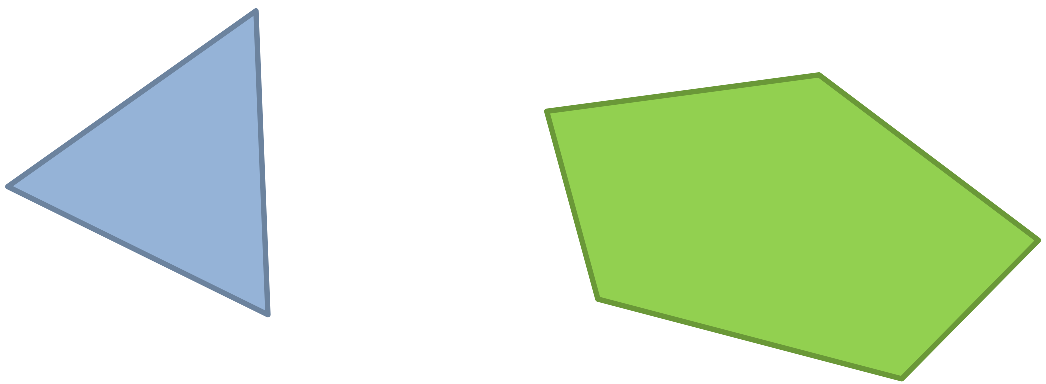
Computing Distance

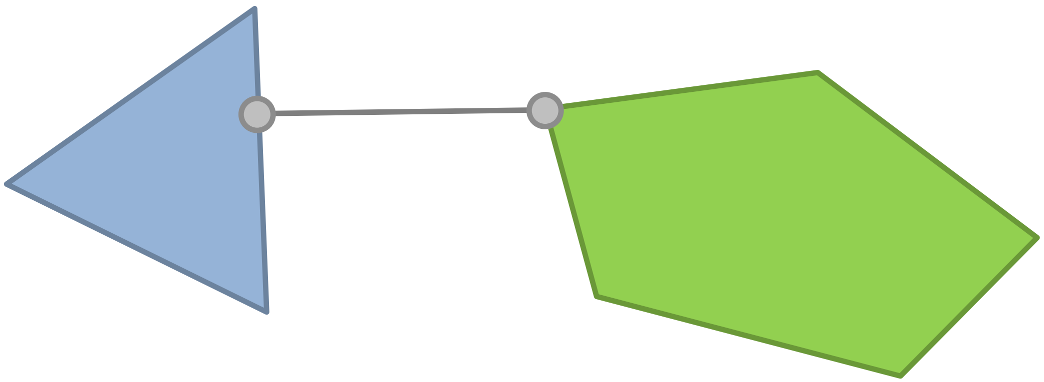
Erin Catto

Blizzard Entertainment

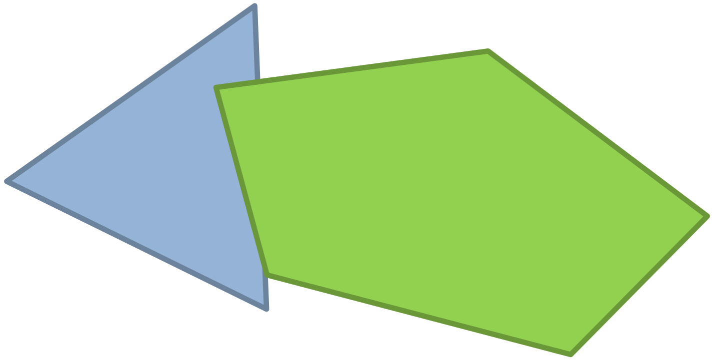
Convex polygons



Closest points



Overlap



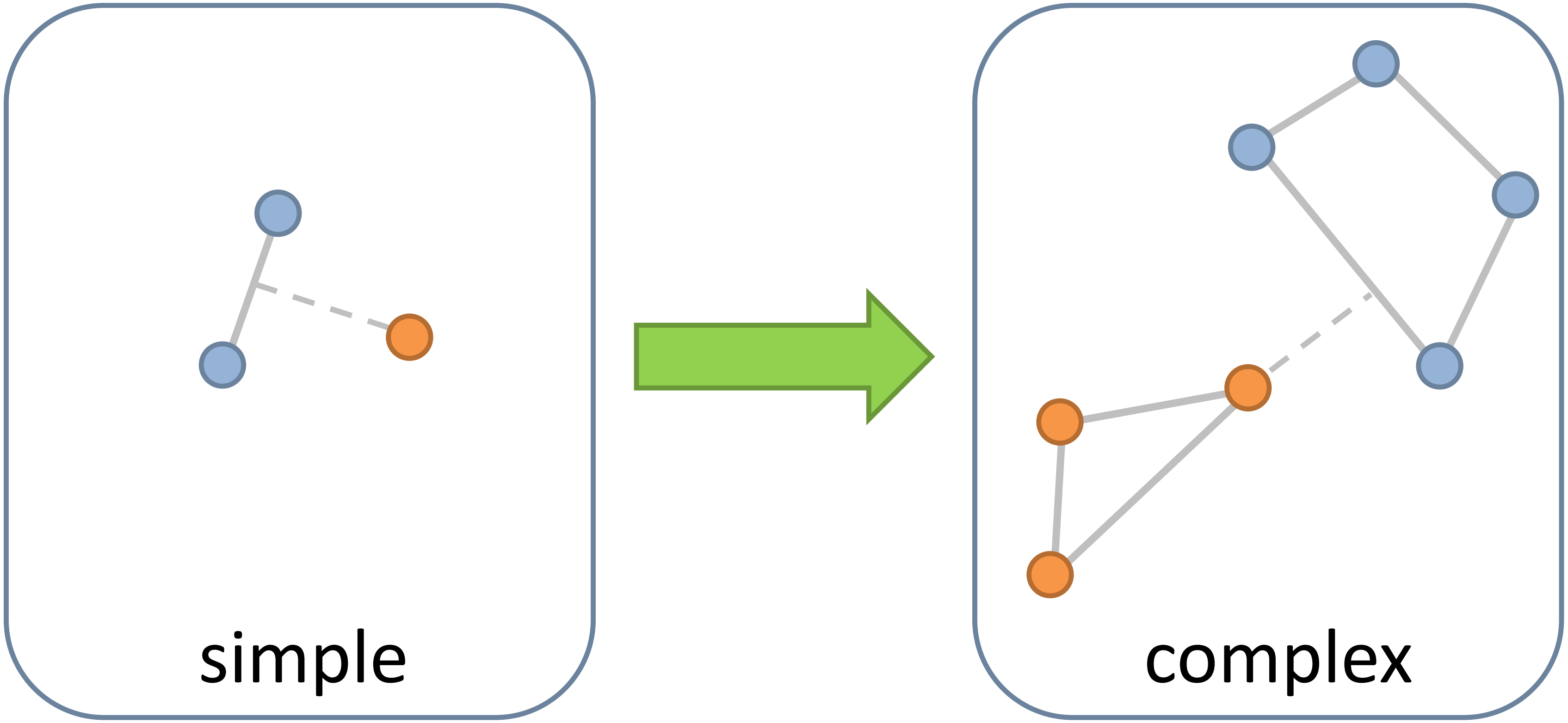
Goal

- Compute the distance between convex polygons

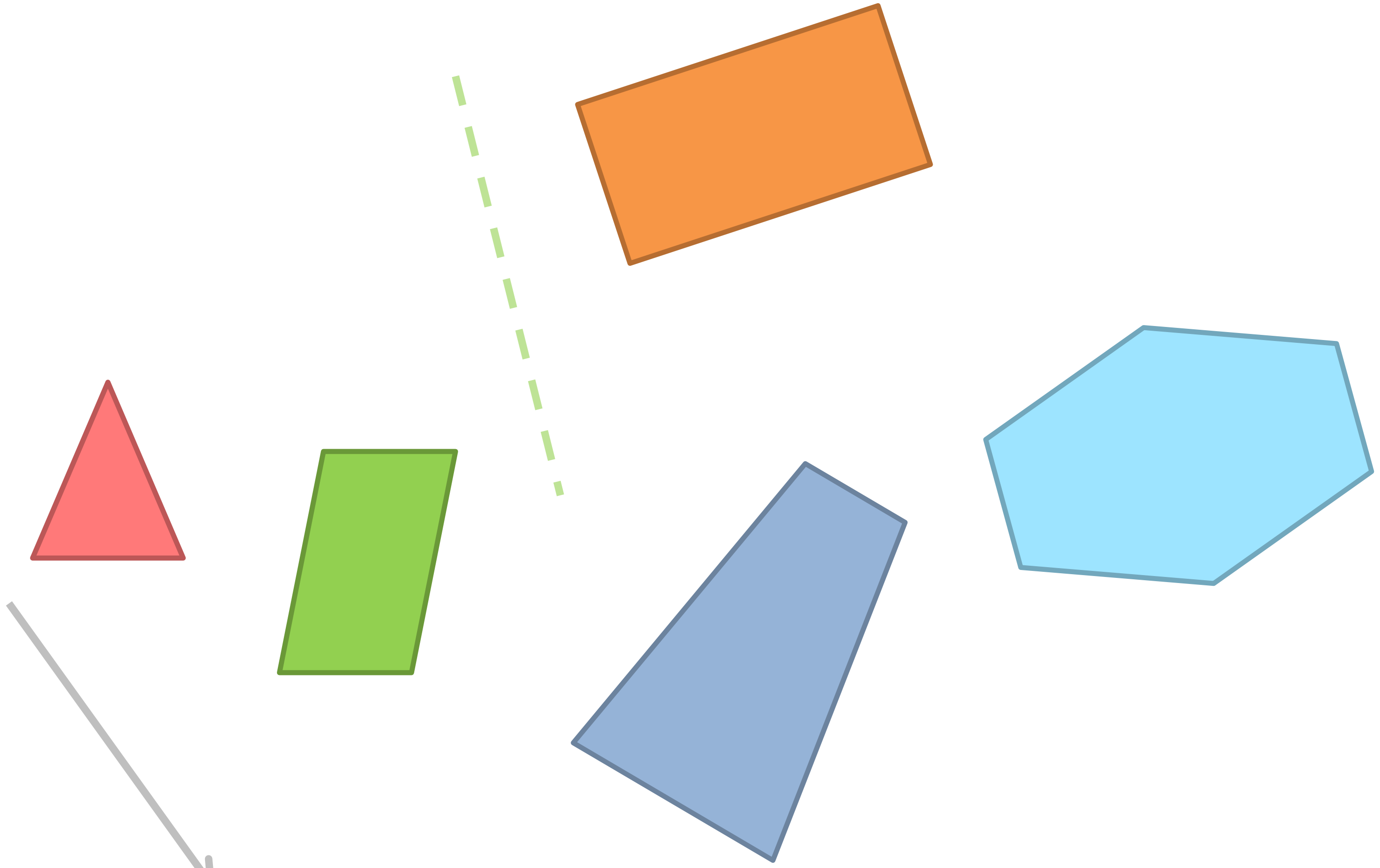
Keep in mind

- 2D
- Code not optimized

Approach



Geometry



If all else fails …

```
// Input for the distance function.
truct Input

    Polygon polygon1;
    Polygon polygon2;
    Transform transform1;
    Transform transform2;
;

// Output for the distance function.
truct Output

    enum
    {
        e_maxSimplices = 20
    };

    Vec2 point1;        ///< closest point on polygon 1
    Vec2 point2;        ///< closest point on polygon 2
    float distance;
    int iterations;     ///< number of GJK iterations used

    Simplex simplices[e_maxSimplices];
    int simplexCount;
;

// Get the closest points between two point clouds.
oid Distance2D(Output* output, const Input& input);
```

DEMO!

Outline

1. Point to line segment
2. Point to triangle
3. Point to convex polygon
4. Convex polygon to convex polygon

Concepts

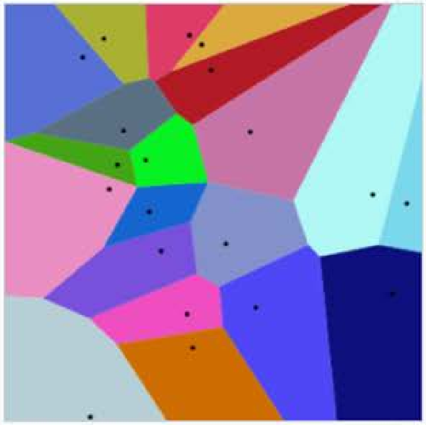
1. Voronoi regions
2. Barycentric coordinates
3. GJK distance algorithm
4. Minkowski difference

Voronoi diagram

From Wikipedia, the free encyclopedia

In mathematics, a **Voronoi diagram** is a partition of a plane into regions close to each of a given set of objects. In the simplest case, these objects are just finitely many points in the plane (called seeds, sites, or generators). For each seed there is a corresponding region, called **Voronoi cells**, consisting of all points of the plane closer to that seed than to any other. The Voronoi diagram of a set of points is dual to its Delaunay triangulation.

20 points and their Voronoi cells (larger version below)



20 points and their Voronoi cells (larger version below)

Formal definition [edit]

Let X be a metric space with distance function d. Let K be a set of indices and let $(P_k)_{k \in K}$ be a tuple (ordered collection) of nonempty subsets (the sites) in the space X. The Voronoi cell, or Voronoi region, R_k, associated with the site P_k is the set of all points in X whose distance to P_k is not greater than their distance to the other sites P_j, where j is any index different from k. In other words, if $d(x,\, A) = \inf\{d(x,\, a) \mid a \in A\}$ denotes the distance between the point x and the subset A, then

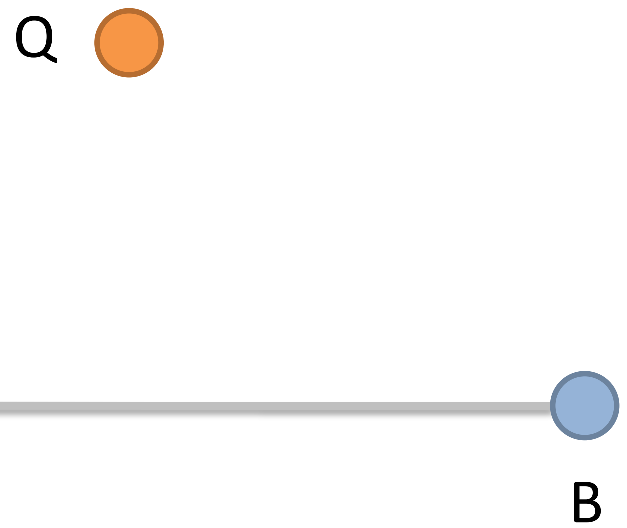
$$R_k = \{x \in X \mid d(x, P_k) \leq d(x, P_j) \text{ for all } j \neq k\}$$

Section 1

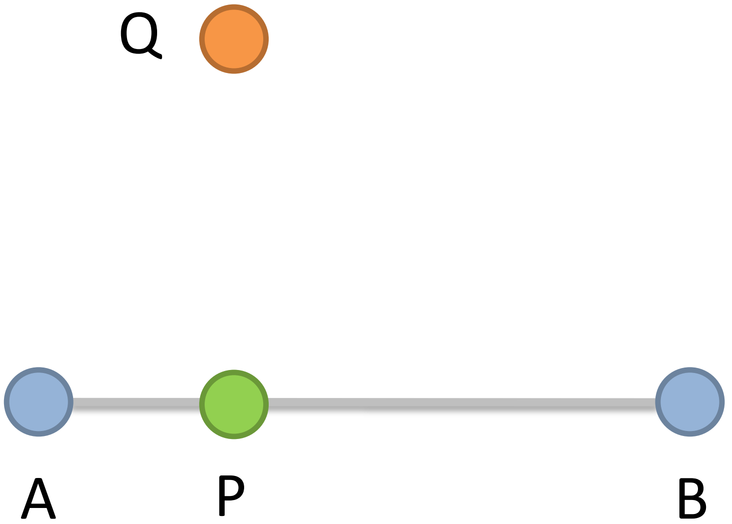
Point to Line Segment

A line segment

Query point

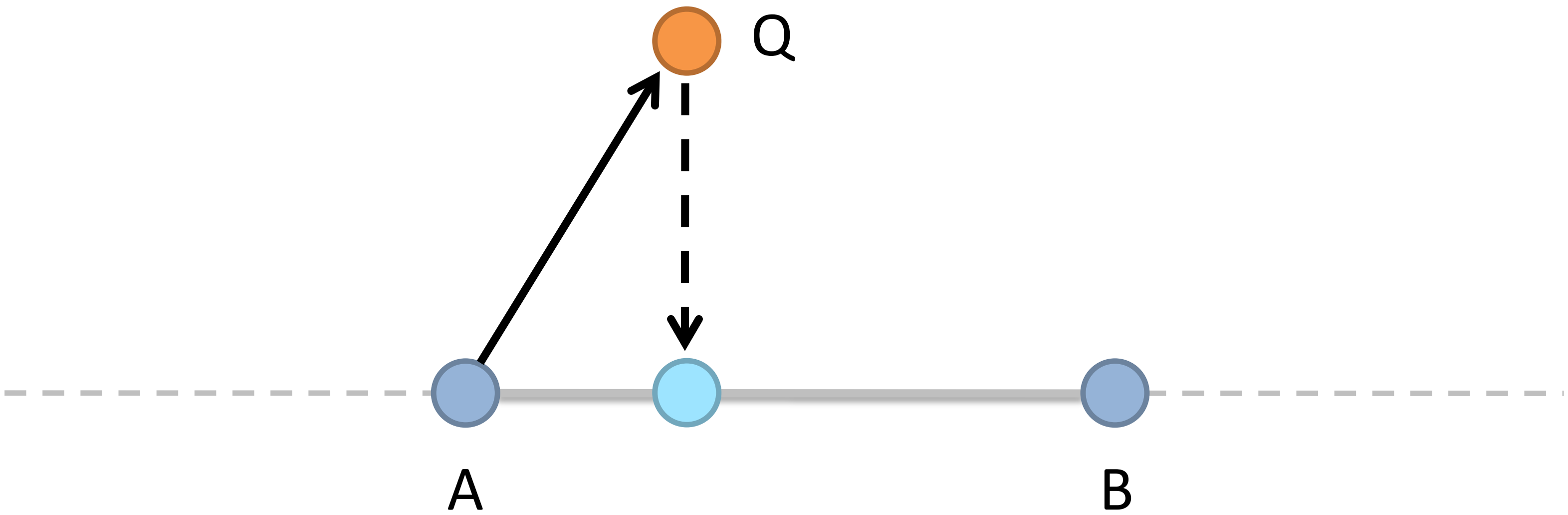


Closest point

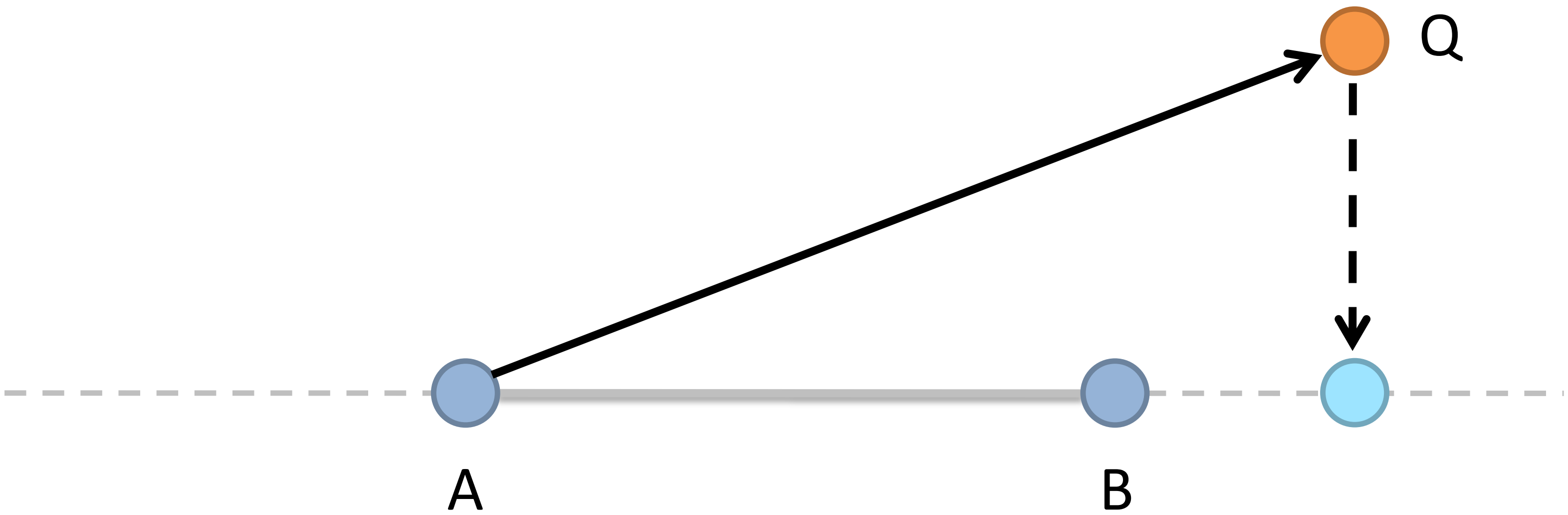


Projection: region A

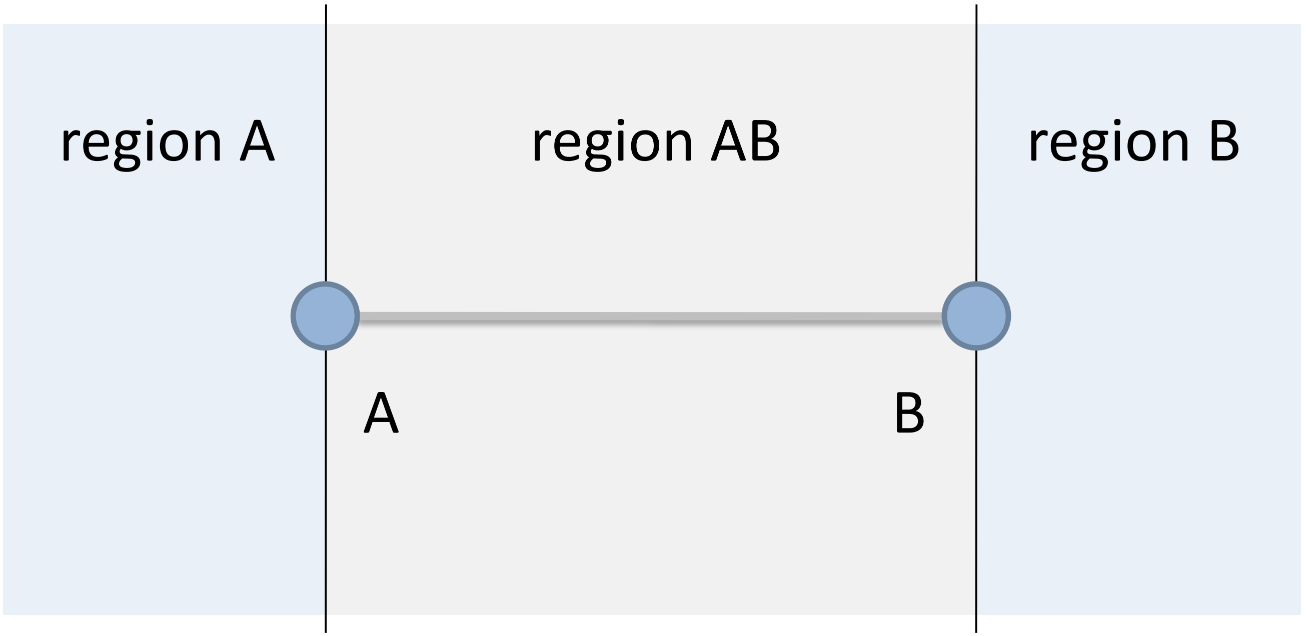
Projection: region AB



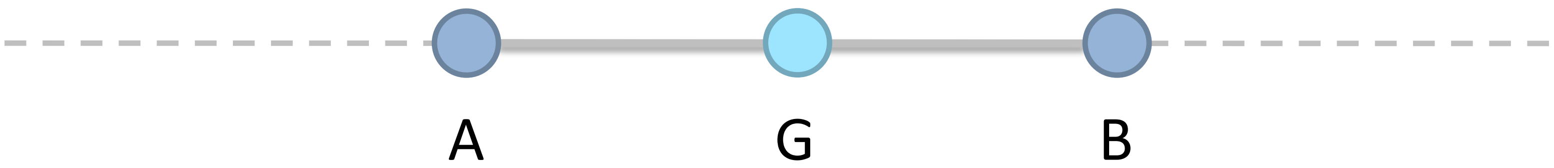
Projection: region B



Voronoi regions



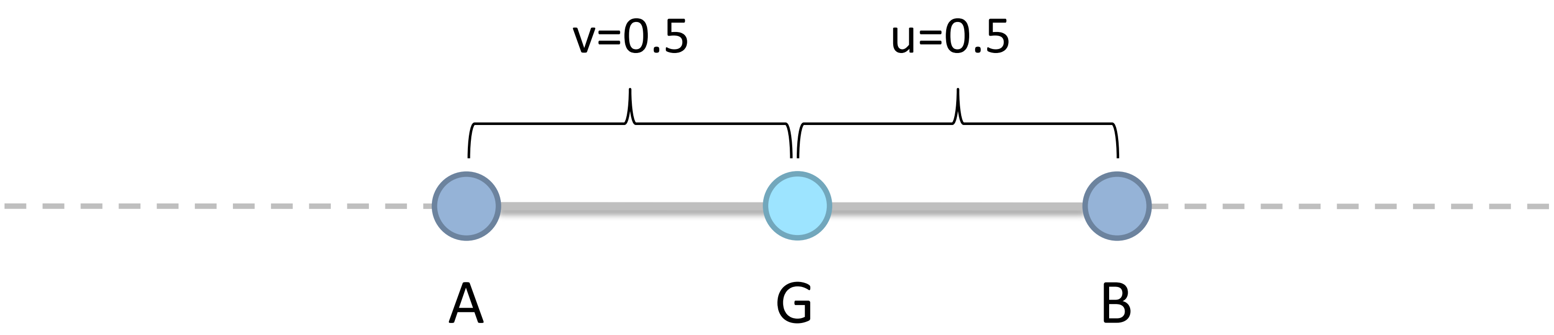
Barycentric coordinates



$$G(u,v)=uA+vB$$

$$u+v=1$$

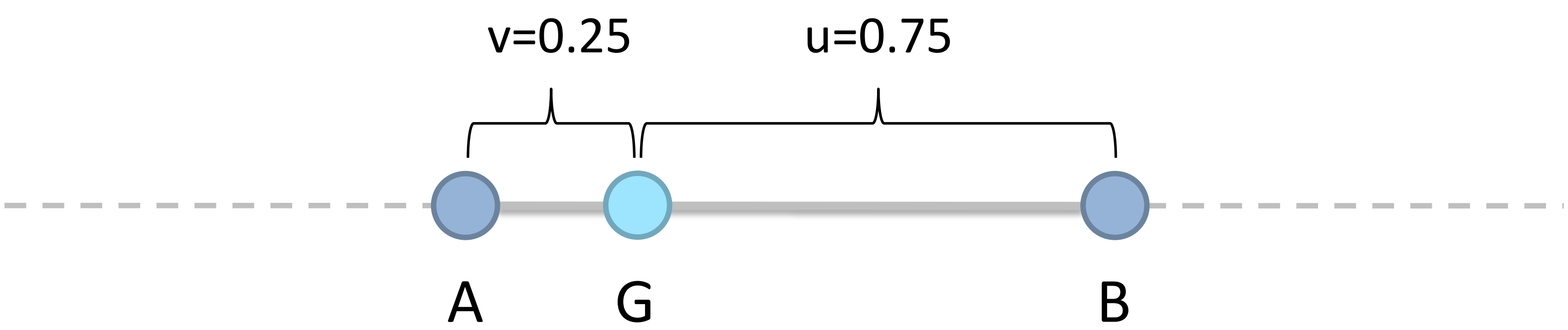
Fractional lengths



$$G(u,v)=uA+vB$$

$$u+v=1$$

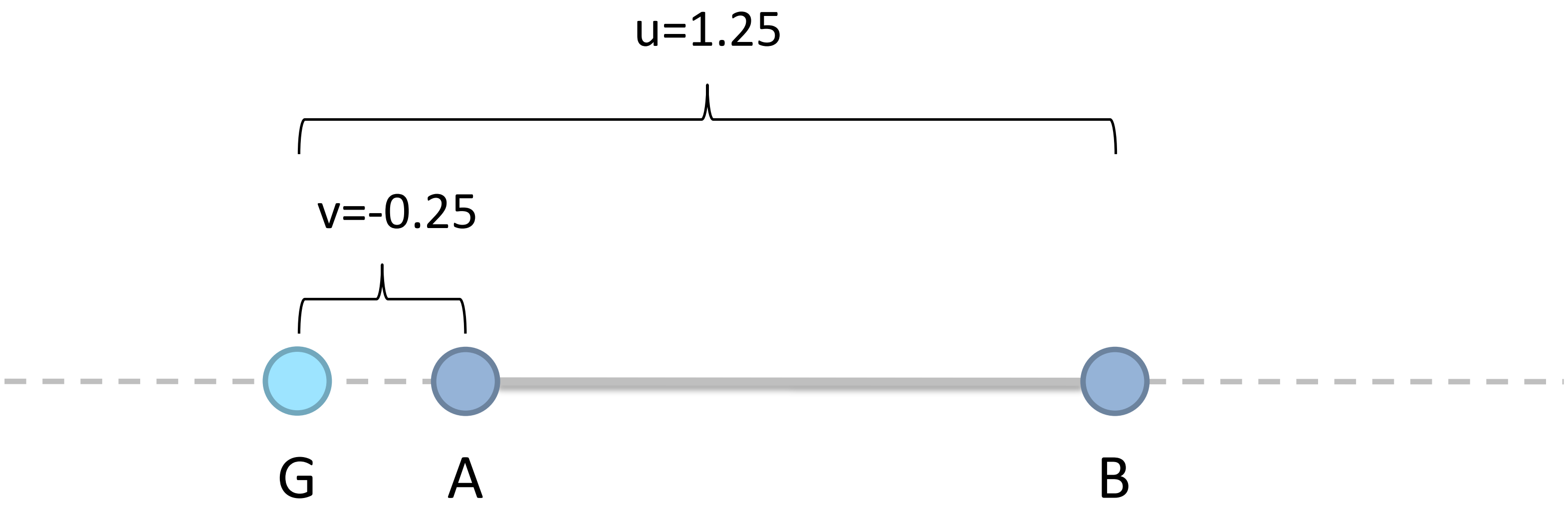
Fractional lengths



G(u,v)=uA+vB

u+v=1

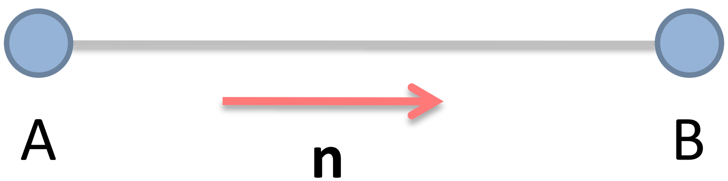
Fractional lengths



G(u,v)=uA+vB

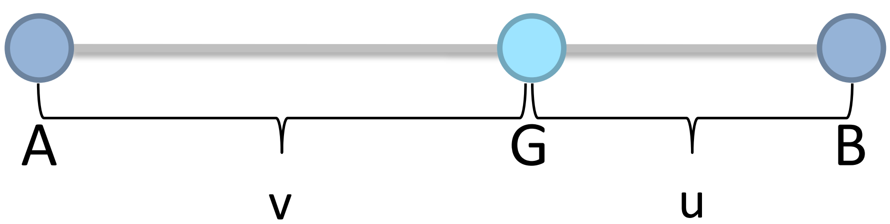
u+v=1

Unit vector



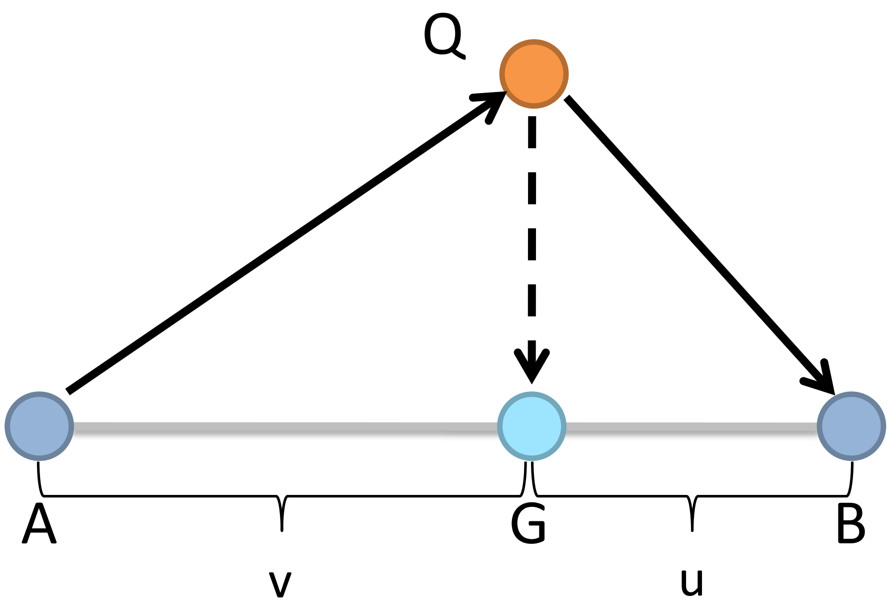
$$\mathbf{n} = \frac{B-A}{\left\| B-A \right\|}$$

(u,v) from G



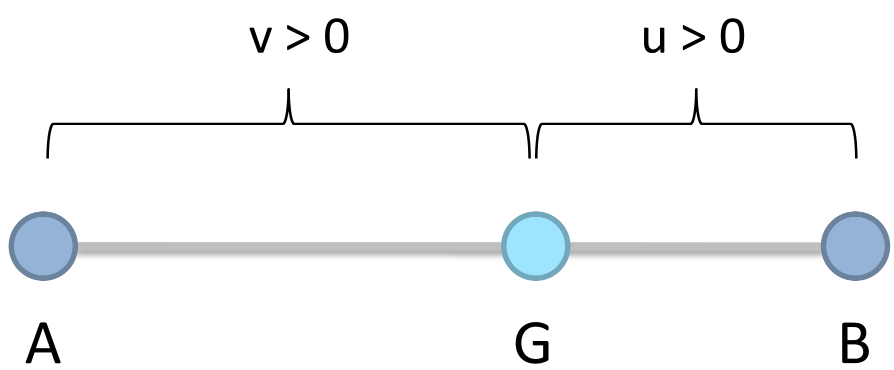
$$v = \frac{(G-A)\cdot n}{\|B-A\|} \qquad u = \frac{(B-G)\cdot n}{\|B-A\|}$$

(u,v) from Q



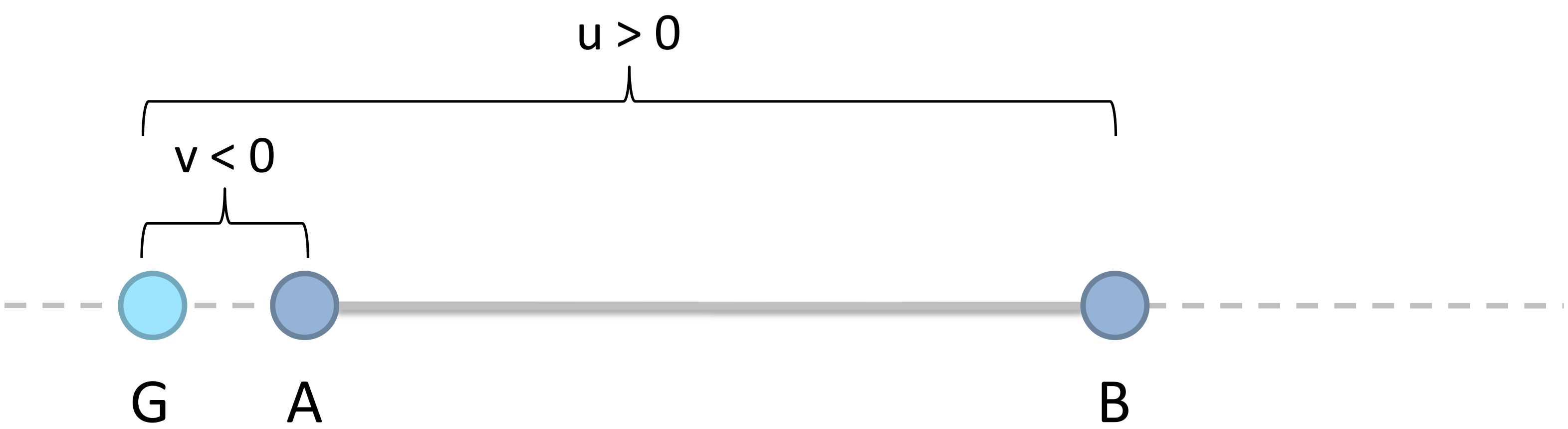
$$v = \frac{(Q-A) \cdot n}{\|B-A\|} \qquad u = \frac{(B-Q) \cdot n}{\|B-A\|}$$

Voronoi region from (u,v)



