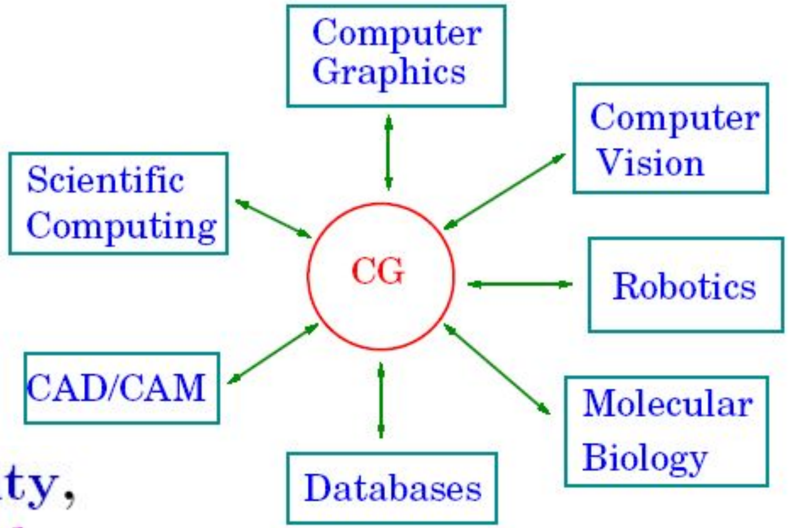
Computational Geometry

- Study of algorithms for geometric problems.
- Deals with discrete shapes: points, lines, polyhedra, polygonal meshes.
- Abstraction of problems in different applied areas.



What does that mean?

- Occlusion, visibility, augmented reality, collision detection, motion or assembly planning, drug design, databases, GIS, layout, fluid dynamics, etc.

Computational Geometry
Some basic algorithms

1. Given directed line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, determine whether $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$ with respect to point p_0?

2. Given two line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$, if we traverse $\overrightarrow{p_0p_1}$ and then $\overrightarrow{p_1p_2}$, do we make a left turn at point p_1?

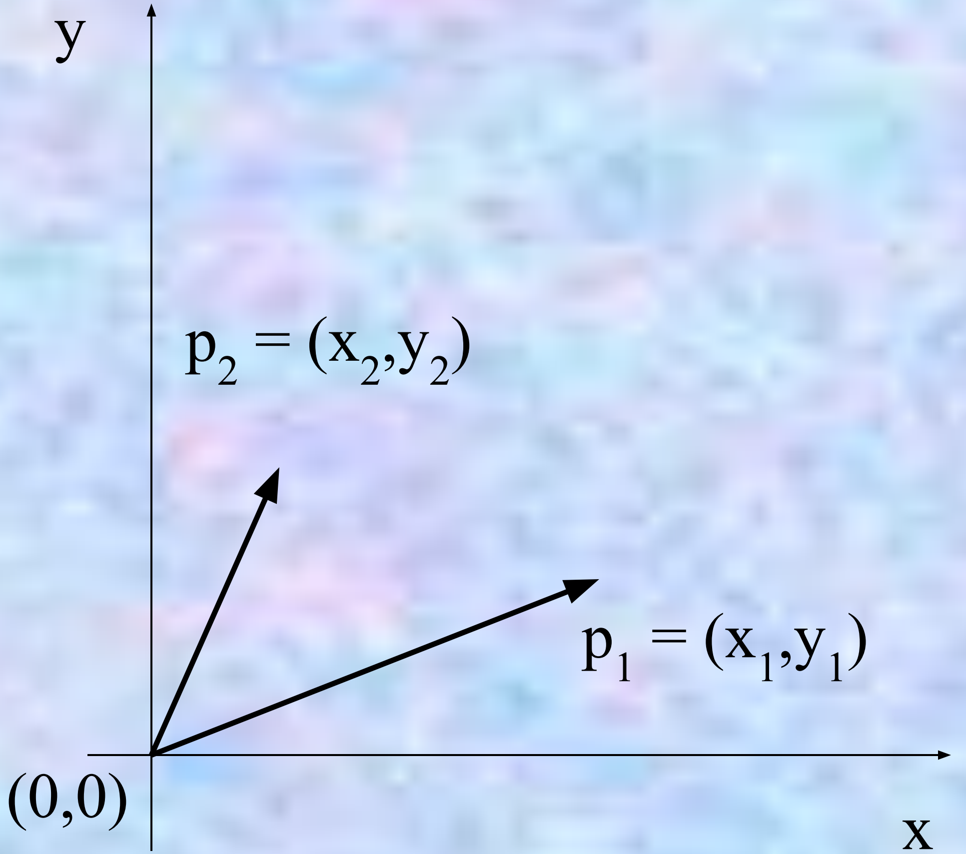
3. Do line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_2p_3}$ intersect?

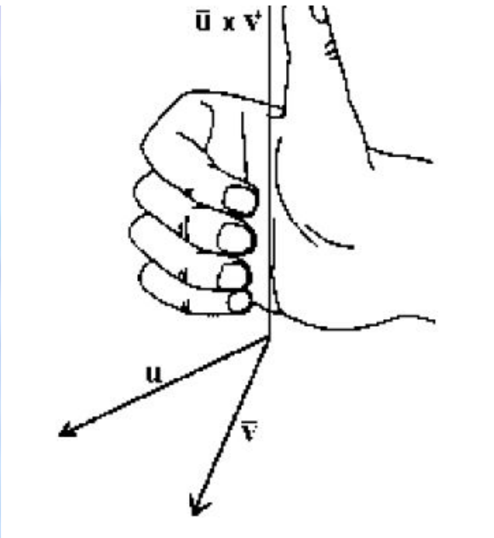
Cross Product

Given two vectors u and v, the **cross product** of u and v is a vector orthogonal to both u and v given by

$$u \mathbf{X} v = |u||v| \sin(\theta) \; n$$

where θ is the smallest angle between u and v and n is the unit vector perpendicular to both u and v, whose direction is given by the **right hand rule**.





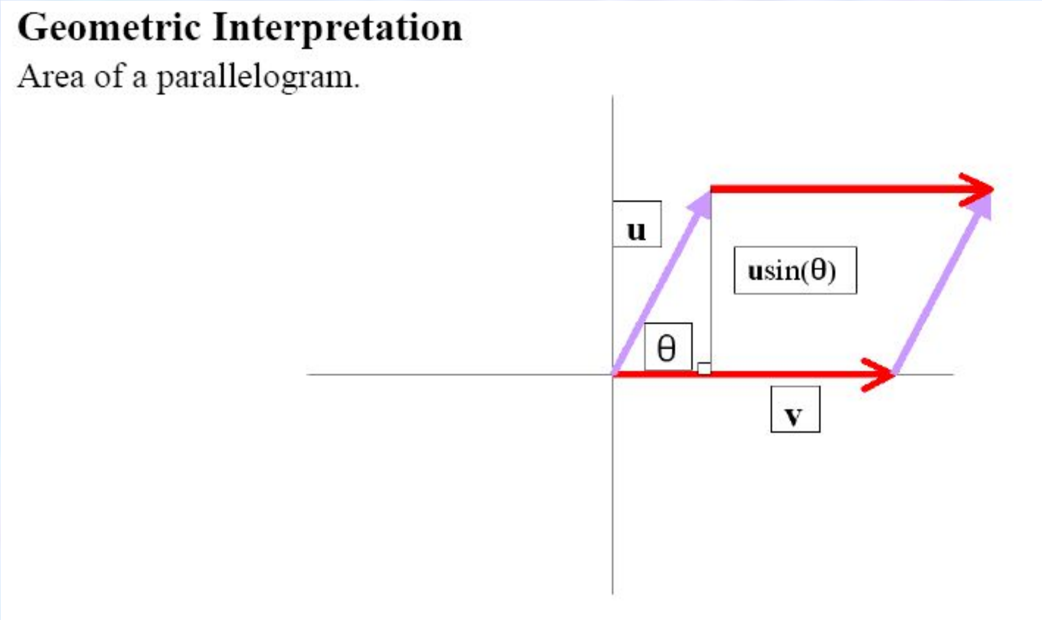
$$p_1 \times p_2 = x_1 y_2 - x_2 y_1$$

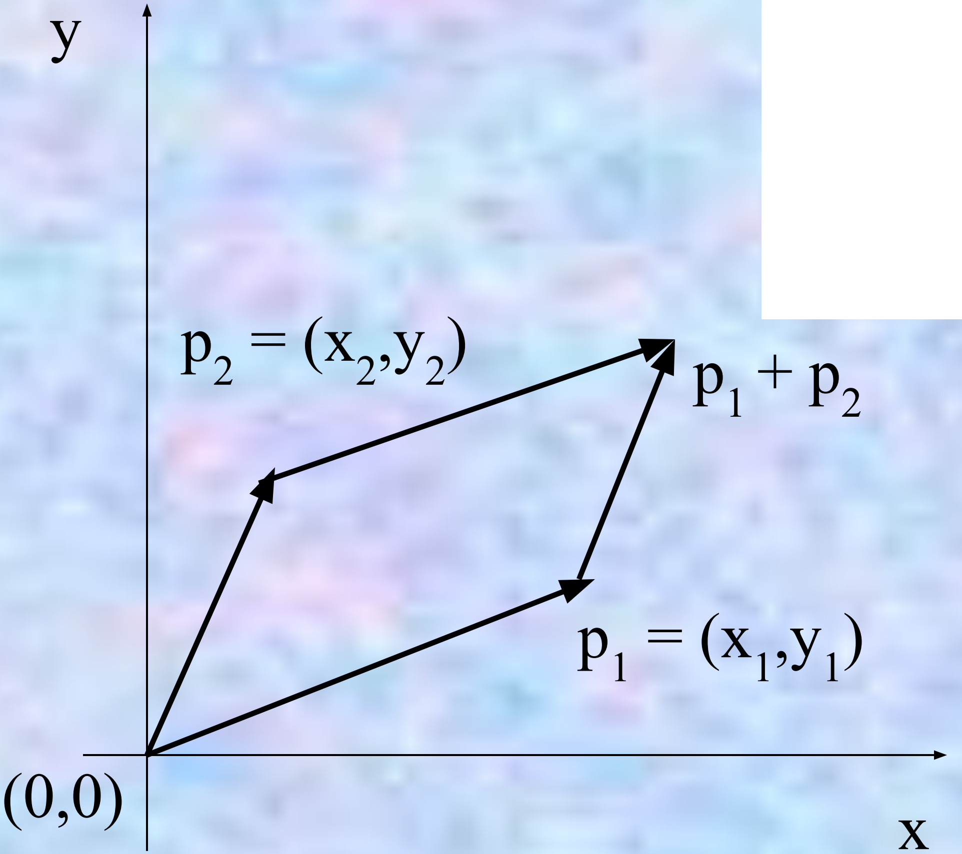
$$= -\, p_2 \times p_1$$

We assume that cross product is scalar given by this formula...

