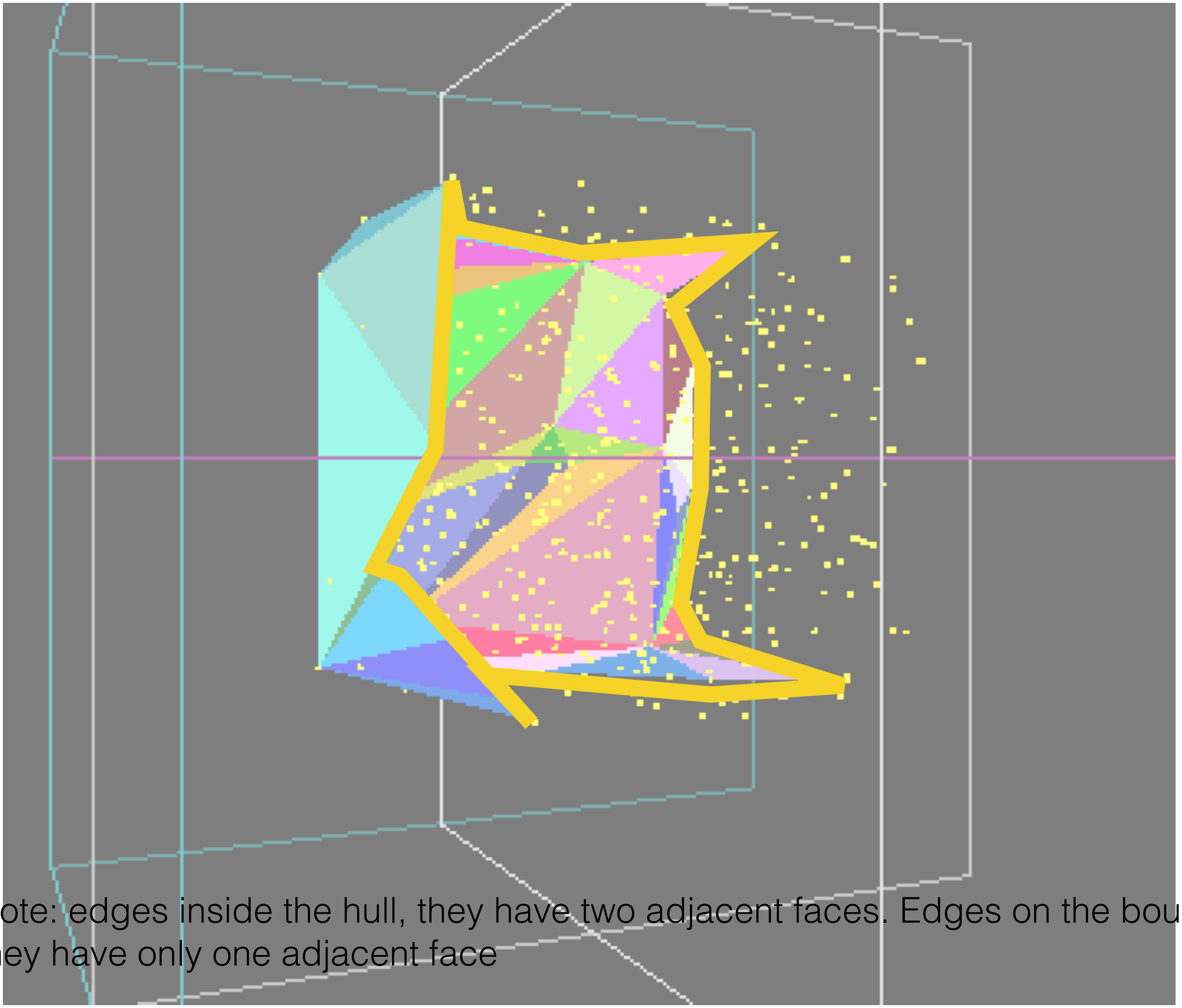
```
/* ************************************************************ */
/* finds and returns a face on the hull3d */
triangle3d find_first_face(vector<point3d>& points) {

    edge3d e = find_first_edge_on_hull(points);
    int first_point = e.ia;
    int second_point = e.ib;

    //first_point and second_point are both on the hull. Find the third point similarly.
    //int third_point = pivot_around_edge(points[first_point], points[second_point], points)
    int third_point = pivot_around_edge(first_point, second_point, points);

    printf("%15s", "third point: ");
    print_point(points[third_point], third_point);
    printf("\n");


    triangle3d t;
    t.a = &points[first_point];
    t.b = &points[second_point];
    t.c = &points[third_point];
   return t;
}
```

Gift wrapping

- First face: find a face guaranteed to be on the CH
- REPEAT
    - find an edge e of a face f that's on the CH, and such that the face on the other side of e has not been found.
    - for all remaining points p_i, find the angle of (e, p_i) with f
    - find point p_i with the minimal angle; add face (e, p_i) to CH

- How to keep track of the hull?
- How to keep track of the boundary of the hull?