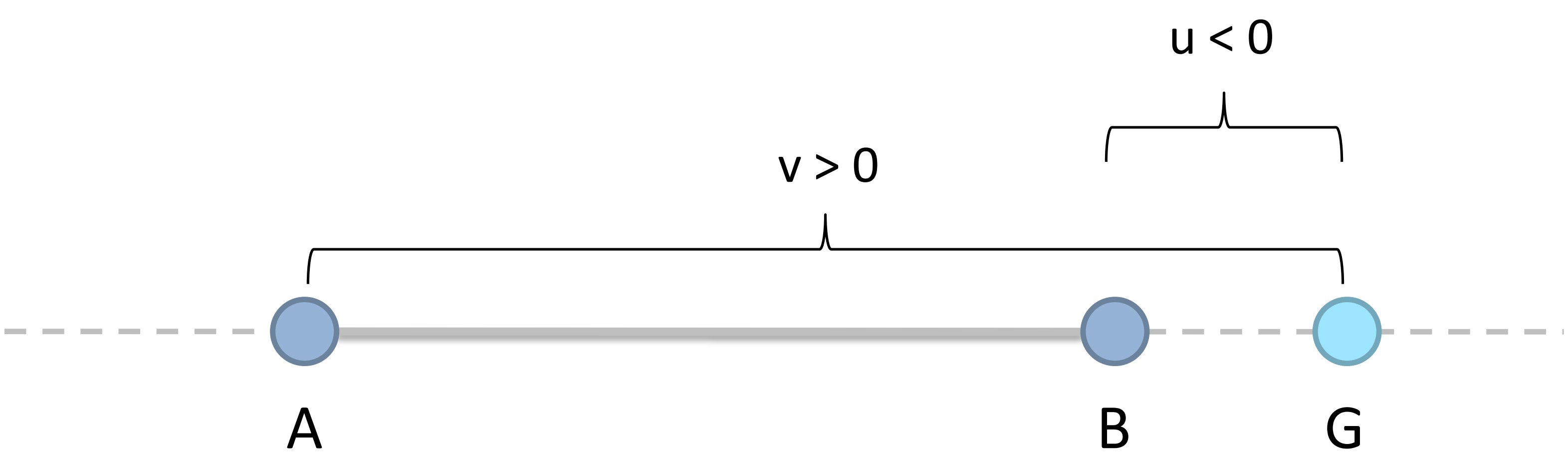
u > 0 and v > 0 $\Rightarrow$ region AB

Voronoi region from (u,v)



v <= 0 ⇨ region A

Voronoi region from (u,v)



u <= 0 ⇒ region B

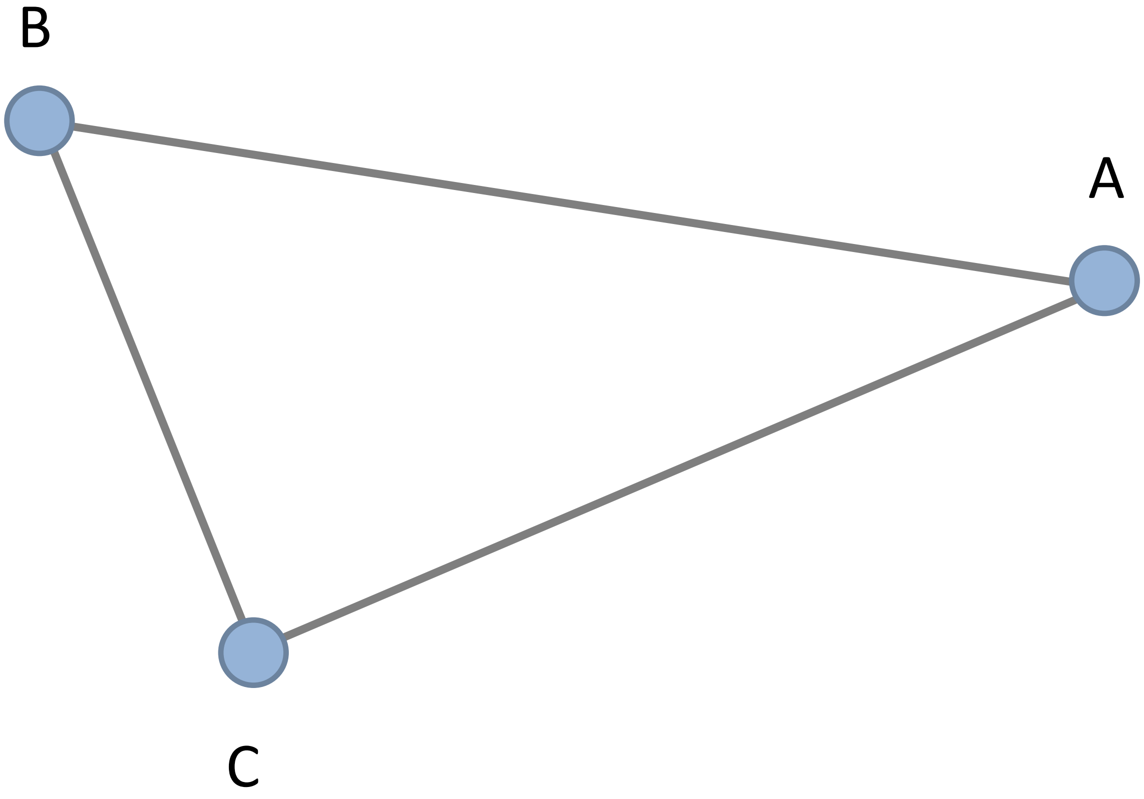
Closet point algorithm

```
input: A, B, Q
compute u and v

if (u <= 0)
  P = B
else if (v <= 0)
  P = A
else
  P = u*A + v*B
```