Computational Geometry

Cross Products

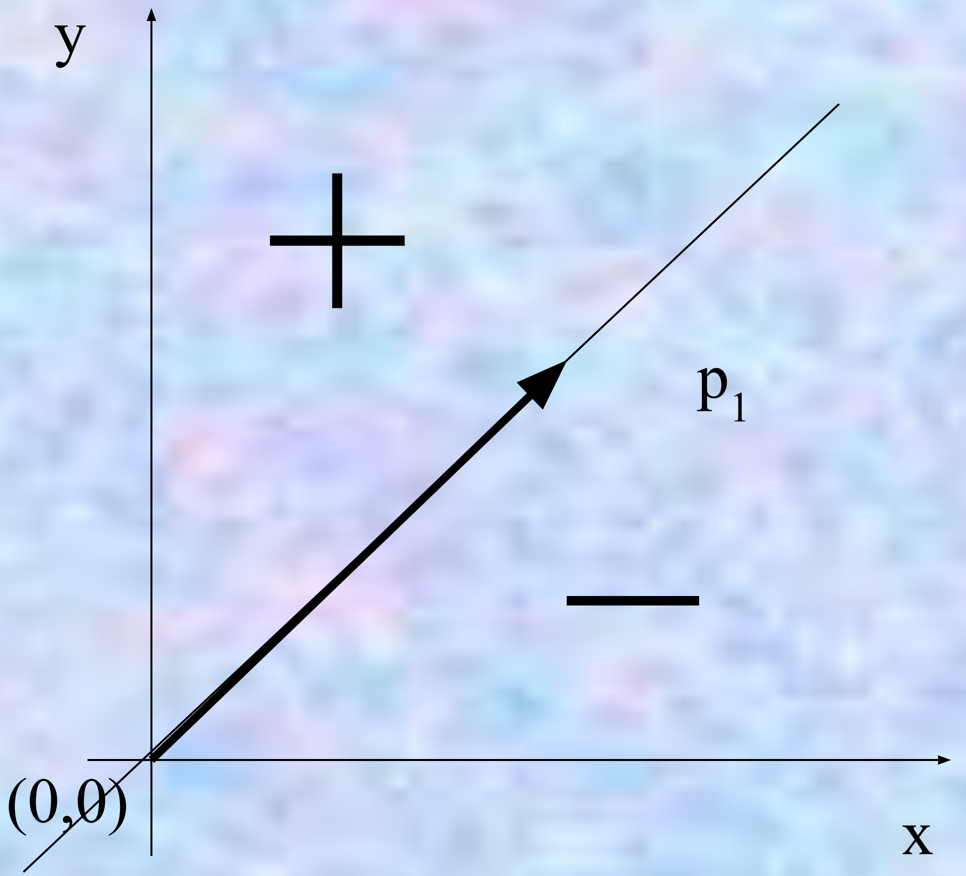




$$p_1 \times p_2 = x_1y_2 - x_2y_1$$

$$= -\, p_2 \times p_1$$

Computational Geometry
Cross Products



The sign (+ or –) of cross product depends in which half plane (relative to p_1) lies p_2

Computational Geometry
Some basic algorithms

1. Given directed line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, determine whether $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$ with respect to point p_0?

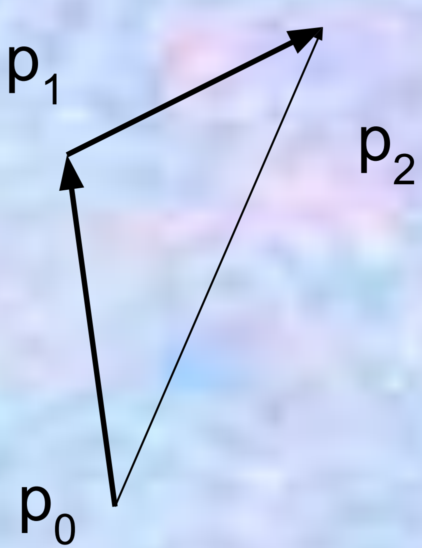
Compute $\Pi = p_1 \times p_2$

if $\Pi > 0$, then $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$
if $\Pi < 0$, then $\overrightarrow{p_0p_1}$ is counterclockwise from $\overrightarrow{p_0p_2}$
or,
if $\Pi > 0$, then $\overrightarrow{p_0p_2}$ is counterclockwise from $\overrightarrow{p_0p_1}$
if $\Pi < 0$, then $\overrightarrow{p_0p_2}$ is clockwise from $\overrightarrow{p_0p_1}$

Computational Geometry

Some basic algorithms

2. Given two line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$, if we traverse $\overrightarrow{p_0p_1}$ and then $\overrightarrow{p_1p_2}$, do we make a left turn at point p_1?



Compute $\Pi = (p_2 - p_0) \times (p_1 - p_0)$

if $\Pi > 0$, then we make a right turn at p_1

if $\Pi < 0$, then we make a left turn at p_1

Computational Geometry
Two Segments Intersect?

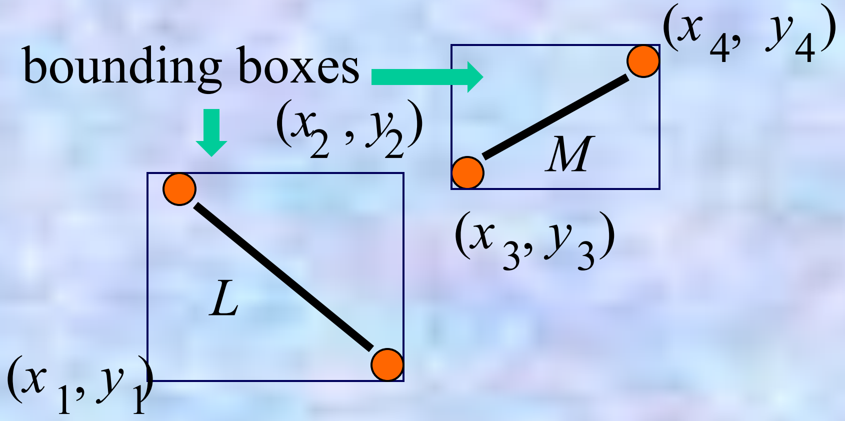
One method: solve for the intersection point of the two lines containing the two line segments, and then check whether this point lies on both segments.

In practice, the two input segments often do ***not*** intersect.

Stage 1: quick rejection if their bounding boxes do not intersect

if and only if $x_4 < x_1 \vee x_3 > x_2 \vee y_4 < y_1 \vee y_3 > y_2$

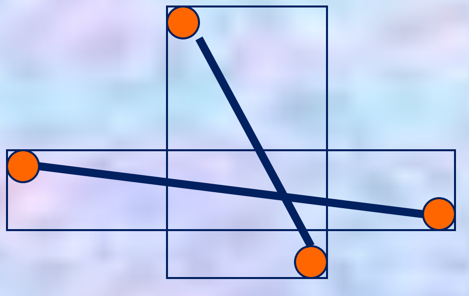
L right of *M*? *L* left of *M*? *L* above *M*? *L* below *M*?

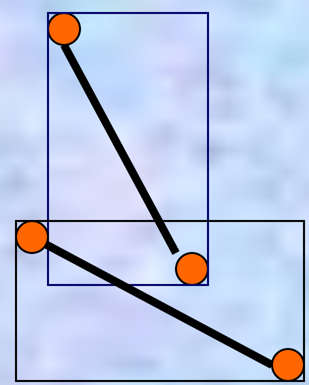


Case 1: bounding boxes do not intersect; neither will the segments.

Computational Geometry
Bounding Box

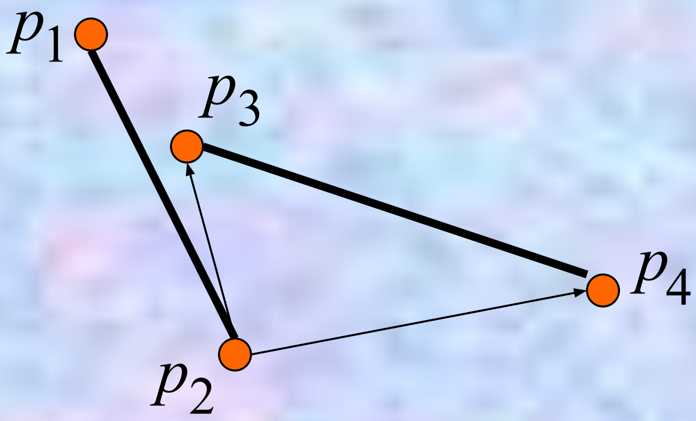
Case 2: Bounding boxes intersect; the segments may or may not intersect. Needs to be further checked in Stage 2.





Computational Geometry
Bounding Box - Stage 2

Two line segments do ***not*** intersect if and only if one segment lies entirely to one side of the line containing the other segment.



$(p_3 - p_2) \times (p_1 - p_2)$ and
$(p_4 - p_2) \times (p_1 - p_2)$ are
both positive!

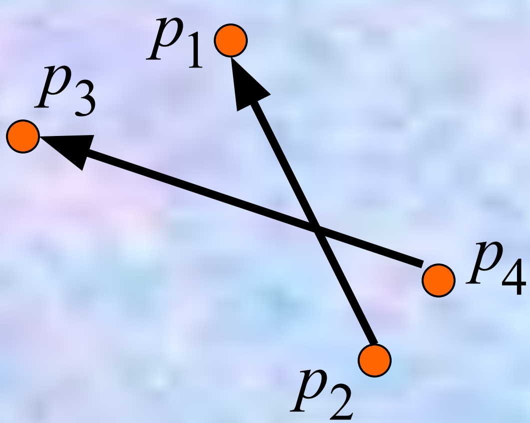
Computational Geometry

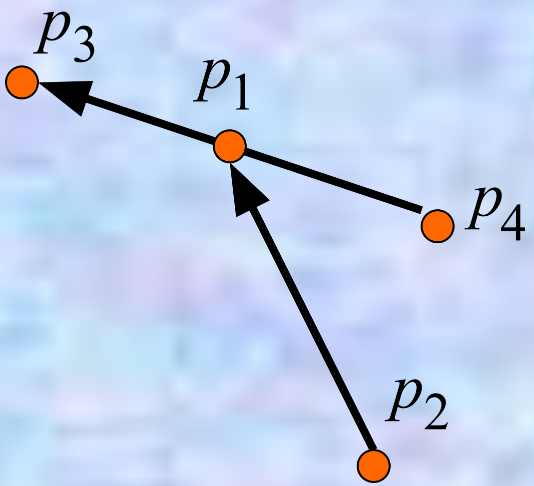
Necessary and Sufficient Condition

Two line segments intersect iff ***each*** of the two pairs of cross products below have different signs (or one cross product in the pair is 0).

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$ // the line through $p_3 p_4$ // intersects $\overline{p_1 p_2}$.