- Note: edges inside the hull, they have two adjacent faces. Edges on the boundary, they have only one adjacent face

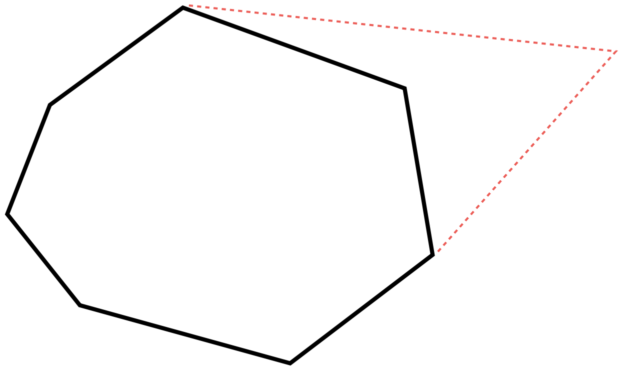
- The hull:
    - vector<triangle3d> hull   (Ideally a triangle has pointers to all adjacent faces)
- Boundary of the hull:
    - a queue
- Edges that are done:
    - need to search (is an edge e done?), and to update (mark edge e as done)

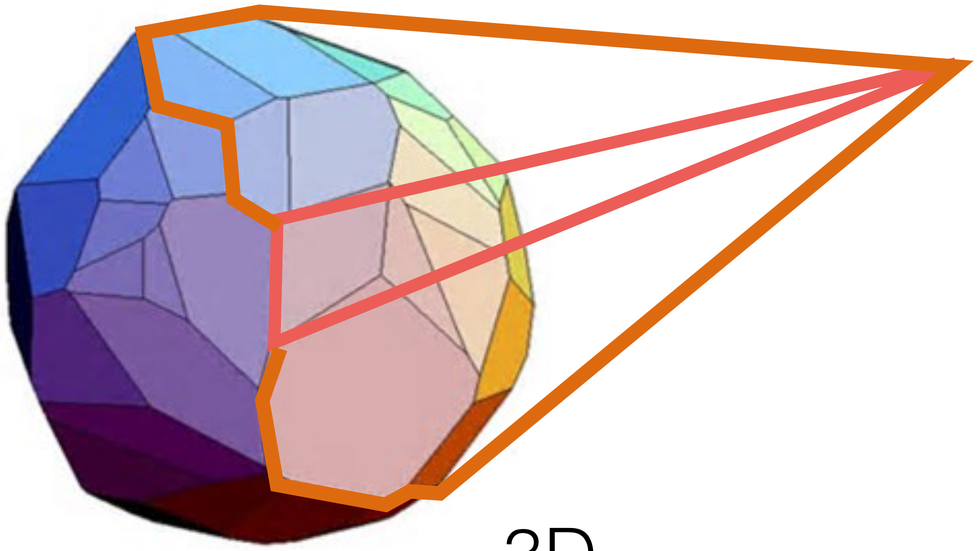
- First face: find a face guaranteed to be on the CH
- $Hull = \{\}$
- $Bnd = \{\}$
- REPEAT
    - Remove an edge e from Bnd and mark is as done.
    - Check if e is done, and if so, continue
    - Find a vertex p so that for all other points p_i, point p is to the right of face (e, p_i)
    - Add face (e, p) to $Hull$, and add the new edges to Bnd

Incremental

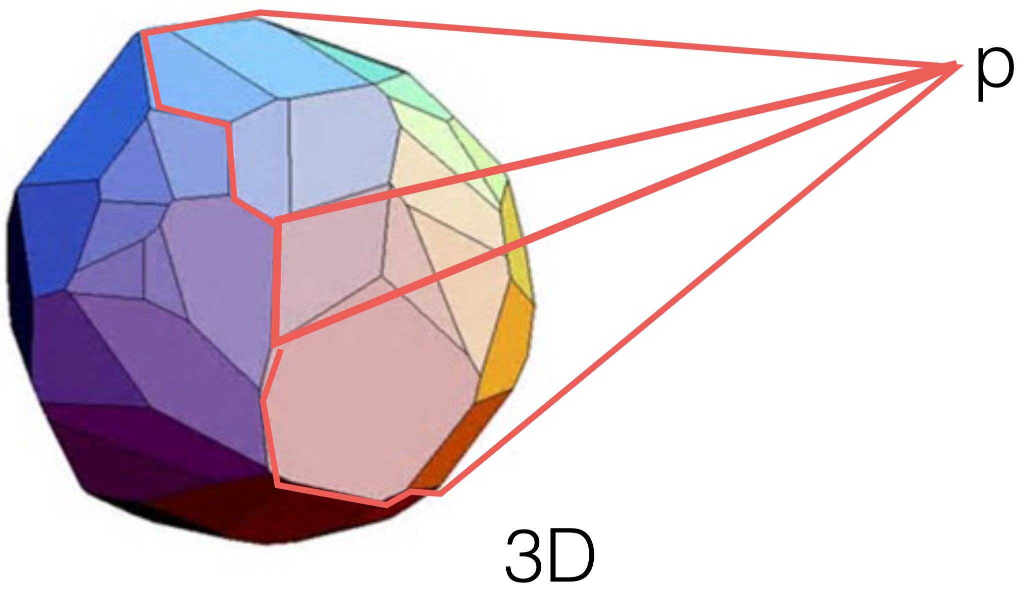
Incremental 3d hull

- initialize hull H of $\{p_1, p_2, p_3, p_4\}$
- for i= 5 to n
    - //invariant: H represents the CH of $p_1, p_2, \ldots, p_{i-1}$
    - add p_i to H and update H to represent the CH of $p_1, p_2, \ldots, p_i$