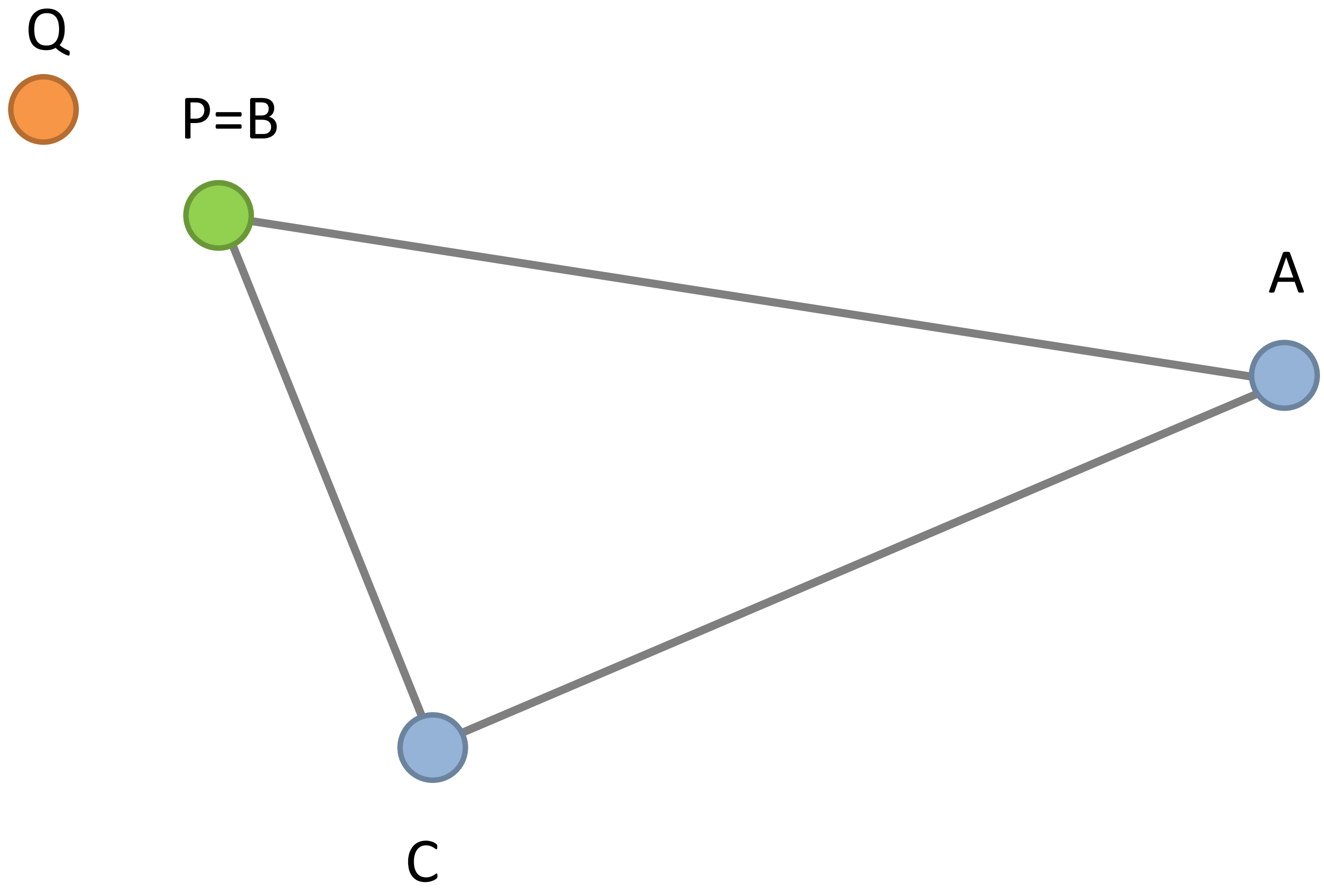
Section 2

Point to Triangle

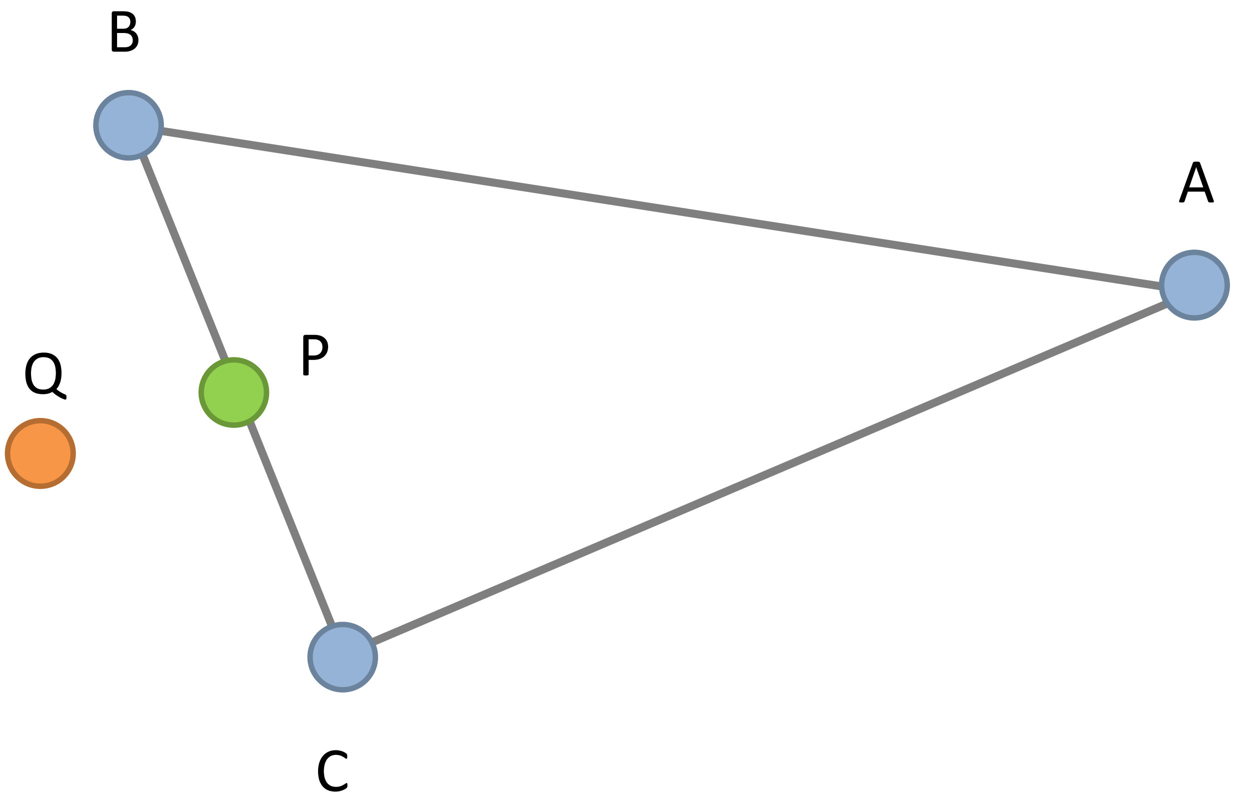
Triangle



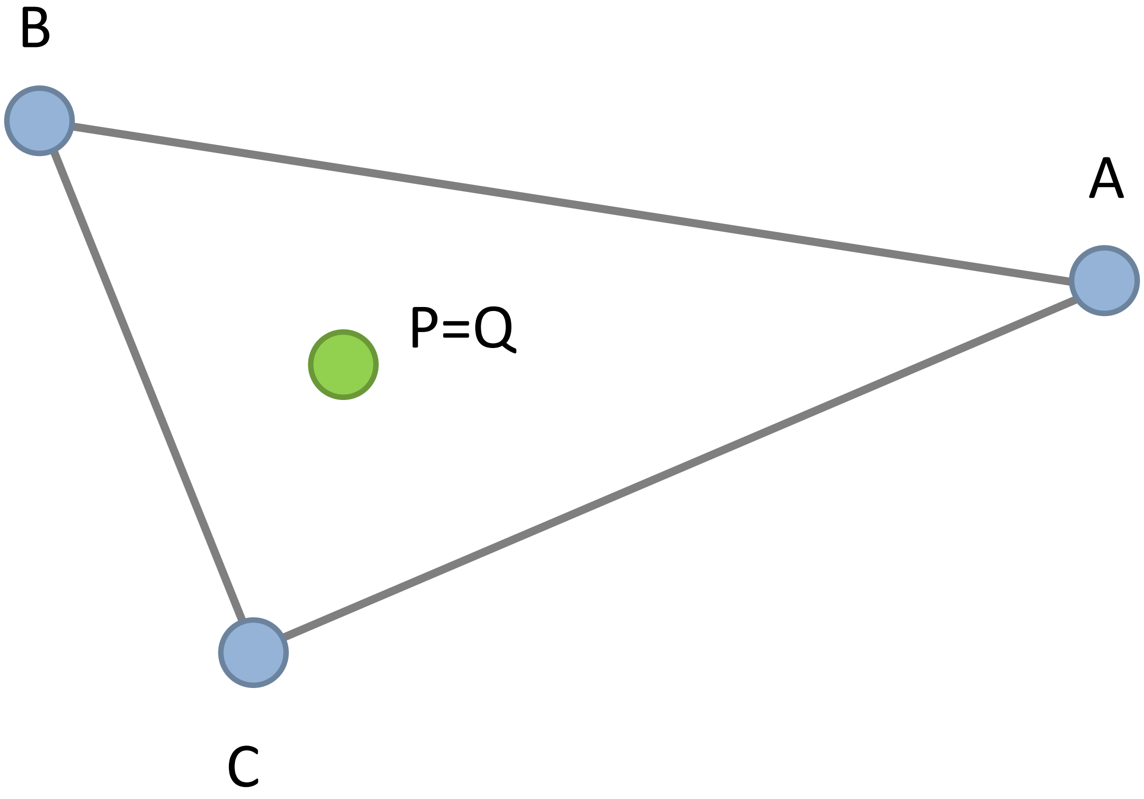
Closest feature: vertex



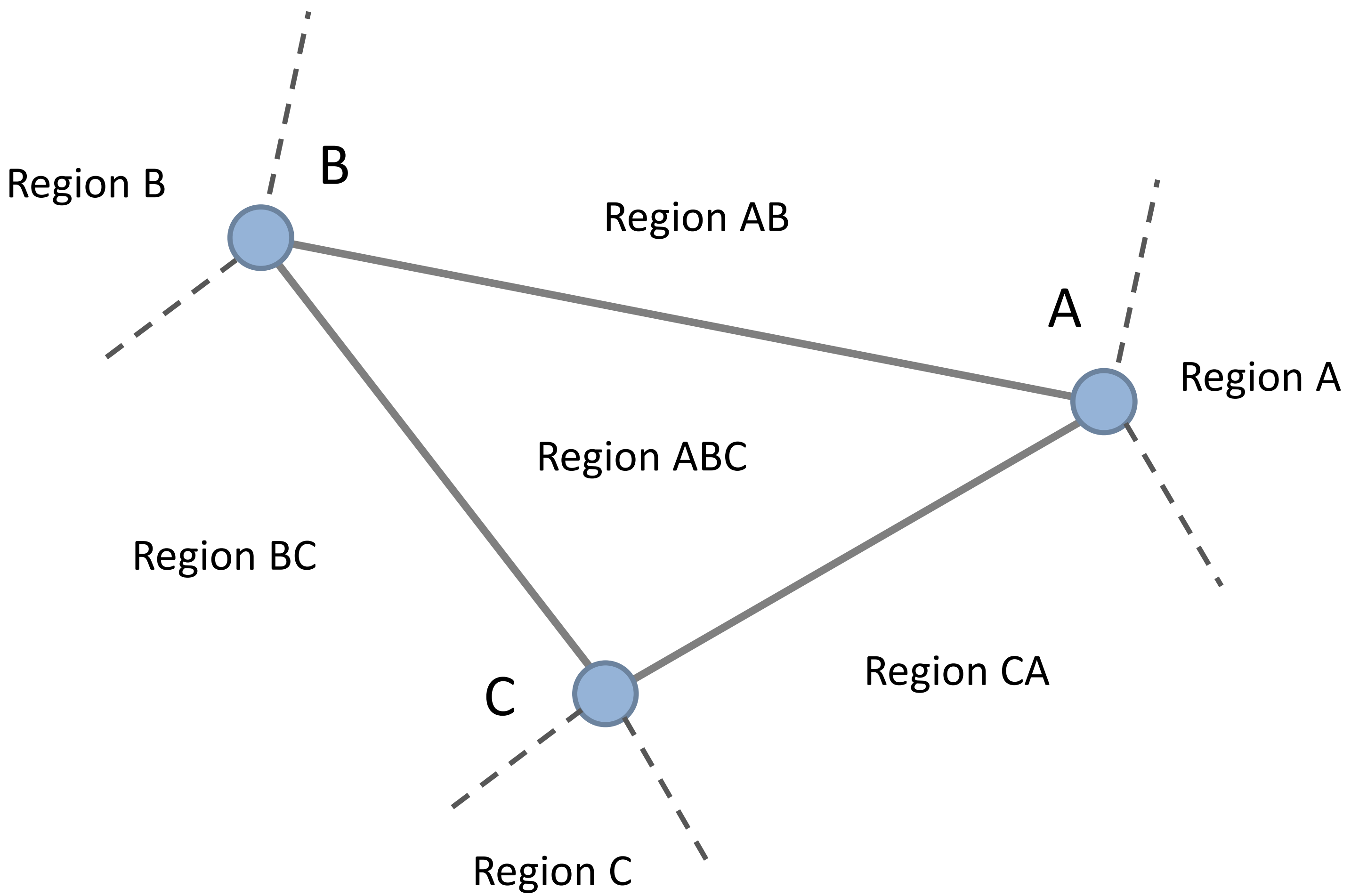
Closest feature: edge



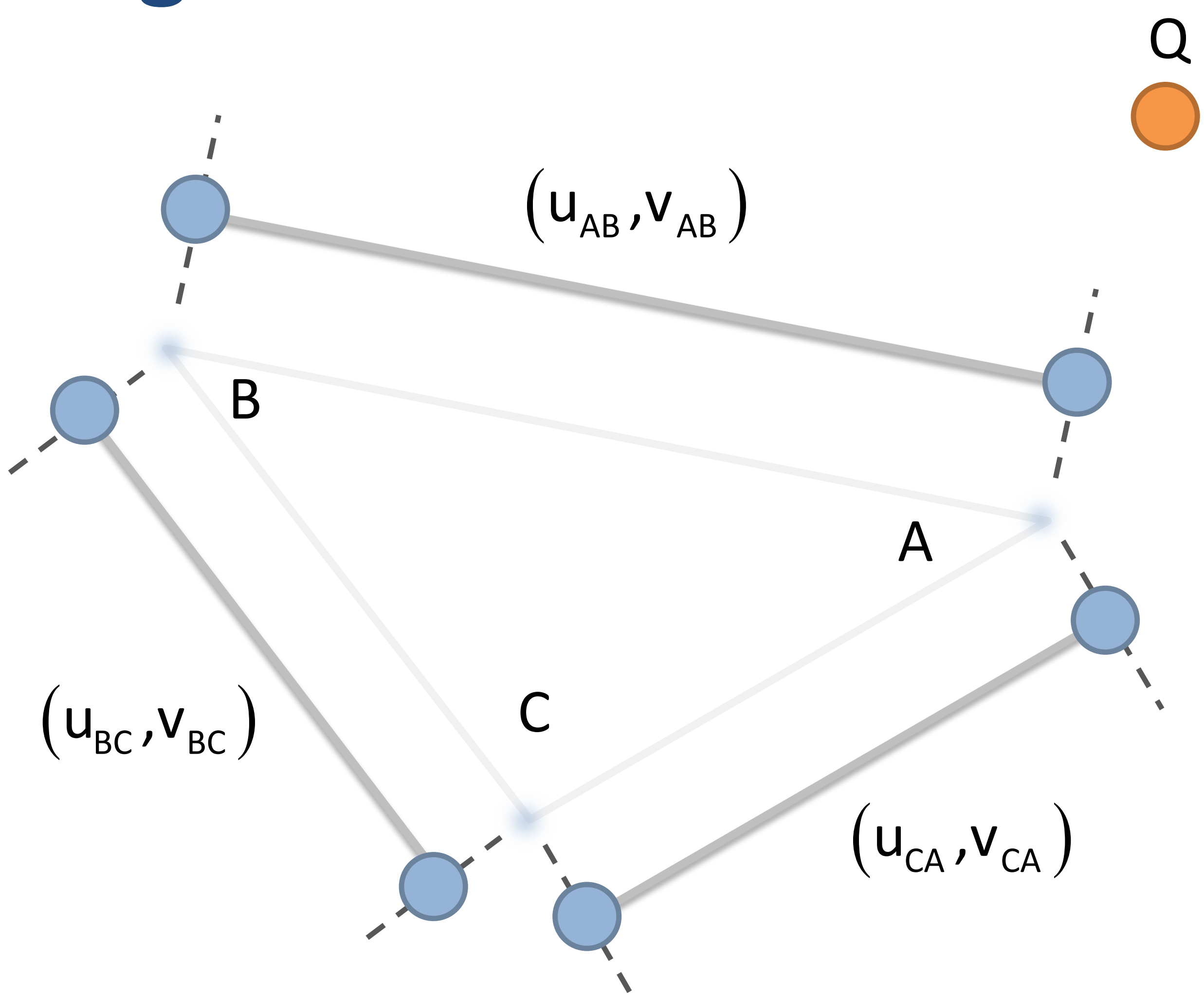
Closest feature: interior



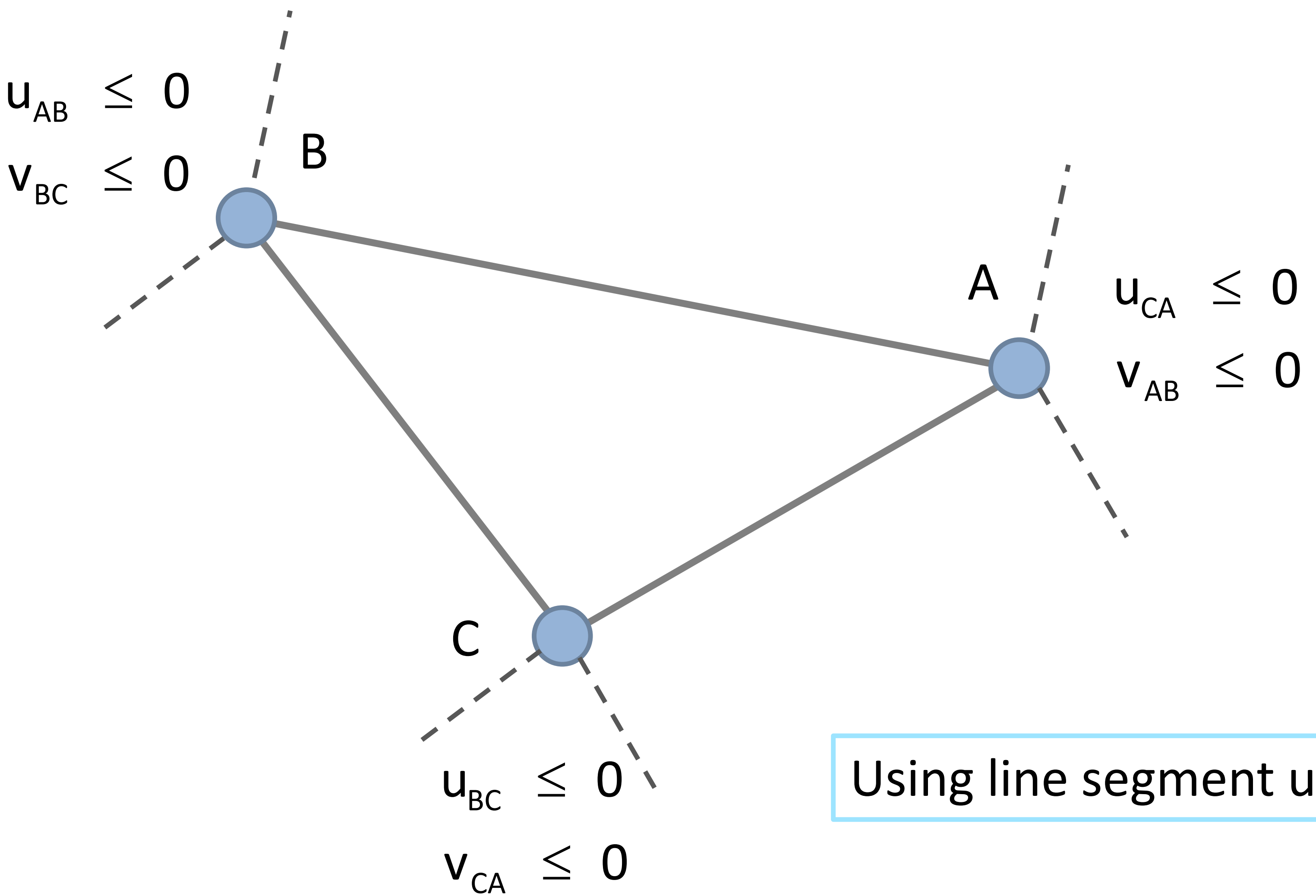
Voronoi regions



3 line segments

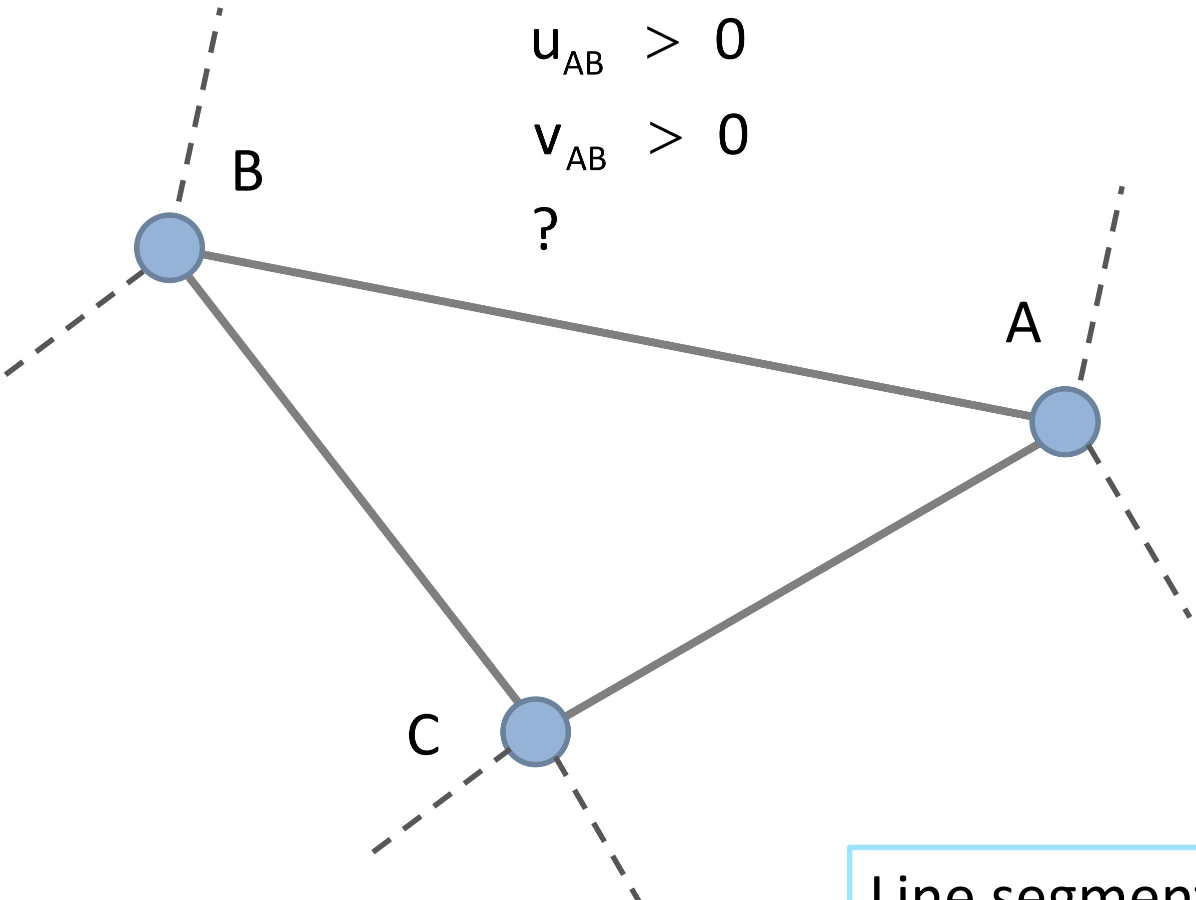


Vertex regions



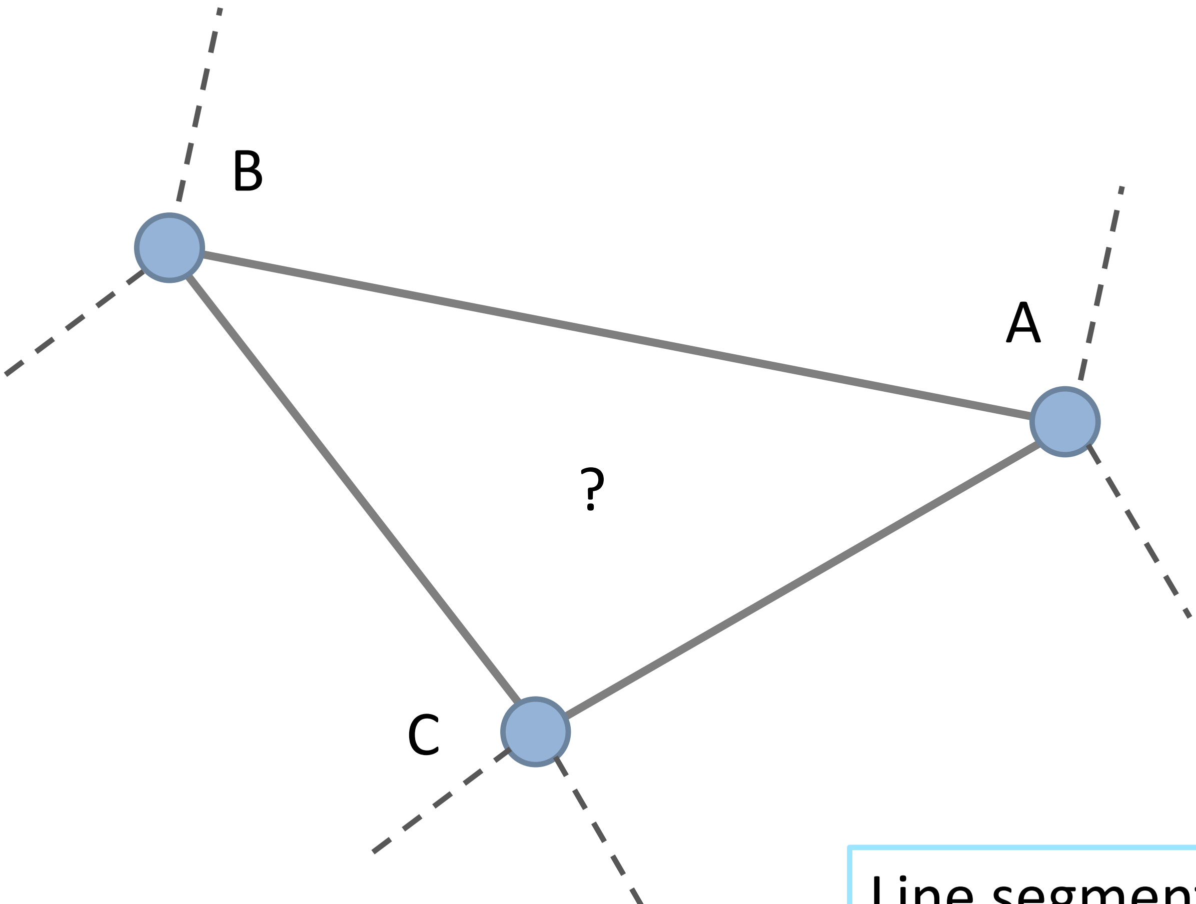
Using line segment uv's

Edge regions



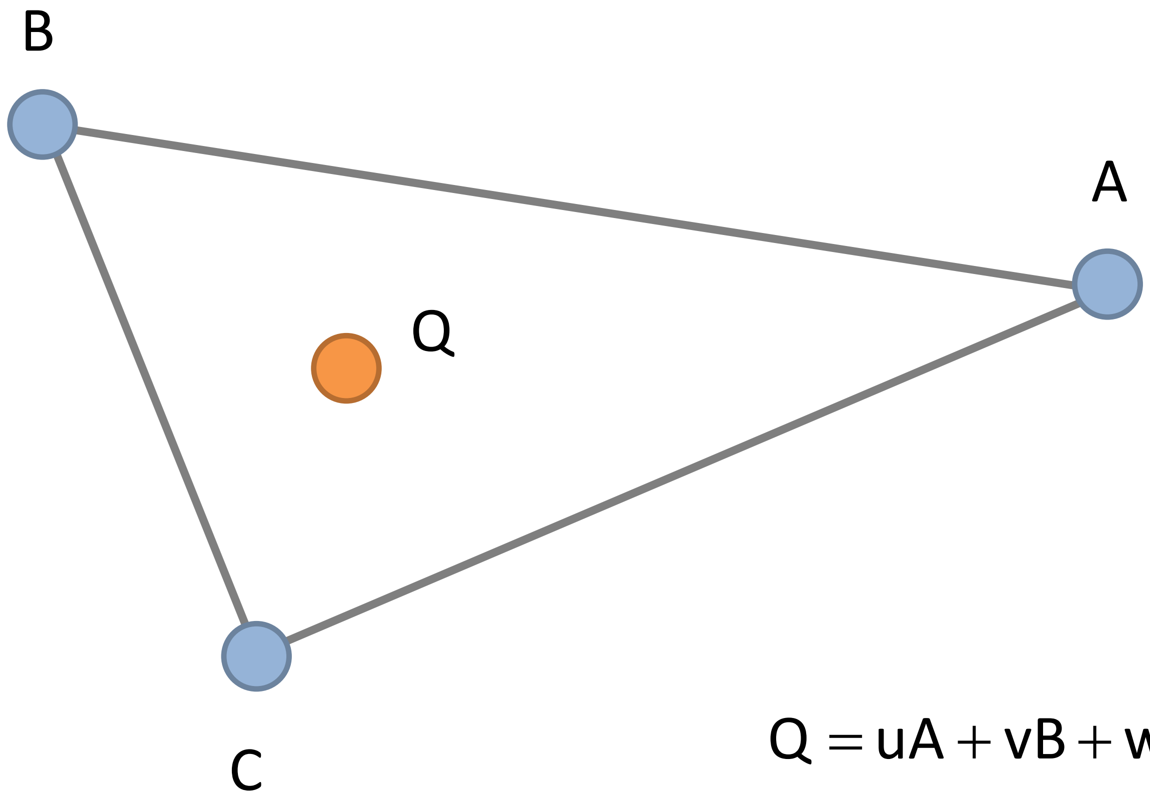
Line segment uv's are not sufficient

Interior region



Line segment uv's don't help at all

Triangle barycentric coordinates



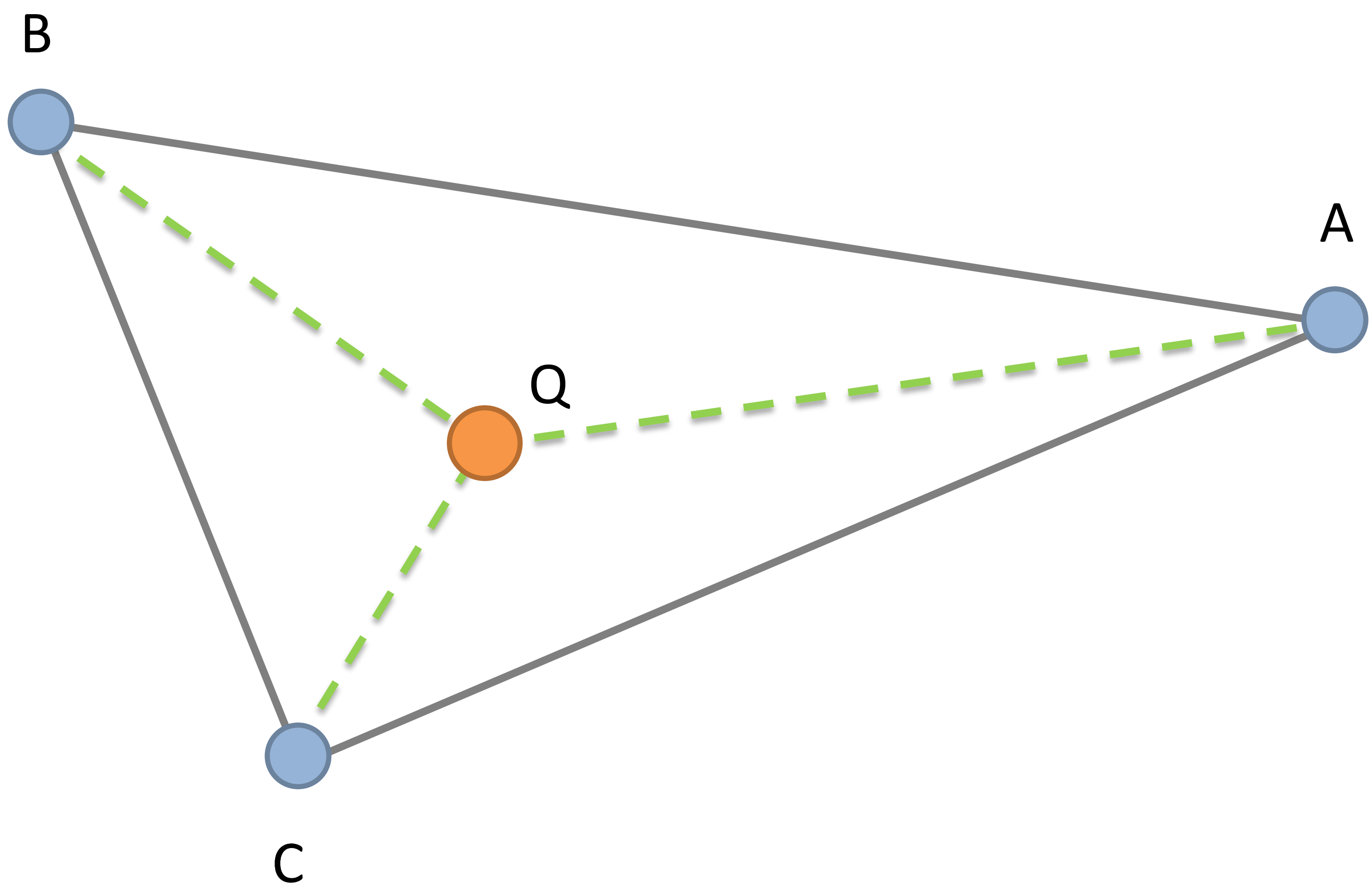
$$Q = uA + vB + wC$$

$$u + v + w = 1$$

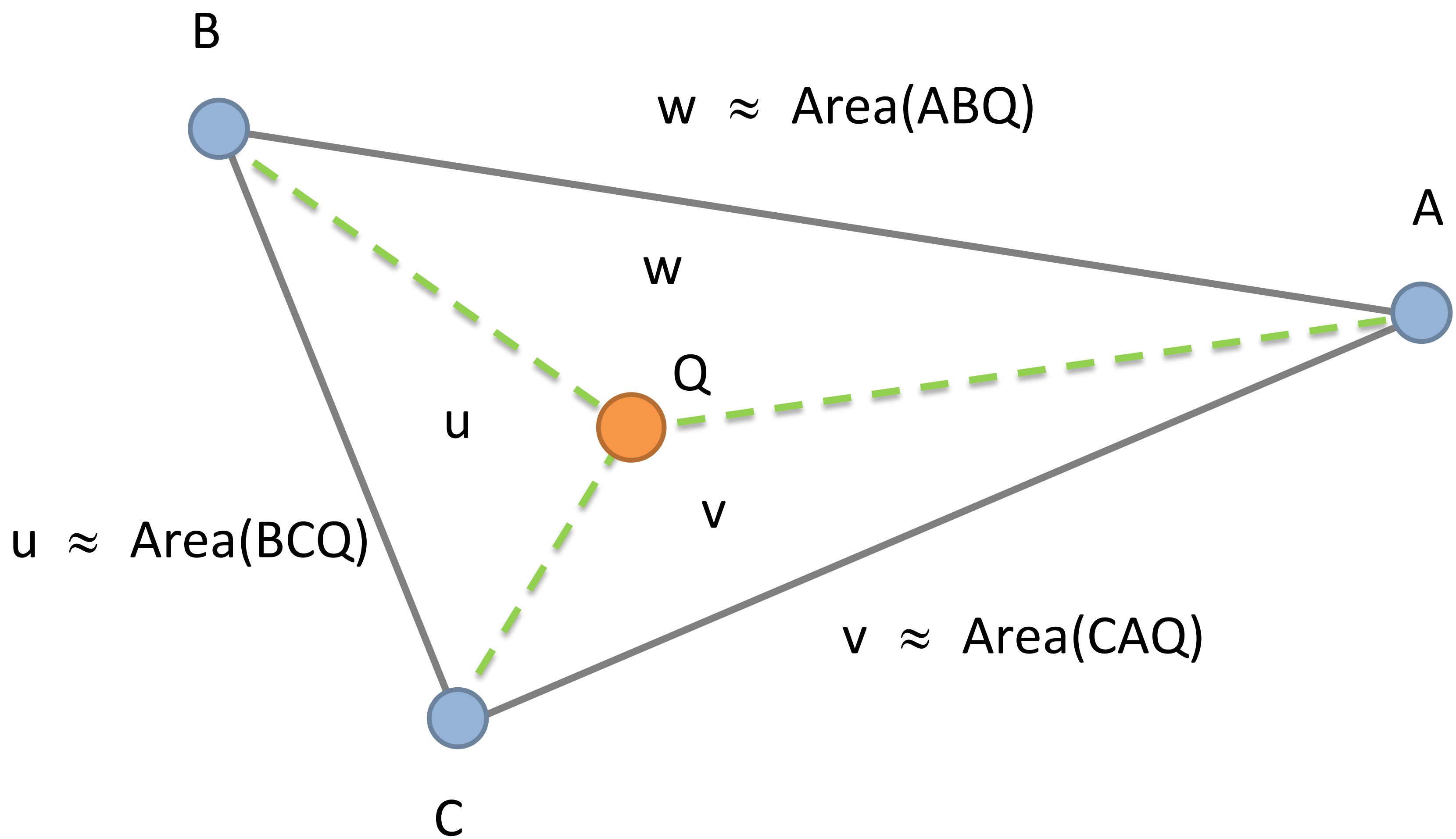
Linear algebra solution

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix}$$

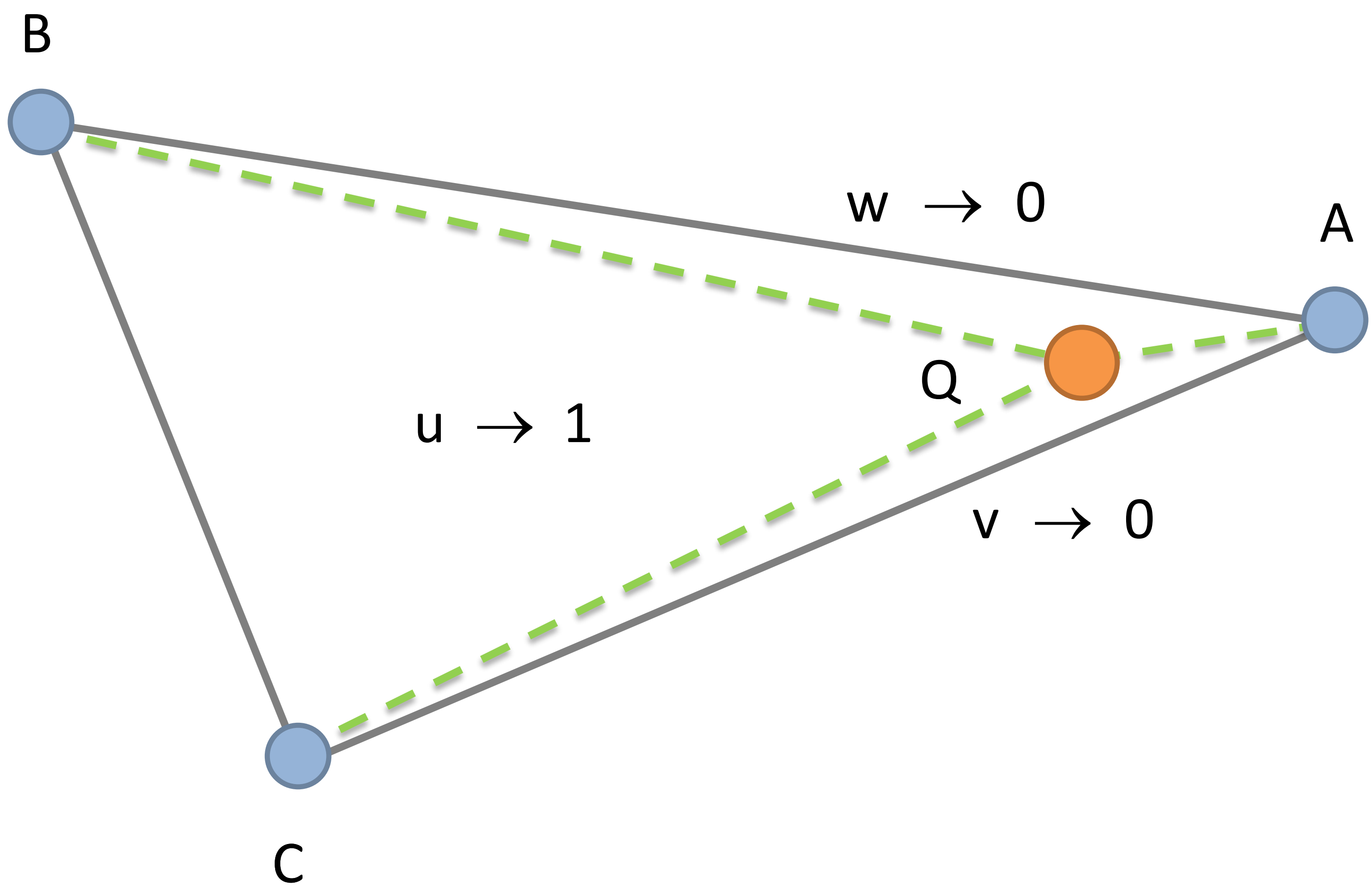
Fractional areas



The barycenctric coordinates are the fractional areas



Barycentric coordinates



Barycentric coordinates

$$u = \frac{\text{area}(QBC)}{\text{area}(ABC)}$$

$$v = \frac{\text{area}(QCA)}{\text{area}(ABC)}$$

$$w = \frac{\text{area}(QAB)}{\text{area}(ABC)}$$

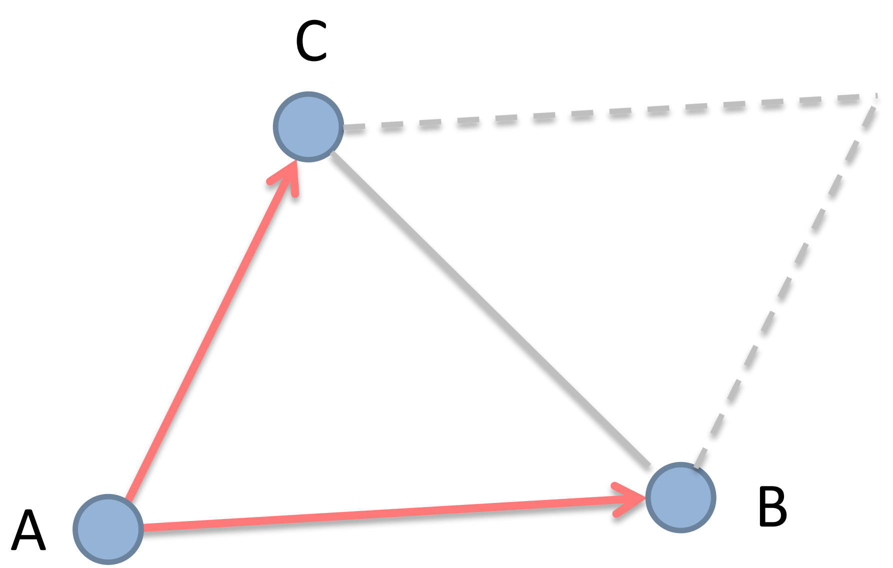
Barycentric coordinates are fractional

line segment : fractional length

triangles : fractional area

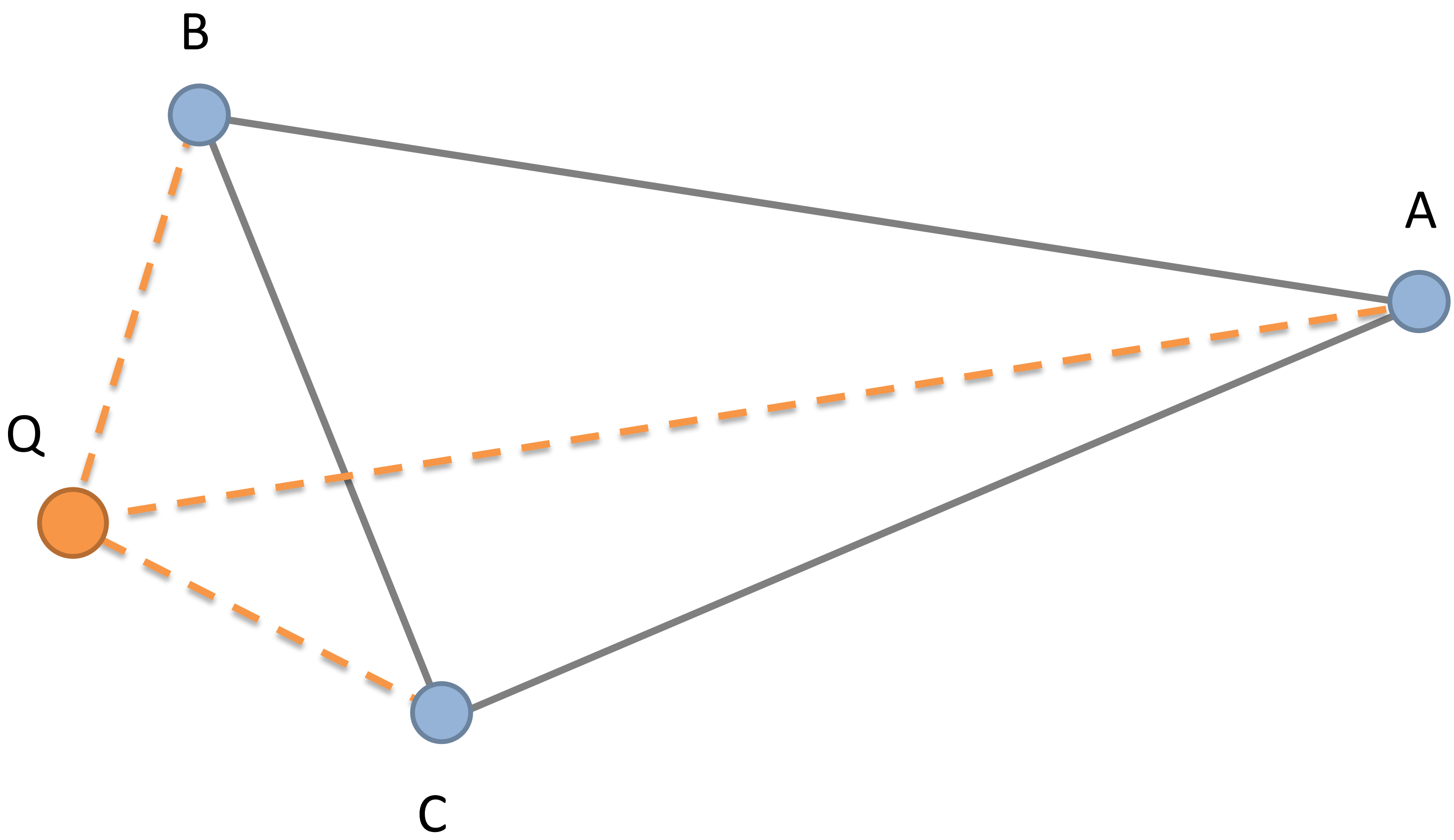
tetrahedrons : fractional volume

Computing Area

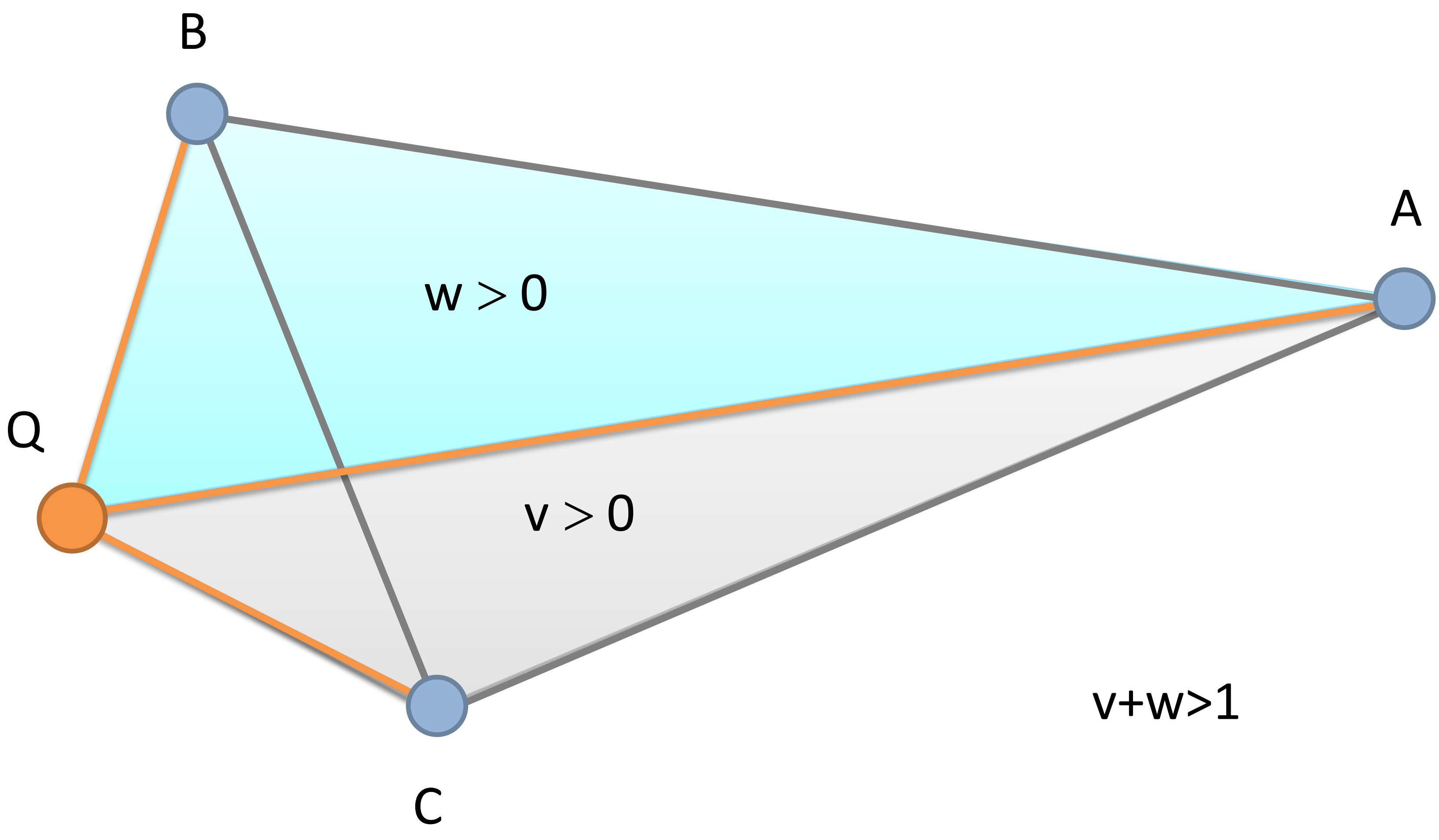


$$\text{signed area} = \frac{1}{2}\,\text{cross}\left(B\text{-}A, C\text{-}A\right)$$

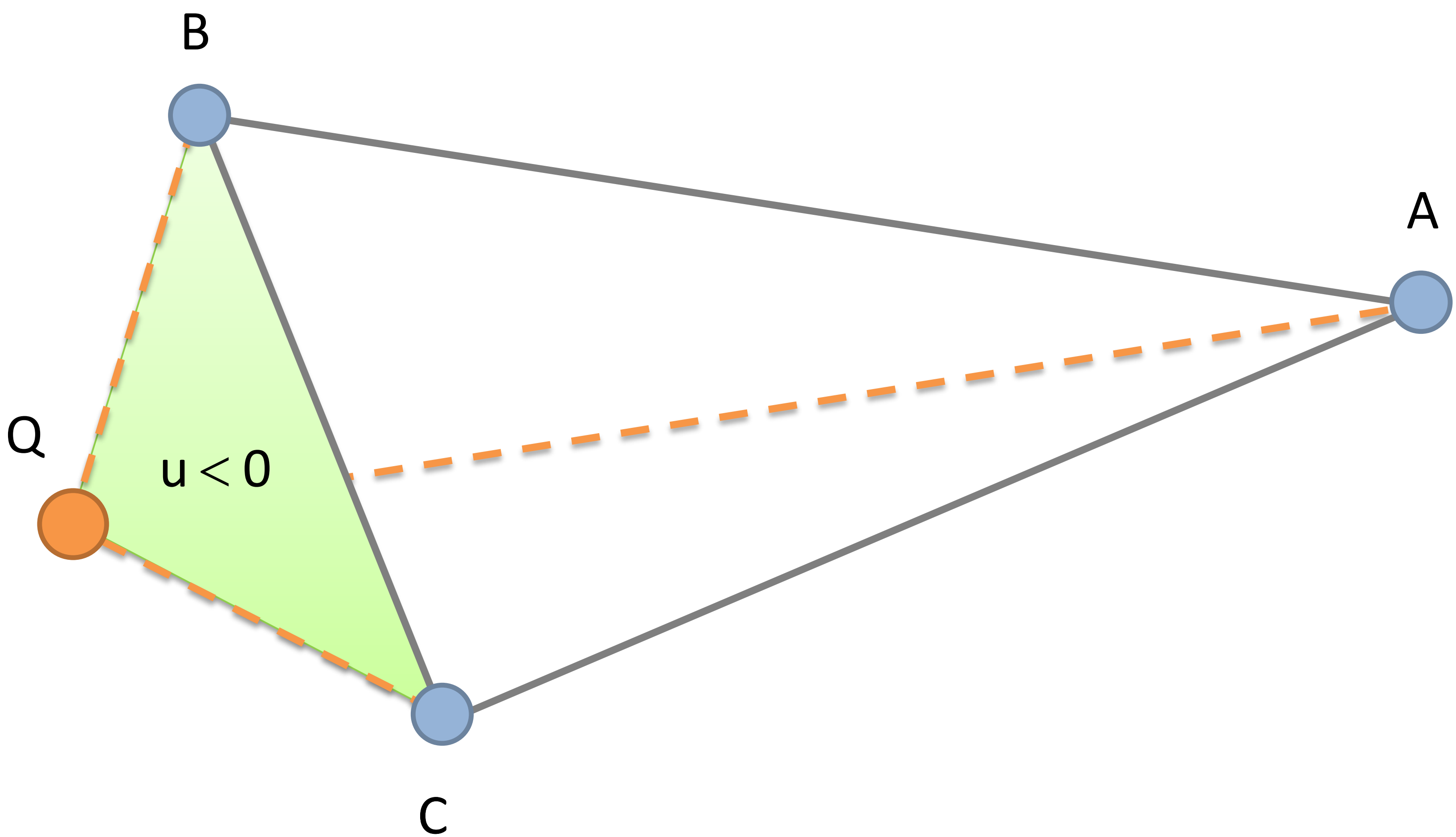
Q outside the triangle



Q outside the triangle



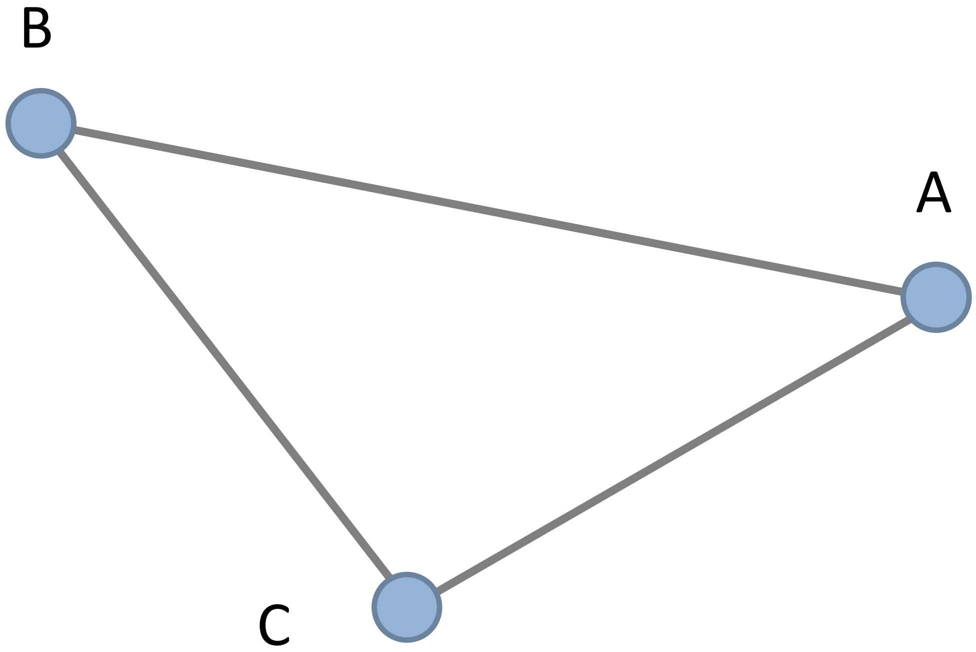
Q outside the triangle



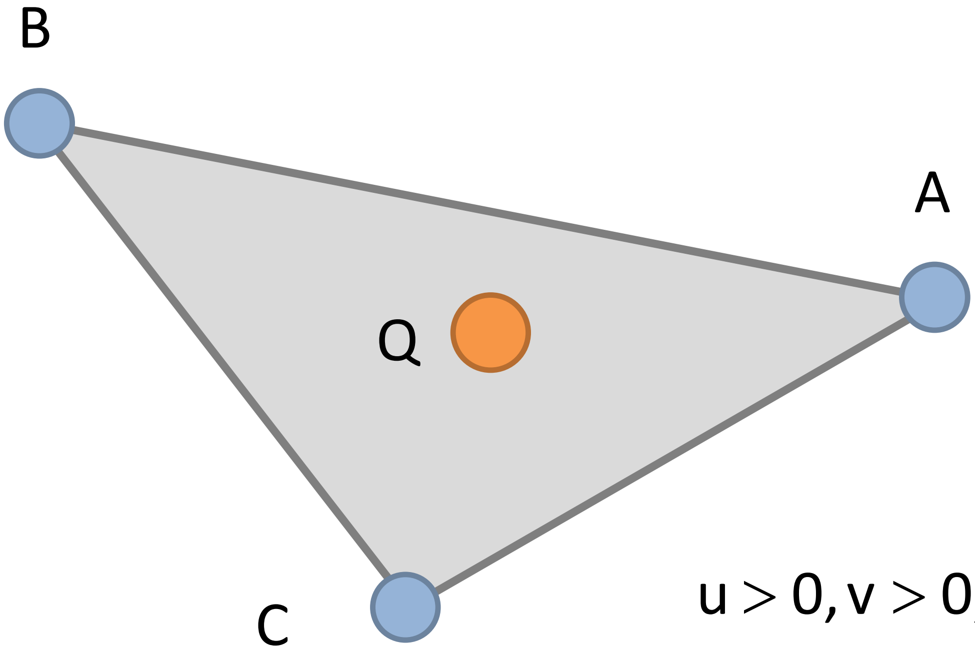
Voronoi versus Barycentric

- Voronoi regions != barycentric coordinate regions
- The barycentric regions are still useful