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ // the line through $p_1 p_2$ // intersects $\overline{p_3 p_4}$.





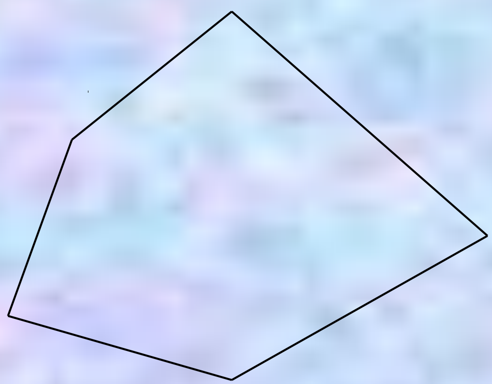
Computational Geometry

Line Segment Intersection Algorithm

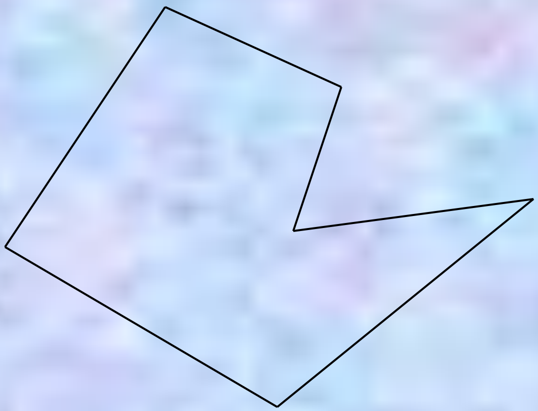
- See page: 937

Computational Geometry

- Data is defined as points, lines, surfaces, polygons etc.



Convex



Non-convex

- Convex Polygon has every in degree less than 180
- Non Convex Polygon may have one or more in degrees greater than 180

Computational Geometry

Convex Set

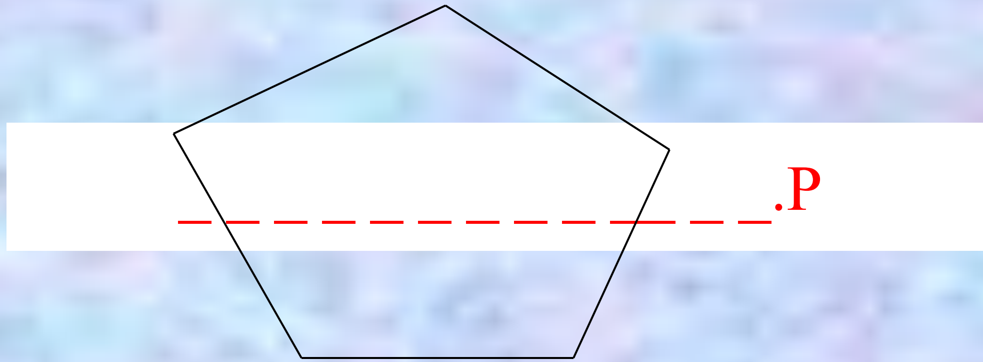
- an object is **convex** if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.

Computational Geometry

- *Problem:* To determine whether a given point lies outside or inside a given polygon.
- *Answer:* Yes if the point lies inside the polygon, No otherwise
- *Solution:* Draw a line along the X –axis from the point in one direction i.e. the line can have a increasing as well as decreasing X – axis.
- If the line thus drawn intersects *ONLY once* with any edge of the polygon then the point lies inside the polygon else it lies outside the polygon

Computational Geometry

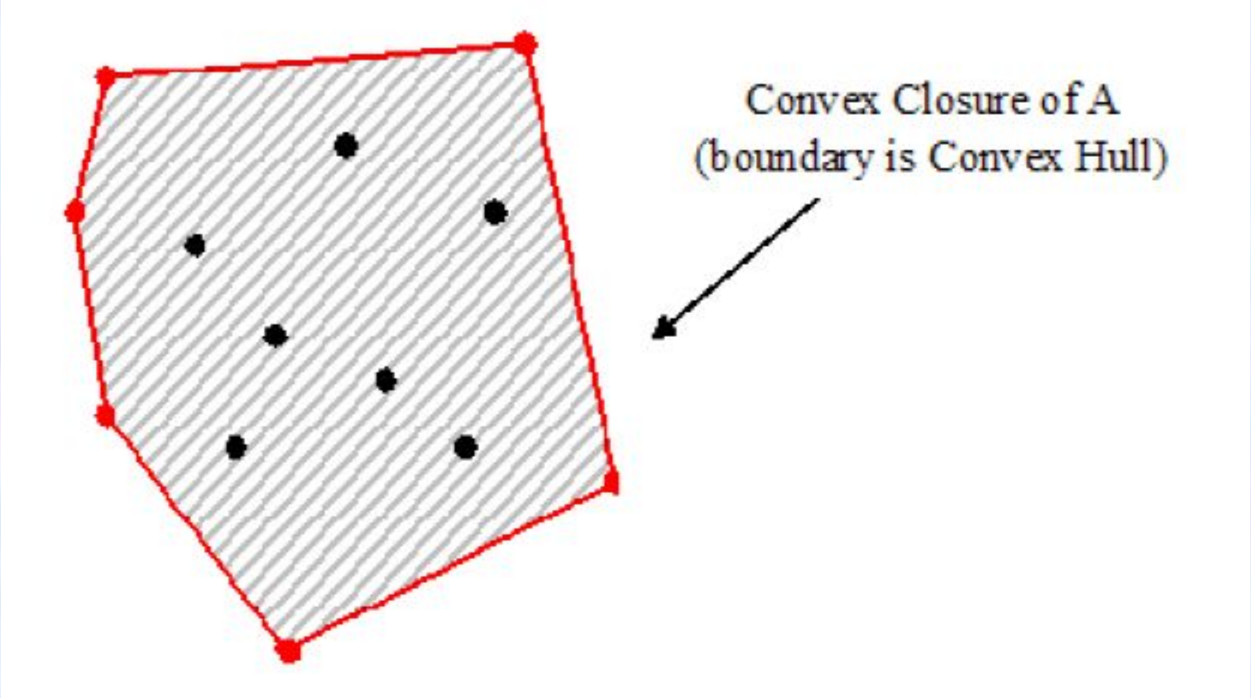
- Example:



The above point intersects only once with an edge of the polygon hence it lies inside polygon

Computational Geometry
Convex Hull - Problem

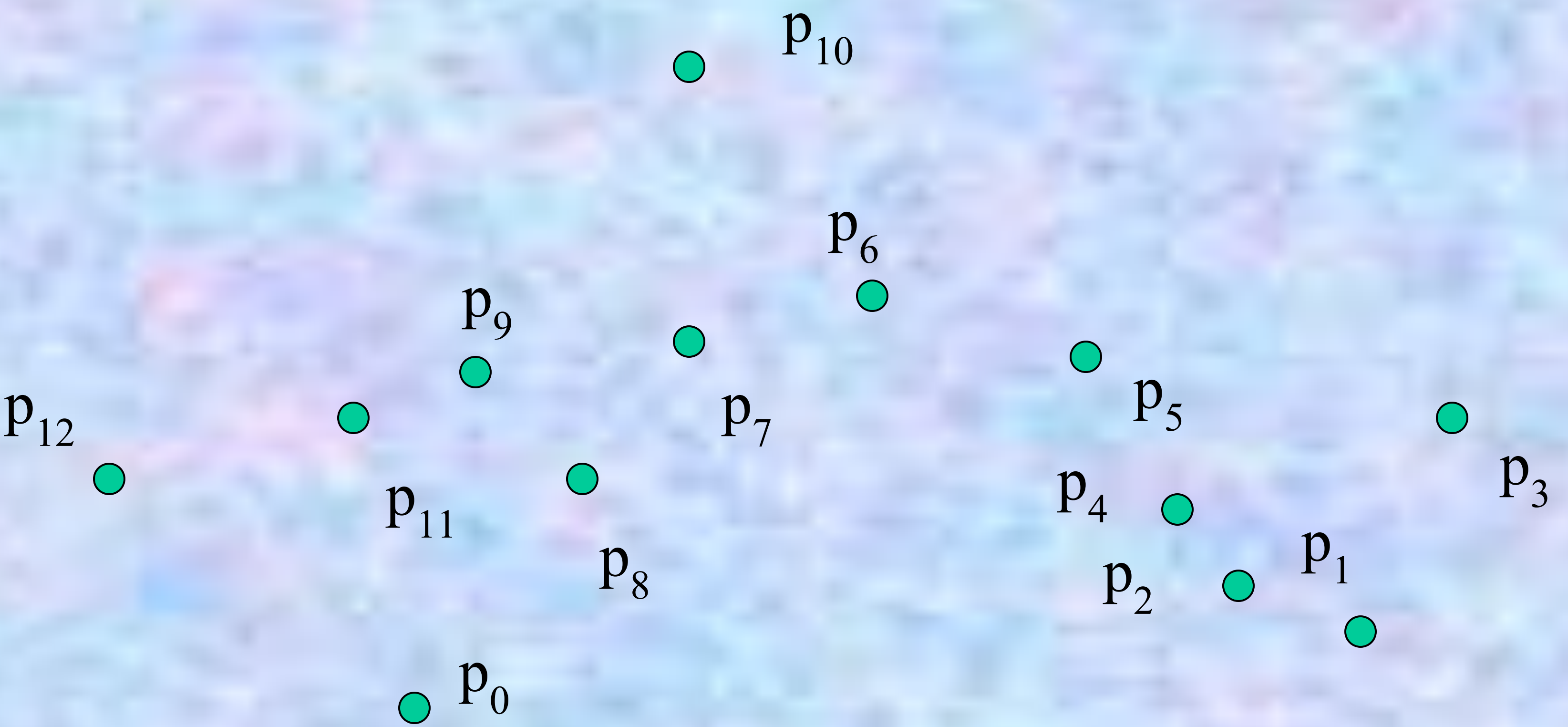
Given n points on plane $p_1, p_2, \ldots, p_n$, find the smallest convex polygon that contains all points $p_1, p_2, \ldots, p_n$.