2D


3D

Incremental 3d hull

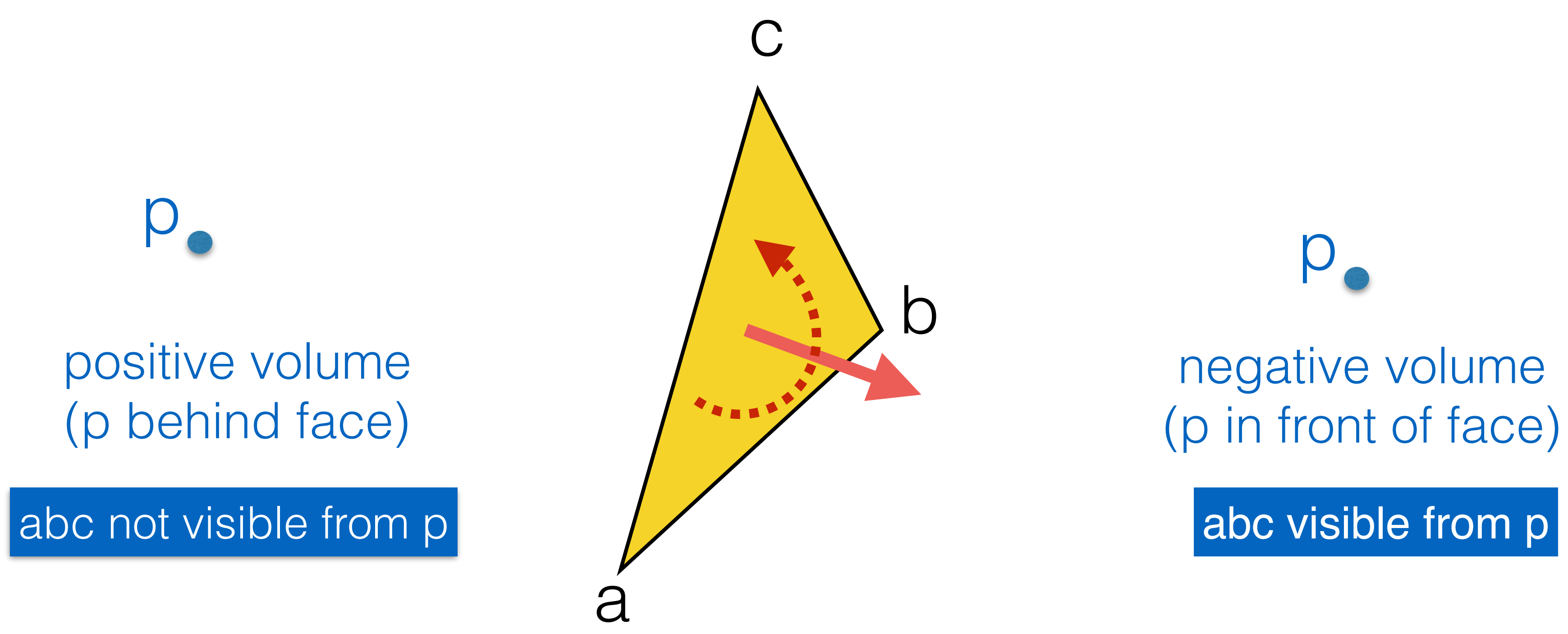


Imagine standing at p and looking towards the hull

The faces that are visible are precisely those that need to be discarded

The edges on the border of the visible region become the basis of the cone

- The convention is that all faces of a polyhedron are oriented so that their normals determined by the right-hand rule point towards the **outside**.



```
//return True if face abc is visible from point p
is_visible(a, b, c, p):
        return signedVolume(a,b,c,p) < 0
```

Incremental 3d hull

- (sort points lexicographically)
- initialize H of p_1, p_2, p_3, p_4
- for each remaining point p:

$O(n)$ • for each face f of H: check if f is visible from p

$O(n)$ • find border edges

$O(n)$ • for each border edge e construct a face (e, p) and add it to H

$O(n)$ • for each visible face f: delete f from H

- Incremental 3d-hull runs in time $O(n^2)$
- RIC: Randomly permute the vertices and then process them in that order, while maintaining a “conflict” graph to speed up finding faces that are visible and need to be deleted. Runs in $O(n \lg n)$