Barycentric regions of a triangle

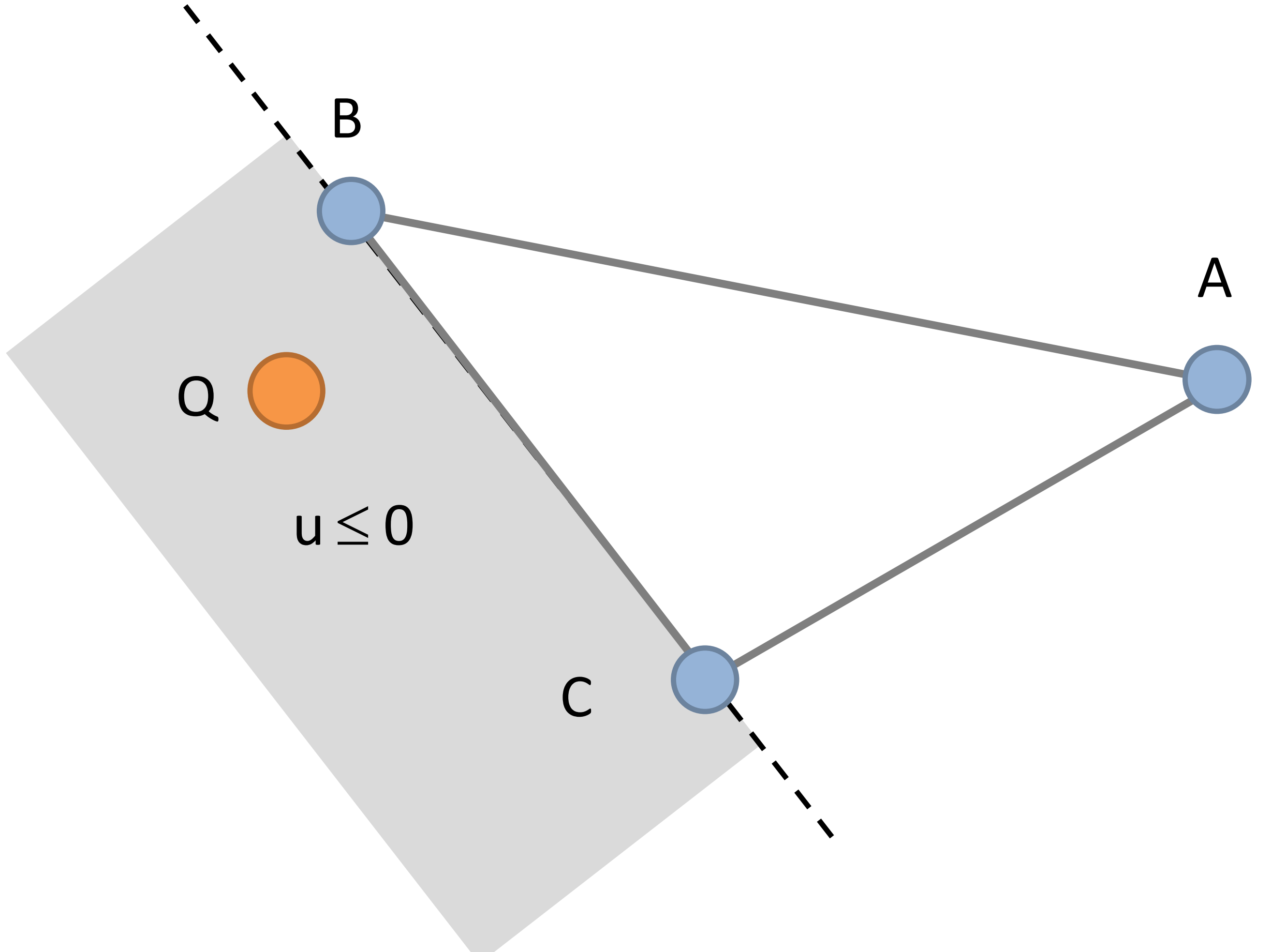


Interior

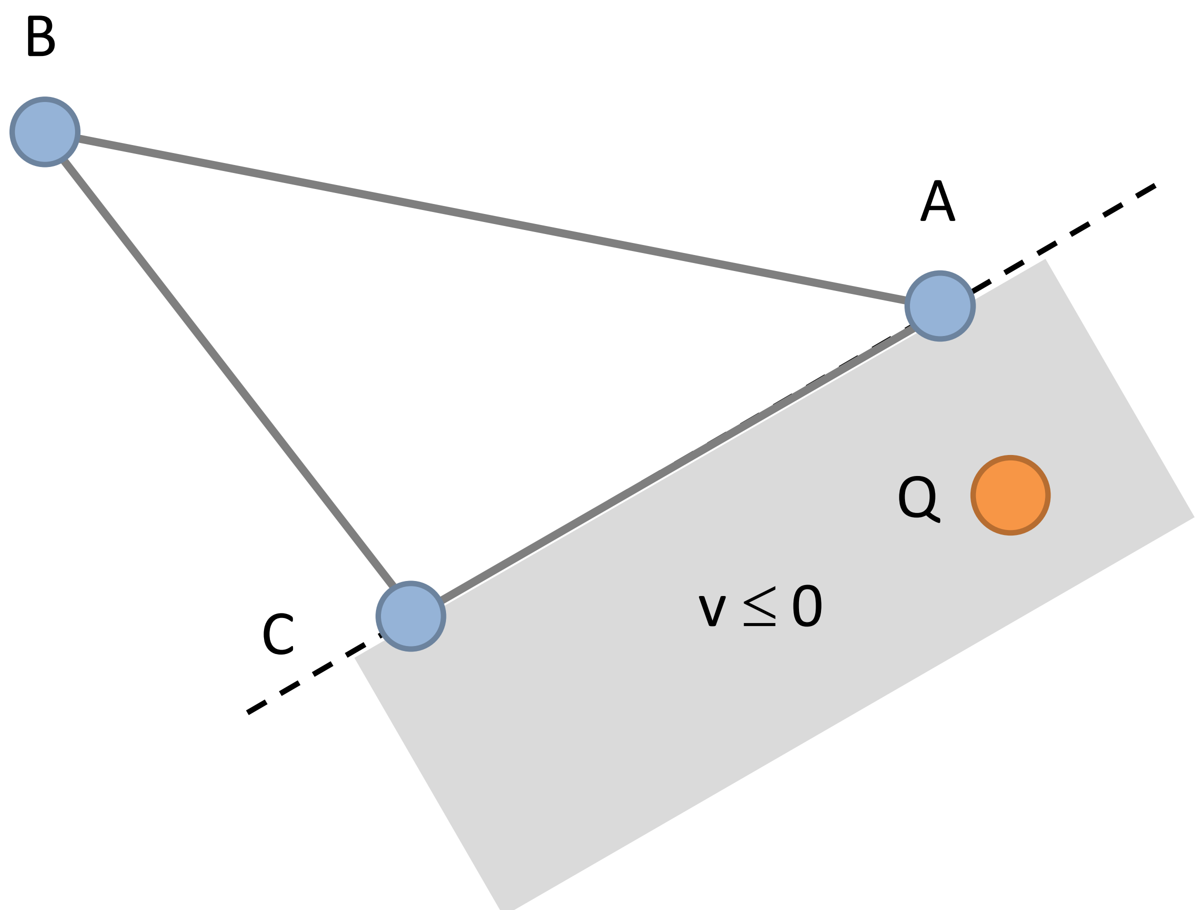


$u > 0, v > 0, w > 0$

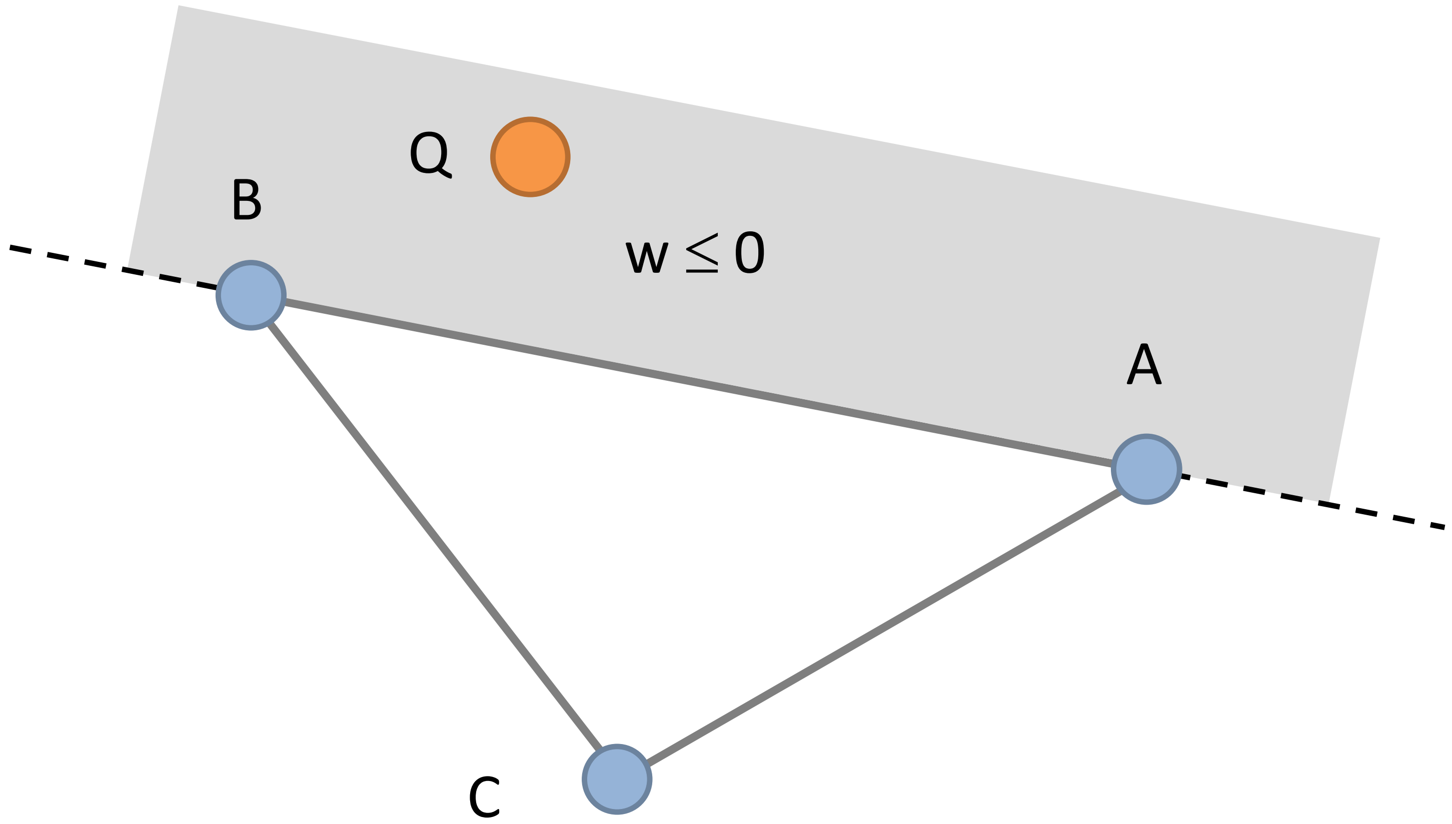
Negative u



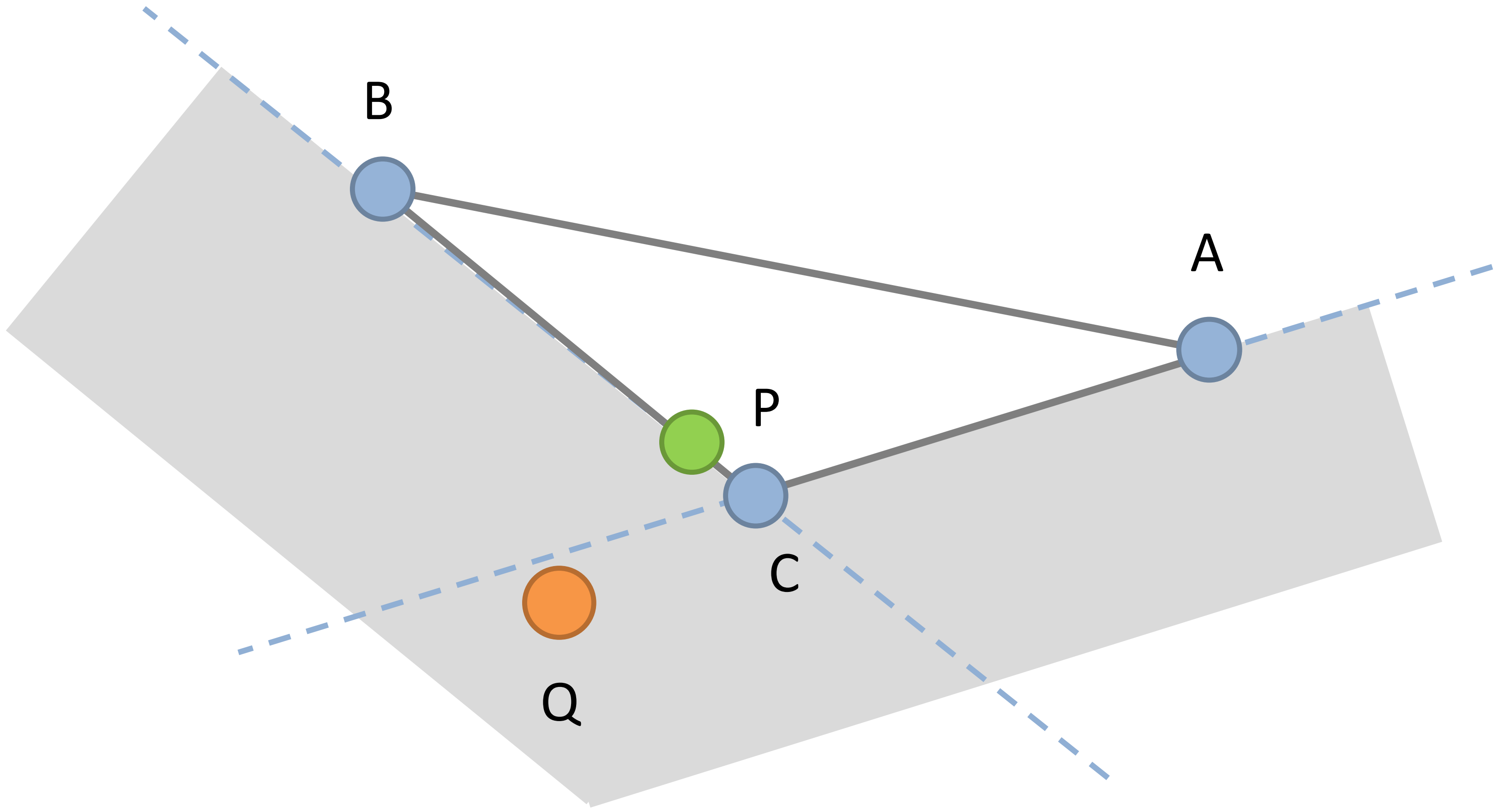
Negative v



Negative w



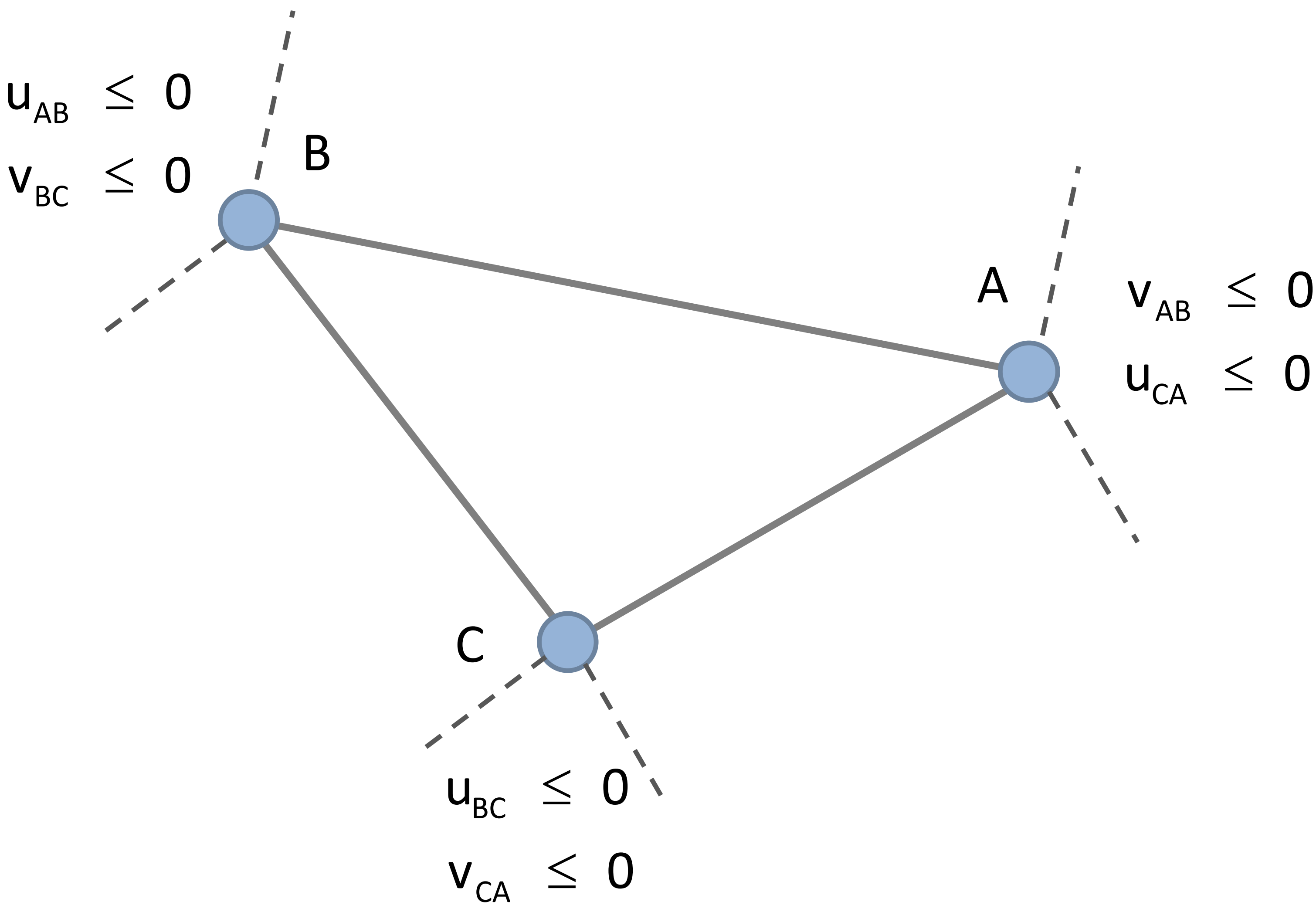
The uv regions are not exclusive



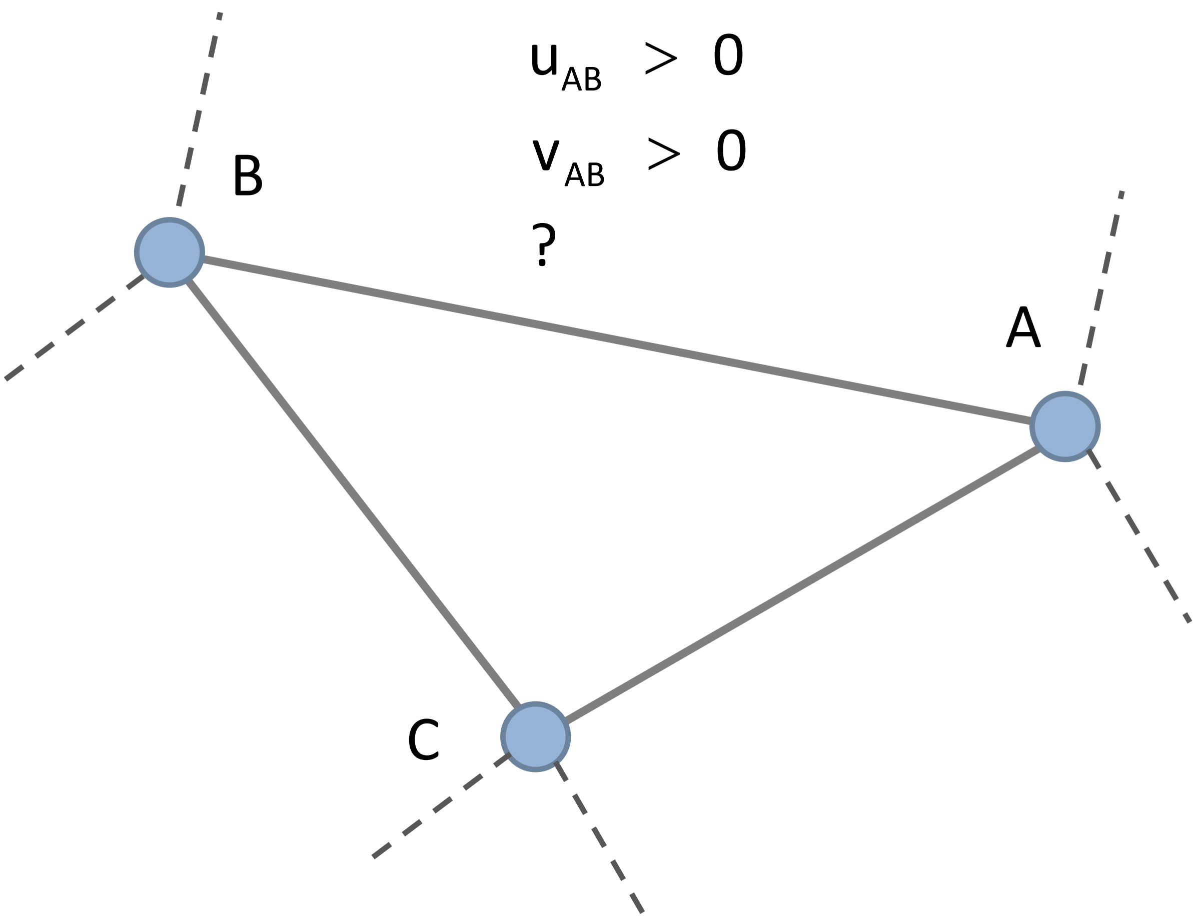
Finding the Voronoi region

- Use the barycentric coordinates to identify the Voronoi region
- Coordinates for the 3 line segments and the triangle
- Regions must be considered in the correct order

First: vertex regions

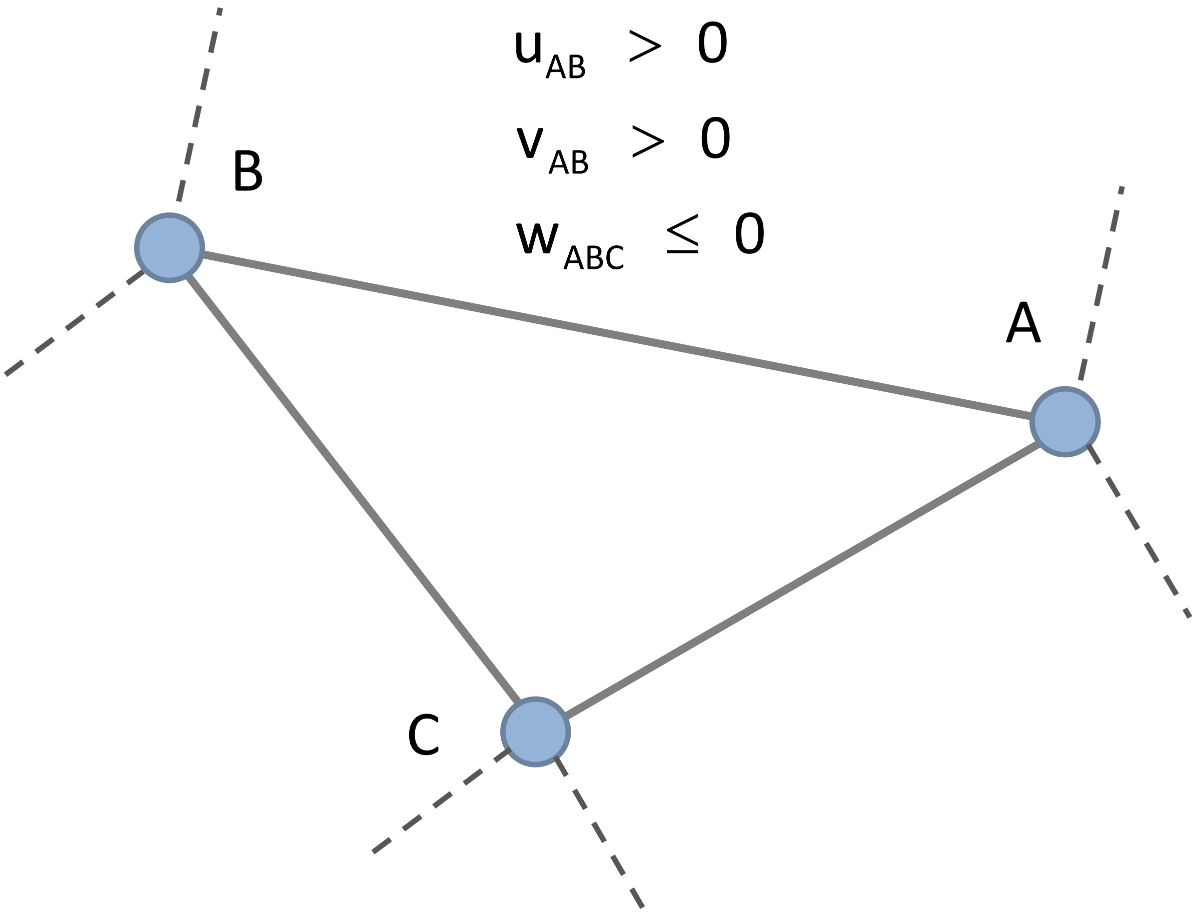


Second: edge regions

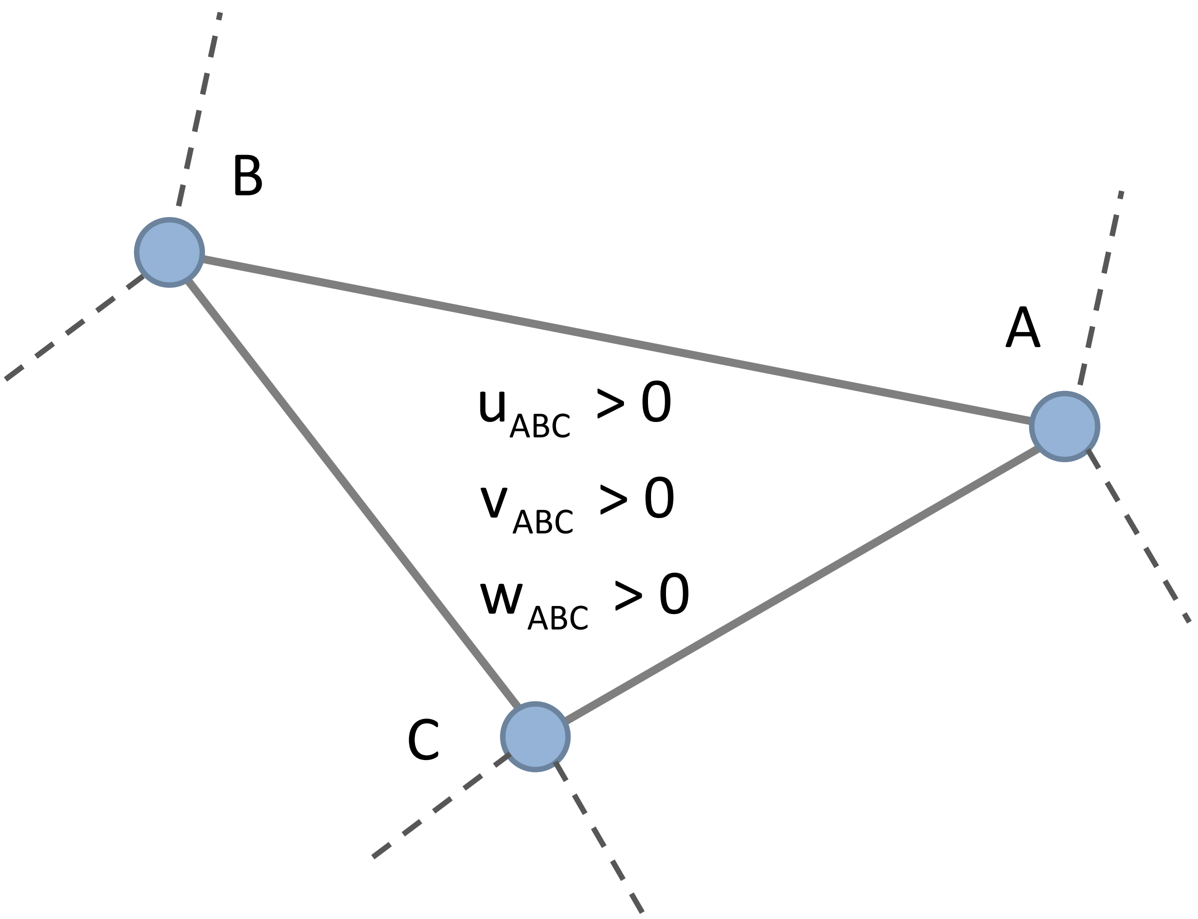


Second: edge regions solved

$$u_{AB} > 0$$
$$v_{AB} > 0$$
$$w_{ABC} \leq 0$$



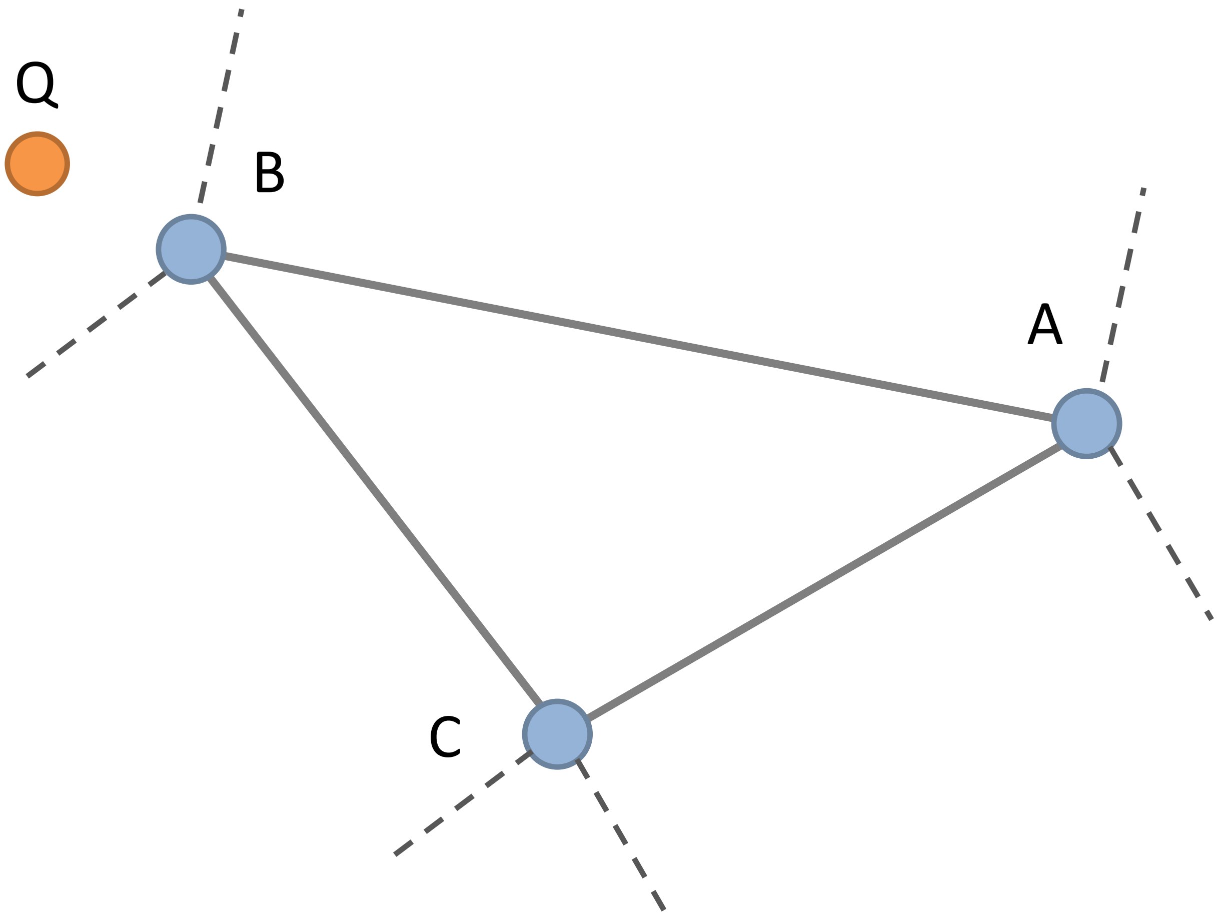
Third: interior region



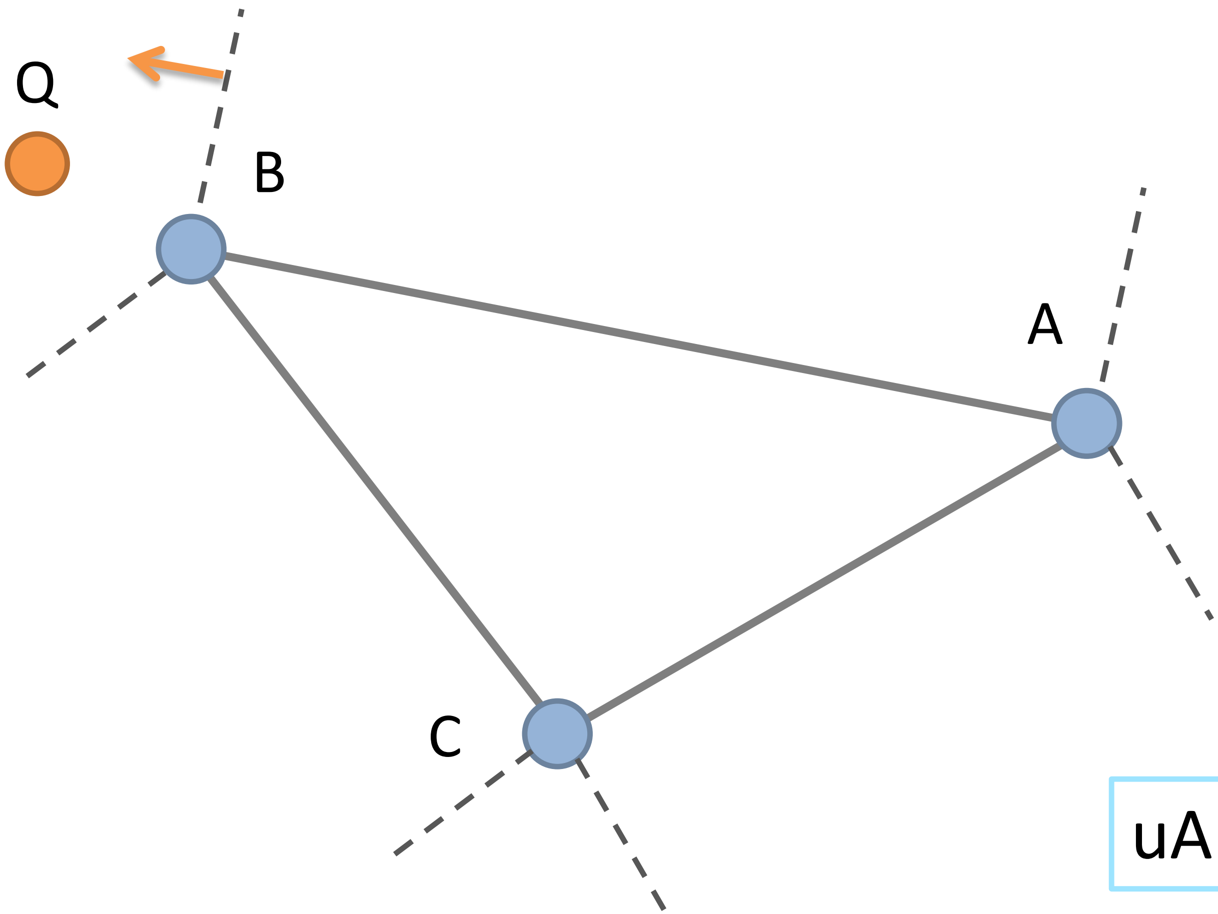
Closest point

- Find the Voronoi region for point Q
- Use the barycentric coordinates to compute the closest point Q