[Adapted from G.Miller]

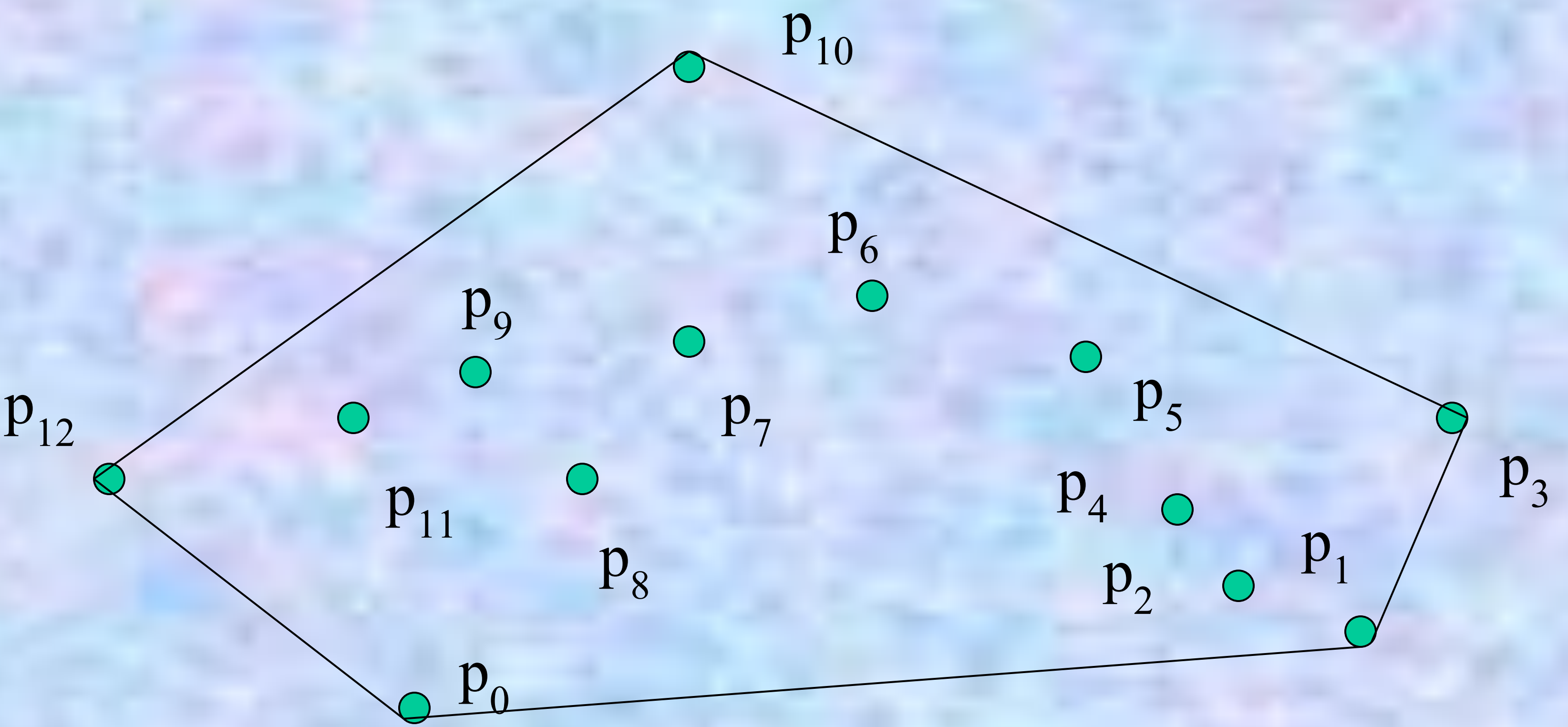
Computational Geometry
Convex Hull - Example

Given n line points on plane p_1, p_2, …,p_n, find the smallest convex polygon that contains all points p_1, p_2, …,p_n.



Computational Geometry
Convex Hull - Example

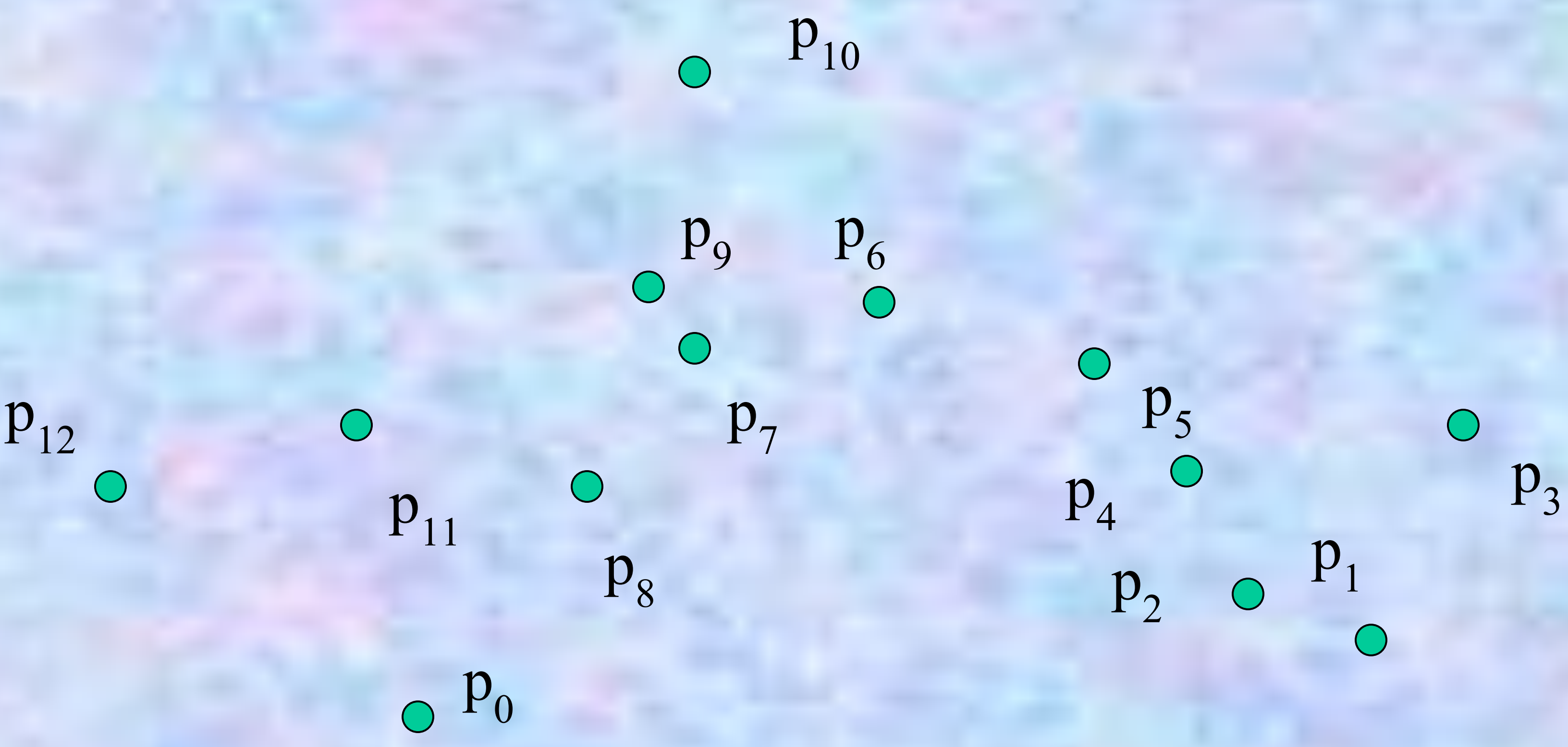
Given n line points on plane p_1, p_2, …,p_n, find the smallest convex polygon that contains all points p_1, p_2, …,p_n.



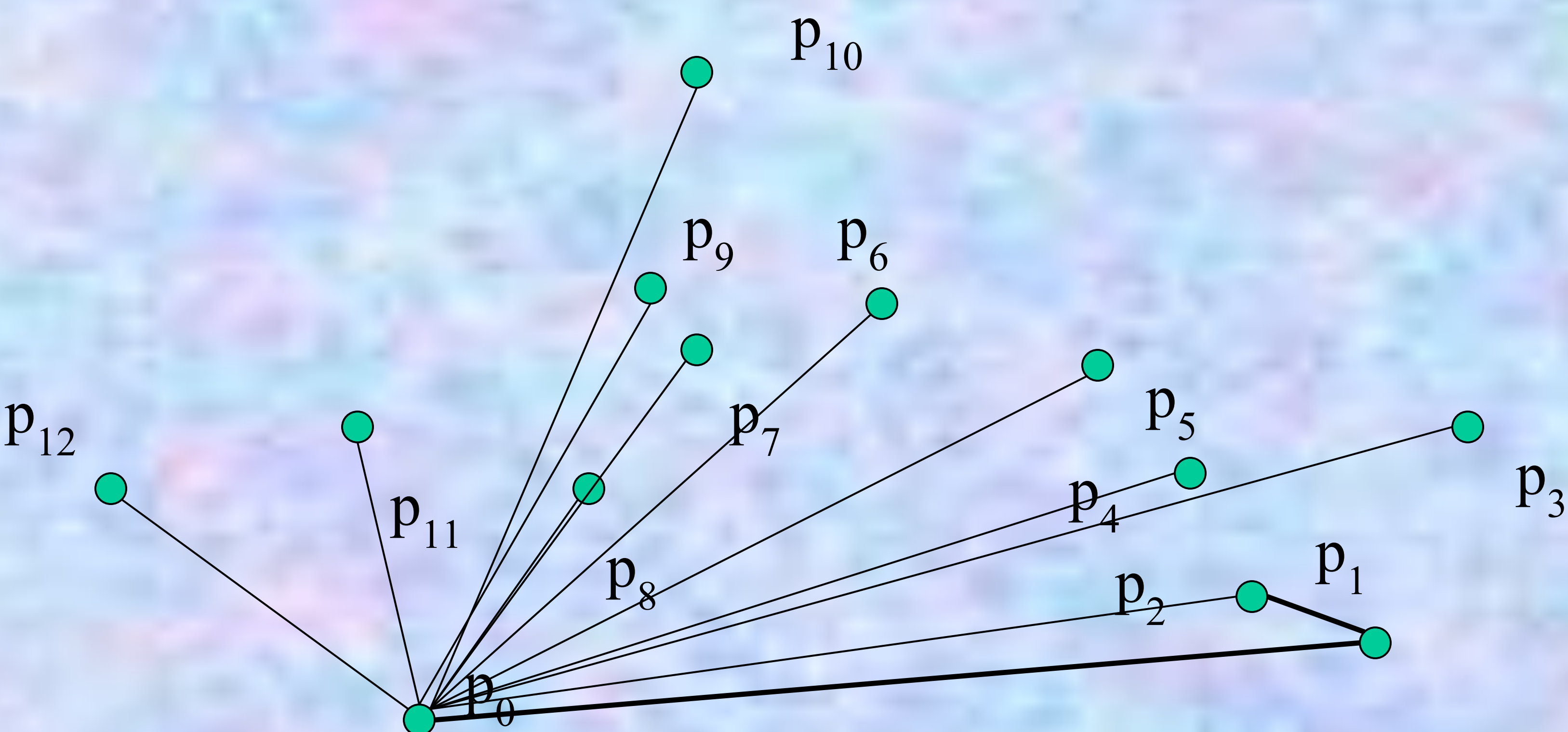
Computational Geometry
Graham Scan - Algorithm

procedure *GrahamScan*(**set** Q)
  let p_0 be the point with the minimum y-coordinate
  let $\langle p_1, \ldots, p_m \rangle$ be the remaining points in Q, sorted by the
    angle in counterclockwise order around p_0
  Top(S) $\leftarrow$ 0
  Push(p_0, S); *Push*(p_1, S); *Push*(p_2, S)
  for i $\leftarrow$ 3 **to** m **do**
    while the angle formed by points *NextToTop*(S), *Top*(S)
      and p_i makes a non-left turn **do**
        Pop(S)
    Push(p_i, S)
  return S