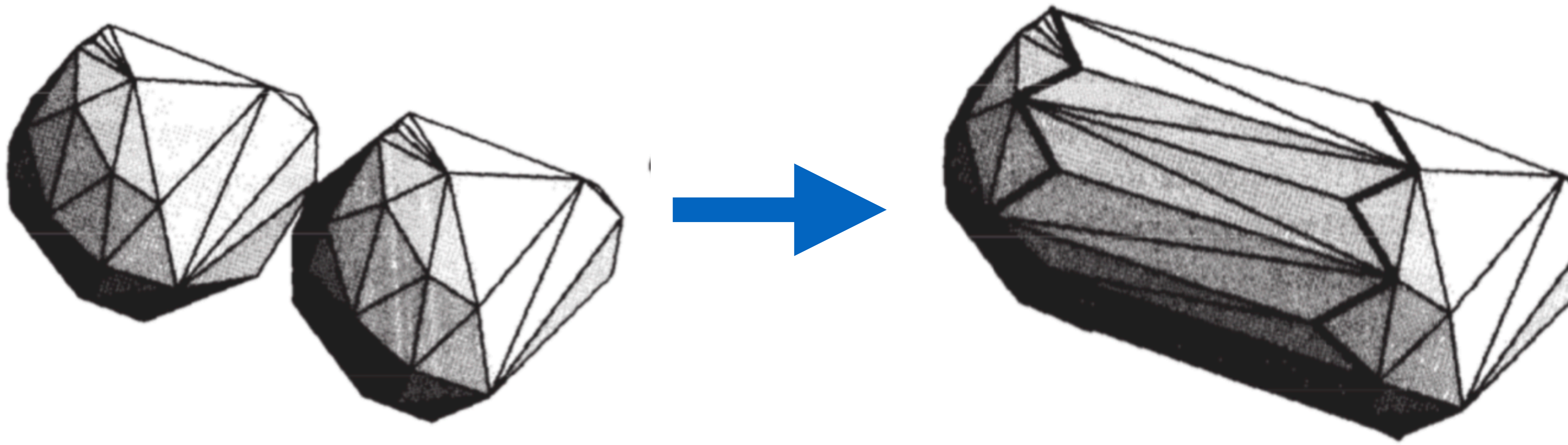
3D hull via divide & conquer

3d hull via divide & conquer

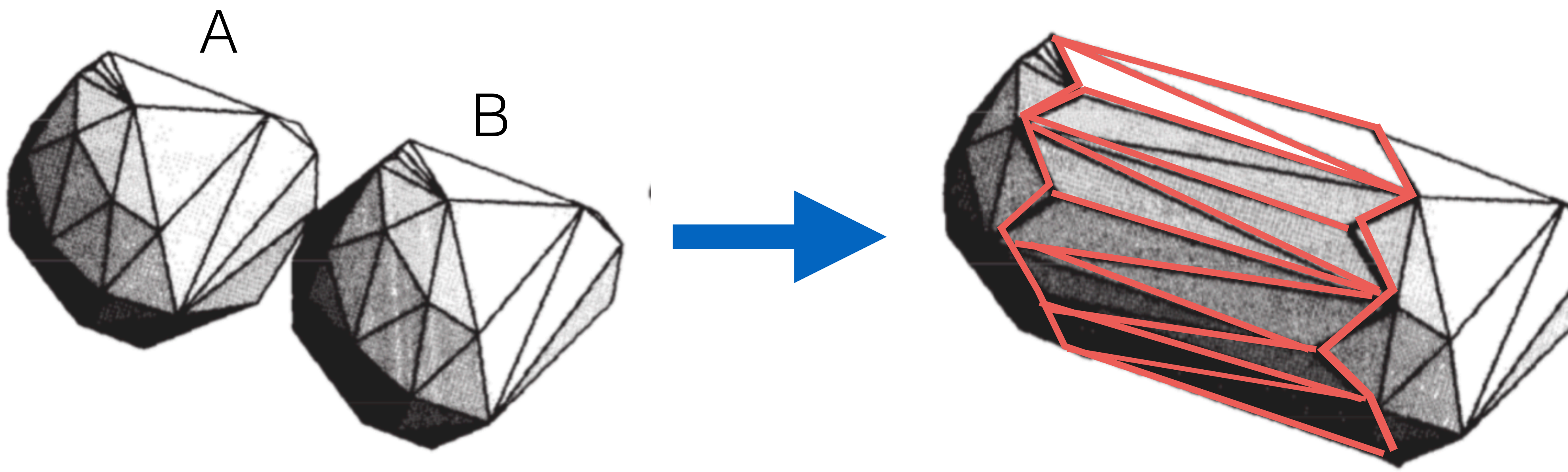
- divide points in two halves P1 and P2
- recursively find CH(P1) and CH(P2)
- merge

Result: Merging can be done in O(n) time ==> 3d-hull via divide-and-conquer runs in O(n lg n) time.