Example 1

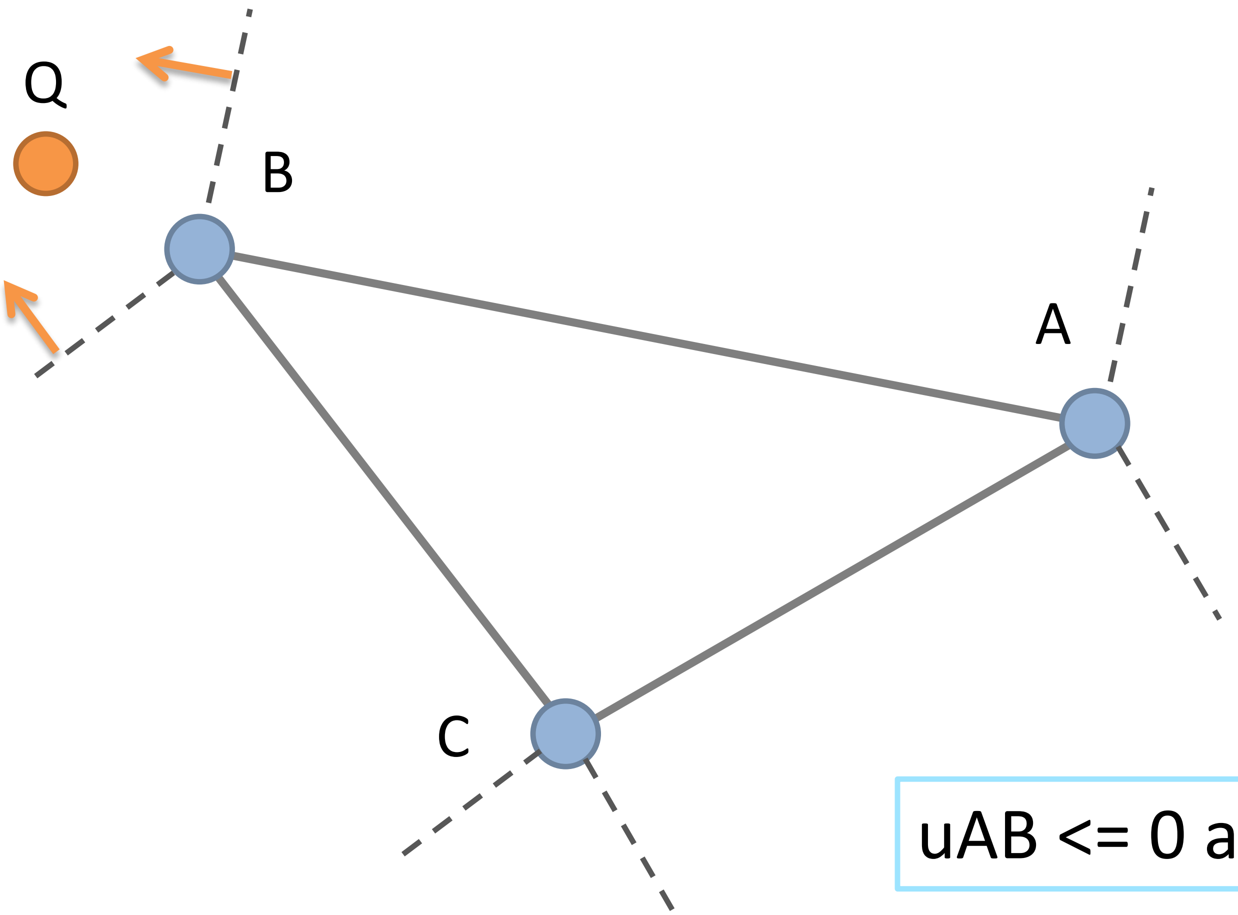


Example 1



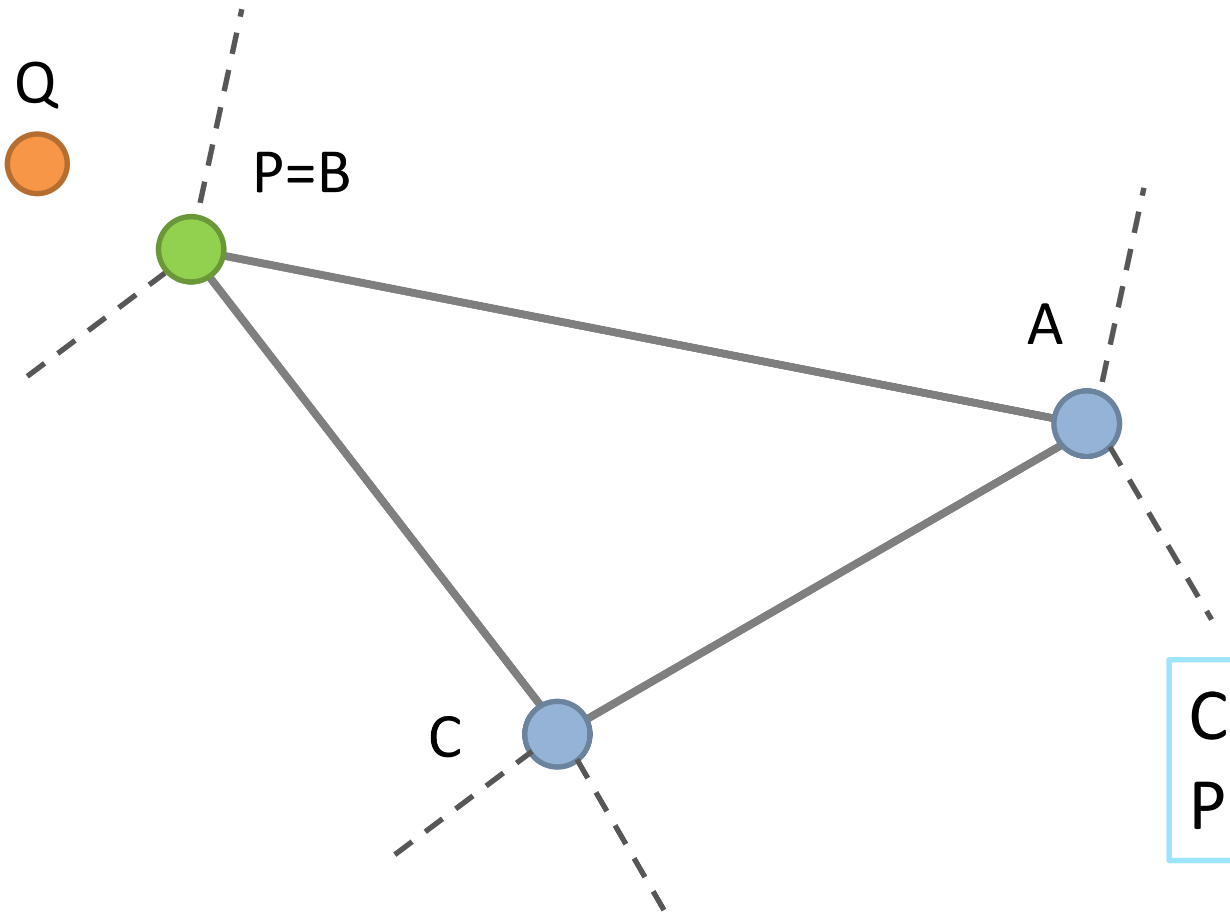
uAB <= 0

Example 1



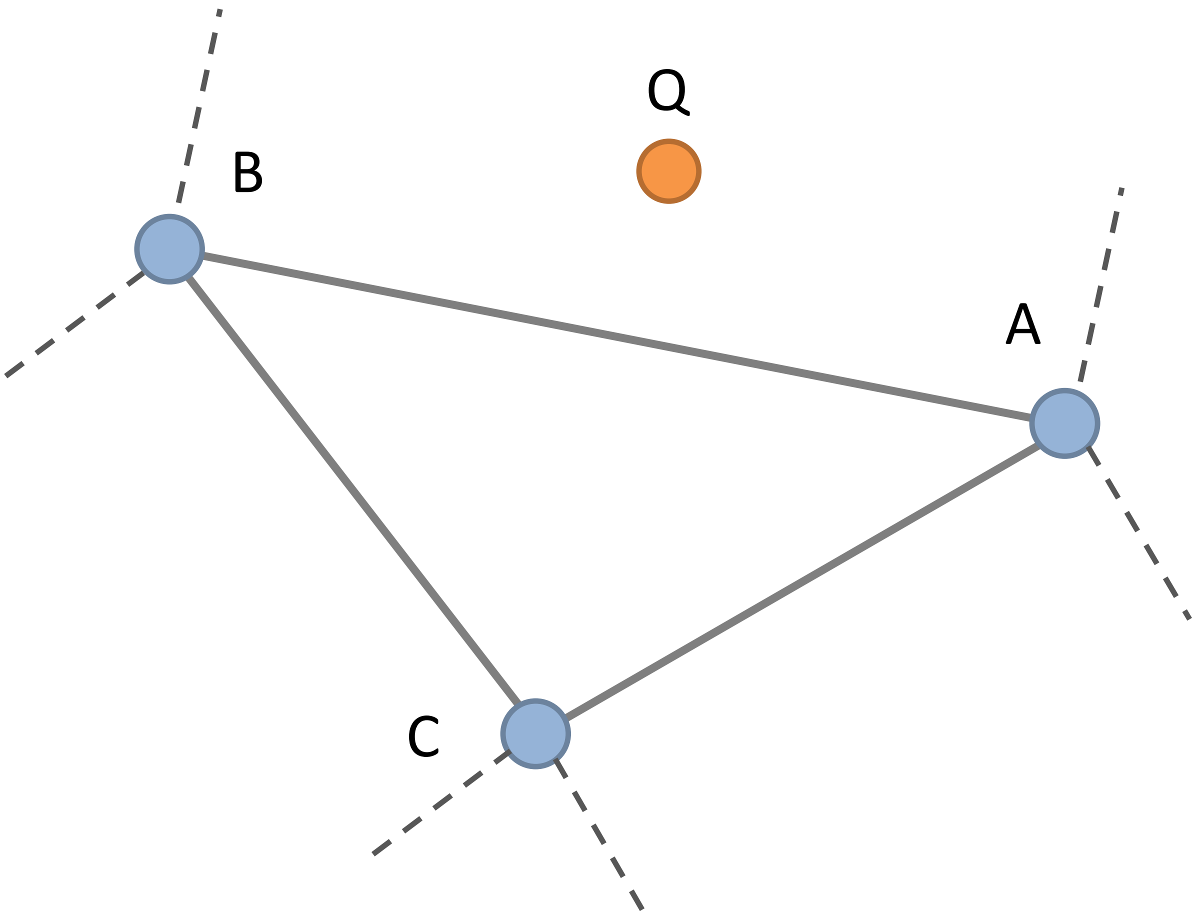
uAB <= 0 and vBC <= 0

Example 1

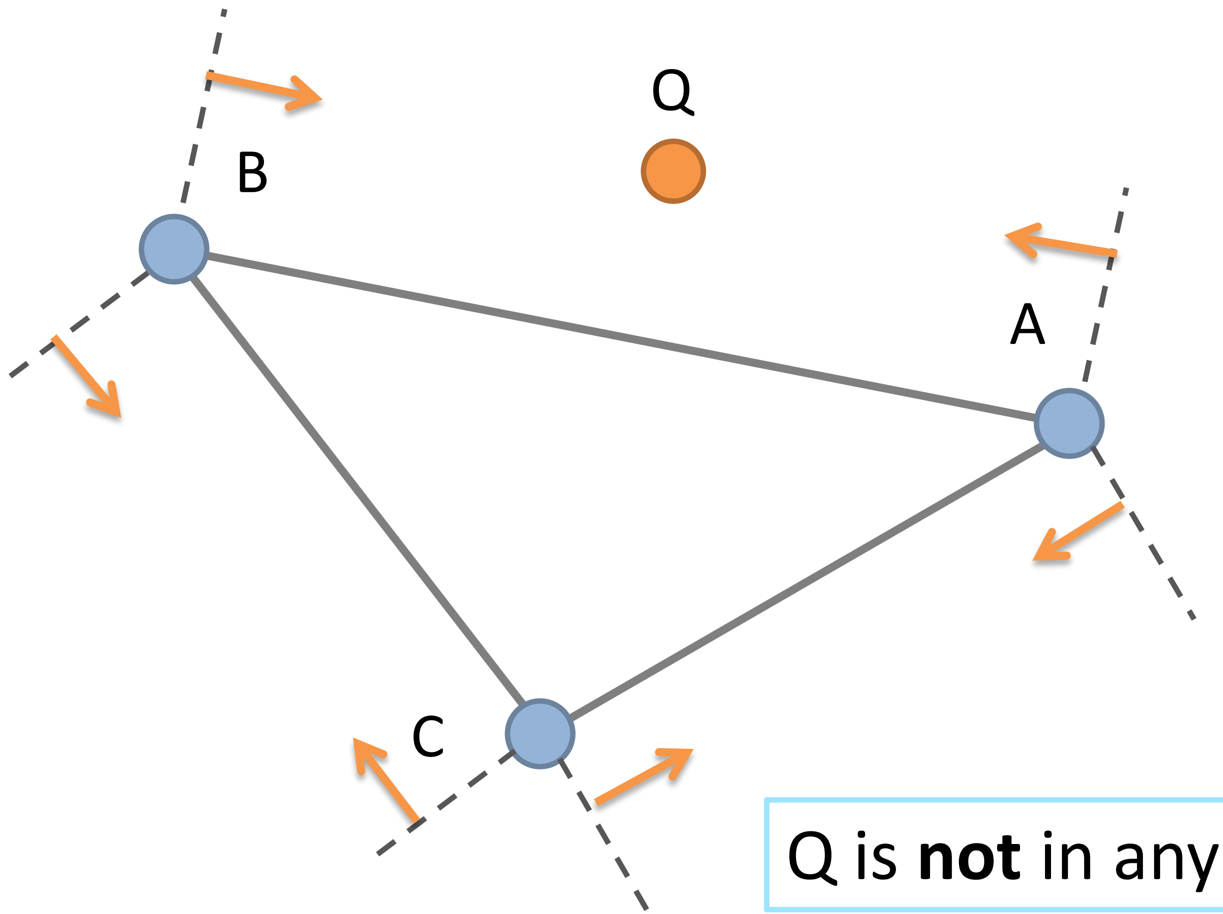


Conclusion:
P = B

Example 2

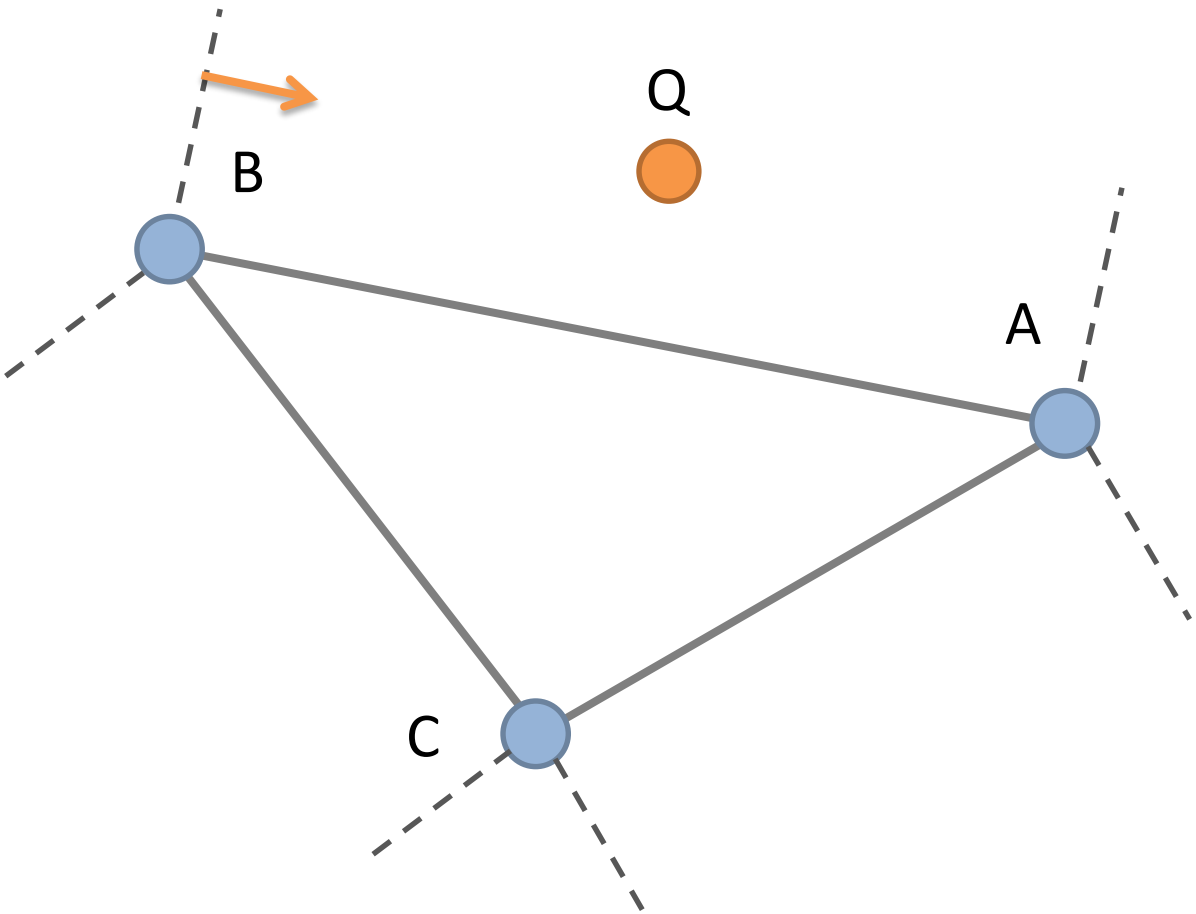


Example 2



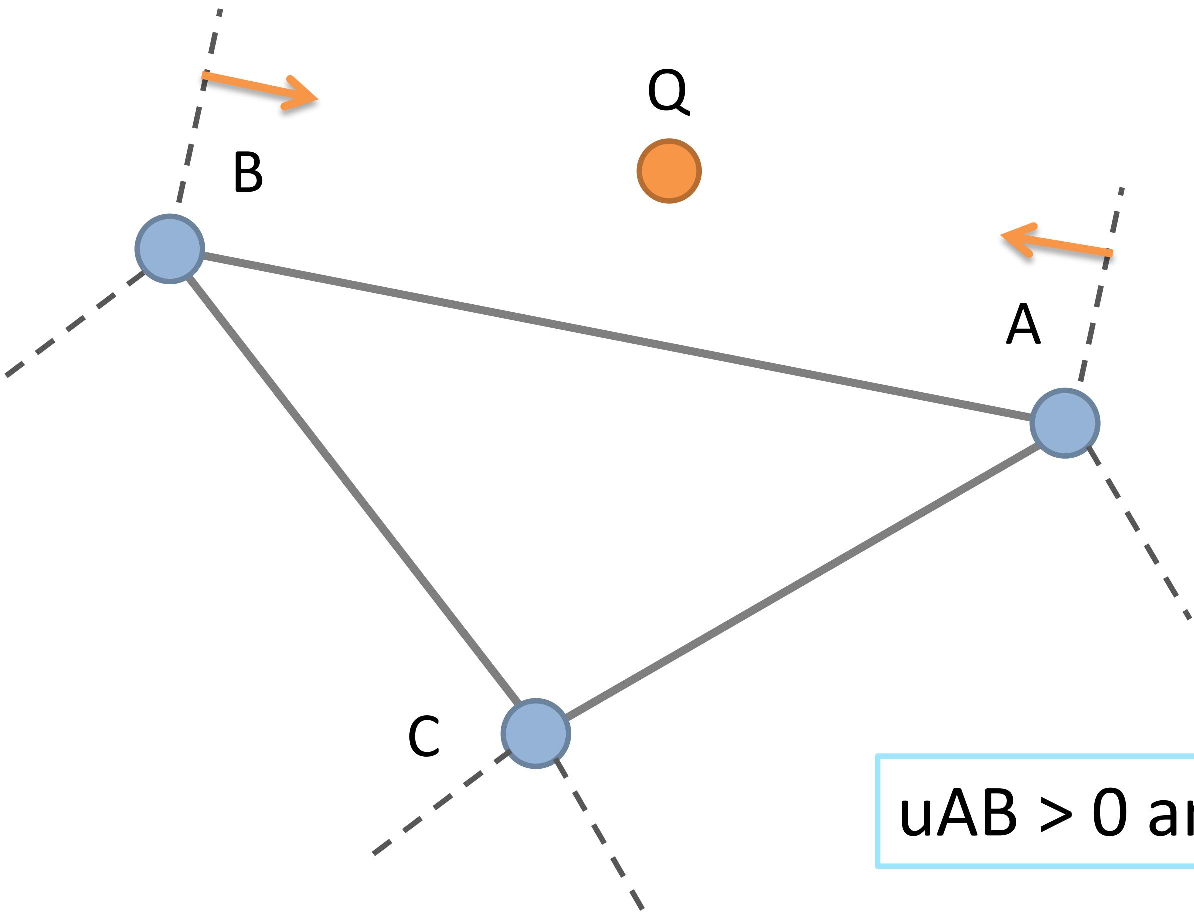
Q is **not** in any vertex region

Example 2



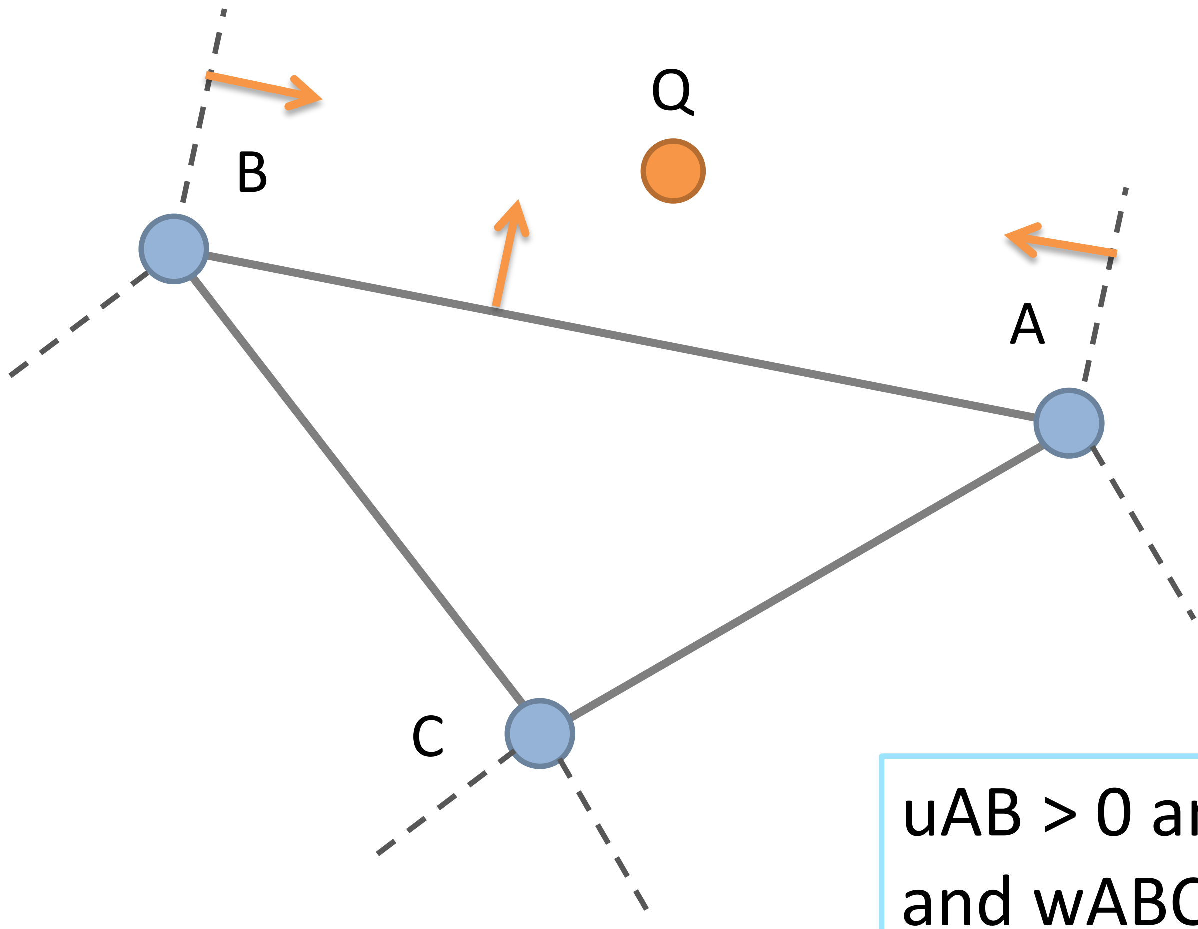
uAB > 0

Example 2



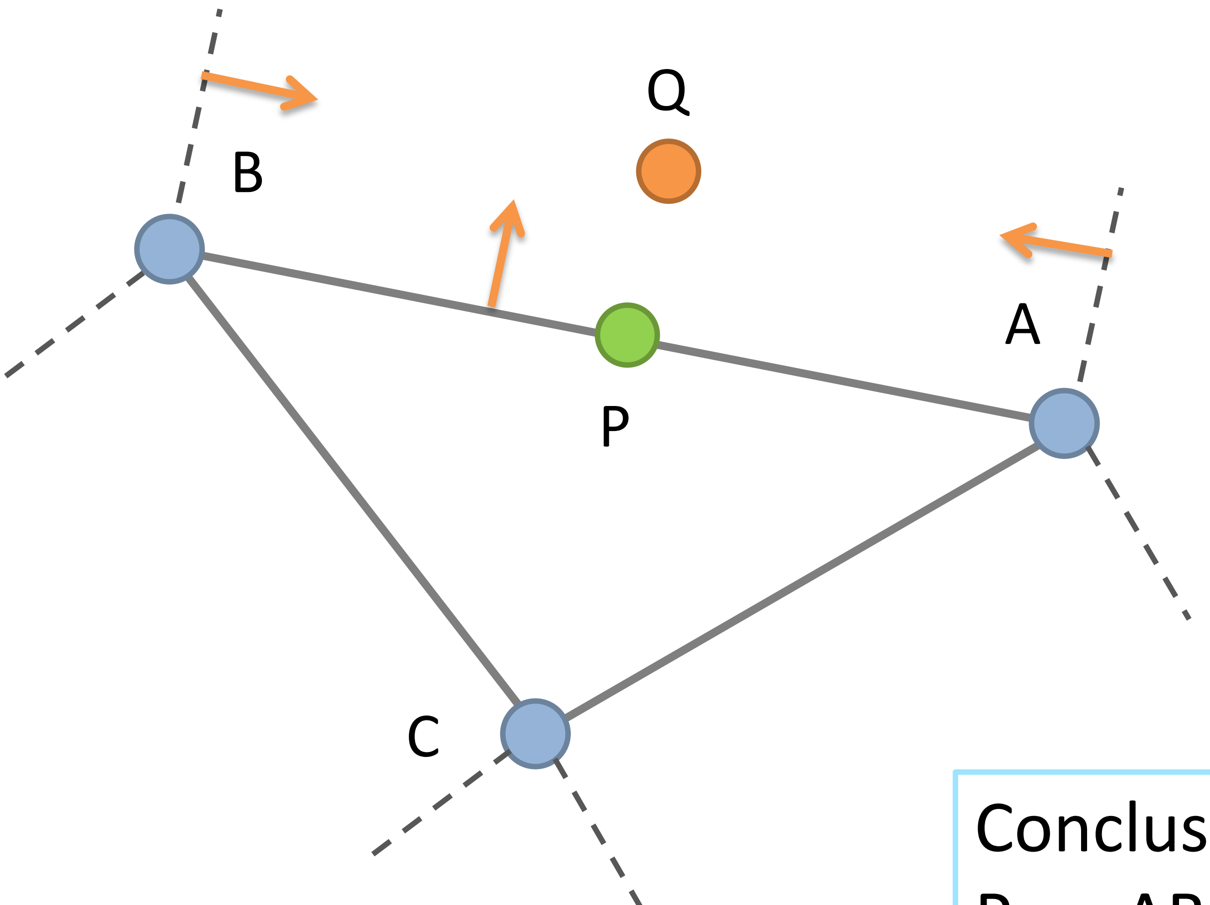
uAB > 0 and vAB > 0

Example 2



uAB > 0 and vAB > 0
and wABC <= 0

Example 2



Conclusion:

P = uAB*A + vAB*B

Implementation

```
input: A, B, C, Q
compute uAB, vAB, uBC, vBC, uCA, vCA
compute uABC, vABC, wABC

// Test vertex regions
…

// Test edge regions
…

// Else interior region
…
```

Testing the vertex regions

```
// Region A
if (vAB <= 0 && uCA <= 0)
  P = A
  return

// Similar tests for Region B and C
```

Testing the edge regions

```
// Region AB
if (uAB > 0 && vAB > 0 && wABC <= 0)
  P = uAB * A + vAB * B
  return

// Similar for Regions BC and CA
```

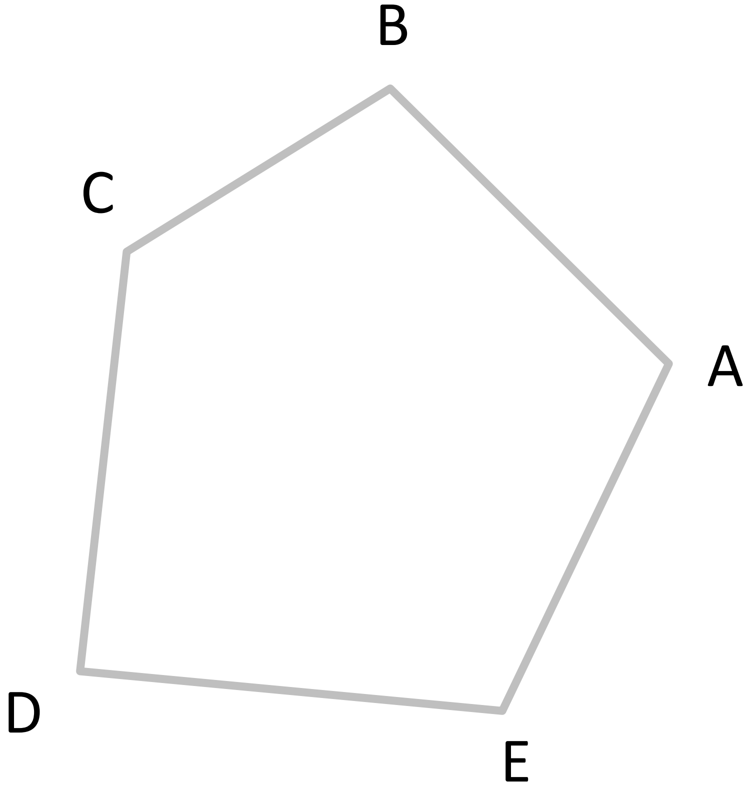
Testing the interior region

```
// Region ABC
assert(uABC > 0 && vABC > 0 && wABC > 0)
P = Q
return
```

Section 3

Point to Convex Polygon

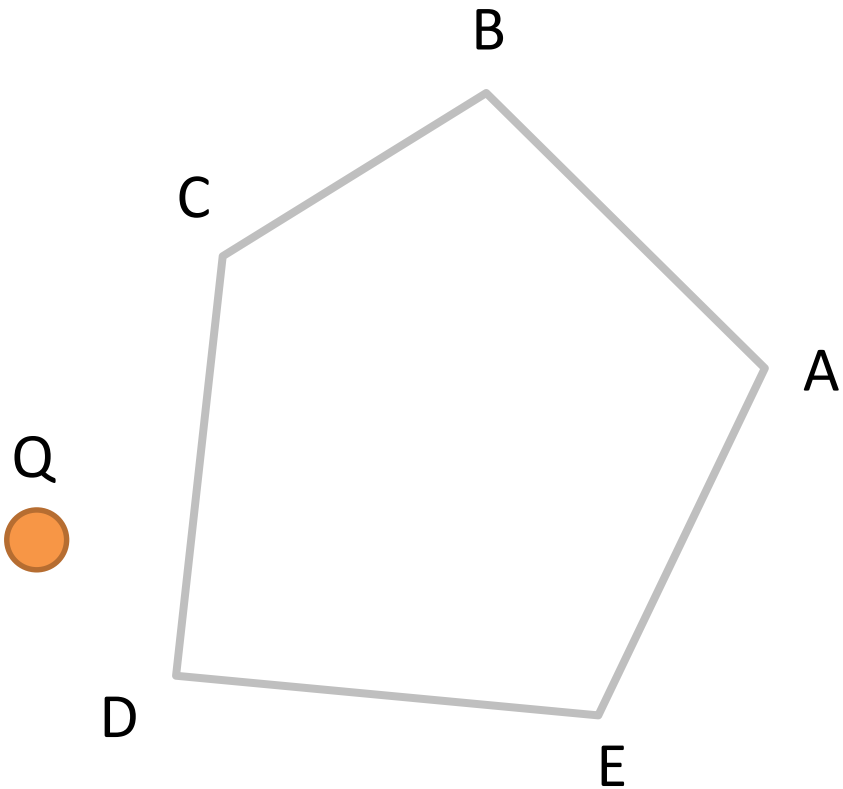
Convex polygon



Polygon structure

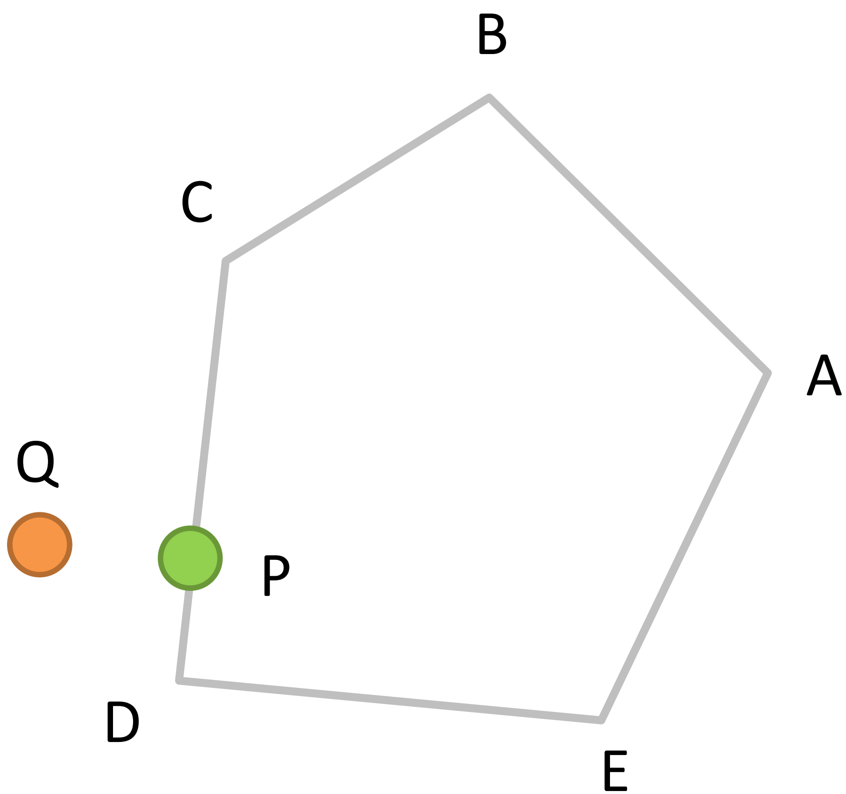
```
struct Polygon
{
  Vec2* points;
  int count;
};
```

Convex polygon: closest point



Query point Q

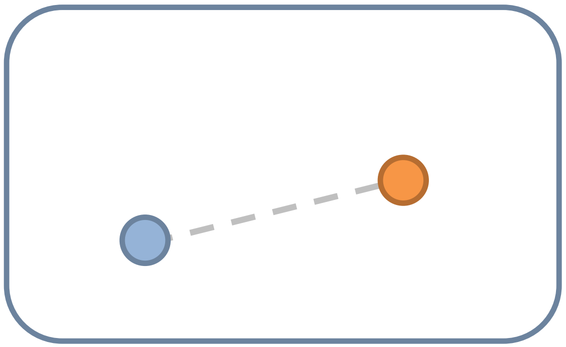
Convex polygon: closest point



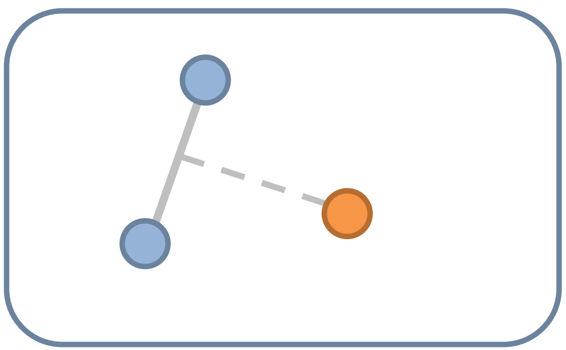
Closest point Q

How do we compute P?

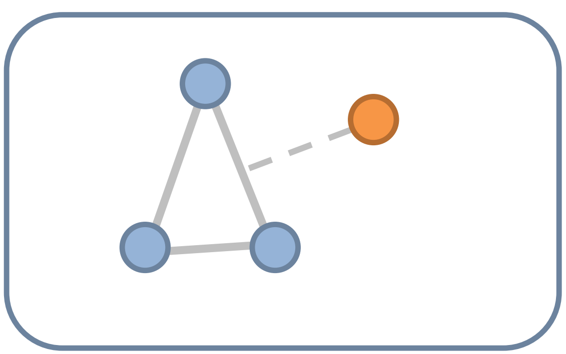
What do we know?



Closest point to point

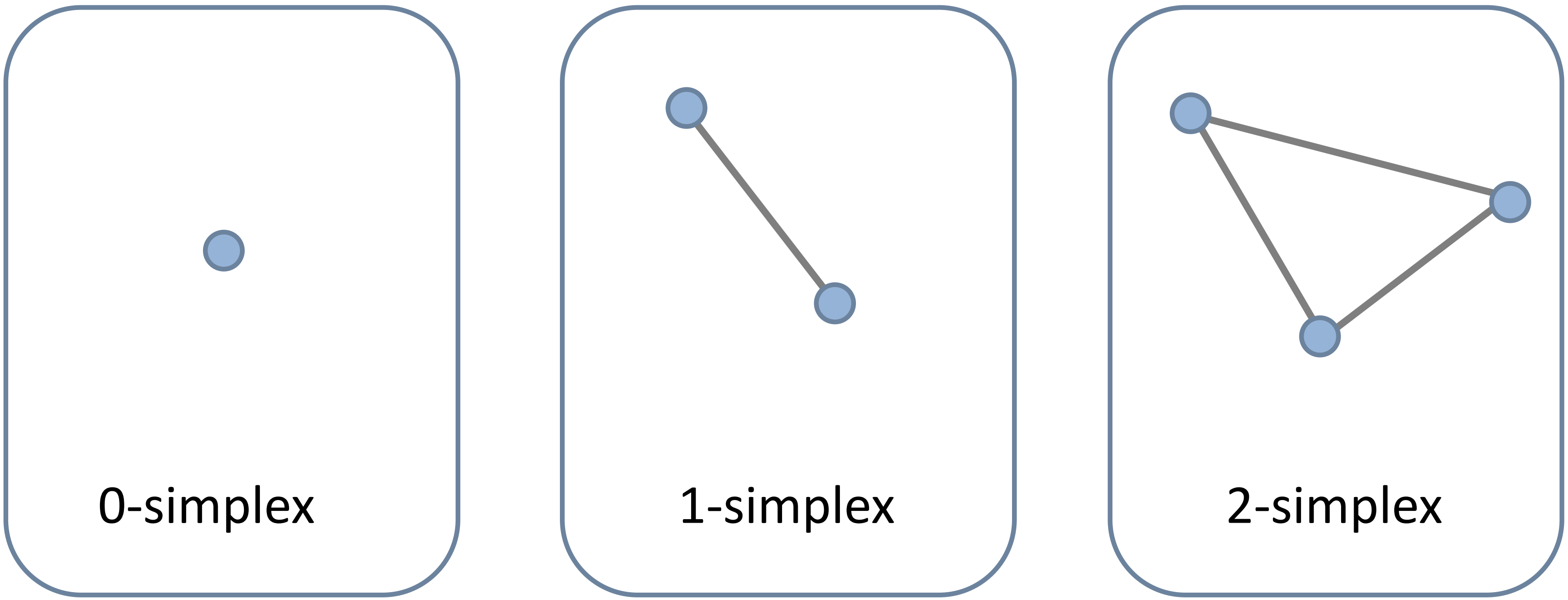


Closest point to line segment

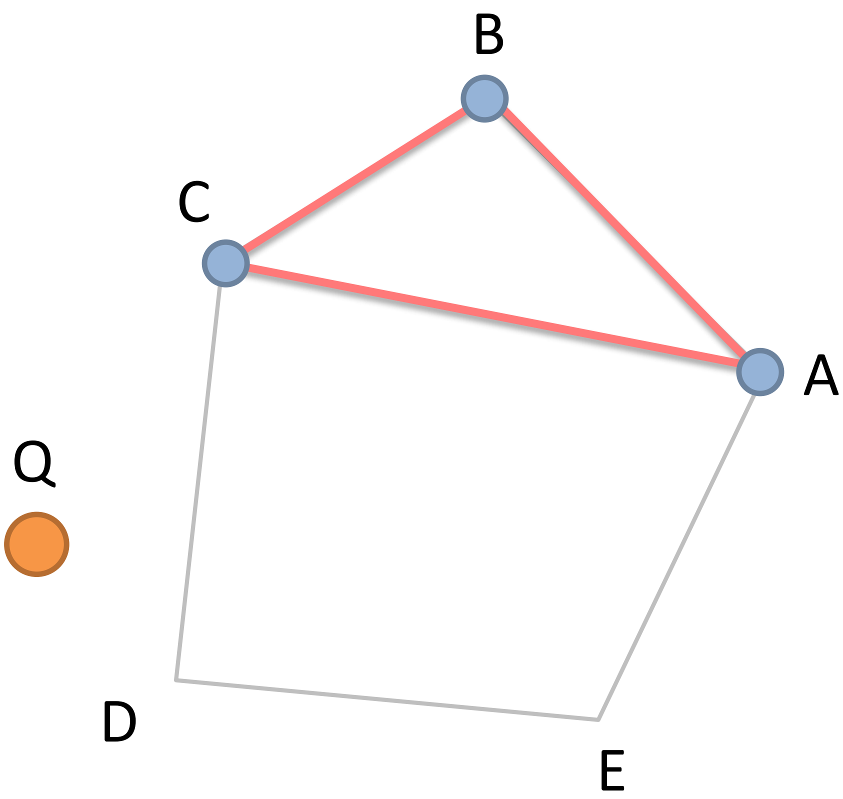


Closest point to triangle

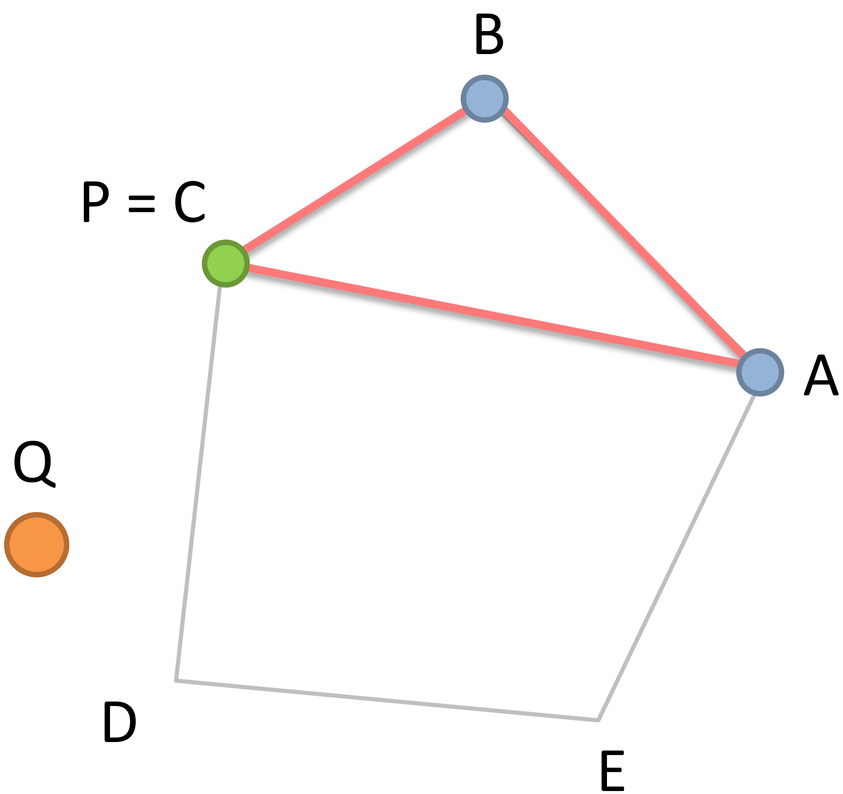
Simplex



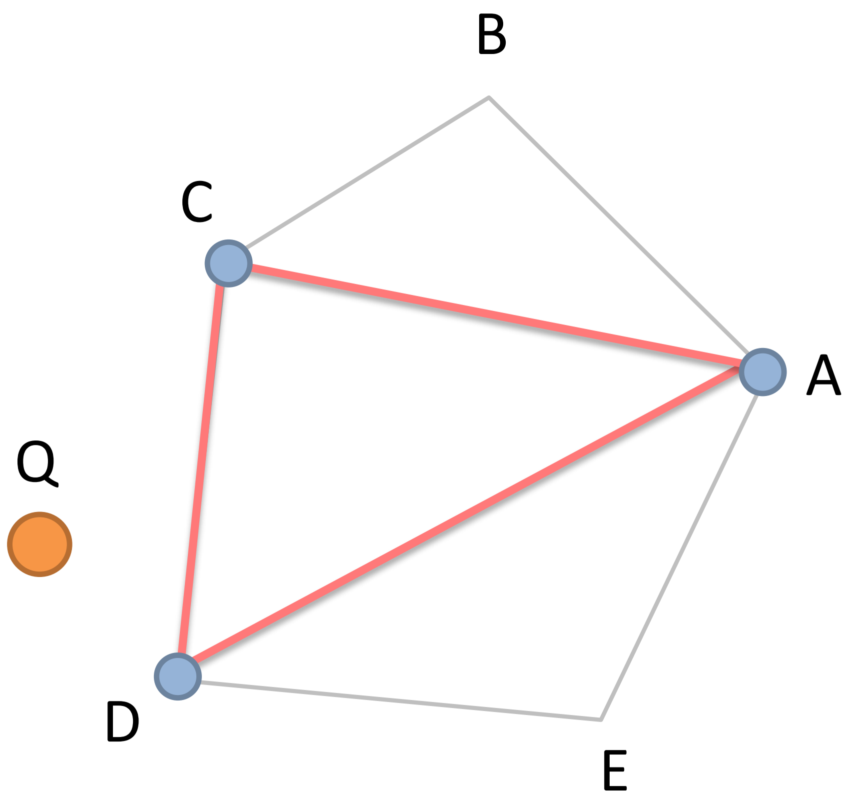
Idea: inscribe a simplex



Idea: closest point on simplex



Idea: evolve the simplex



Simplex vertex

```
struct SimplexVertex
{
    Vec2 point;
    int index;
    float u;
};
```

Simplex

```
struct Simplex
{
  SimplexVertex vertexA;
  SimplexVertex vertexB;
  SimplexVertex vertexC;
  int count;
};
```

We are onto a winner!

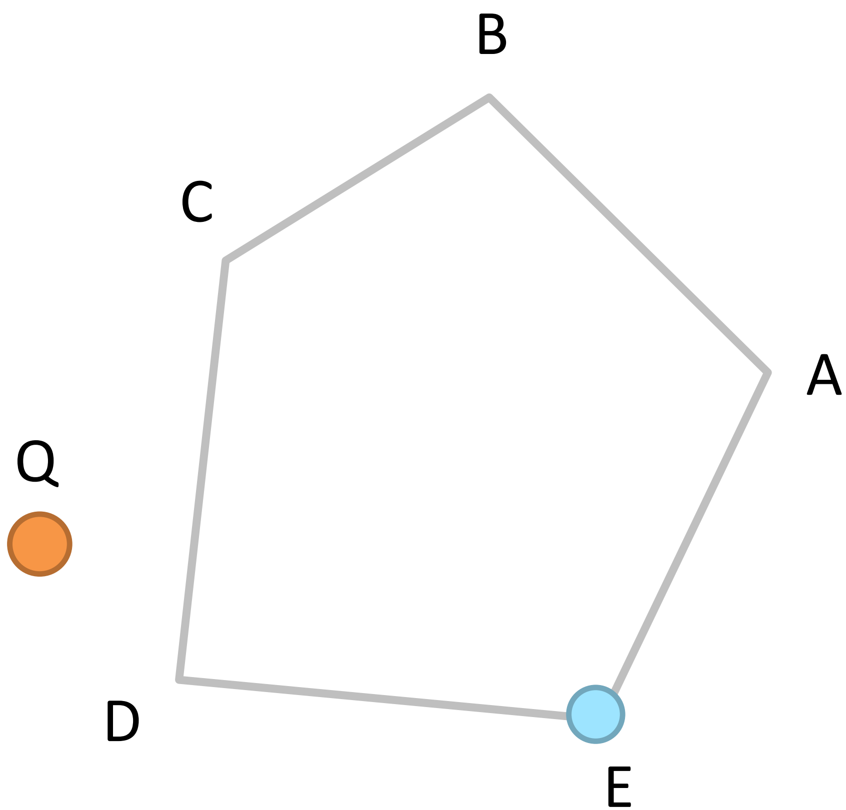
The GJK distance algorithm

- Computes the closest point on a convex polygon
- Invented by Gilbert, Johnson, and Keerthi

The GJK distance algorithm

- Inscribed simplexes
- Simplex evolution

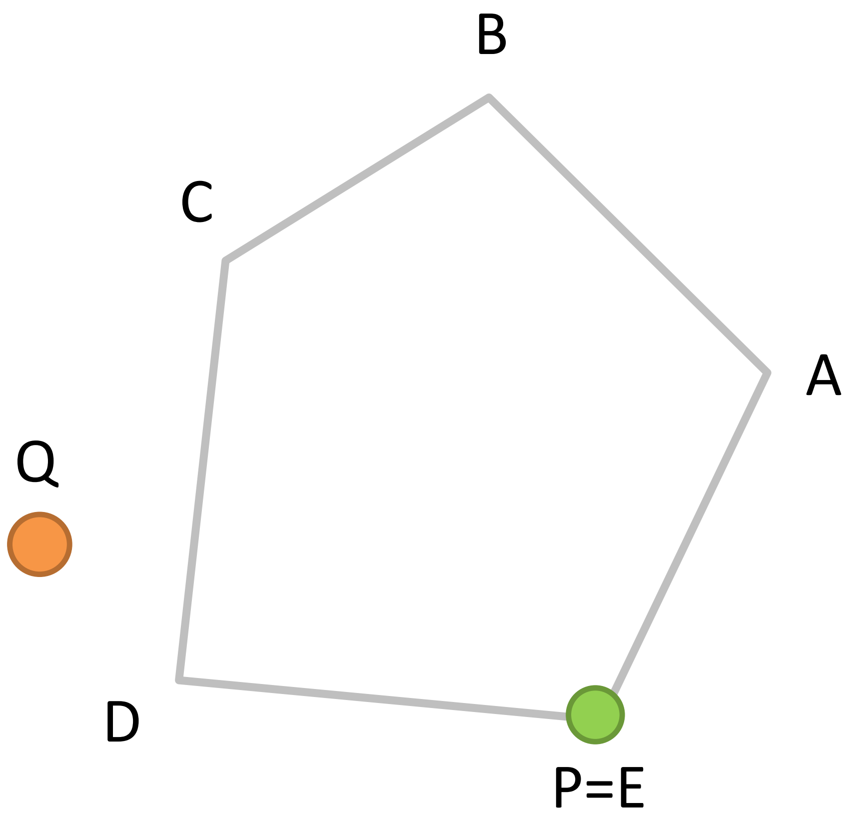
Starting simplex



Start with arbitrary vertex. Pick E.

