CS 532: 3D Computer Vision
11th Set of Notes

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu
Office: Lieb 215

Lecture Outline

- Line Intersection
- Polygon triangulation

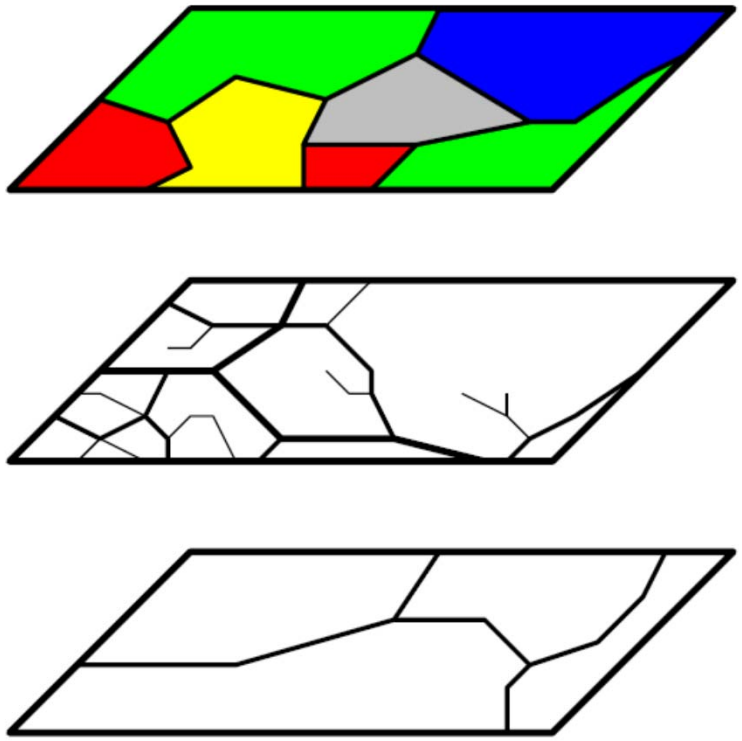
- David M. Mount, CMSC 754: Computational Geometry lecture notes, Department of Computer Science, University of Maryland, Spring 2012
  - Lectures 5 and 6
- Slides by:
  - P. Indyk and J.C. Yang (MIT)
  - M. van Kreveld (Utrecht University)

Line Segment Intersection

Slides by M. van Kreveld

Map Layers

- In a geographic information system (GIS) data is stored in separate layers
- A layer stores the geometric information about some theme, like land cover, road network, municipality boundaries, red fox habitat, …