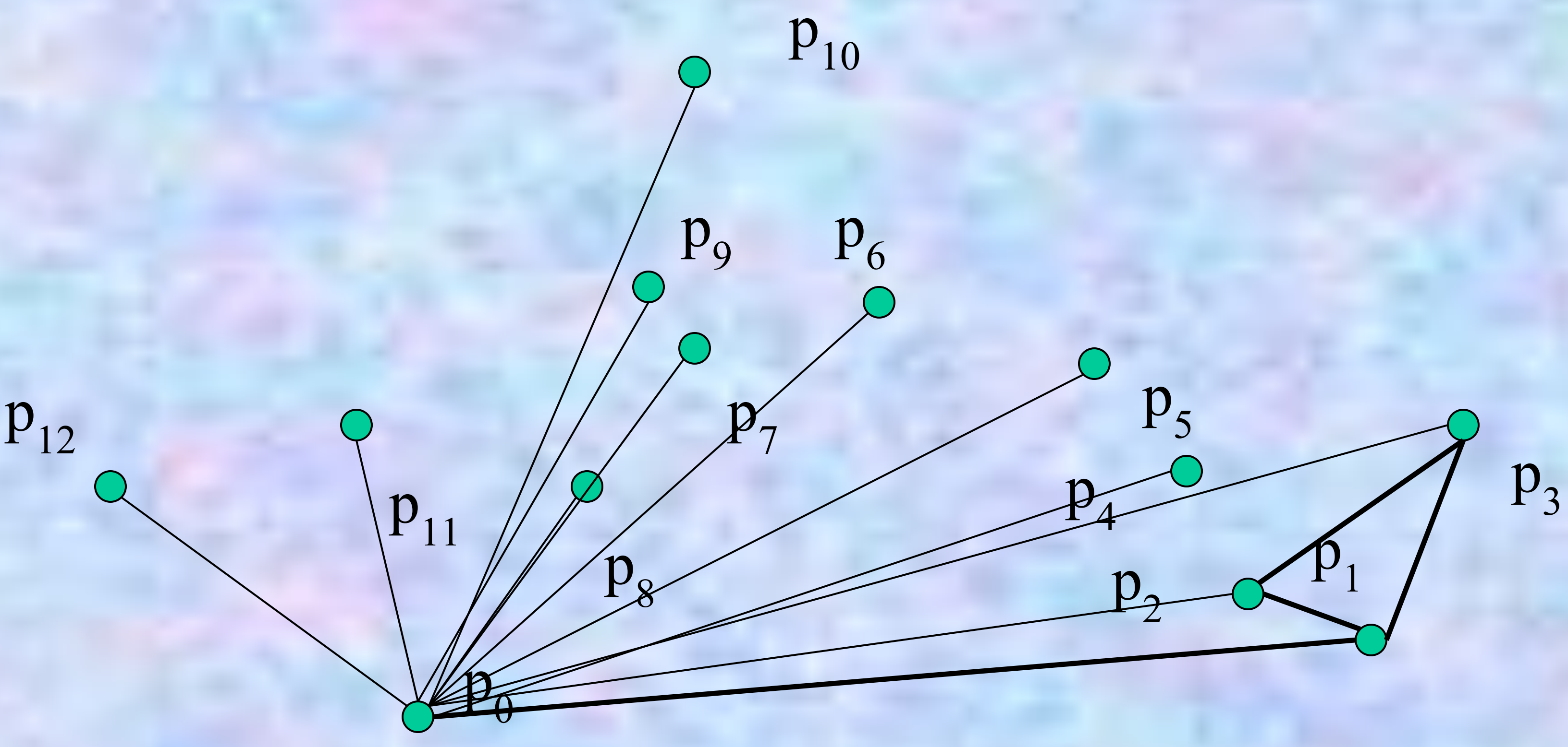
Computational Geometry
Graham Scan - Example



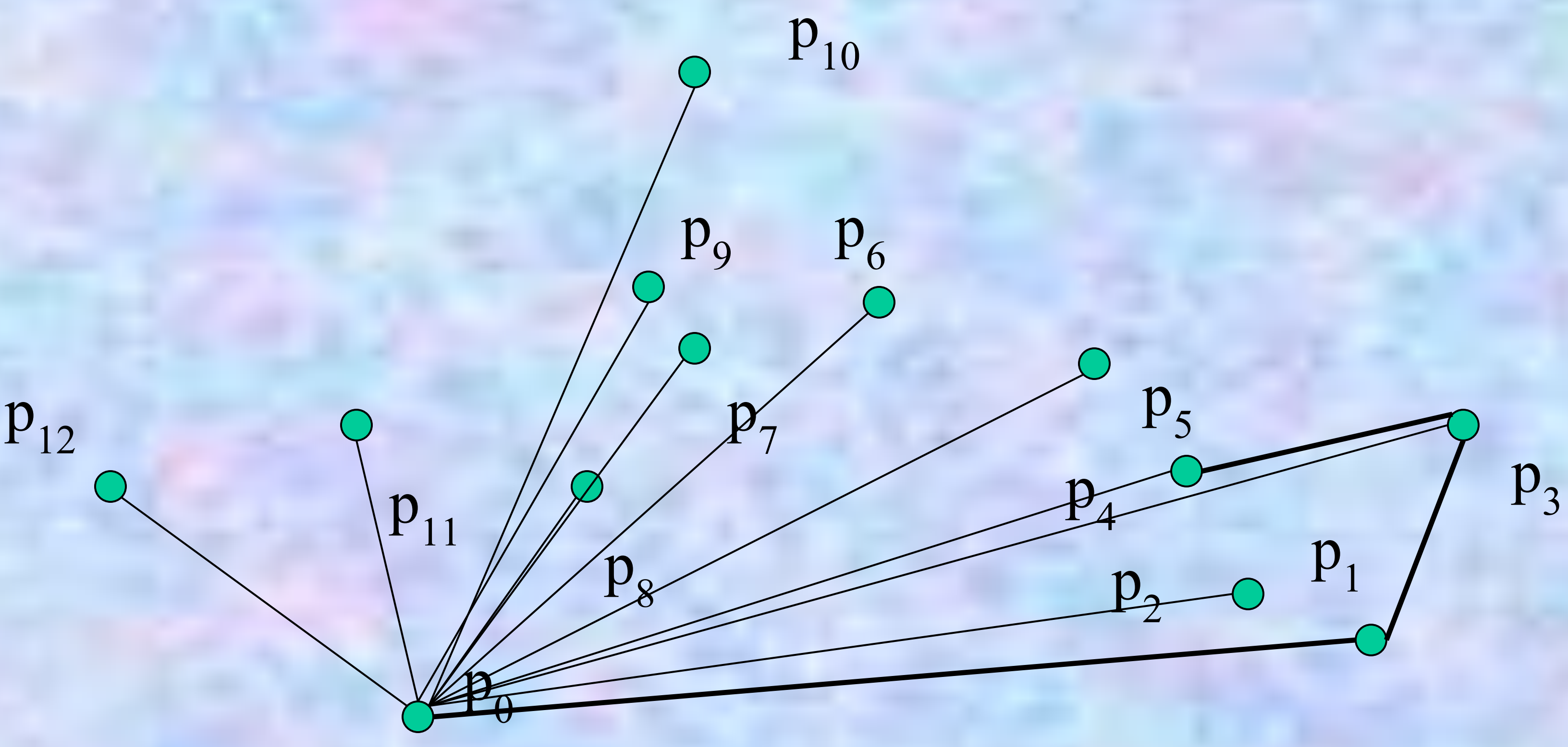
Computational Geometry
Graham Scan - Example



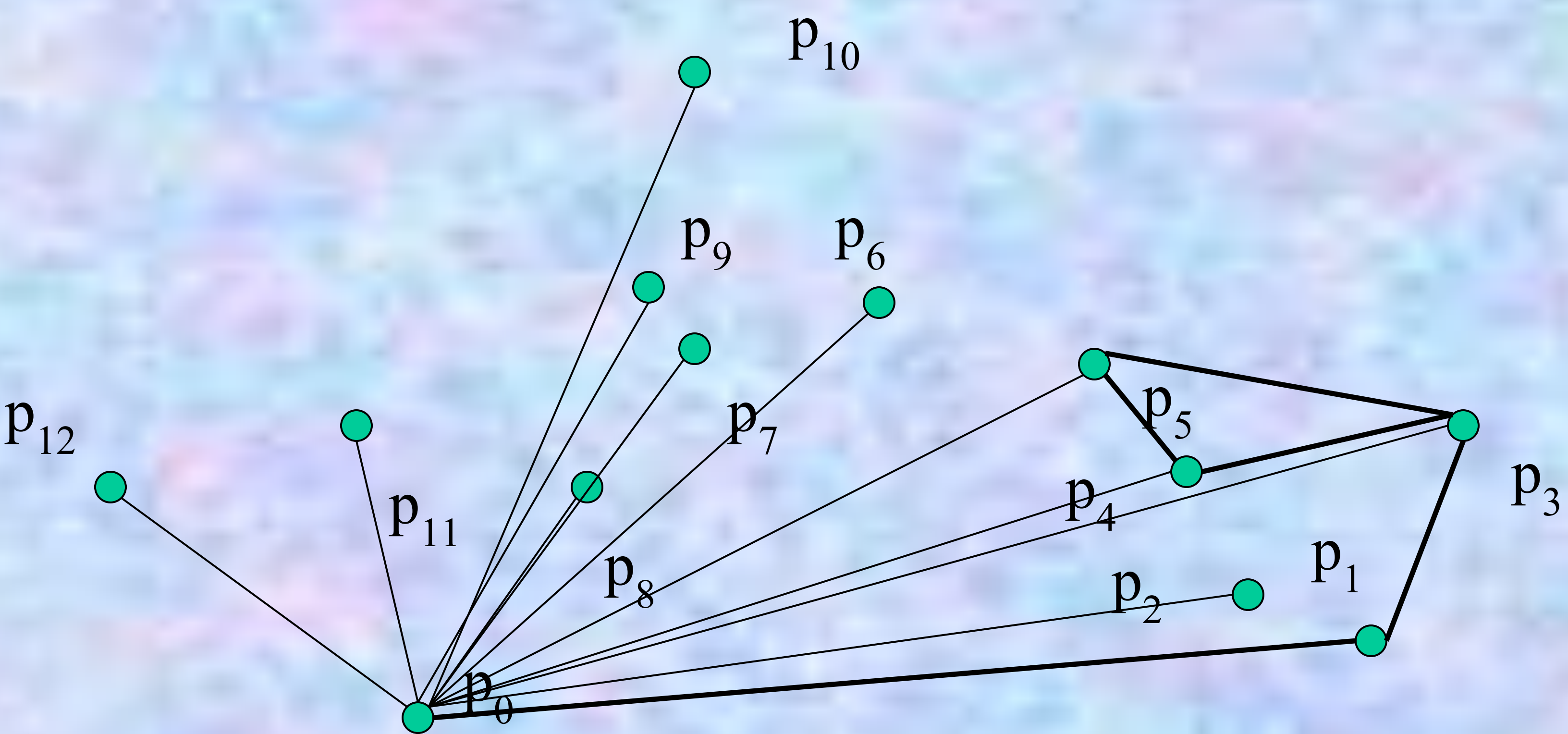
Computational Geometry
Graham Scan - Example