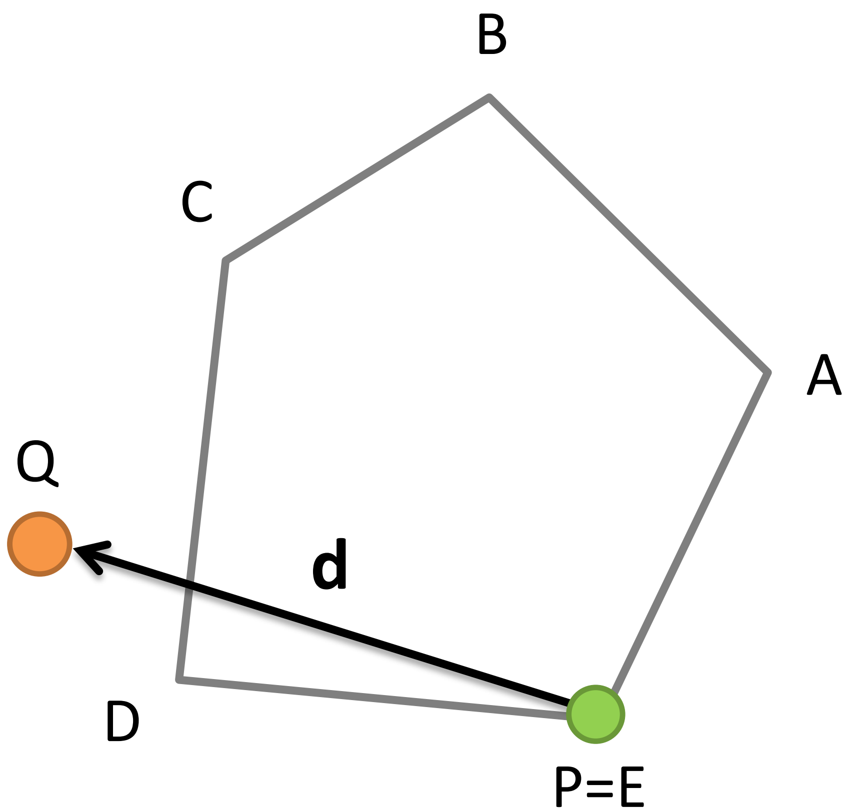
This is our starting simplex.

Closest point on simplex



P is the closest point.

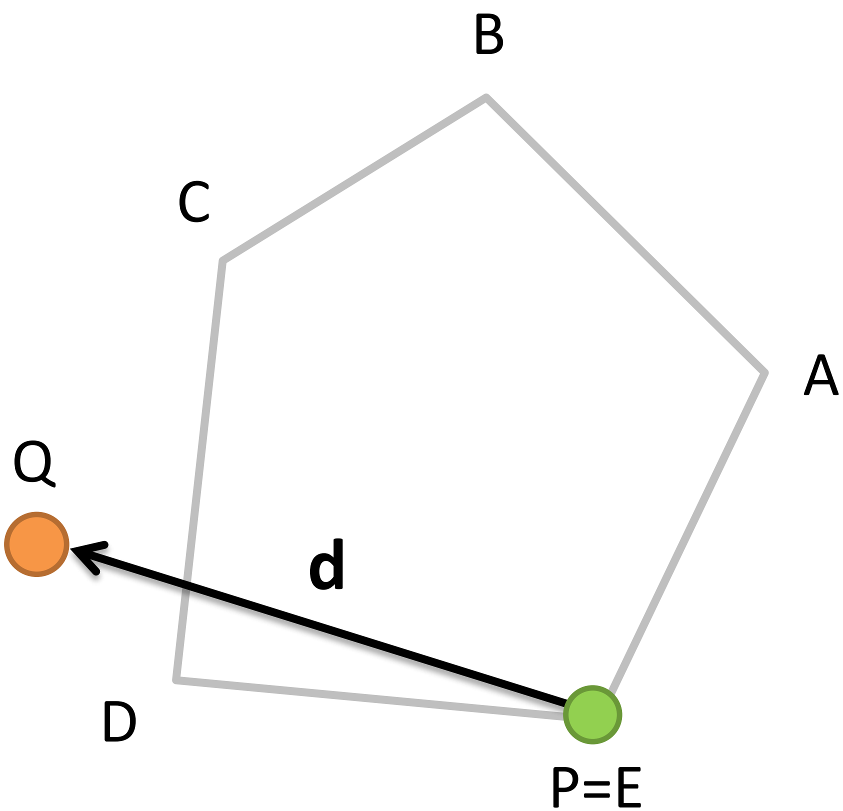
Search vector



Draw a vector from P to Q.

Call this vector **d**.

Find the support point



Find the vertex on polygon furthest in direction **d**.

This is the *support* point.

Support Points

The support point of a polygon is the vertex that is the farthest along a given direction. If two vertices have equal distances along the given direction, either one is acceptable.

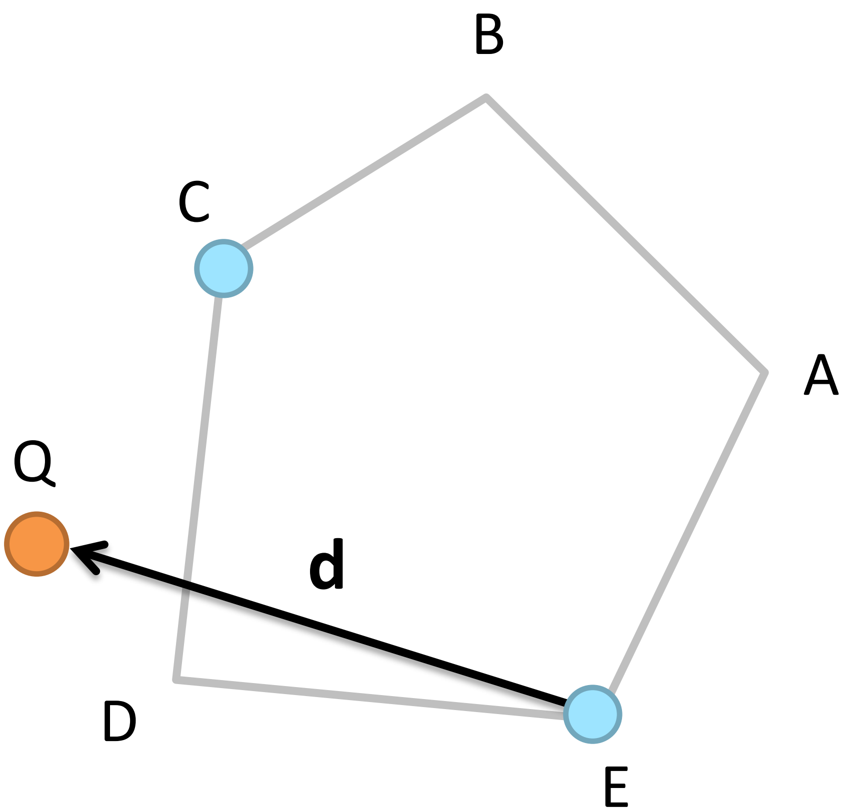
In order to compute a supporting point, the dot product must be used to find a signed distance along a given direction. Since this is very simple, I'll show a quick example within this article:

```
// The extreme point along a direction within a polygon
Vec2 GetSupport( const Vec2& dir )
{
  real bestProjection = -FLT_MAX;
  Vec2 bestVertex;

  for(uint32 i = 0; i < m_vertexCount; ++i)
  {
    Vec2 v = m_vertices[i];
    real projection = Dot( v, dir );

    if(projection > bestProjection)
    {
      bestVertex = v;
      bestProjection = projection;
    }
  }

  return bestVertex;
}
```

Support point code

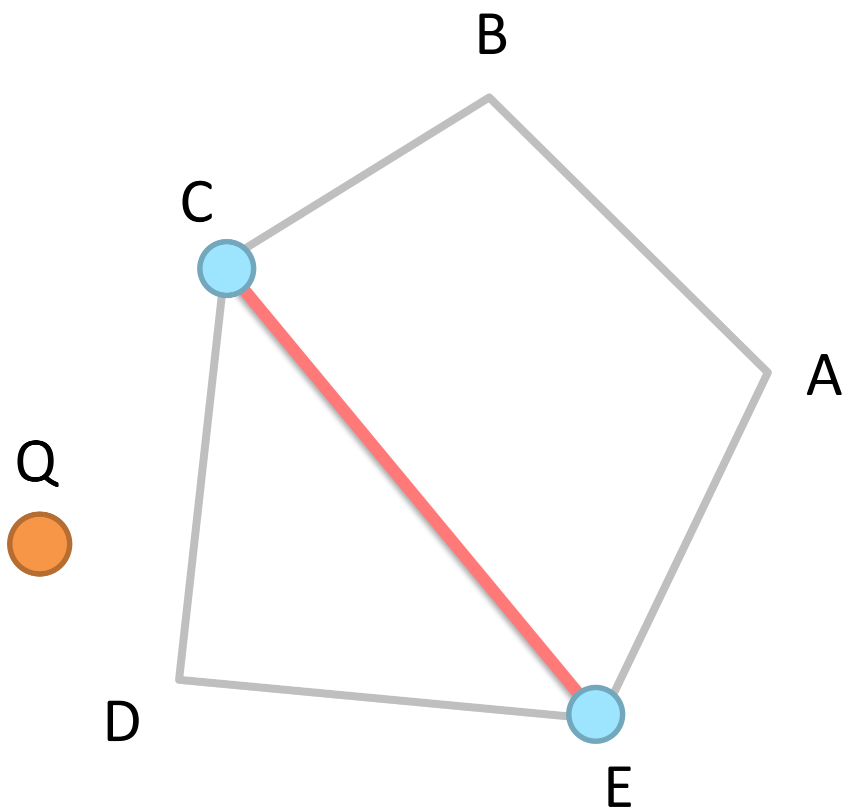
```
int Support(const Polygon& poly, const Vec2& d)
{
  int index = 0;
  float maxValue = Dot(d, poly.points[index]);
  for (int i = 1; i < poly.count; ++i)
  {
    float value = Dot(d, poly.points[i]);
    if (value > maxValue)
    {
      index = i;
      maxValue = value;
    }
  }
  return index;
}
```

Support point found



C is the support point.

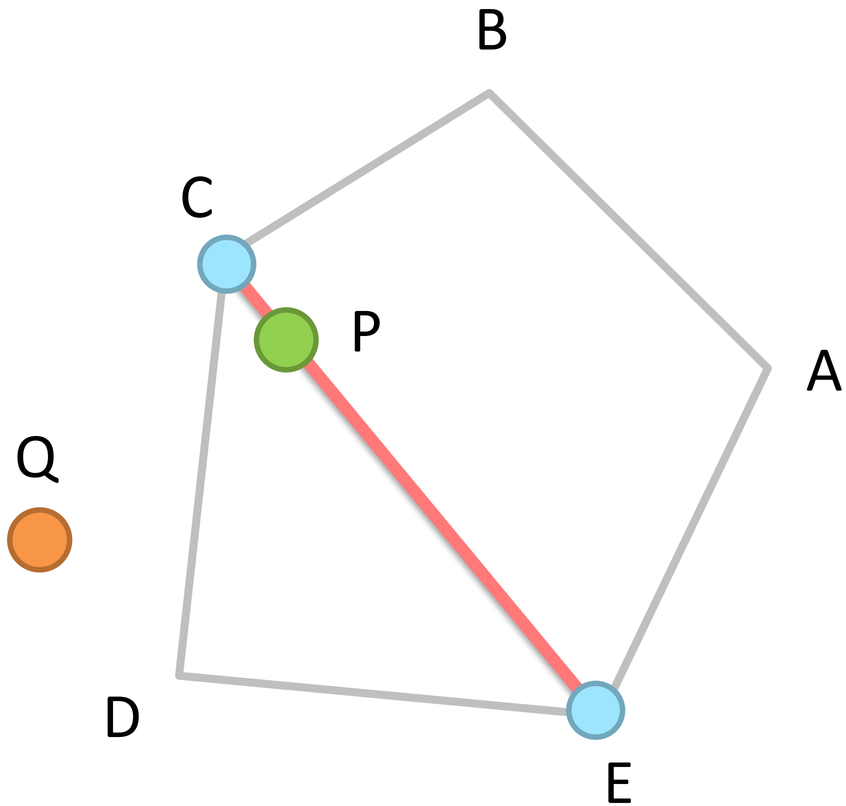
Evolve the simplex



Create a line segment CE.

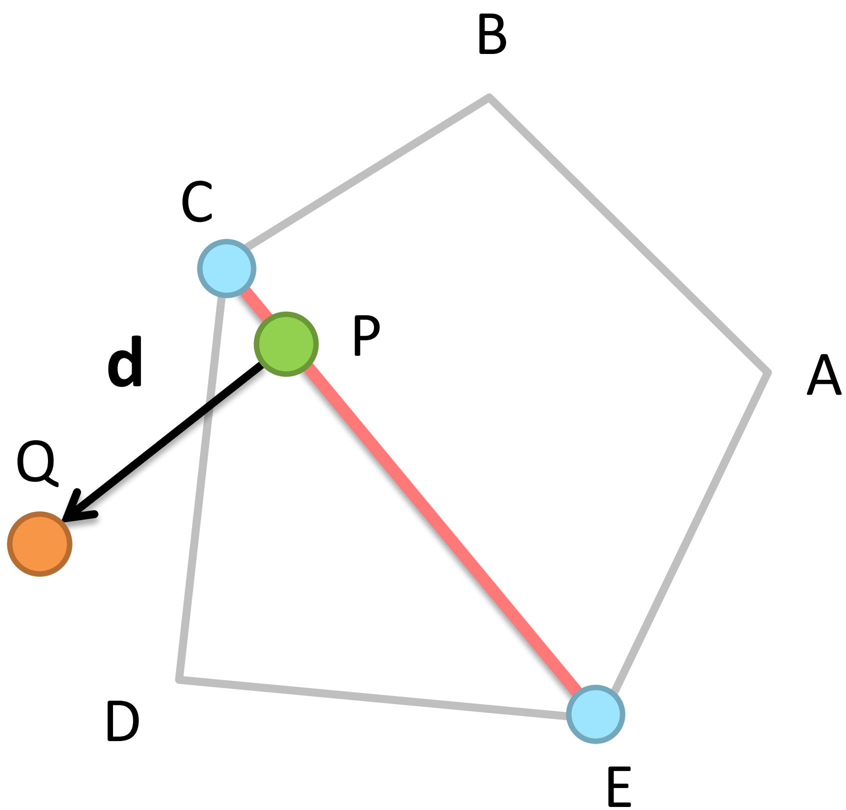
We now have a 1-simplex.

Repeat the process



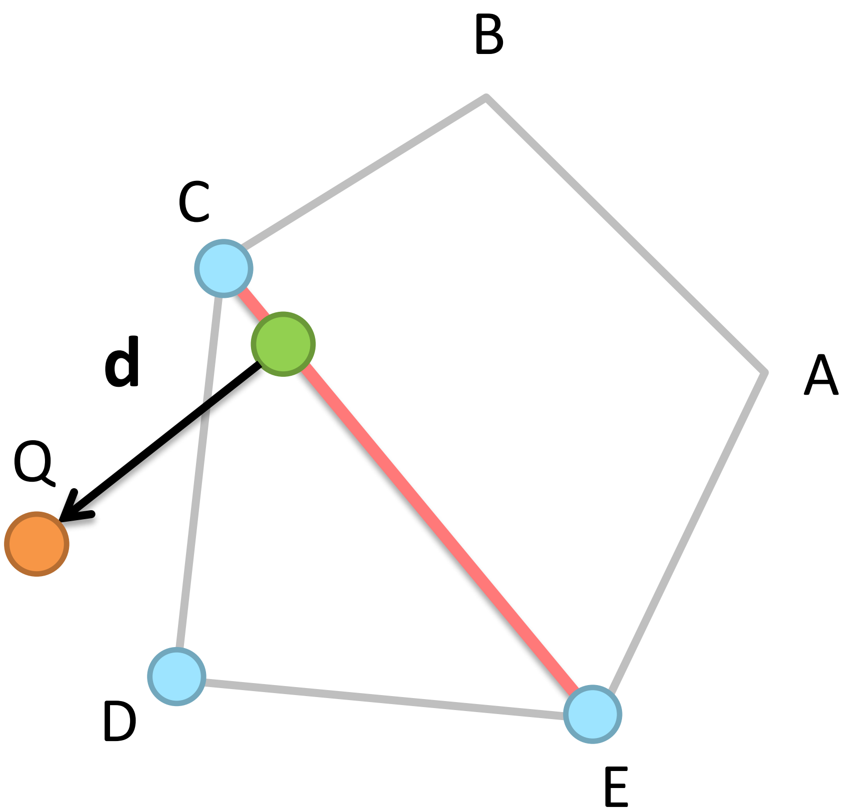
Find closest point P on CE.

New search direction



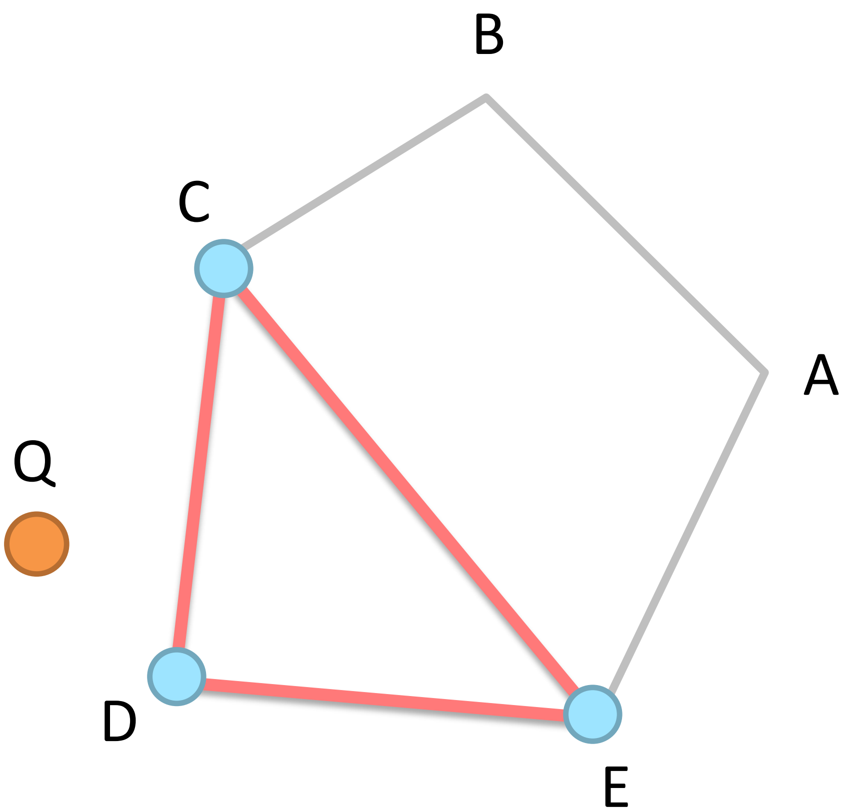
Build **d** as a line pointing from P to Q.

New support point



D is the support point.

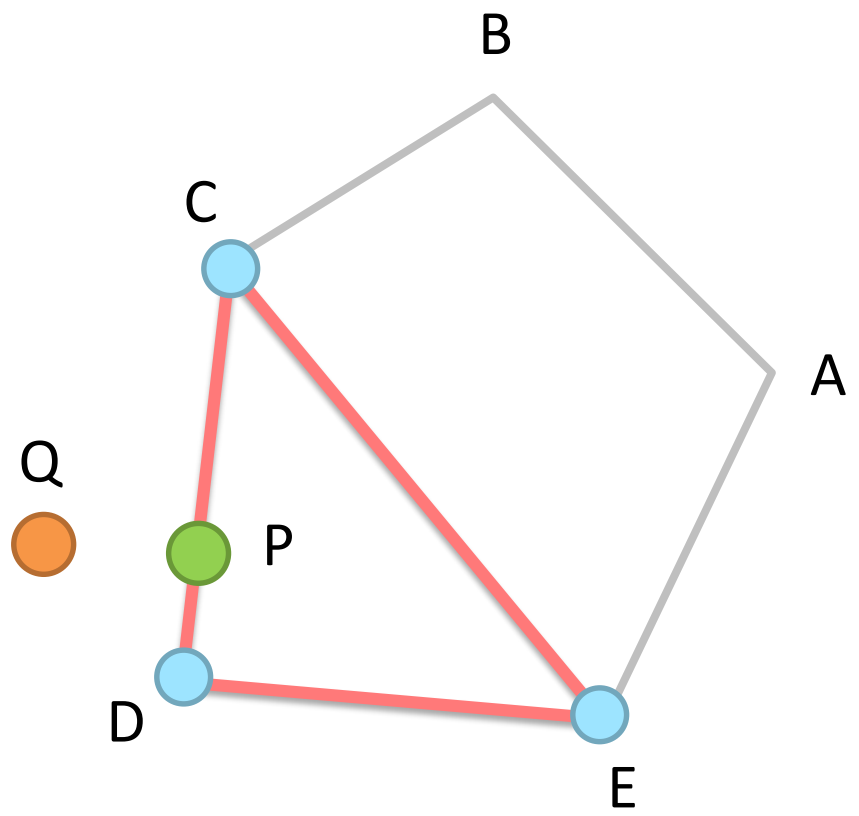
Evolve the simplex



Create triangle CDE.

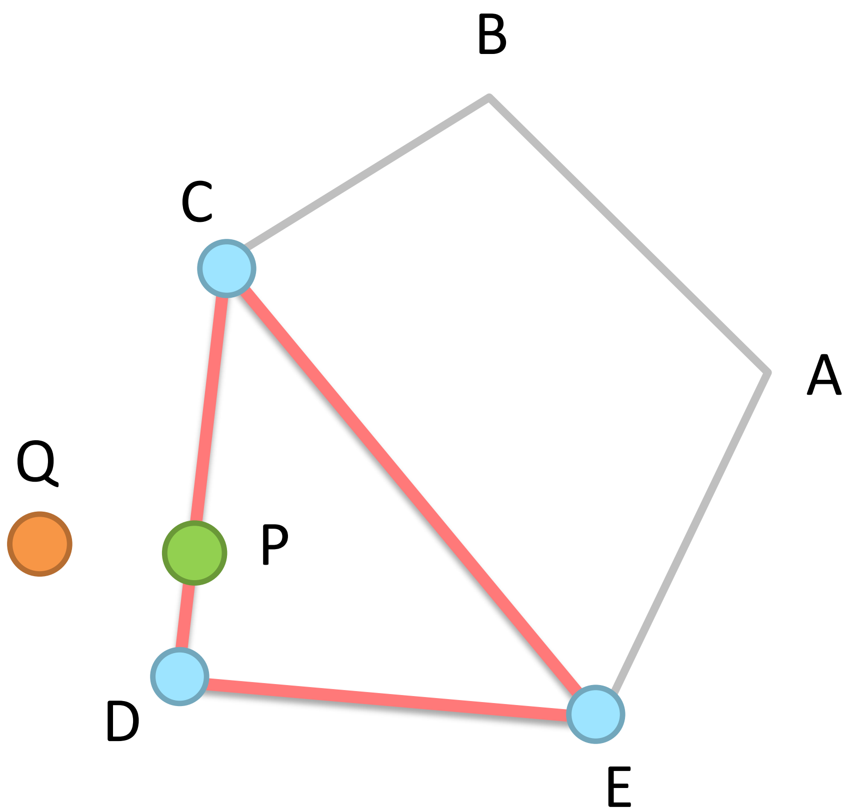
This is a
2-simplex.

Closest point



Compute closest point on CDE to Q.

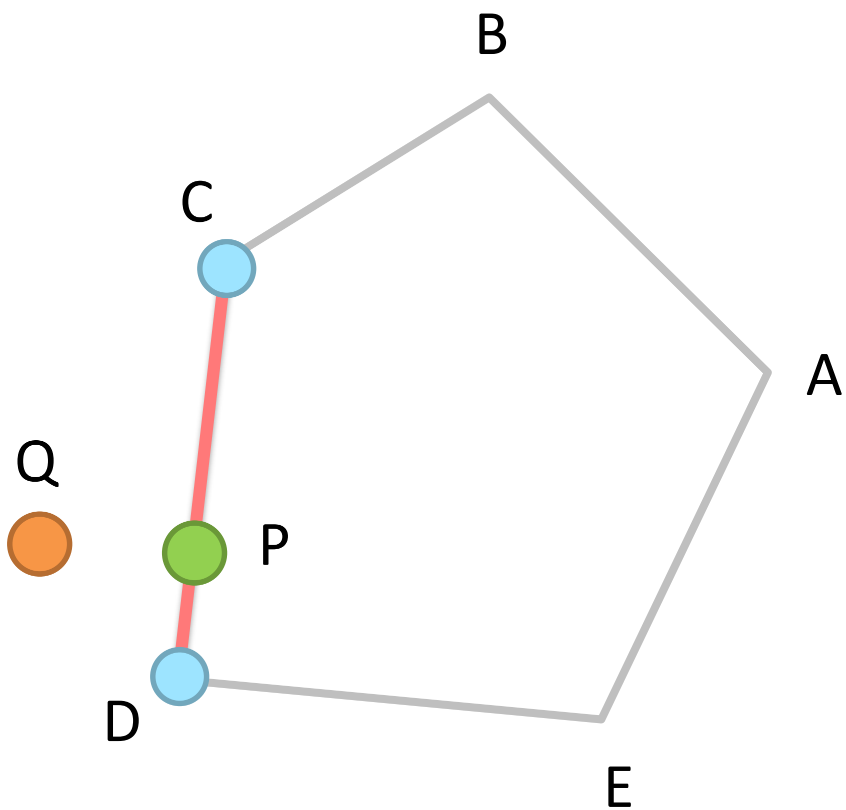
E is worthless



Closest point is on CD.

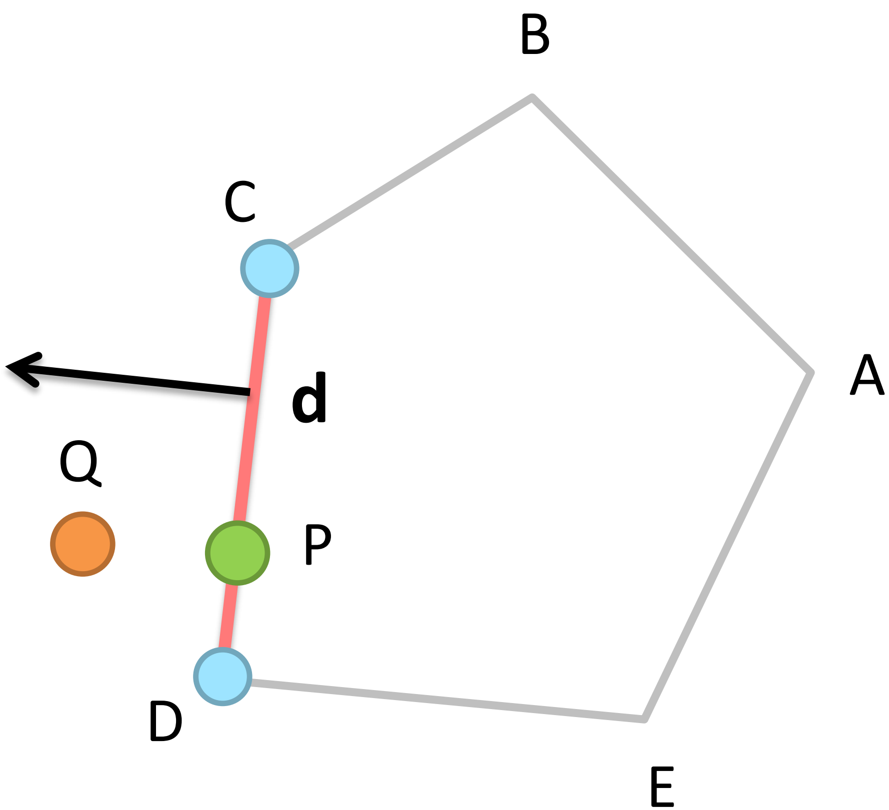
E does not *contribute*.

Reduced simplex



We dropped E,
so we now have
a 1-simplex.

Termination



Compute support point in direction **d**.

We find either C or D. Since this is a repeat, we are done.

GJK algorithm

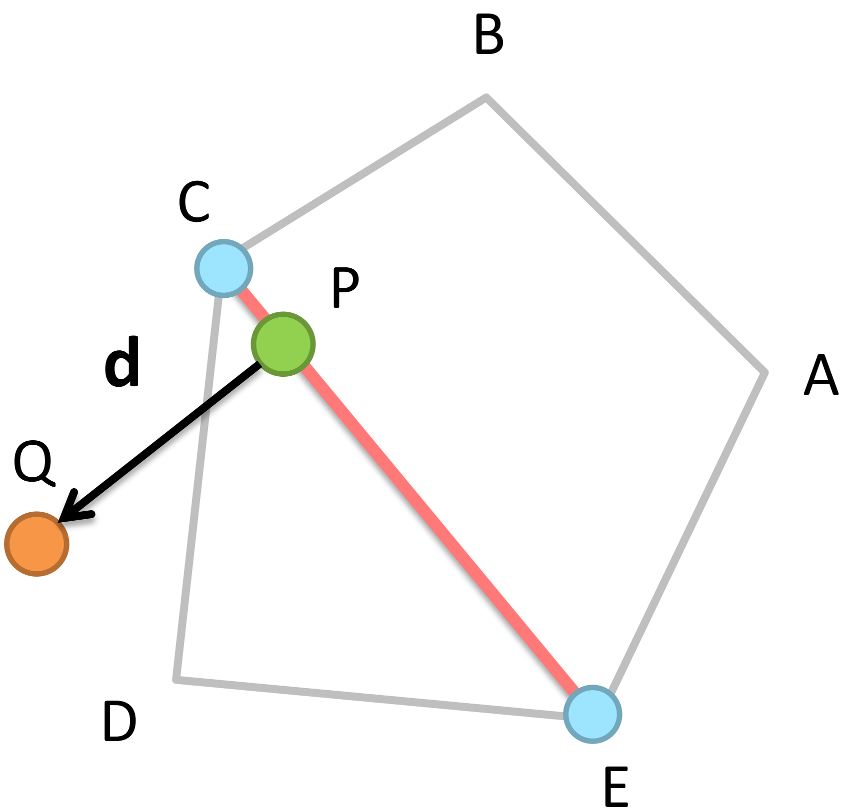
```
Input: polygon and point Q
pick arbitrary initial simplex S
loop
  compute closest point P on S
  cull non-contributing vertices from S
  build vector d pointing from P to Q
  compute support point in direction d
  add support point to S
end
```

DEMO!!!

Numerical Issues

- Search direction
- Termination
- Poorly formed polygons

A bad direction



d can be built from PQ.

Due to round-off:

dot(Q-P, C-E) != 0

A real example in single precision

Line Segment
A = [0.021119118, 79.584320]
B = [0.020964622, -31.515678]

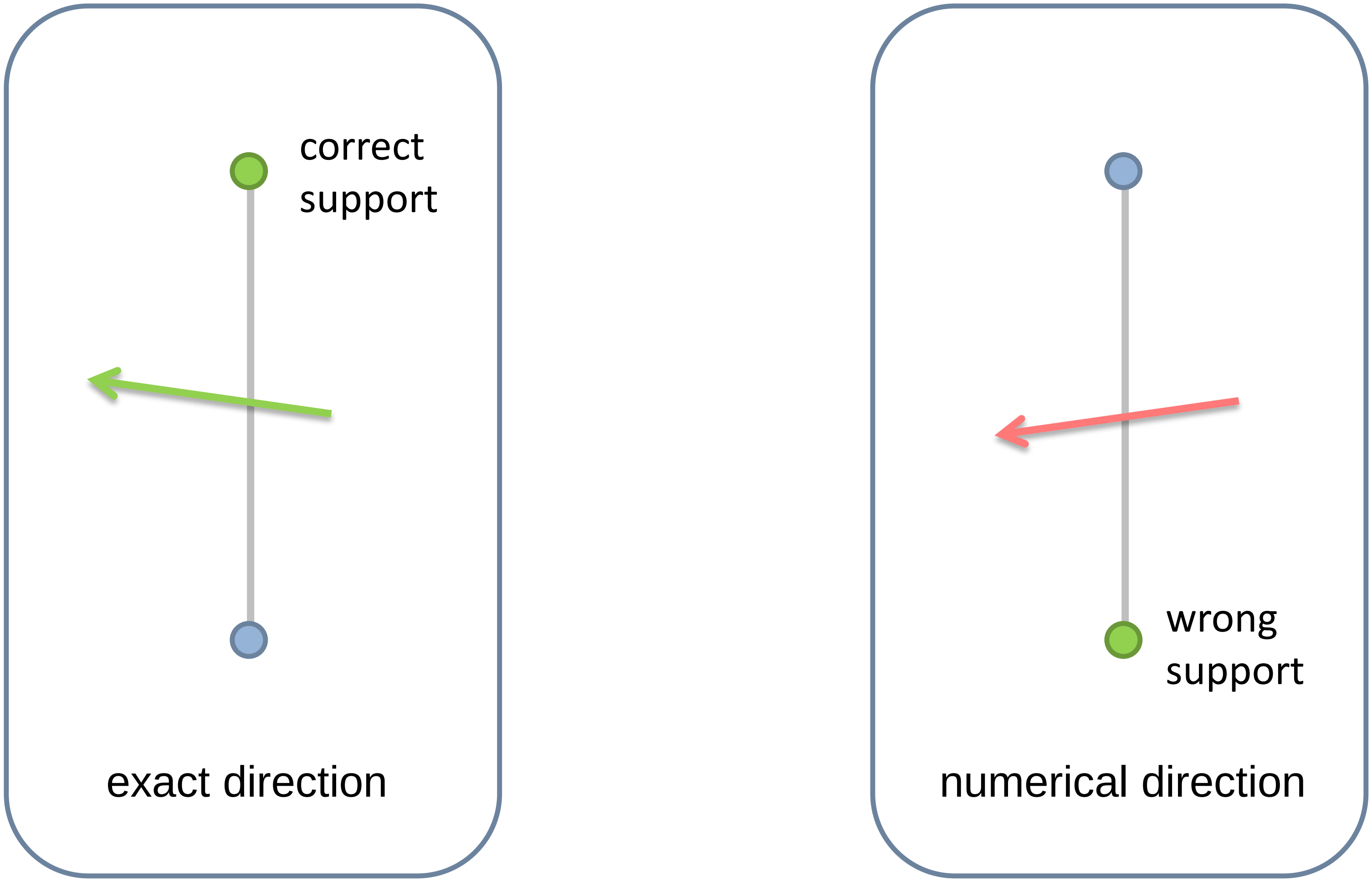
Query Point
Q = [0.0 0.0]

Barycentric Coordinates
(u, v) = (0.28366947, 0.71633047)

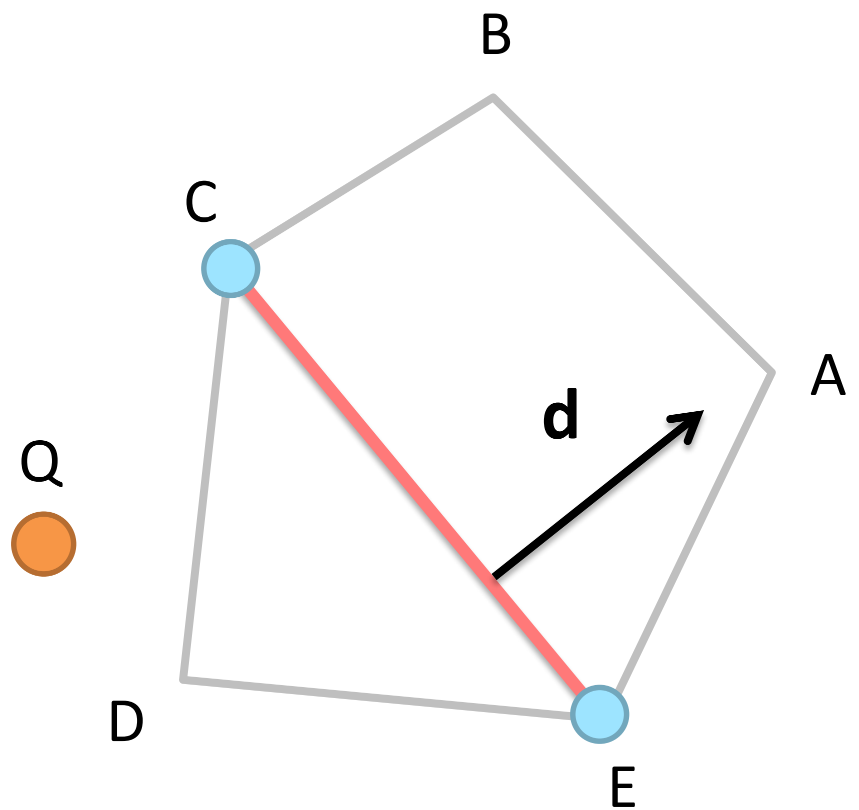
Search Direction
d = Q – P = [-0.021008447, 0.0]

dot(d, B – A) = 3.2457051e-006

Small errors matter



An accurate search direction



Directly compute a vector perpendicular to CE.

d = cross(C-E,**z**)

Where **z** is normal to the plane.