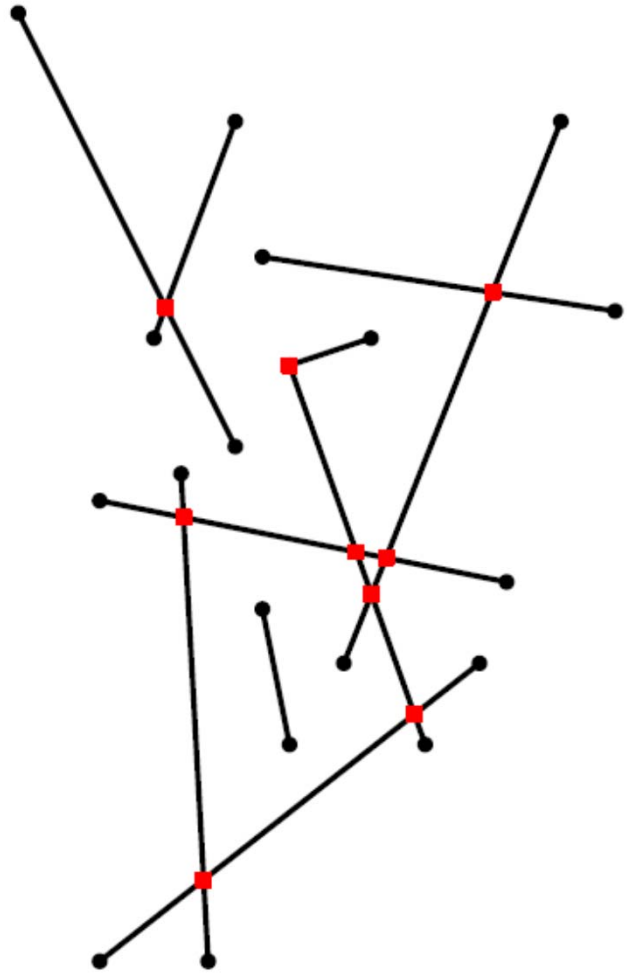


Map Overlay

- Map overlay is the combination of two (or more) map layers
- It is needed to answer questions like:
  - What is the total length of roads through forests?
  - What is the total area of corn fields within 1km from a river?
- To solve map overlay questions, we need (at the least) intersection points from two sets of line segments (possibly, boundaries of regions)

The Easy Problem

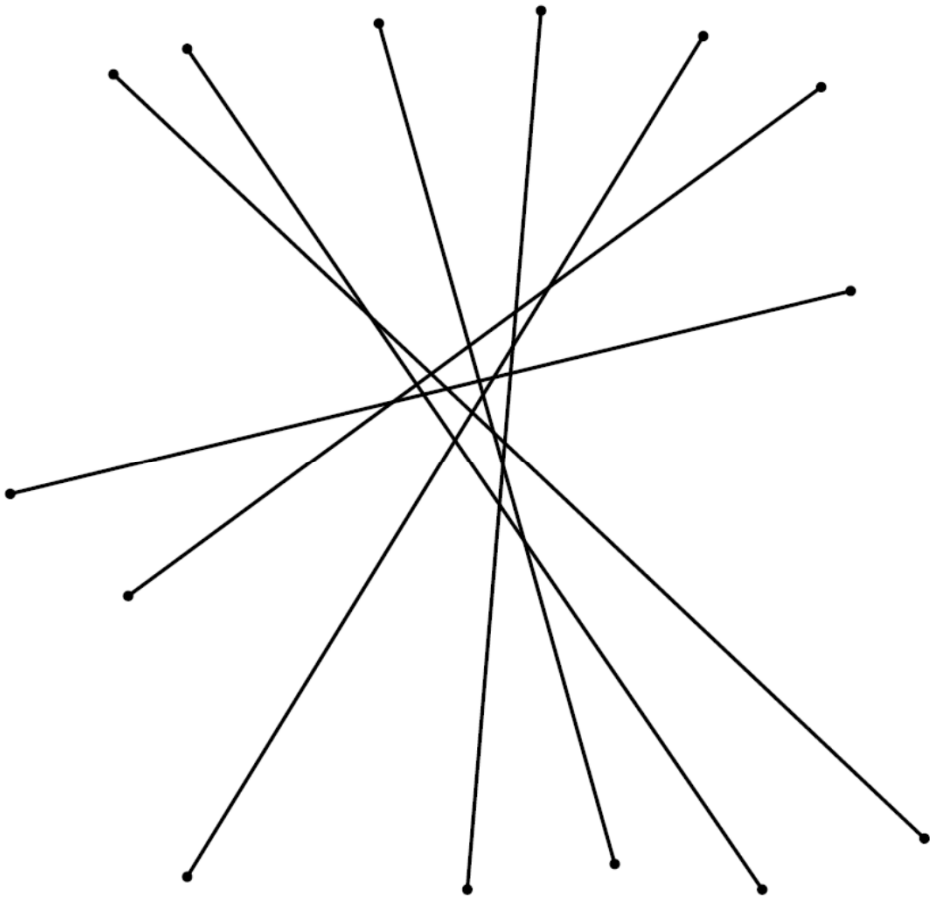
- Given a set of n line segments in the plane, find all intersection points efficiently



An Easy Algorithm

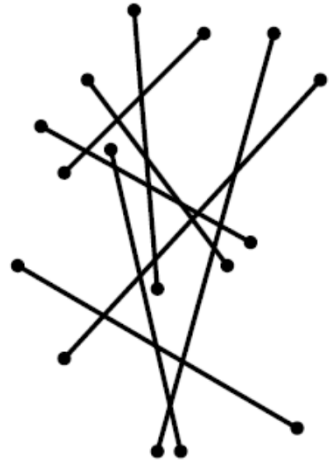
Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The set of intersection points among the segments in S.
1. **for** each pair of line segments $e_i, e_j \in S$
2. **do if** e_i and e_j intersect
3. **then** report their intersection point

Is it Optimal?