Merging

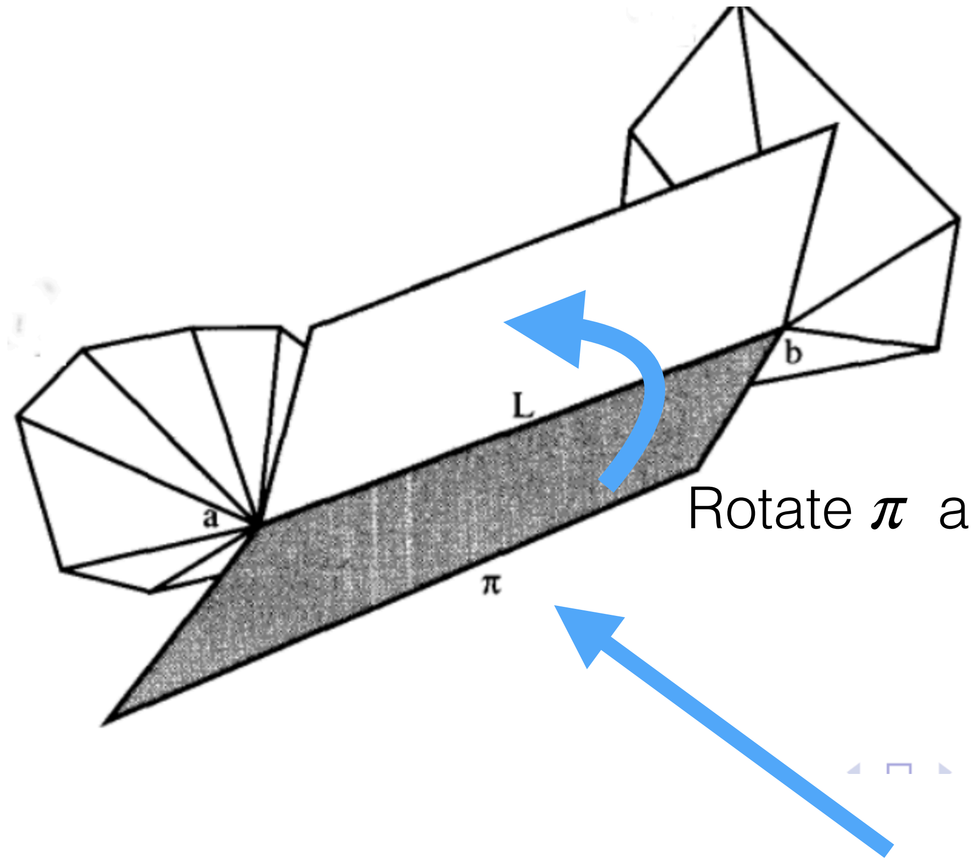


Merging



The merged hull will add a “band” of faces between A and B

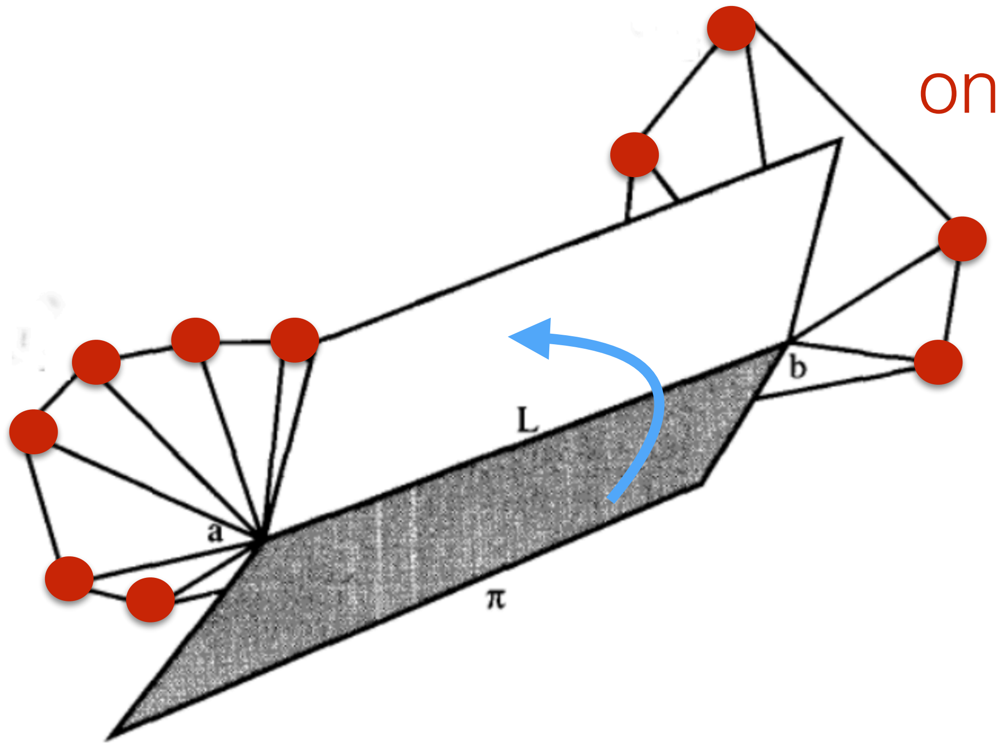
- Imagine rotating the plane around ab, until it touches A and B



Rotate π around ab

Let π be a plane touching A in a and B in b

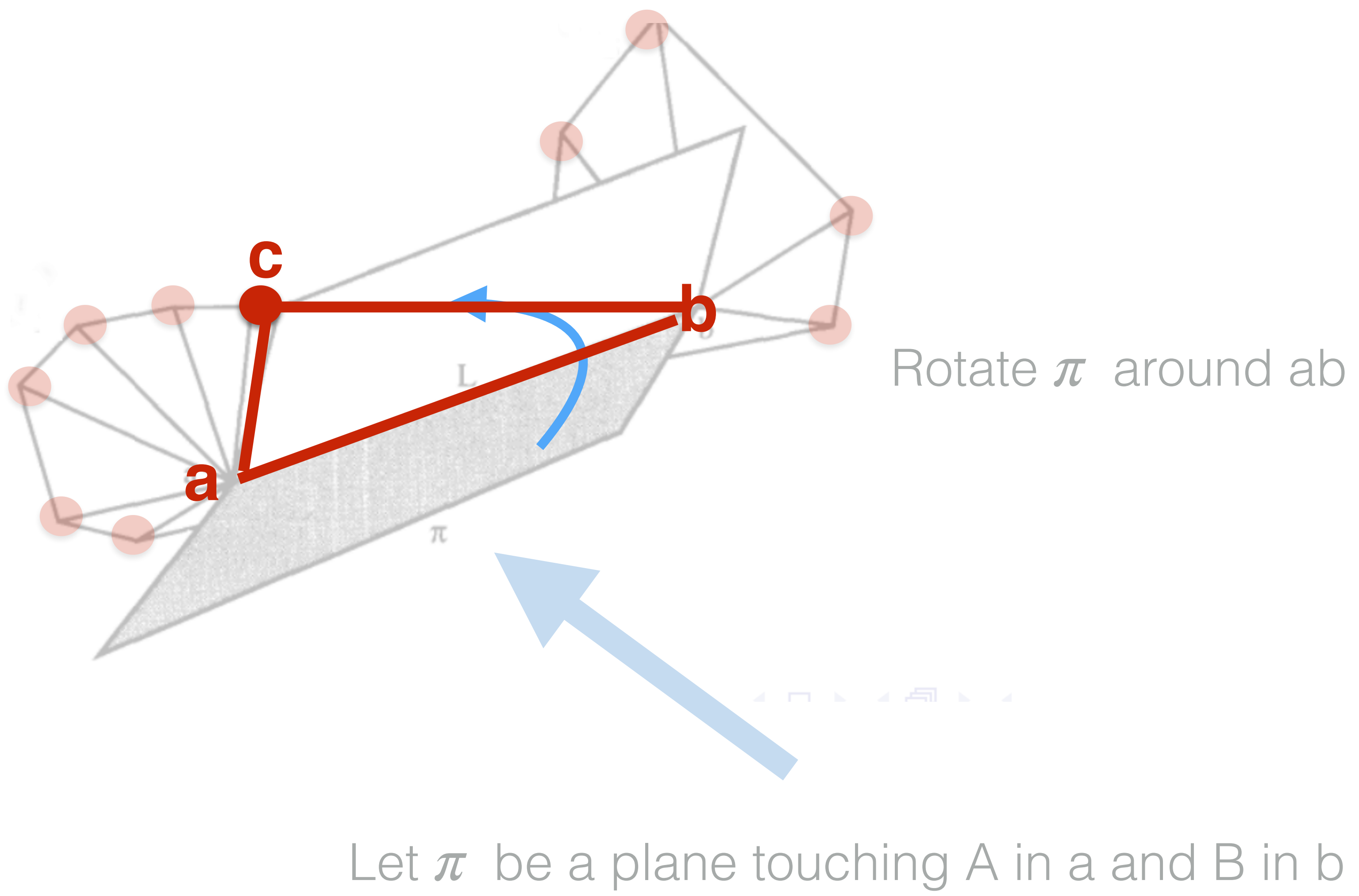
- Claim: When we rotate π around ab, the first vertex hit is a vertex c adjacent to a or b and vertex c has the smallest angle among all neighbors of a,b