The dot product is exactly zero

edge direction:

search direction:

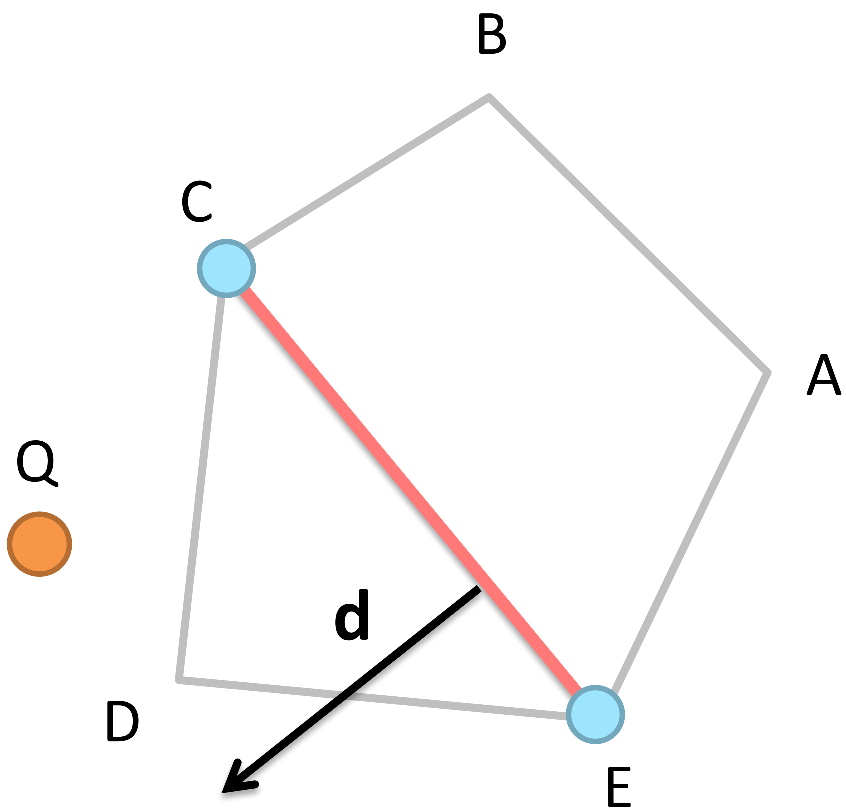
dot product:

$$\mathbf{e} = \begin{pmatrix} x & y \end{pmatrix}$$

$$\mathbf{d} = \begin{pmatrix} -y & x \end{pmatrix}$$

$$\mathbf{e} \cdot \mathbf{d} = -xy+yx=0$$

Fixing the sign



Flip the sign of **d** so that:

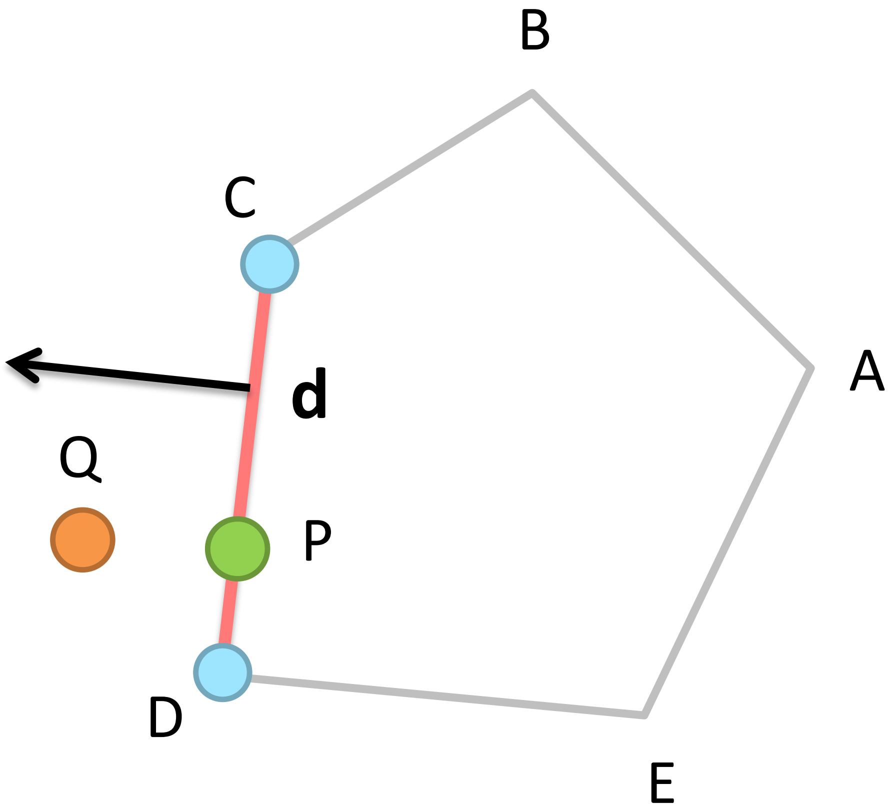
dot(d, Q – C) > 0

Perk: no divides

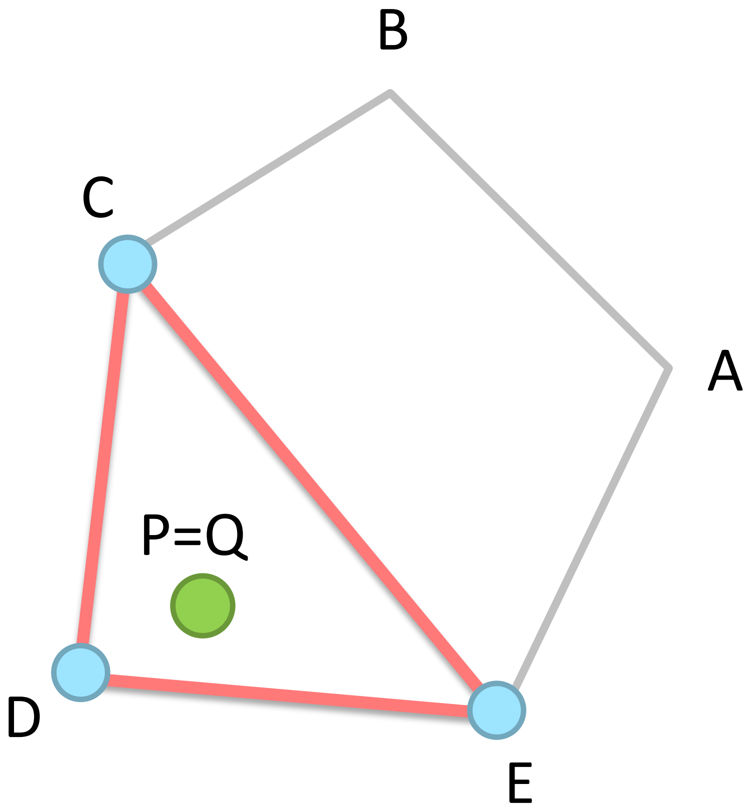
Termination conditions

}

Case 1: repeated support point

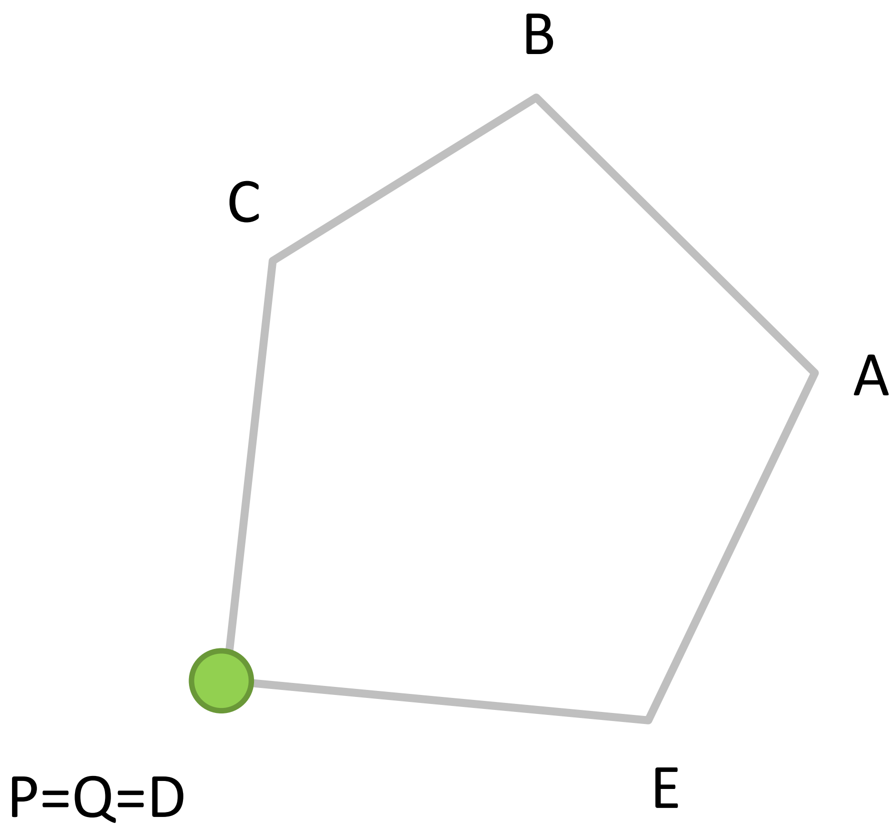


Case 2: containment



We find a 2-simplex and all vertices contribute.

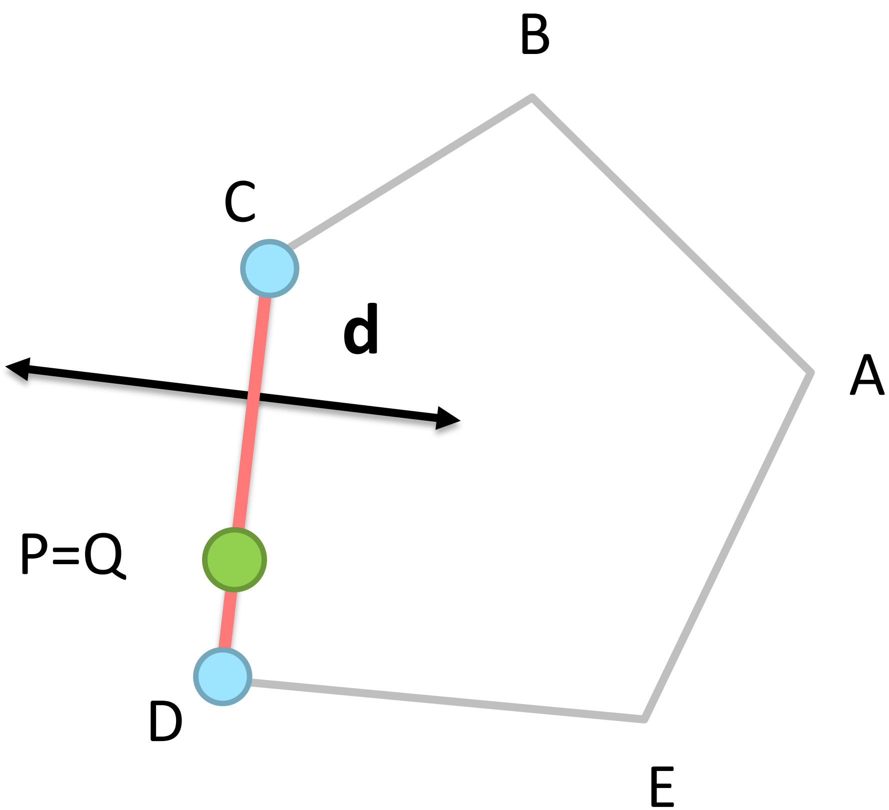
Case 3a: vertex overlap



We will compute **d**=Q-P as zero.

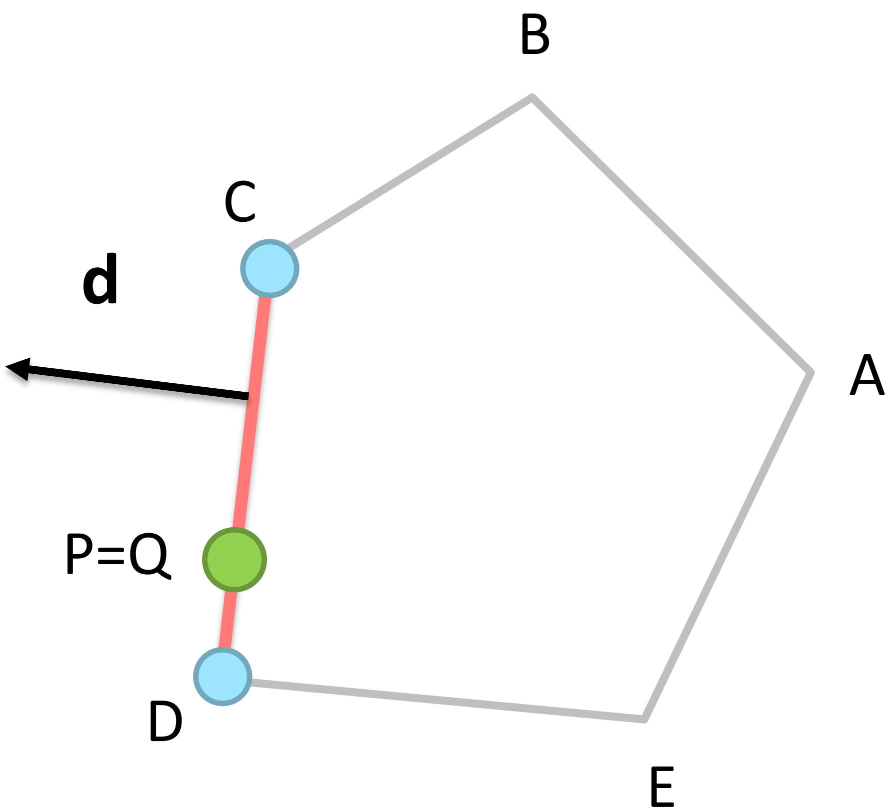
So we terminate if **d**=**0**.

Case 3b: edge overlap



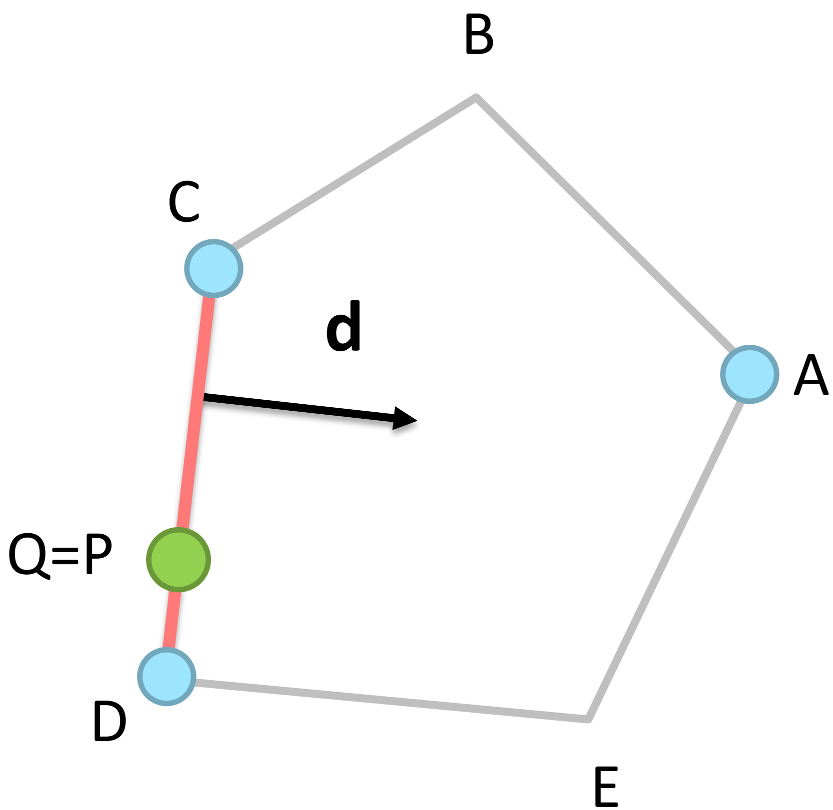
d will have an arbitrary sign.

Case 3b: d points left



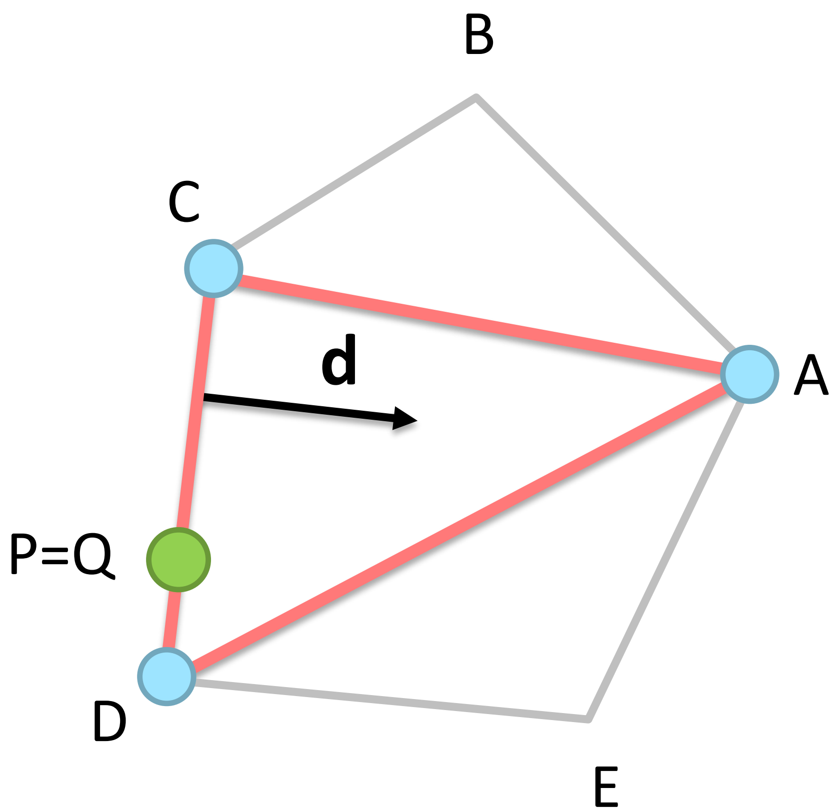
If we search left, we get a duplicate support point.
In this case we terminate.

Case 3b: d points right



If we search right, we get a new support point (A).

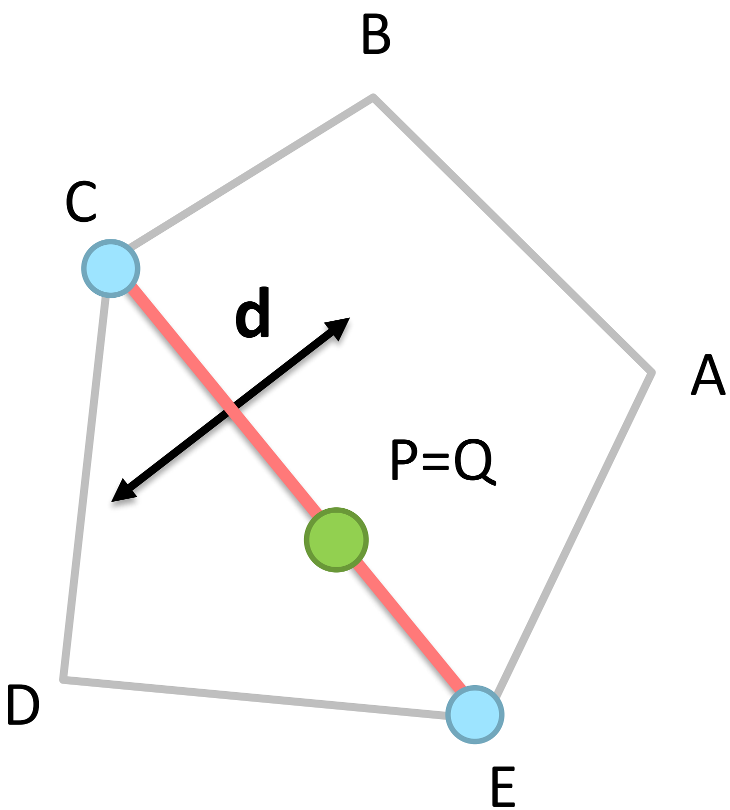
Case 3b: d points right



But then we get back the same P, and then the same **d**.

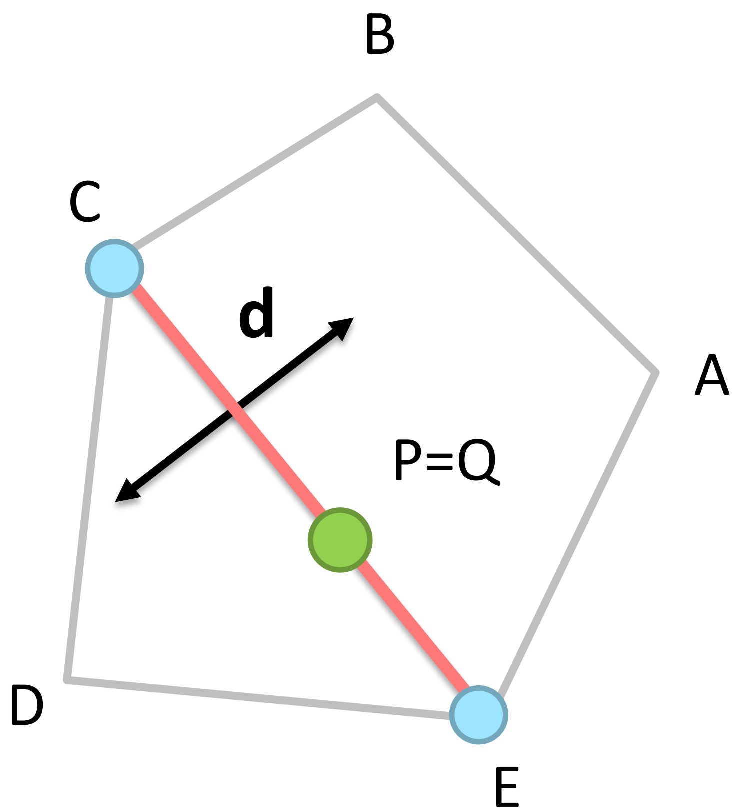
Soon, we detect a repeated support point or detect containment.

Case 4: interior edge



d will have an arbitrary sign.

Case 4: interior edge

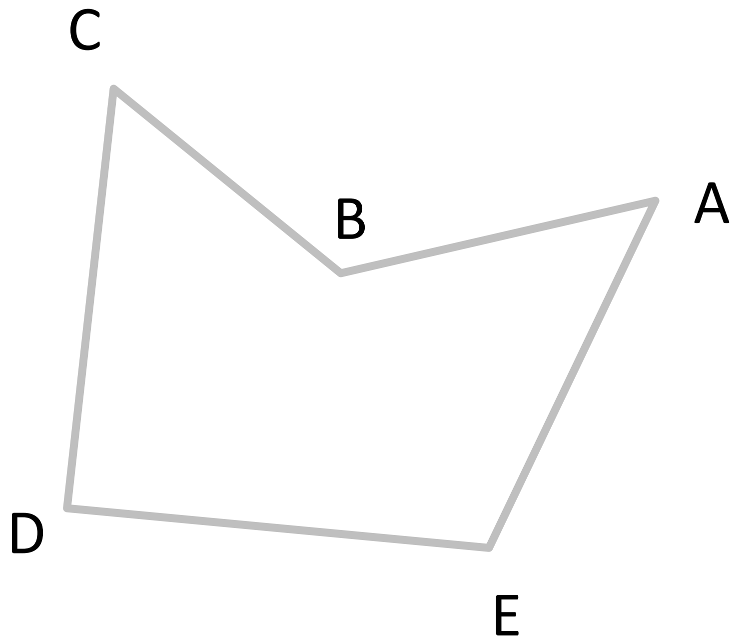


Similar to Case 3b

Termination in 3D

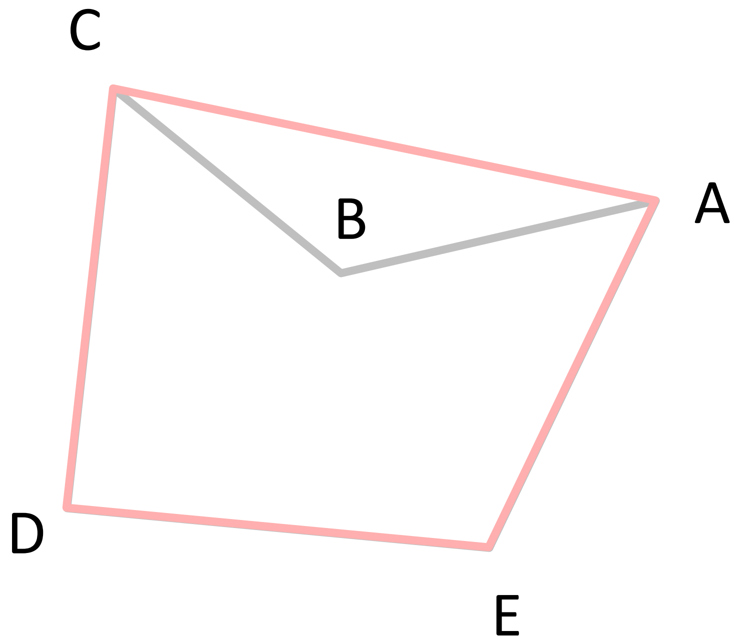
- May require new/different conditions
- Check for distance progression

Non-convex polygon



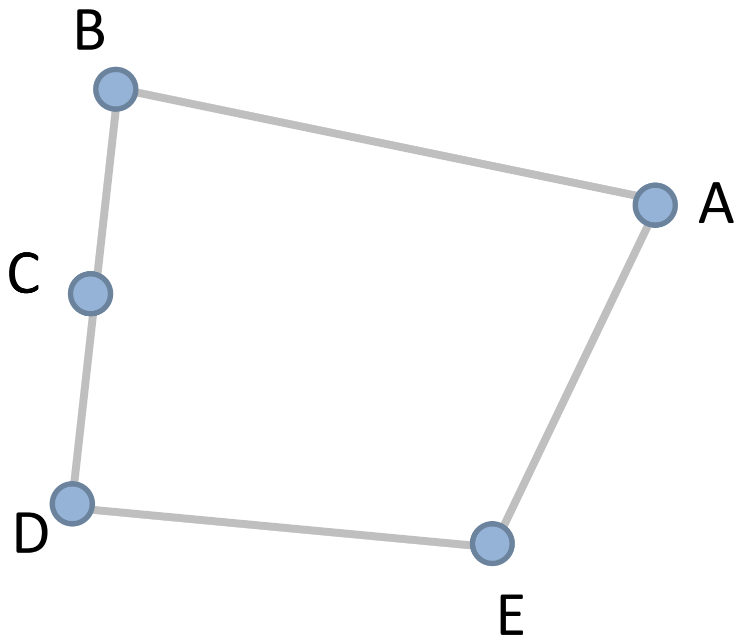
Vertex B is non-convex

Non-convex polygon



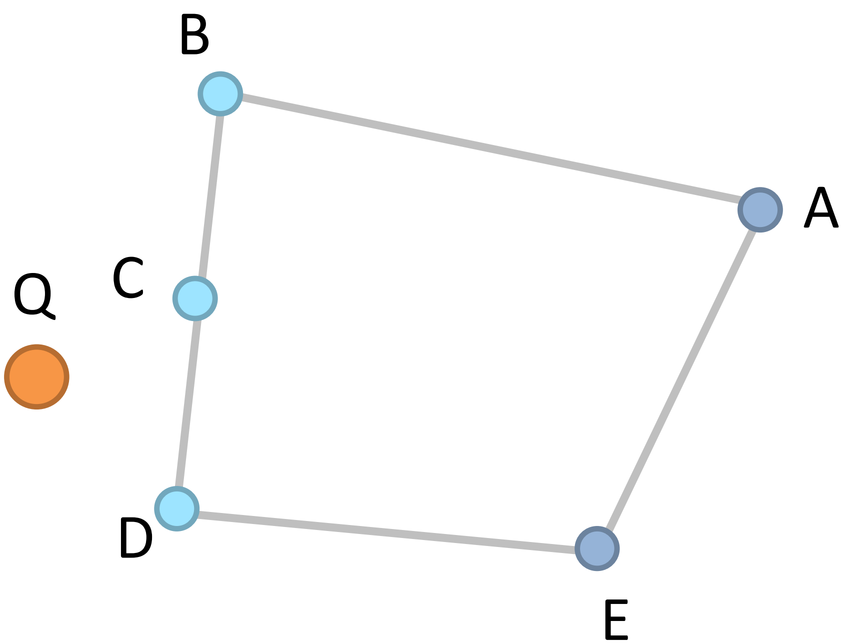
B is never a support point

Collinear vertices



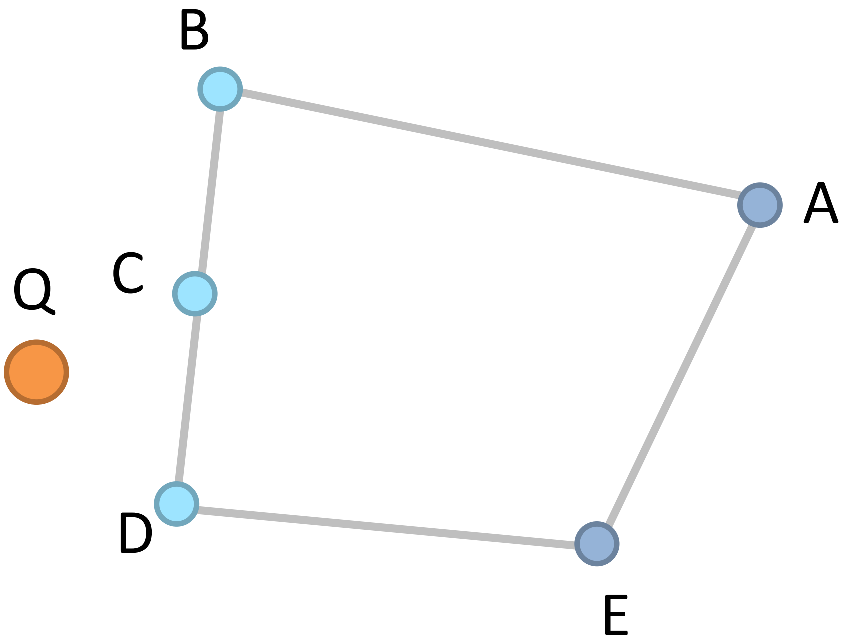
B, C, and D are collinear

Collinear vertices



2-simplex BCD

Collinear vertices

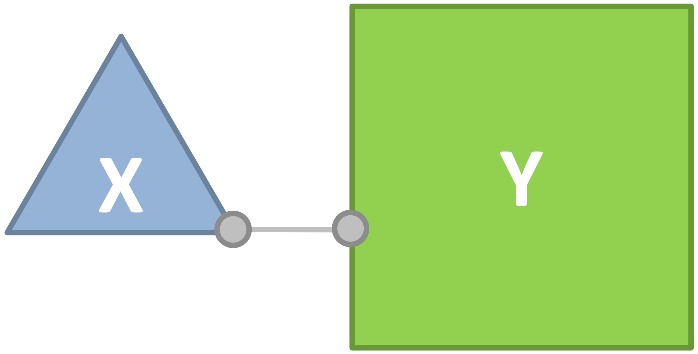


area(BCD) = 0

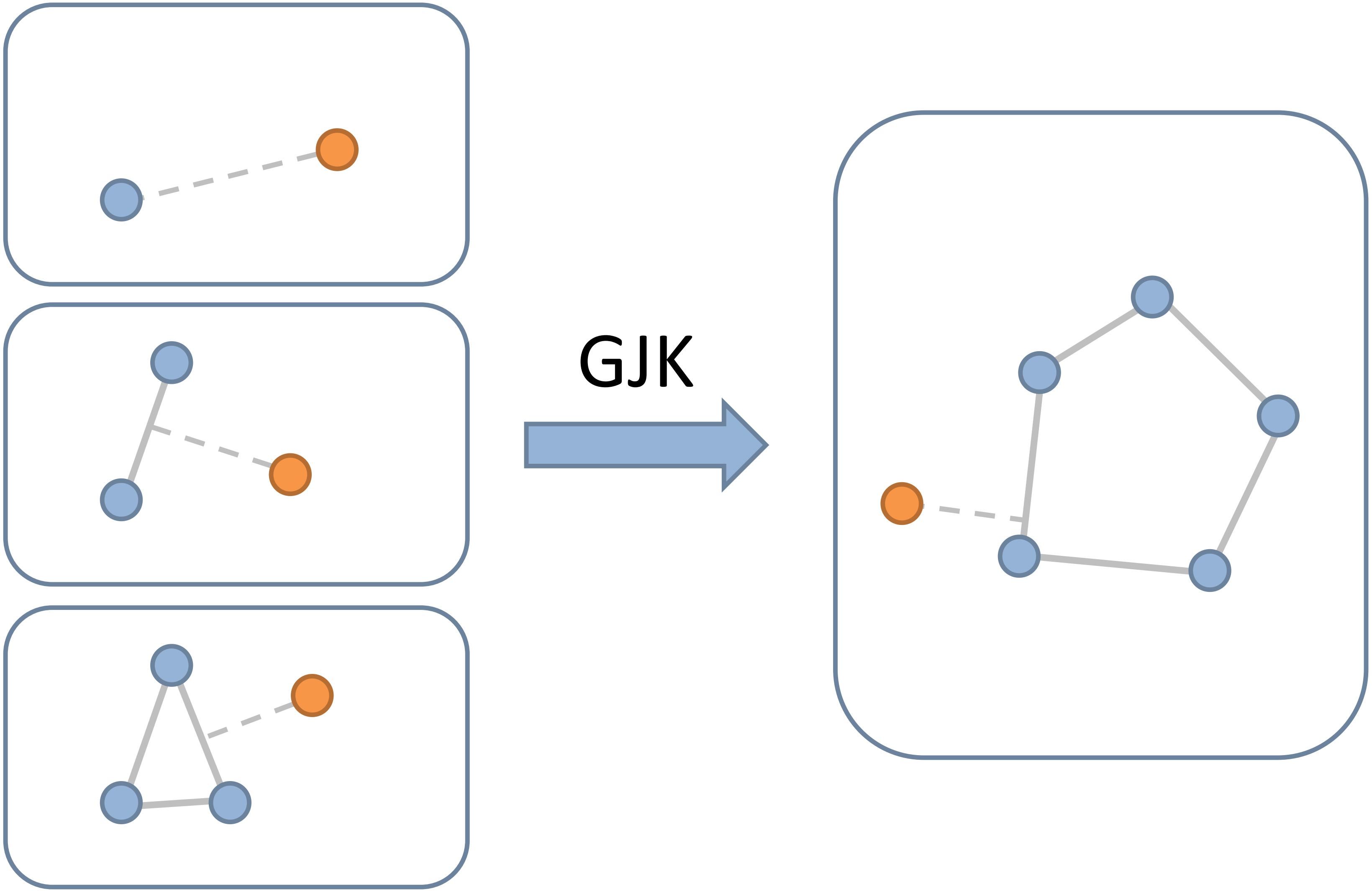
Section 4

Convex Polygon to Convex Polygon

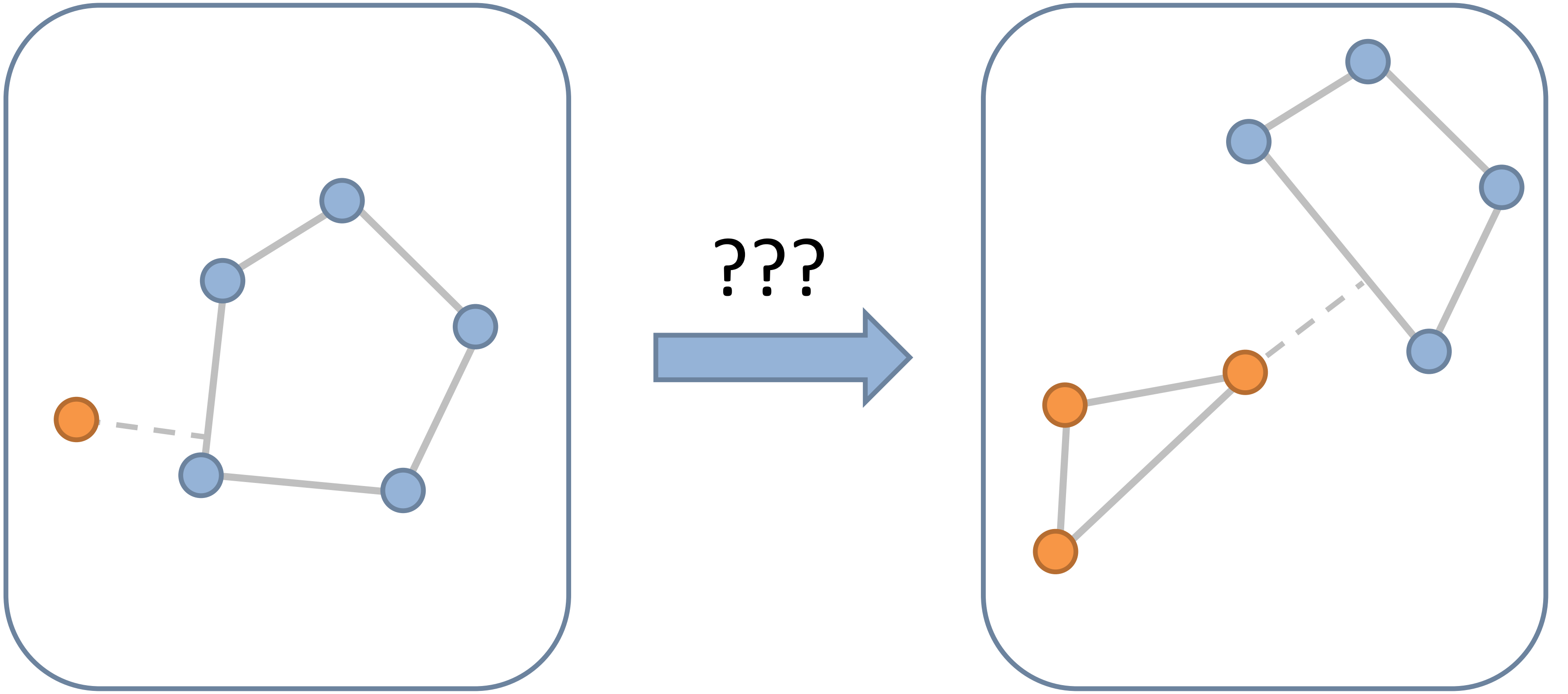
Closest point between convex polygons



What do we know?