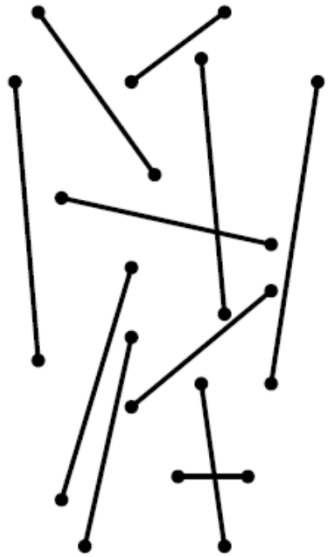


Output-sensitive Algorithms

- The asymptotic running time of an algorithm is always input-sensitive (depends on n)



- We may also want the running time to be output-sensitive: if the output is large, it is fine to spend a lot of time, but if the output is small, we want a fast algorithm



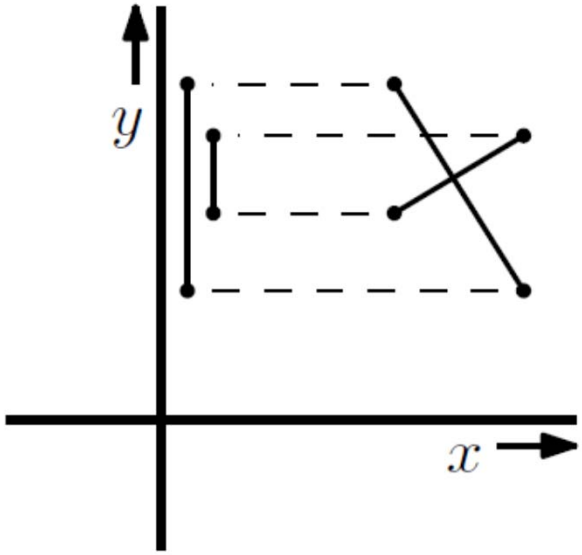
Intersection Points in Practice

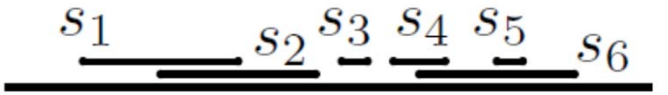
- Question: How many intersection points do we typically expect in our application?

- If this number is k, and if k = O(n), it would be nice if the algorithm runs in O(nlogn) time

First Attempt

- Observation: Two line segments can only intersect if their y-spans have an overlap
- So, how about only testing pairs of line segments that intersect in the y-projection?
- 1D problem: Given a set of intervals on the real line, find all partly overlapping pairs





$(s_1, s_2),\ (s_4, s_6),\ (s_5, s_6)$

First Attempt

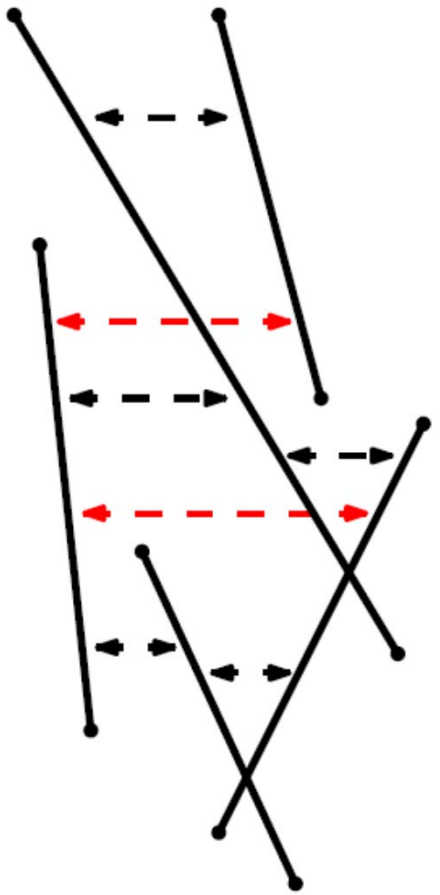
- 1D problem: Given a set of intervals on the real line, find all partly overlapping pairs
- Sort the endpoints and handle them from left to right; maintain currently intersected intervals in a balanced search tree T
  - Left endpoint of s_i: for each s_j in T, report the pair s_i; s_j. Then insert s_i in T
  - Right endpoint of s_i: delete s_i from T
- Question: Is this algorithm output-sensitive for 1D interval intersection?