one of these vertices

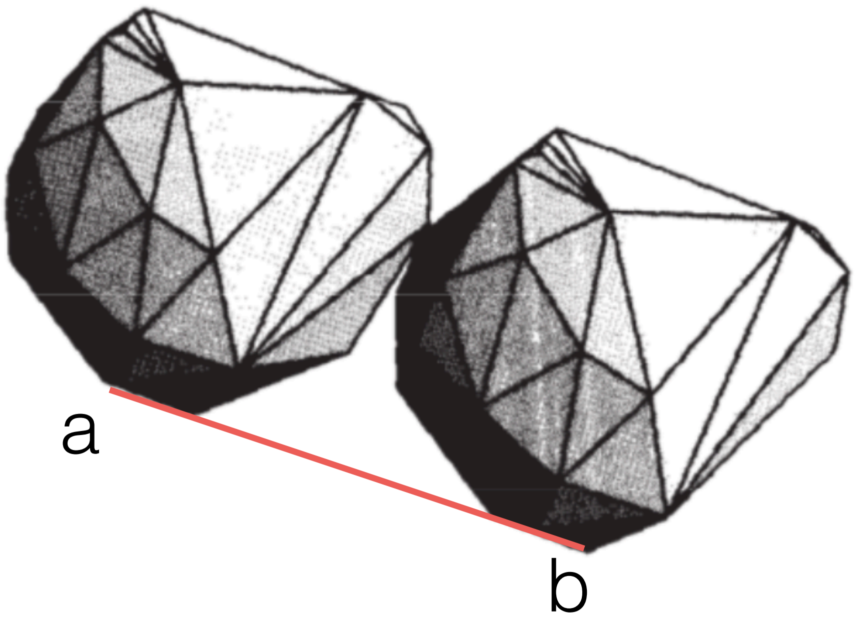
Rotate π around ab

- Once π hits c, a triangular face of the merged hull has been found



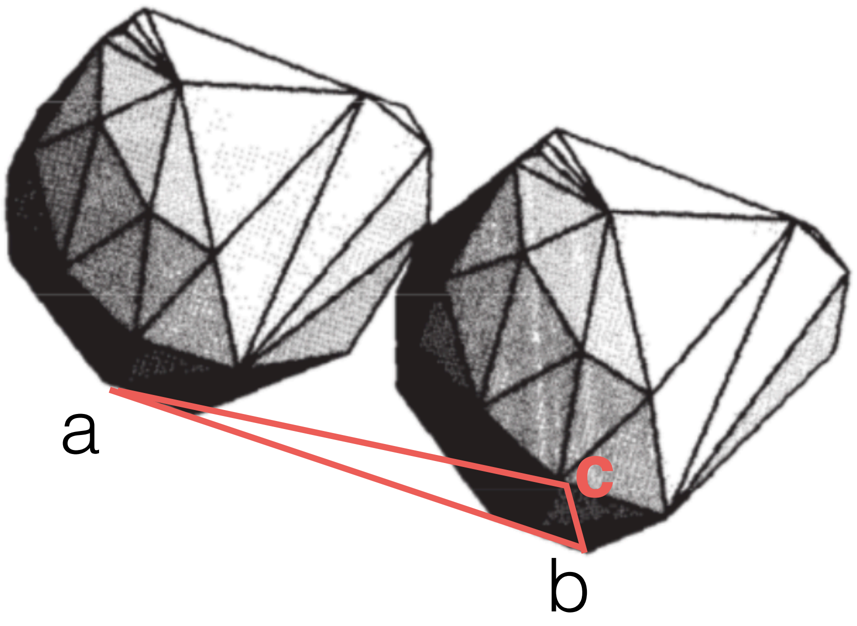
Merge

1. Find a common tangent ab



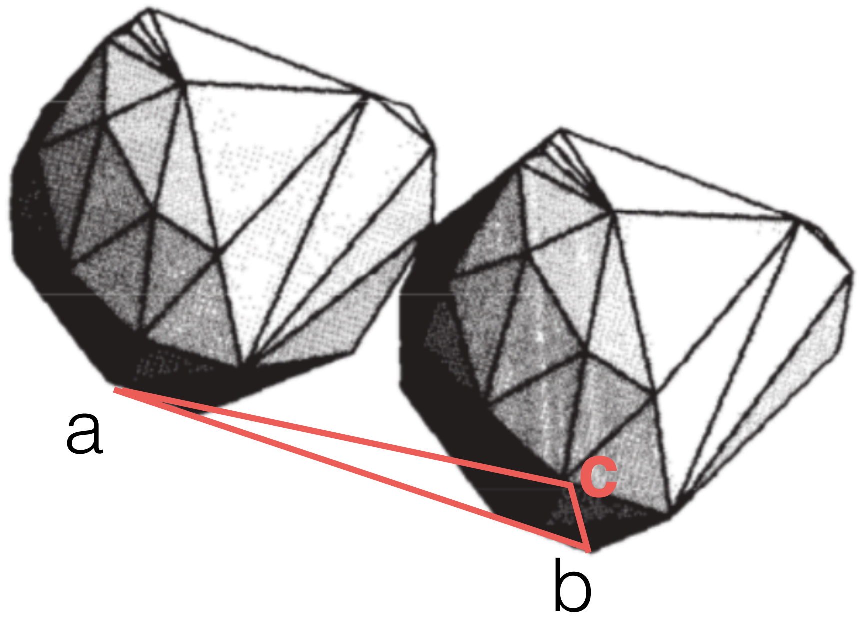
Merge

1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).