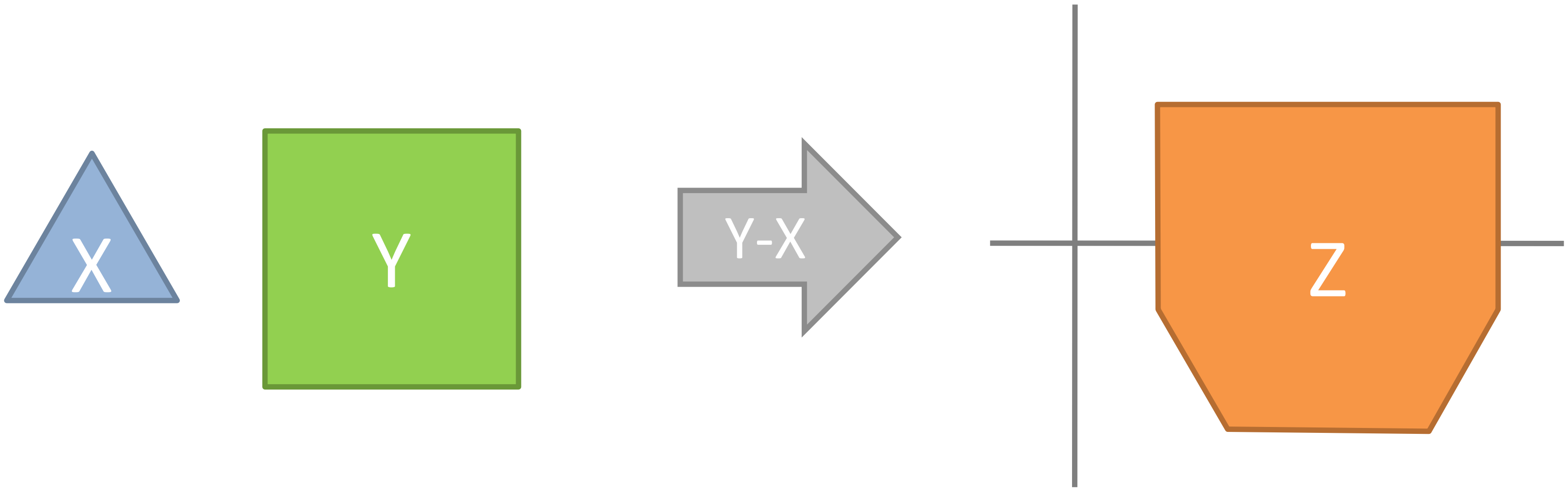
What do we need to know?



Idea

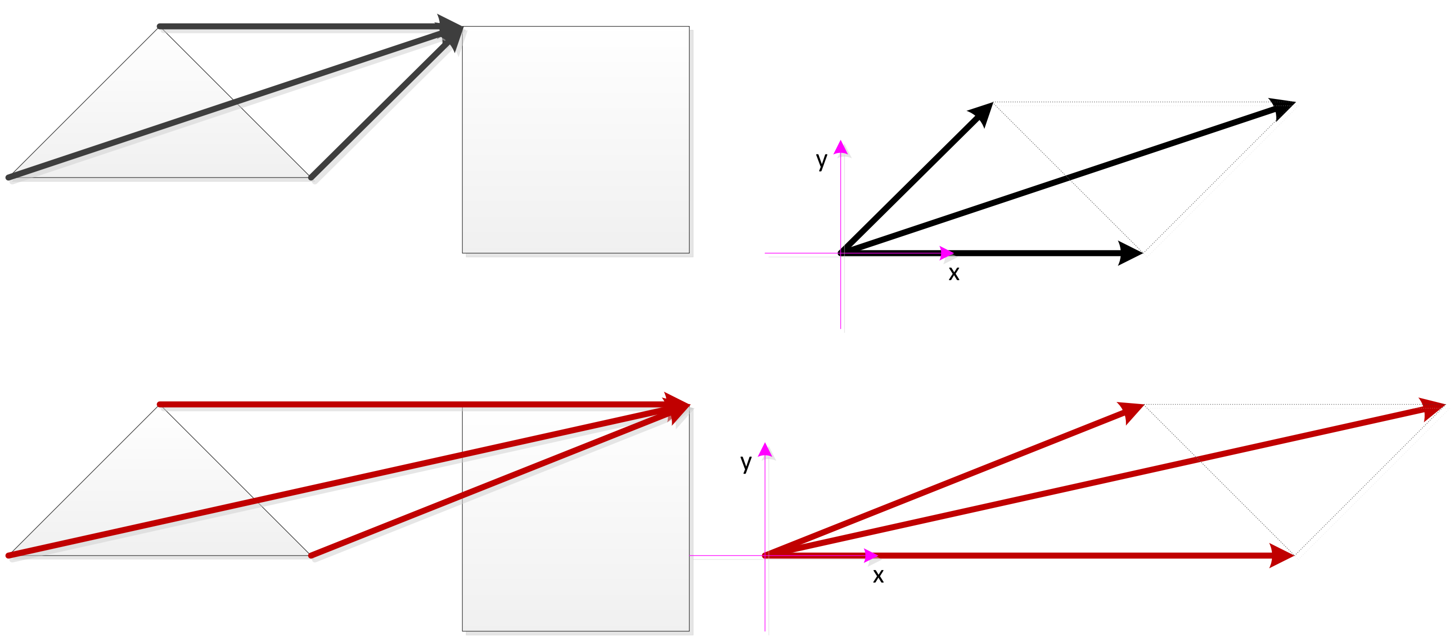
- Convert polygon to polygon into point to polygon
- Use GJK to solve point to polygon

Minkowski difference

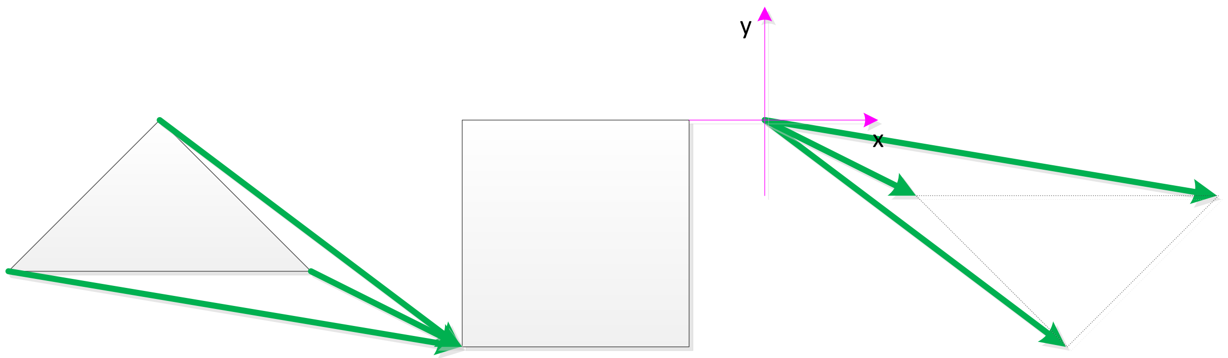


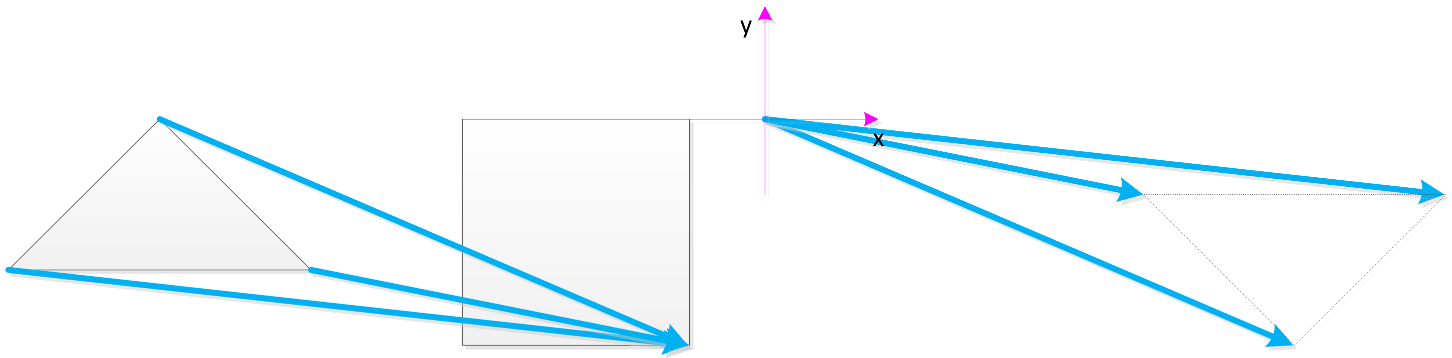
Minkowski difference definition

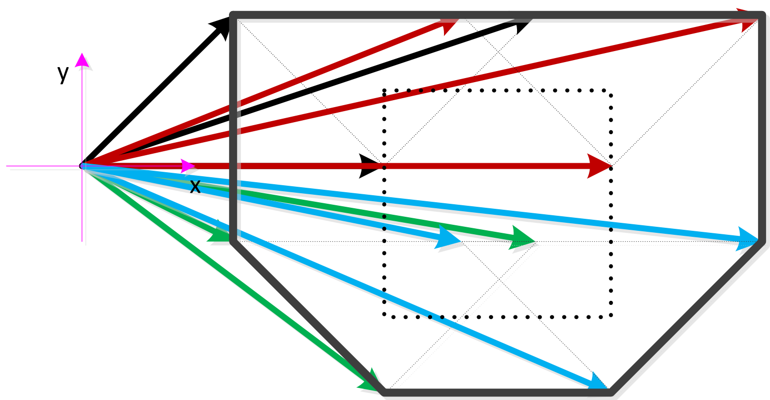
$$Z = \left\{ y_j - x_i : x_i \in X, y_j \in Y \right\}$$

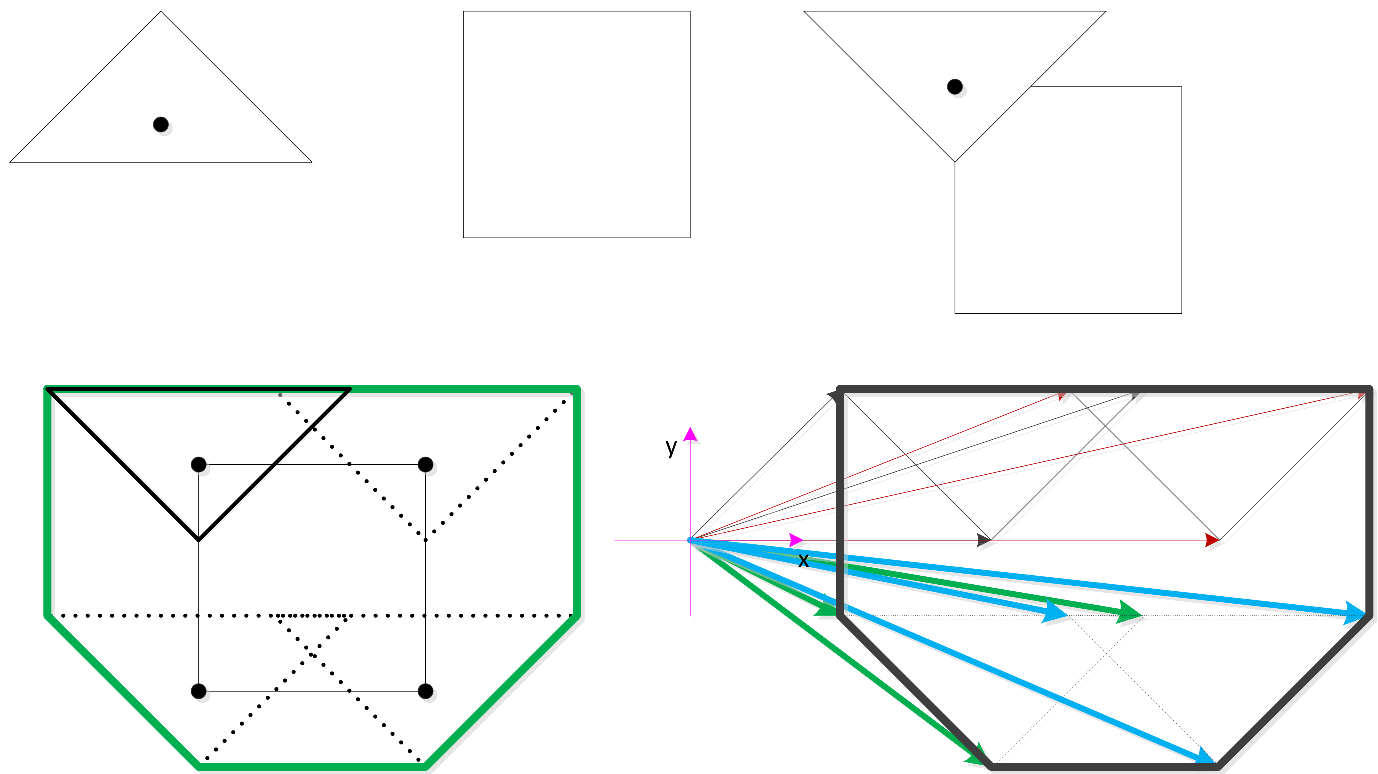


y
x
y
x


y
x


y
x

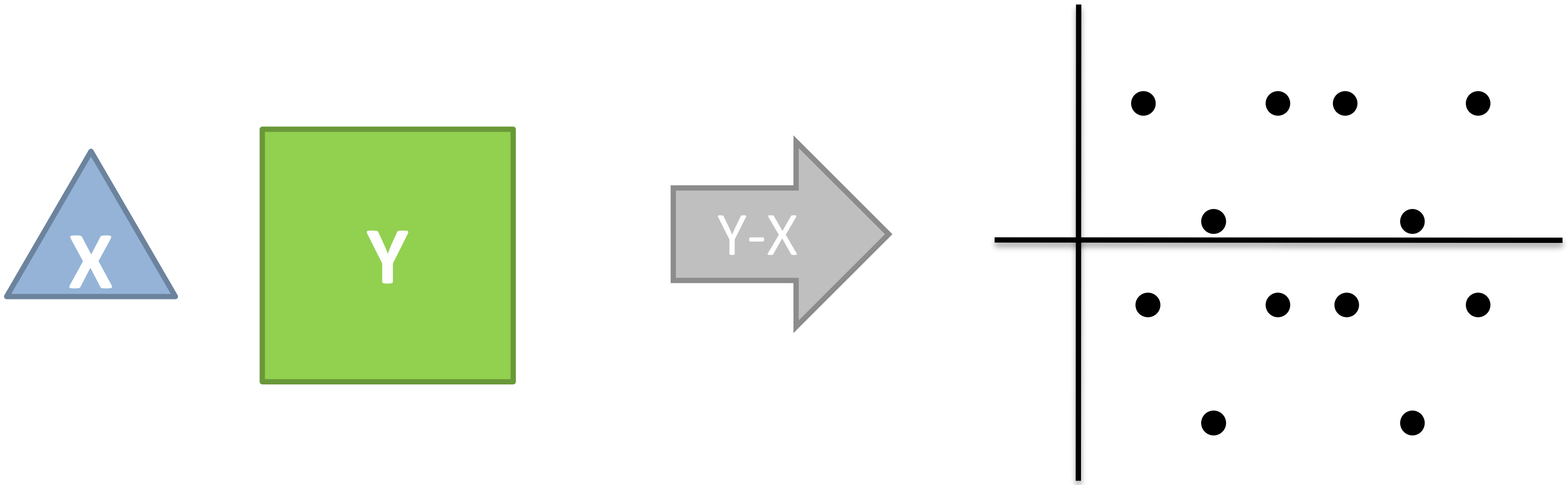

y
x


y
x

Building the Minkowski difference

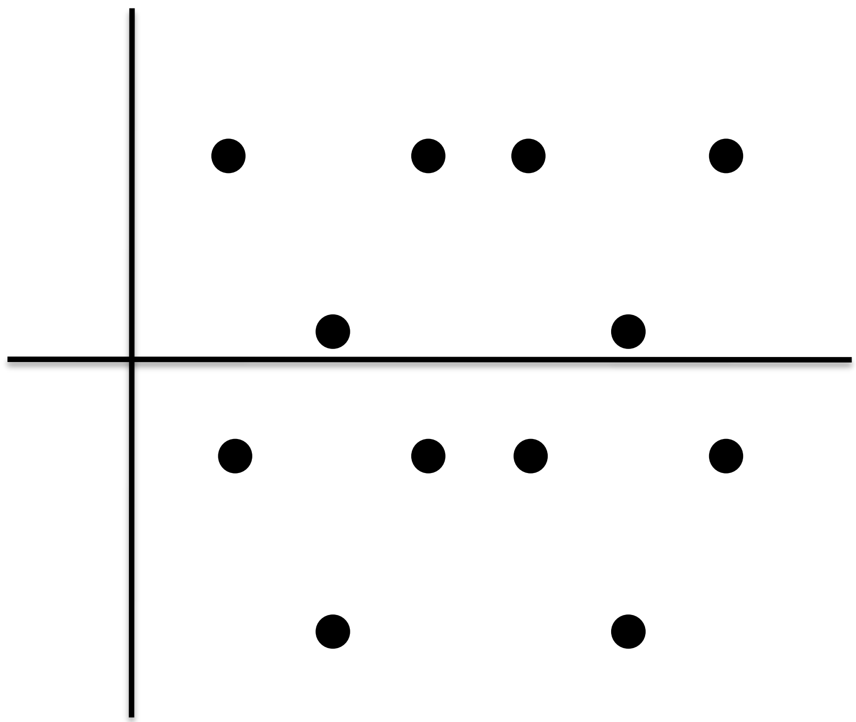
```
Input: polygon X and Y
array points
for all xi in X
  for all yj in Y
    points.push_back(yj – xi)
  end
end

polygon Z = ConvexHull(points)
```

Example point cloud



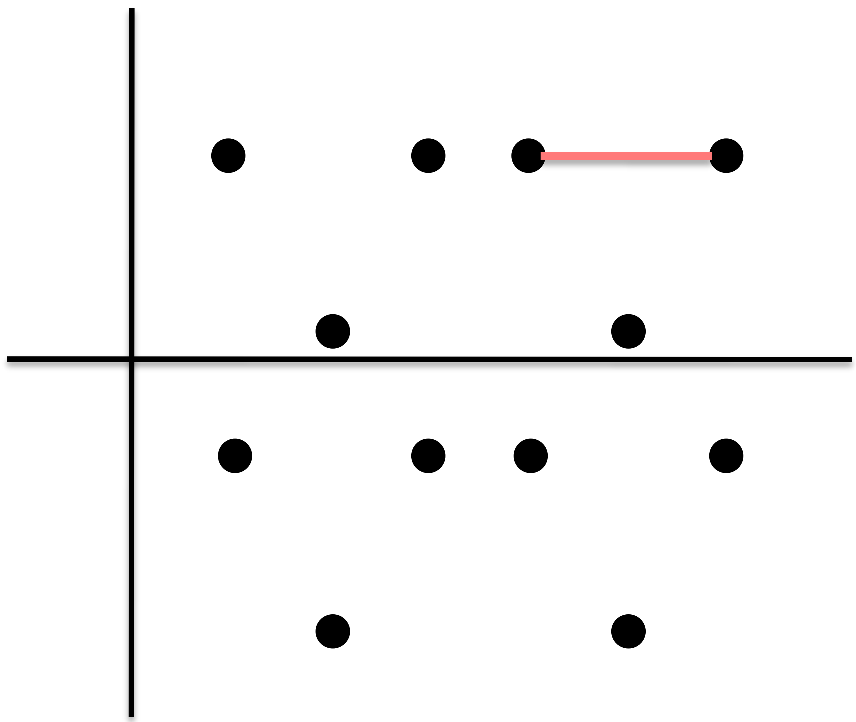
Compute Yi – Xj for i = 1 to 4 and j = 1 to 3

Building the convex hull



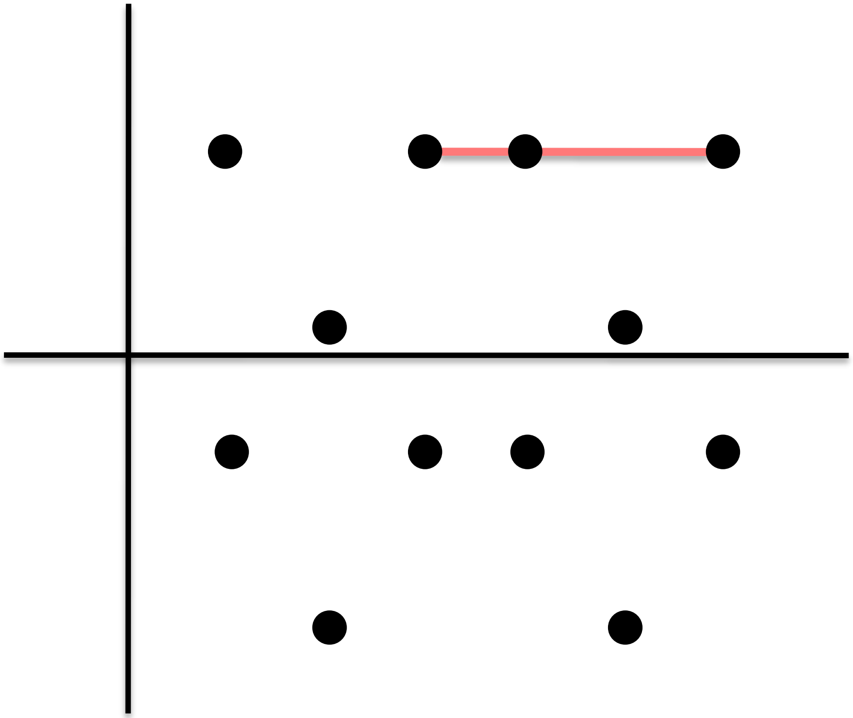
Compute the convex hull by shrink wrapping the points.

Building the convex hull



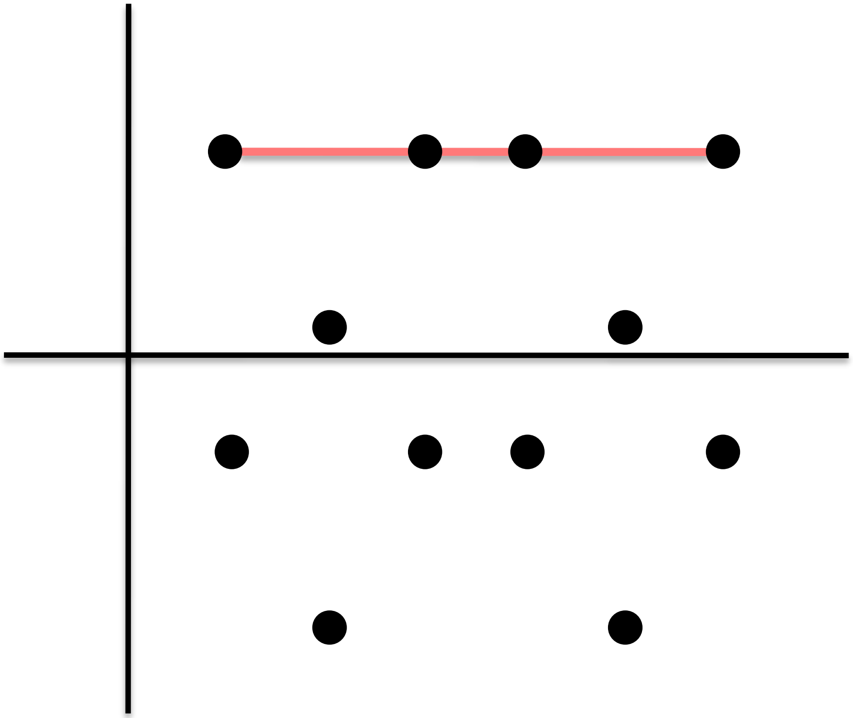
Compute the convex hull by shrink wrapping the points.

Building the convex hull



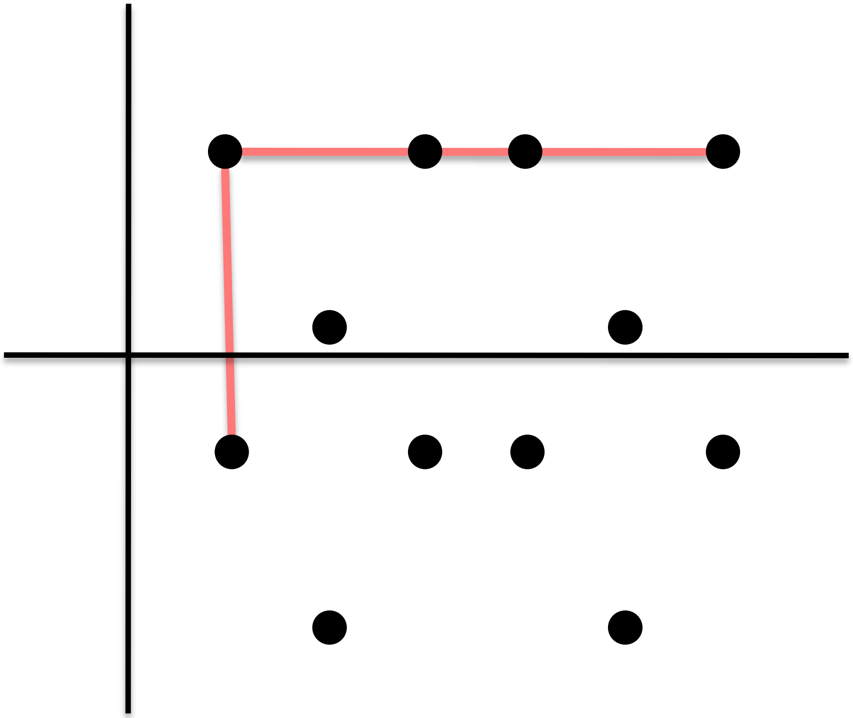
Compute the convex hull by shrink wrapping the points.

Building the convex hull



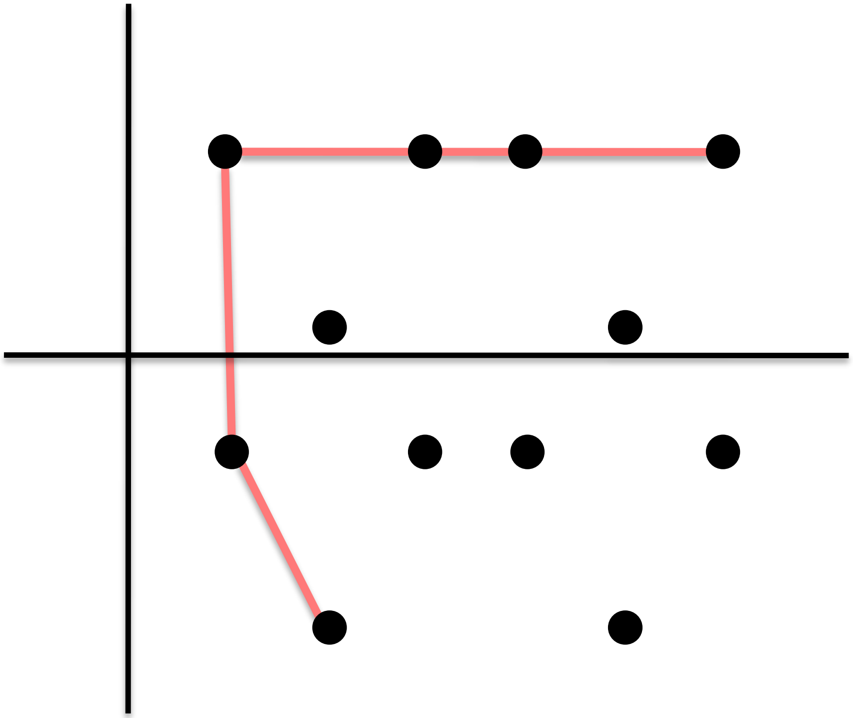
Compute the convex hull by shrink wrapping the points.

Building the convex hull



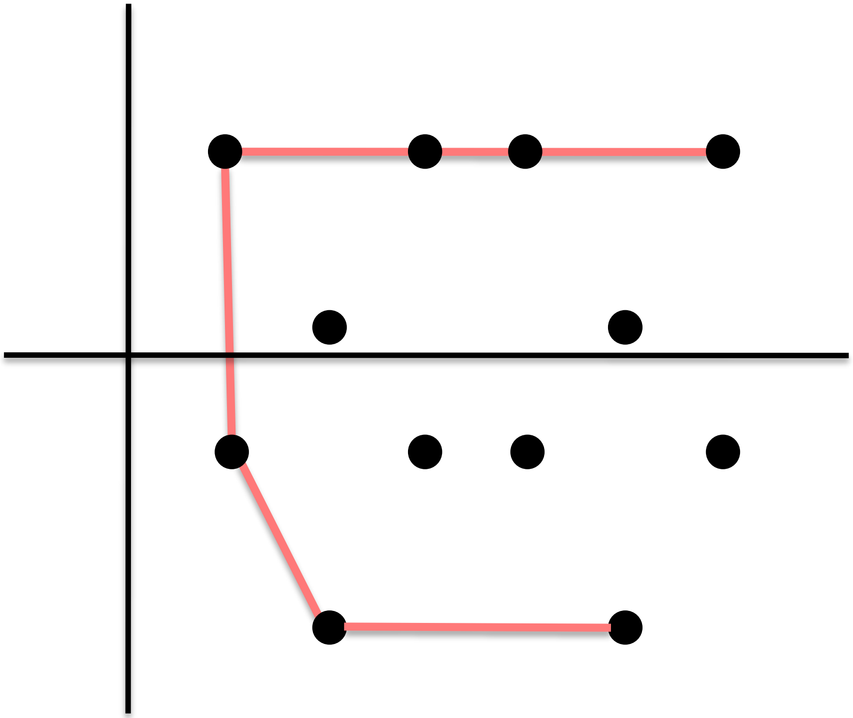
Compute the convex hull by shrink wrapping the points.

Building the convex hull



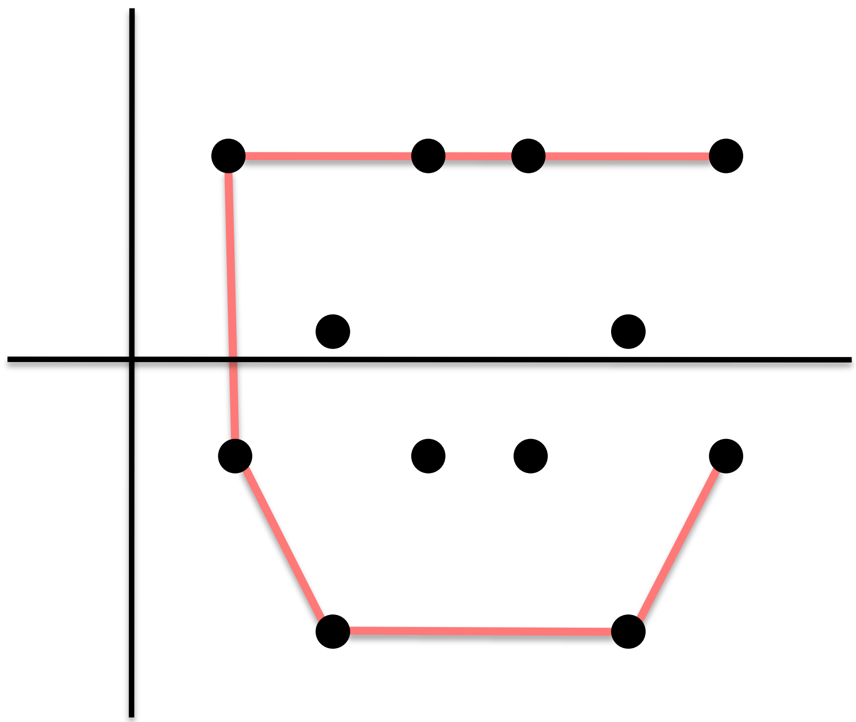
Compute the convex hull by shrink wrapping the points.

Building the convex hull



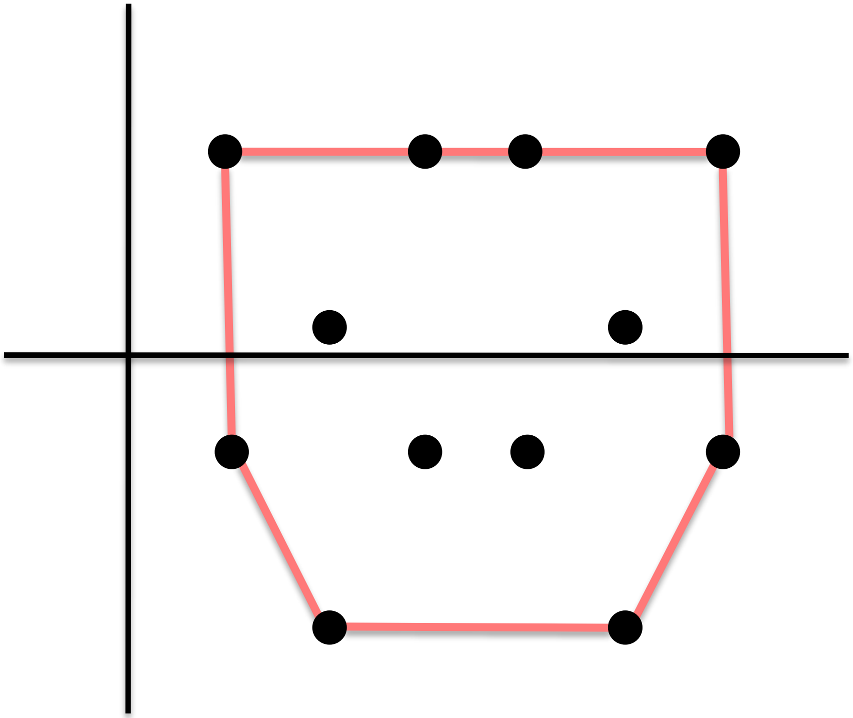
Compute the convex hull by shrink wrapping the points.

Building the convex hull



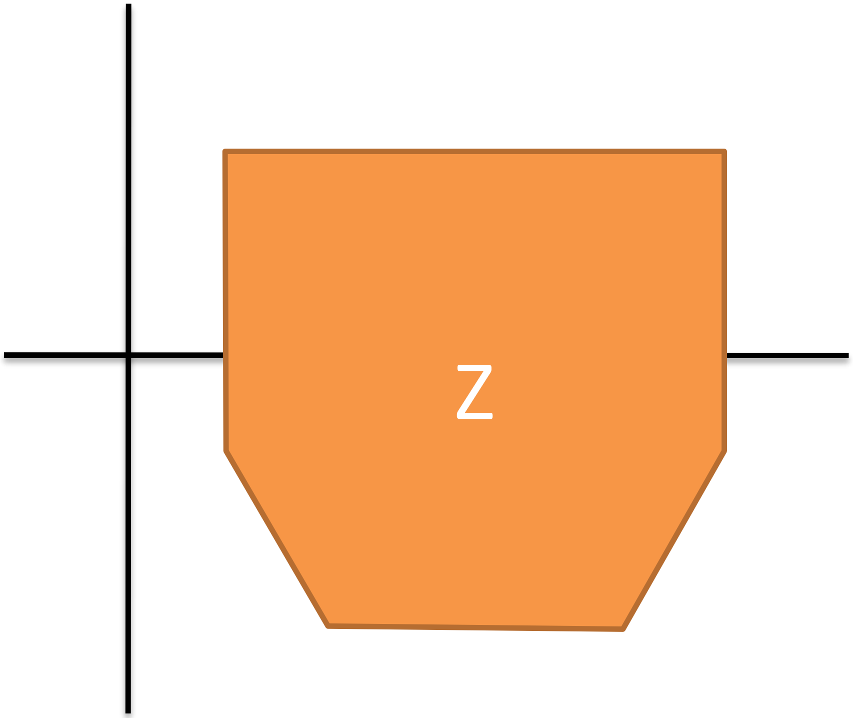
Compute the convex hull by shrink wrapping the points.

Building the convex hull



Compute the convex hull by shrink wrapping the points.

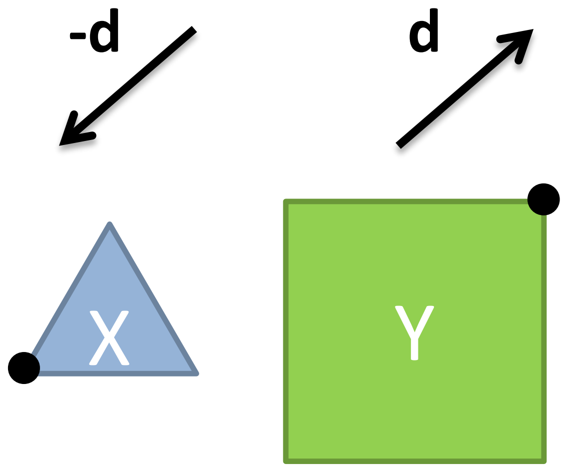
The final polygon

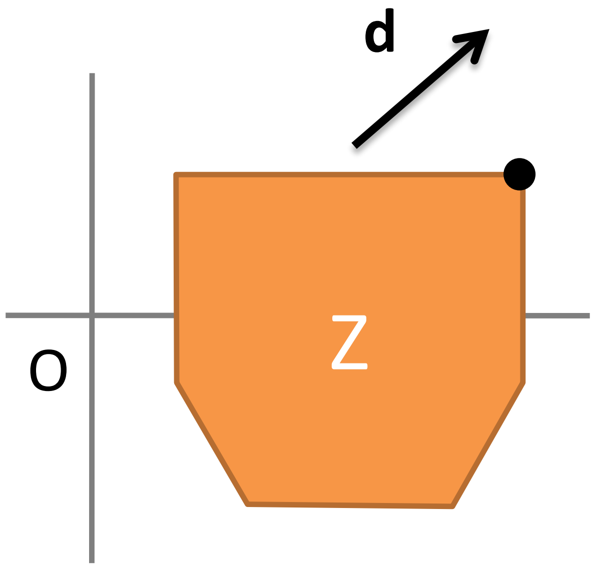


Property 1: distances are equal

distance(X,Y) == distance(O, Y-X)

Property 2: support points





support(Z, **d**) = support(Y, **d**) – support(X, -**d**)

~~Convex Hull?~~

Modifying GJK

- Change the support function
- Simplex vertices hold two indices

Closest point on polygons

- Use the barycentric coordinates to compute the closest points on X and Y
- See the demo code for details

DEMO!!!

Download: box2d.org

Further reading

- Collision Detection in Interactive 3D Environments, Gino van den Bergen, 2004
- Real-Time Collision Detection, Christer Ericson, 2005
- Implementing GJK: http://mollyrocket.com/849, Casey Muratori.

Box2D

- An open source 2D physics engine
- http://www.box2d.org
- Written in C++