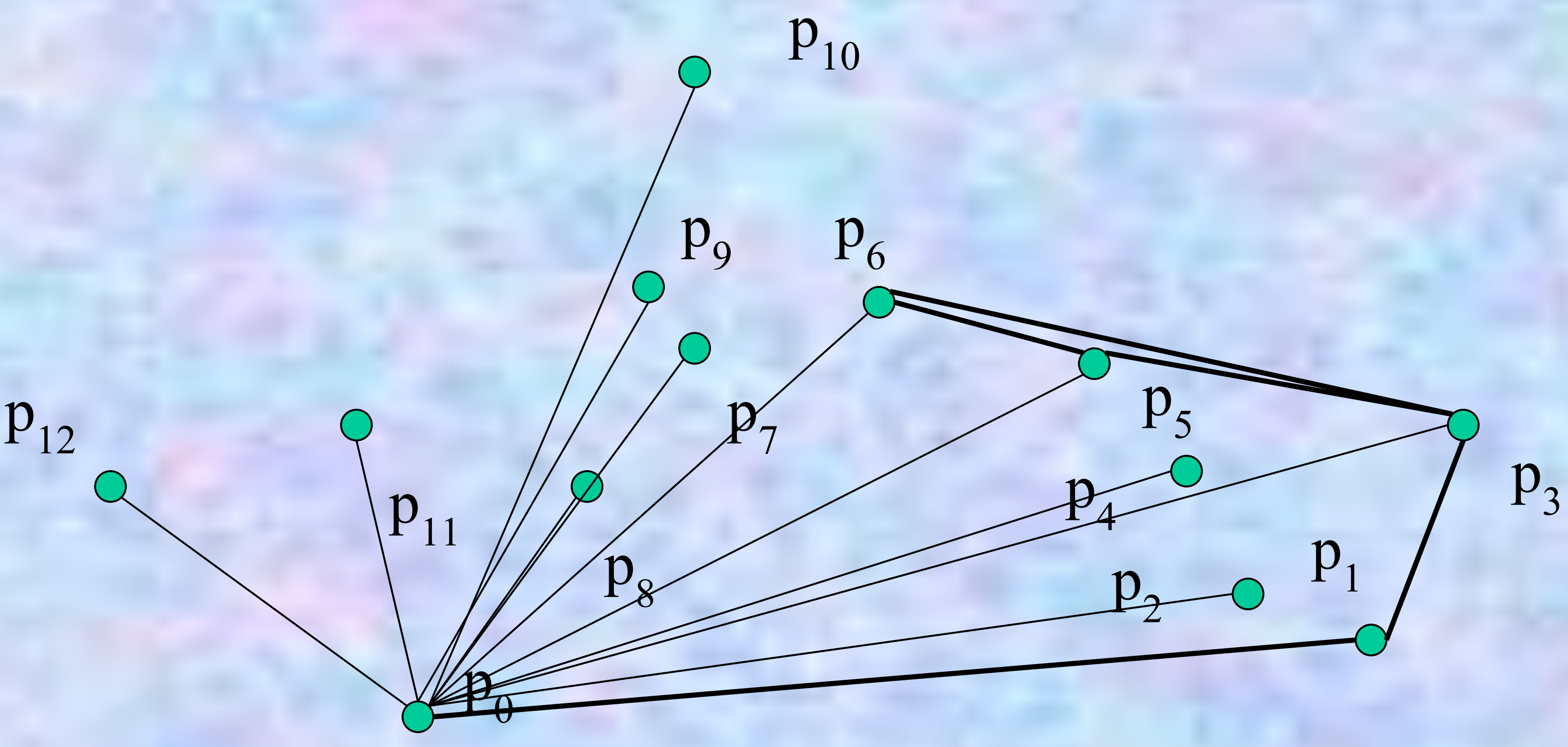
Computational Geometry
Graham Scan - Example



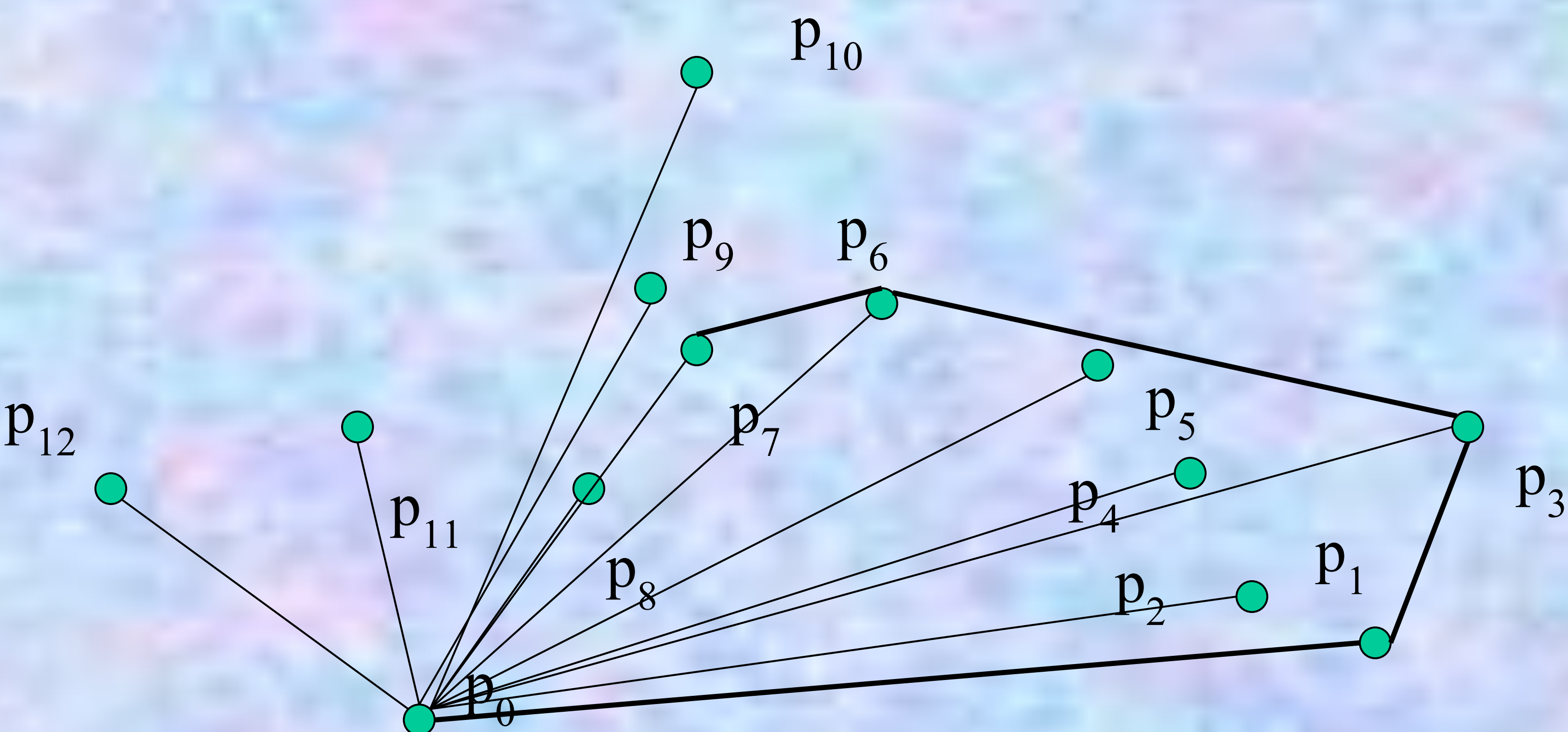
Computational Geometry
Graham Scan - Example



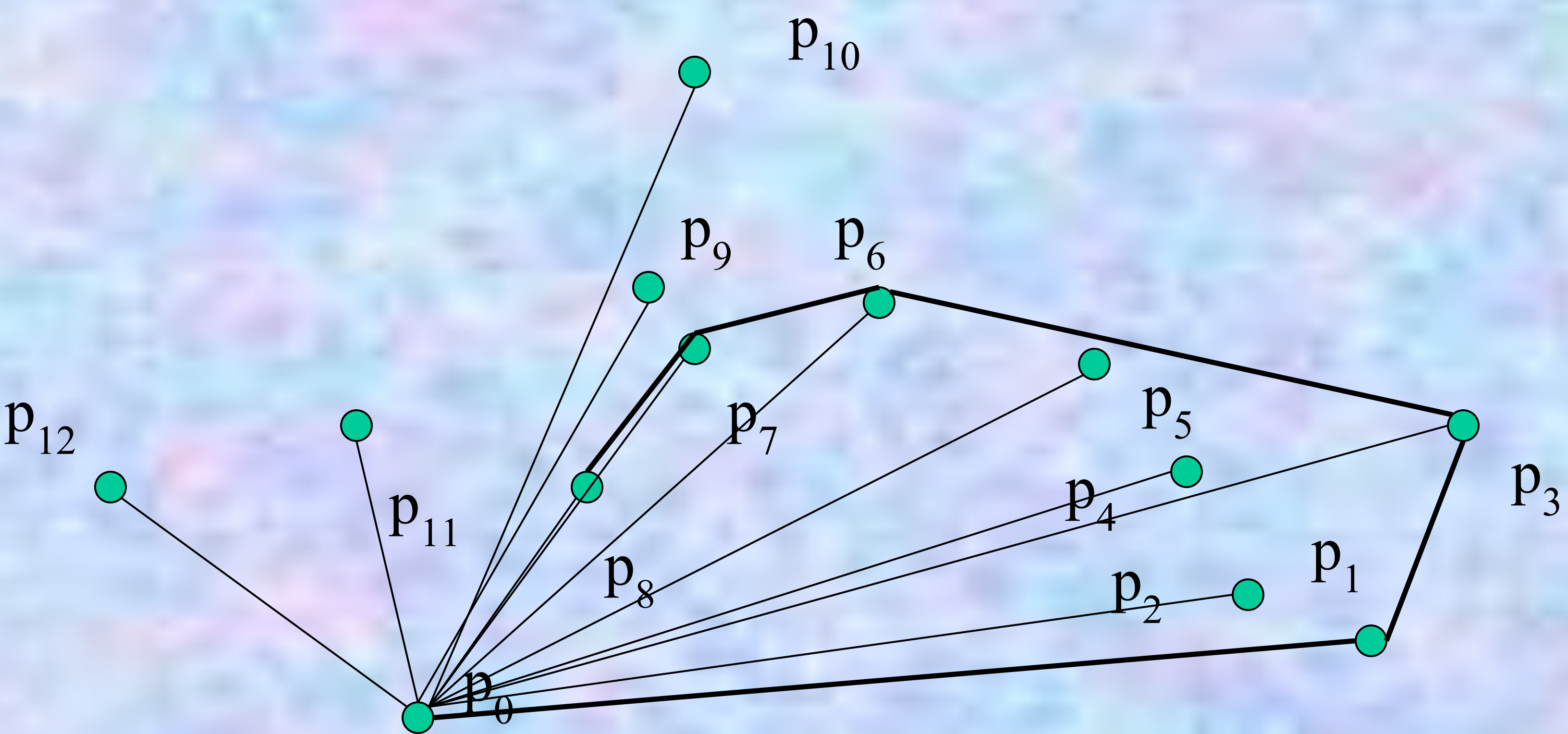
Computational Geometry
Graham Scan - Example