First Attempt

- Back to the 2D problem:
- Determine the y-intervals of the 2D line segments
- Find the intersecting pairs of intervals with the 1D solution
- For every pair of intersecting intervals, test whether the corresponding line segments intersect, and if so, report it
- Question: Is this algorithm output-sensitive for 2D line segment intersection?

Second Attempt

- Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are horizontal neighbors)

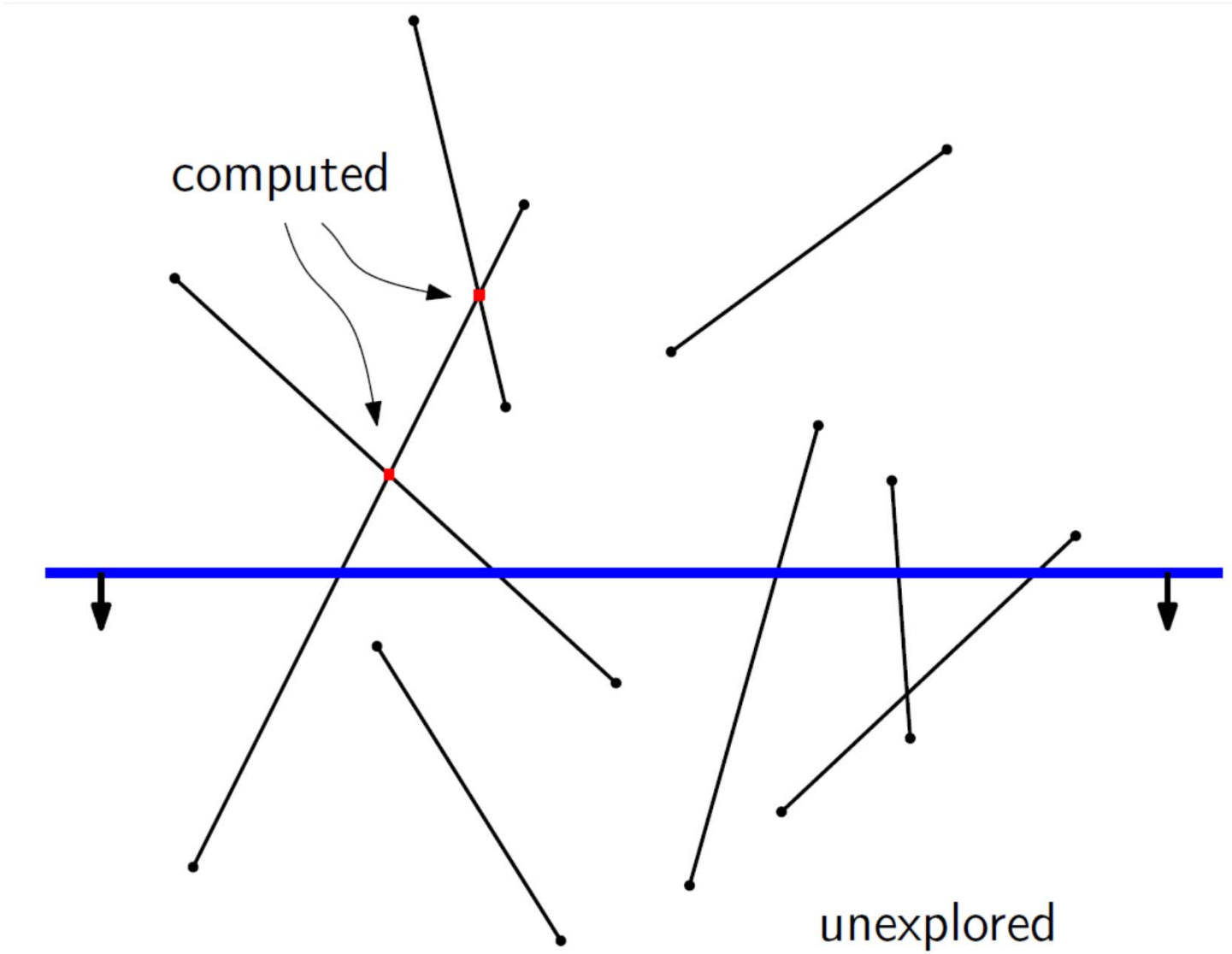


Plane Sweep

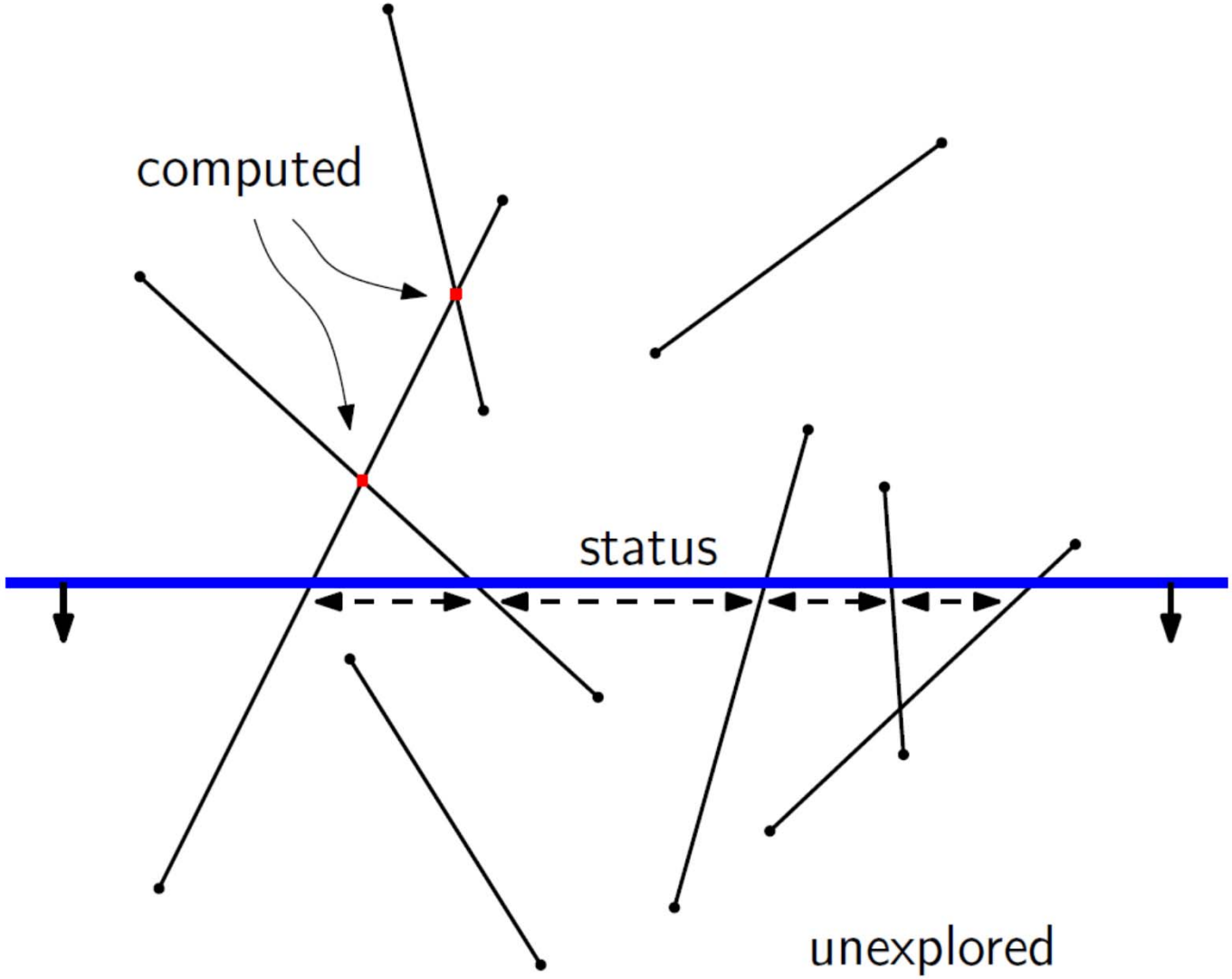
The plane sweep technique: Imagine a horizontal line passing over the plane from top to bottom, solving the problem as it moves