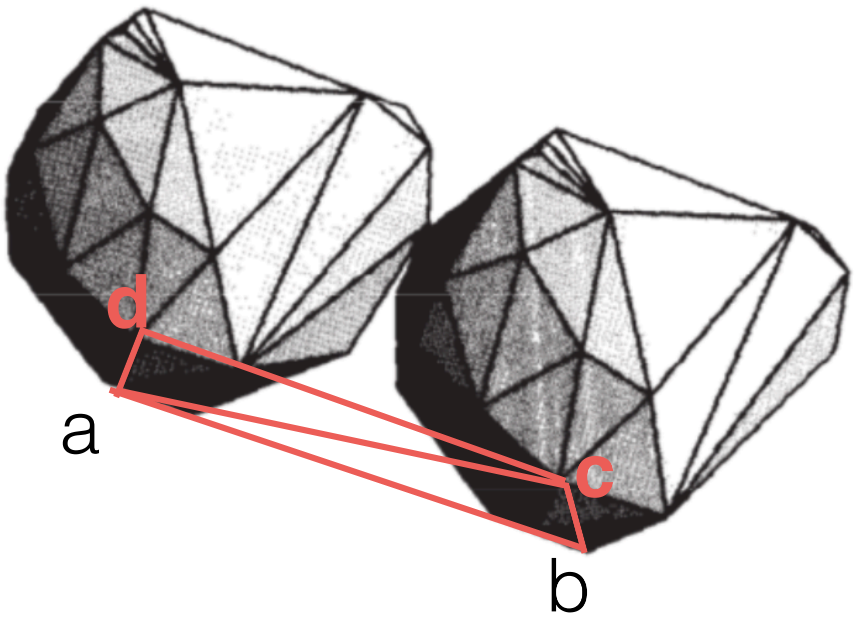
Merge

1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).
3. Repeat from edge ac.



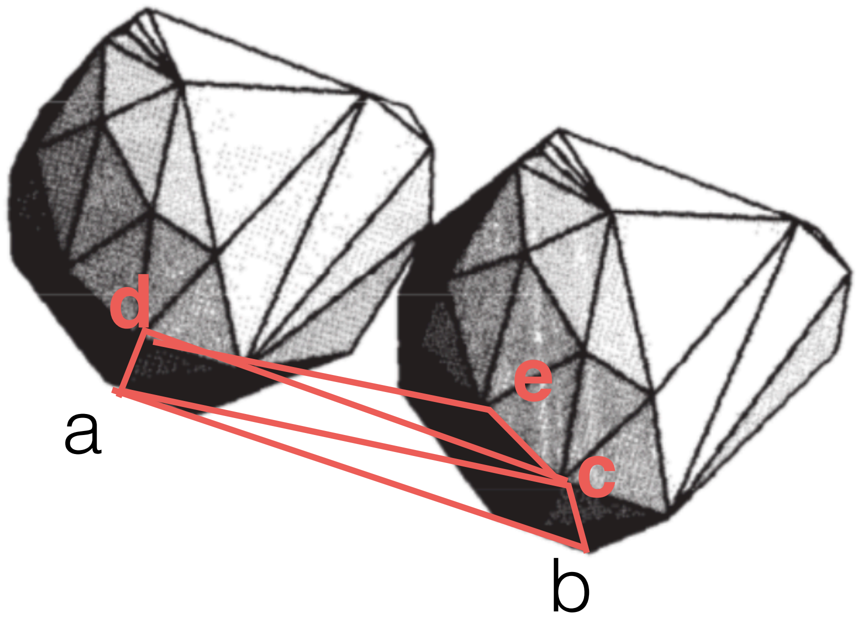
Merge

1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).
3. Repeat from edge ac.