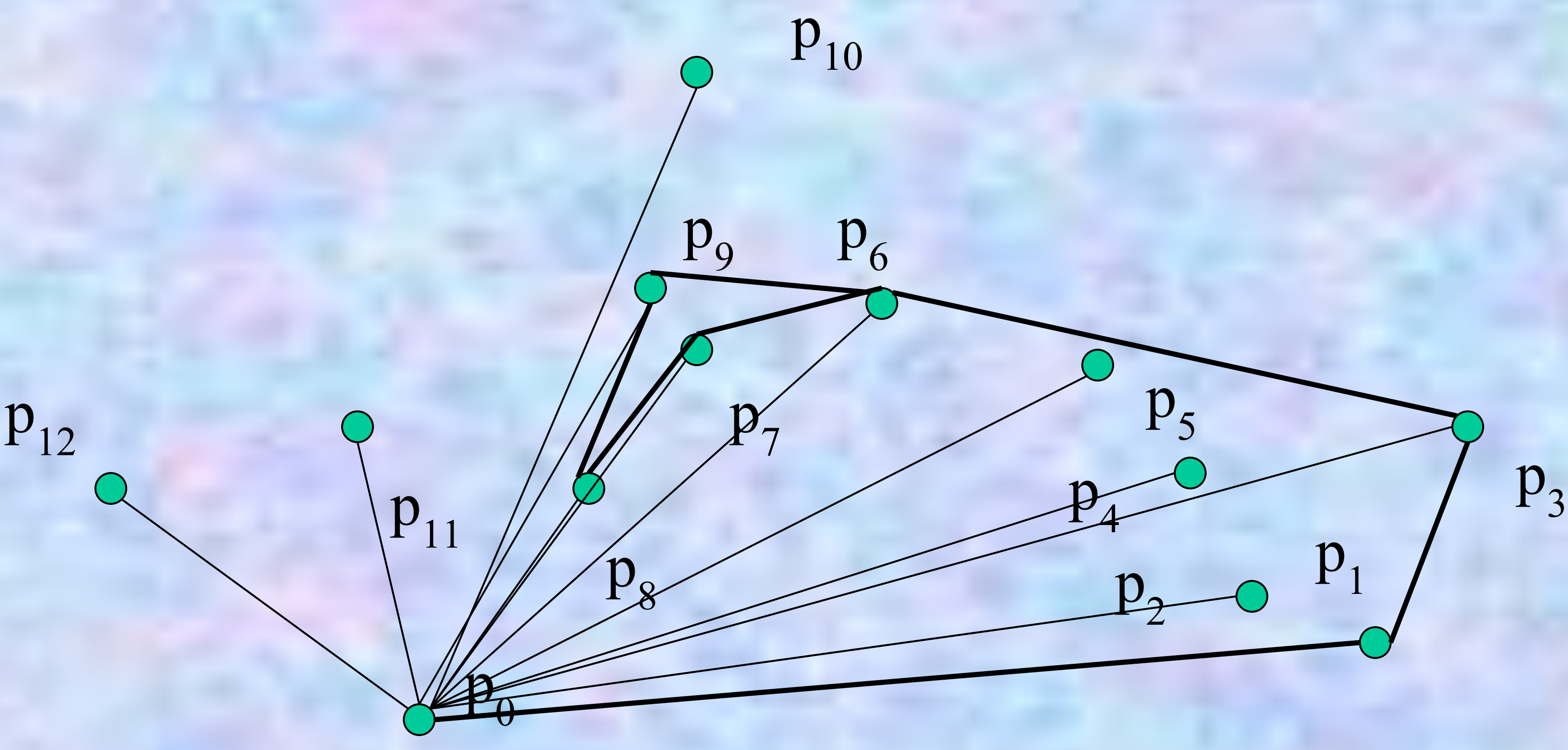
Computational Geometry
Graham Scan - Example



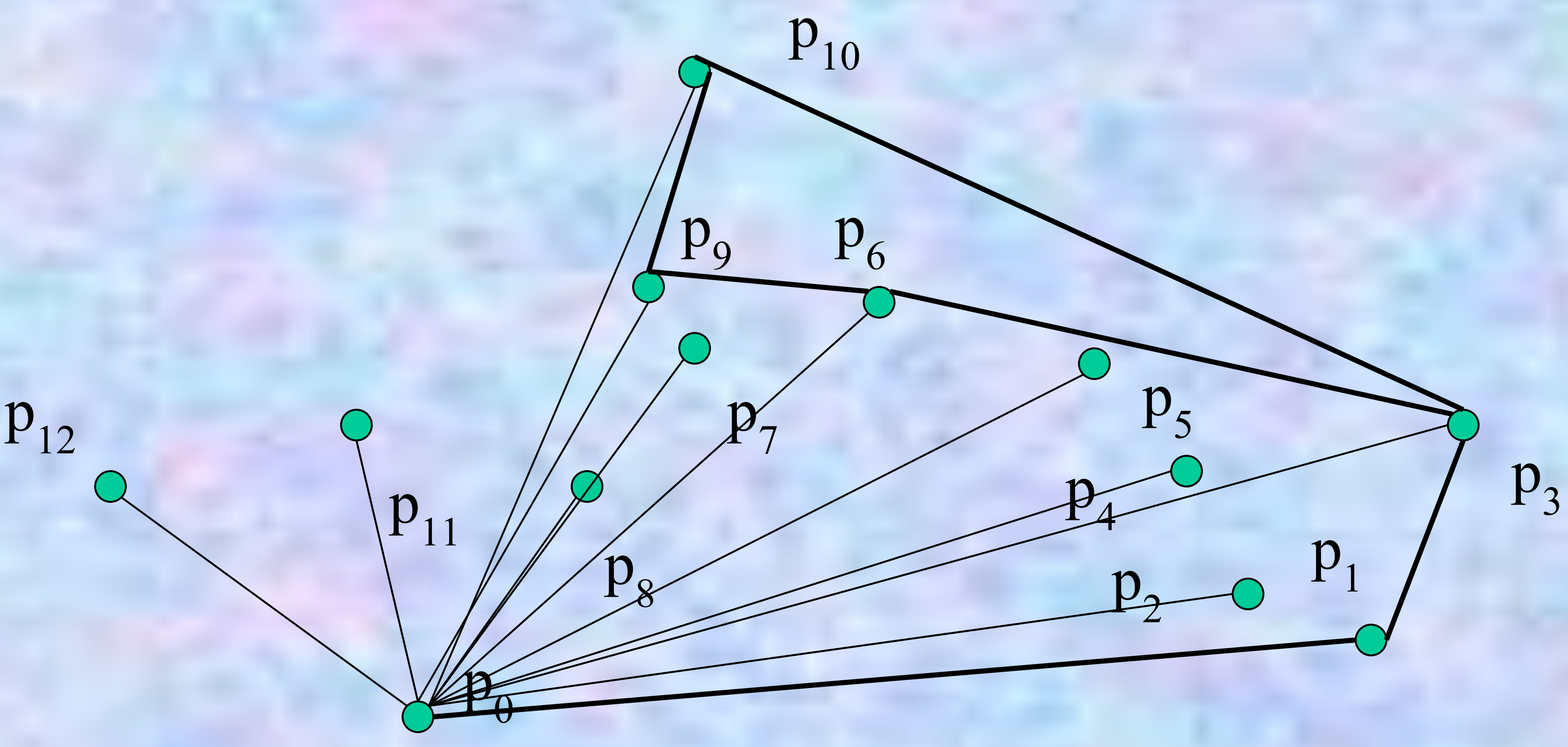
Computational Geometry
Graham Scan - Example



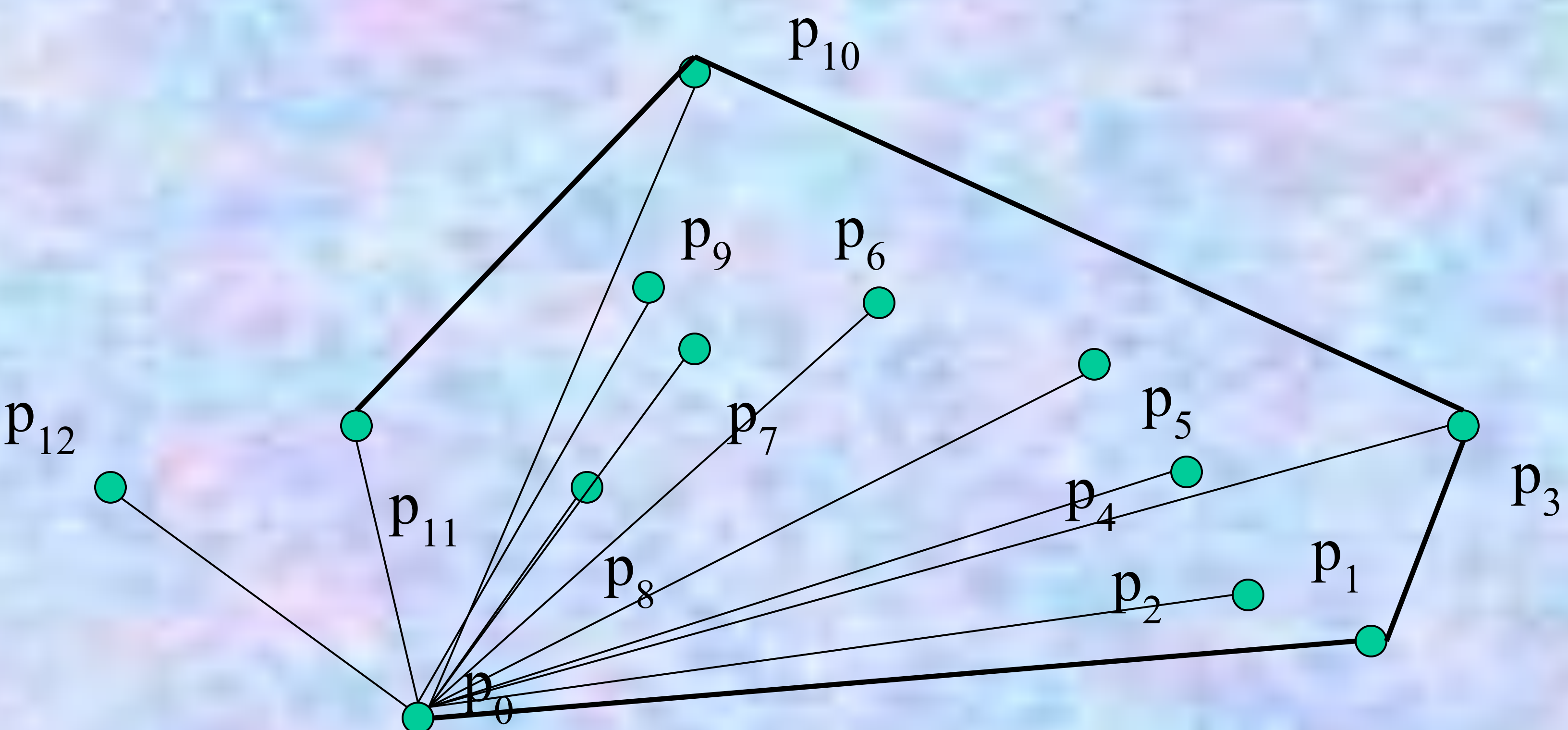
Computational Geometry
Graham Scan - Example