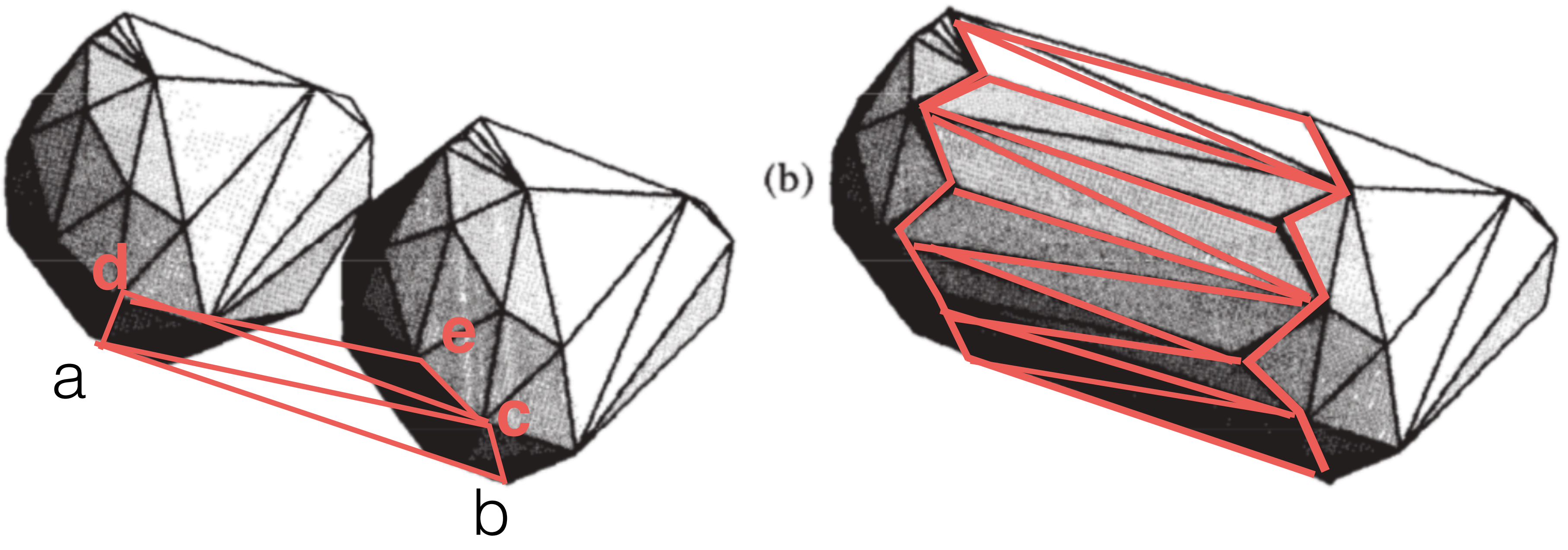
Merge

1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).
3. Repeat from edge ac.



Merge

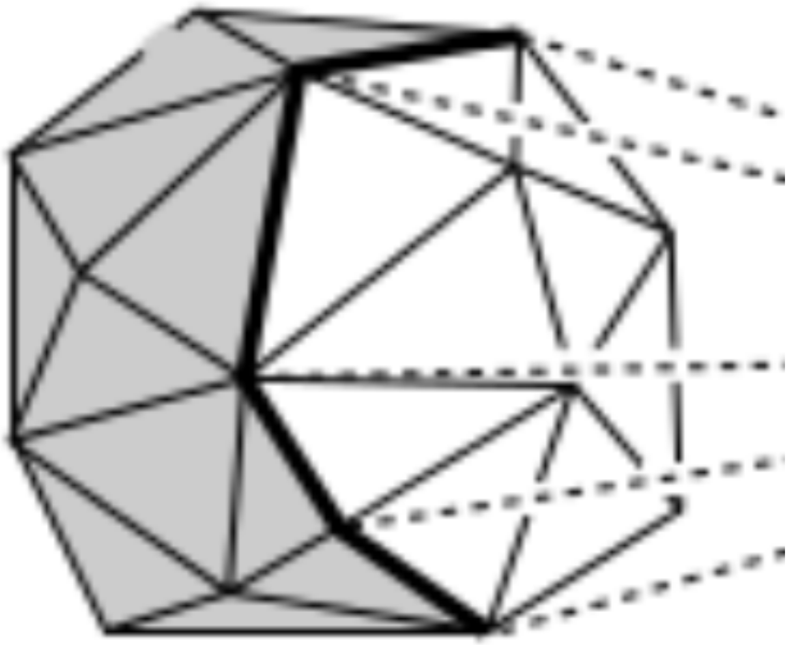
1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).
3. Repeat from edge ac.

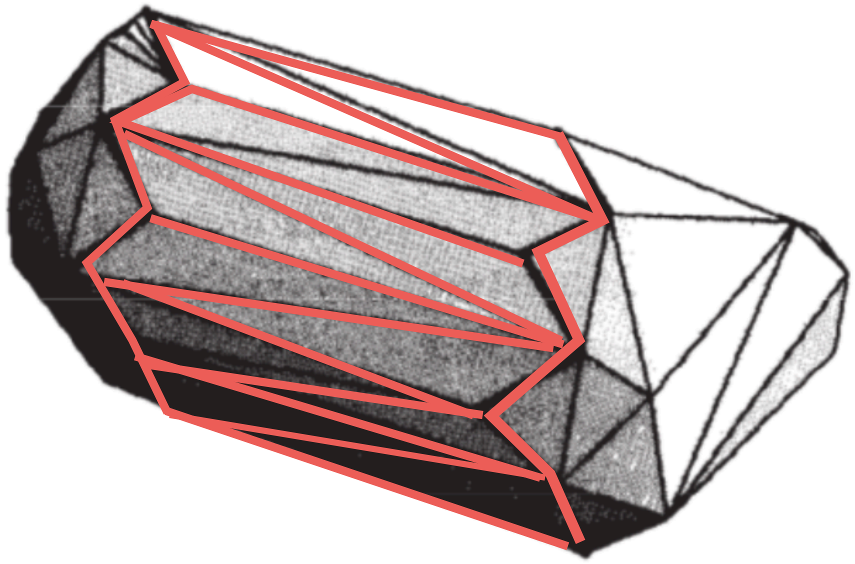


Merge

1. Find a common tangent ab
2. Consider all neighbor vertices of a,b and find the vertex with smallest angle (wrt the plane through ab).
3. Repeat from edge ac.
4. Delete hidden faces

The hidden faces





(b)

- Find the edges on the “boundary” of the cylinder
- BFS or DFS faces “towards” the cylinder
- All faces reached are inside

3d hull: summary

3D hull summary

	2D	3D
Naive	$O(n^3)$	$O(n^4)$
Gift wrapping	$O(nh)$	$O(n \times F)$
Graham scan	$O(n \lg n)$	does not extend to 3D
Quickhull	$O(n \lg n)$, $O(n^2)$	does not extend to 3D
Incremental	$O(n \lg n)$	$O(n^2)$
Divide-and-conquer	$O(n \lg n)$	$O(n \lg n)$

3d hull: Summary

- Of all algorithms that extend to 3D, divide-and-conquer is the only one that achieves optimal $O(n \lg n)$
- But, difficult to implement
- The slower algorithms (gift wrapping, incremental) preferred in practice

Convex hull in higher dimensions

- Surprisingly, have many applications
    - e.g. computing triangulations for points in 3D can be constructed from convex hulls in 4D
- Size of d-hull: $\Omega(n^{\lfloor d/2 \rfloor})$
- In 4D: size is $\Omega(n^2)$
    - $O(n \lg n)$ algorithm not possible
    - $O(n^2)$ algorithms known