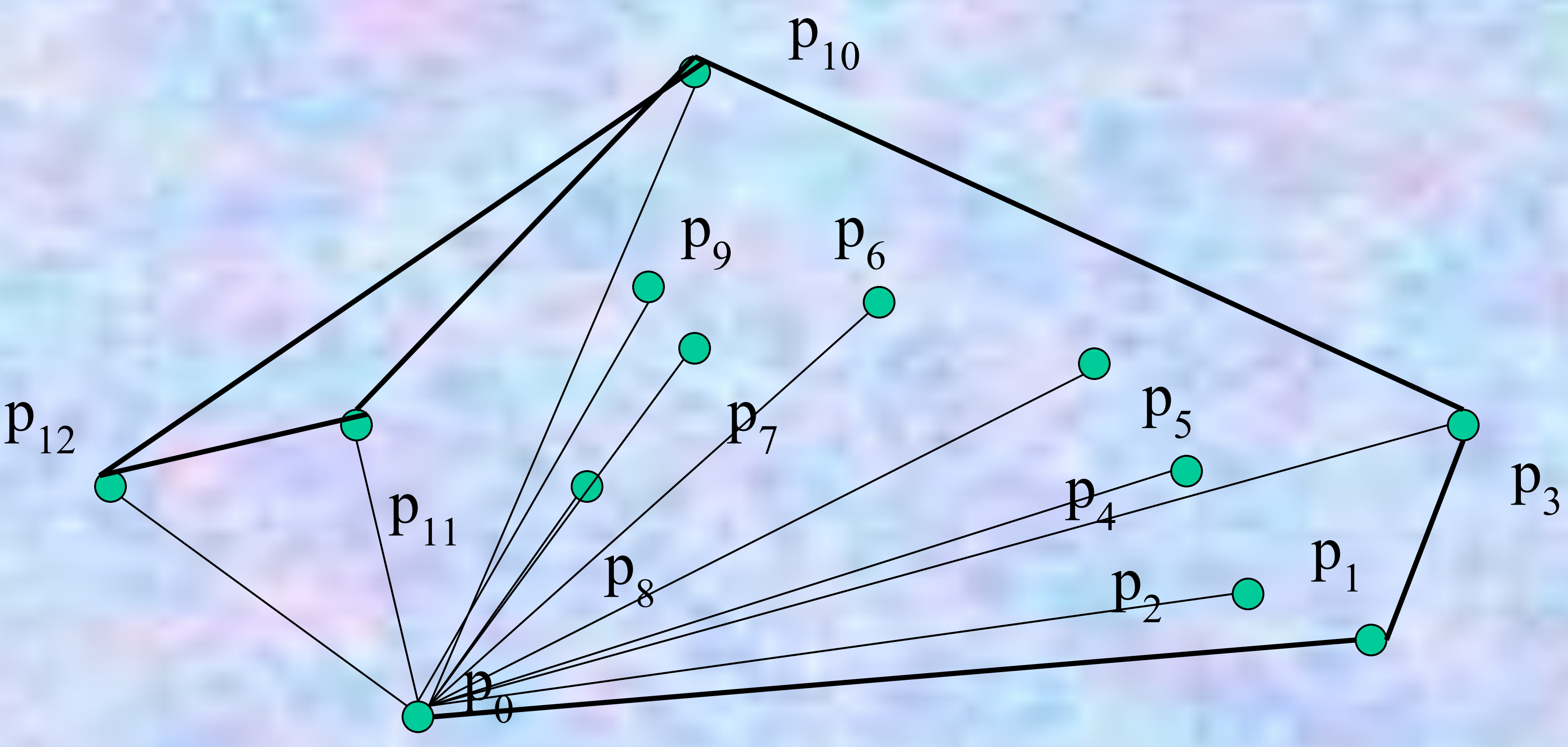
Computational Geometry
Graham Scan - Example



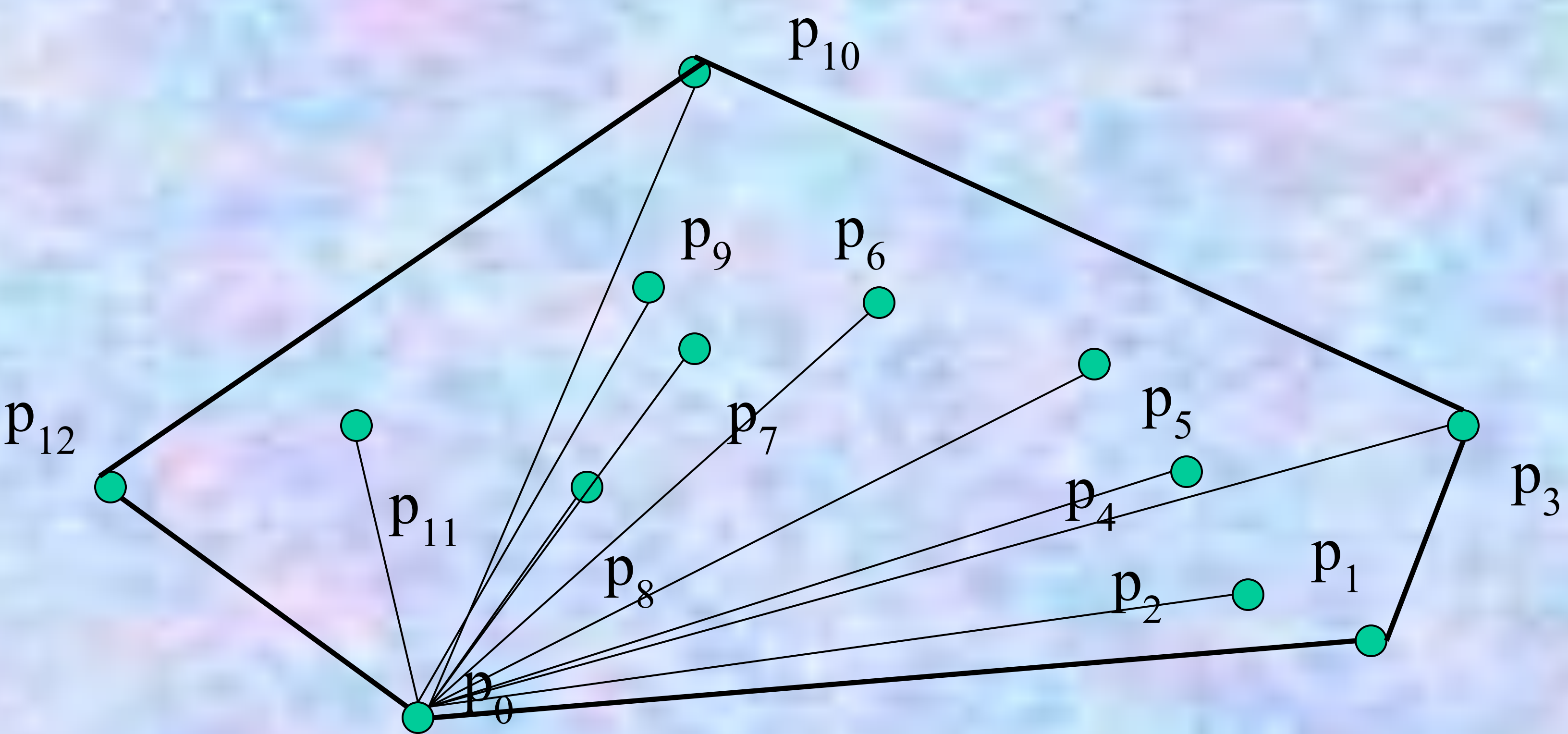
Computational Geometry
Graham Scan - Example



Computational Geometry
Graham Scan - Example



Computational Geometry
Graham Scan - Example



Computational Geometry
Graham Scan - Algorithm

procedure *GrahamScan*(**set** Q)

  let p_0 be the point with the minimum y-coordinate

  let $<p_1, \ldots, p_m>$ be the remaining points in Q, sorted by the
    angle in counterclockwise order around p_0

  Top(S) $\leftarrow$ 0

  Push(p_0, S); *Push*(p_1, S); *Push*(p_2, S)

  for i $\leftarrow$ 3 **to** m **do**

    while the angle formed by points *NextToTop*(S), *Top*(S)
      and p_i makes a non-left turn **do**

        Pop(S)

    Push(p_i, S)

  return S

Computational Geometry

Graham Scan - Complexity

Sorting takes Θ(n log n) time
for loop executes Θ(n) times
each **while** loop might take up Θ(n) time
however, no more than Θ(n) for all while loops together

procedure *GrahamScan*(**set** Q)
  let p_0 be the point with the minimum y-coordinate
  let <p_1, …, p_m> be the remaining points in Q, sorted by the
    angle in counterclockwise order around p_0
  Top(S) ← 0
  Push(p_0, S); *Push*(p_1, S); *Push*(p_2, S)
  for i ← 3 **to** m **do**
    while the angle formed by points *NextToTop*(S), *Top*(S)
      and p_i makes a non-left turn **do**
        Pop(S)
    Push(p_i, S)
  return S

T(n) = Θ(n log n) + const Θ(n) = Θ(n log n)