- The sweep line stops and the algorithm computes at certain positions => events
- The algorithm stores the relevant situation at the current position of the sweep line => status
- The algorithm knows everything it needs to know above the sweep line, and has found all intersection points

Sweep



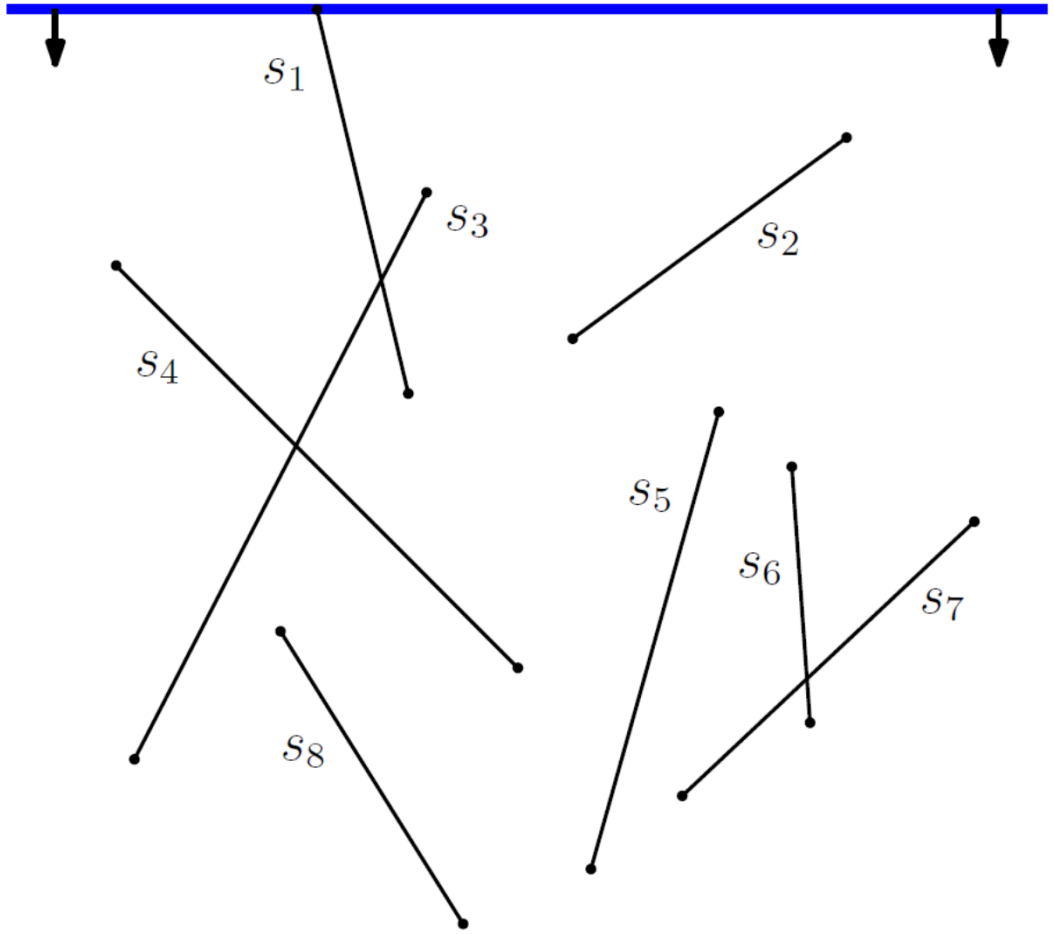
Sweep and Status



Status and Events

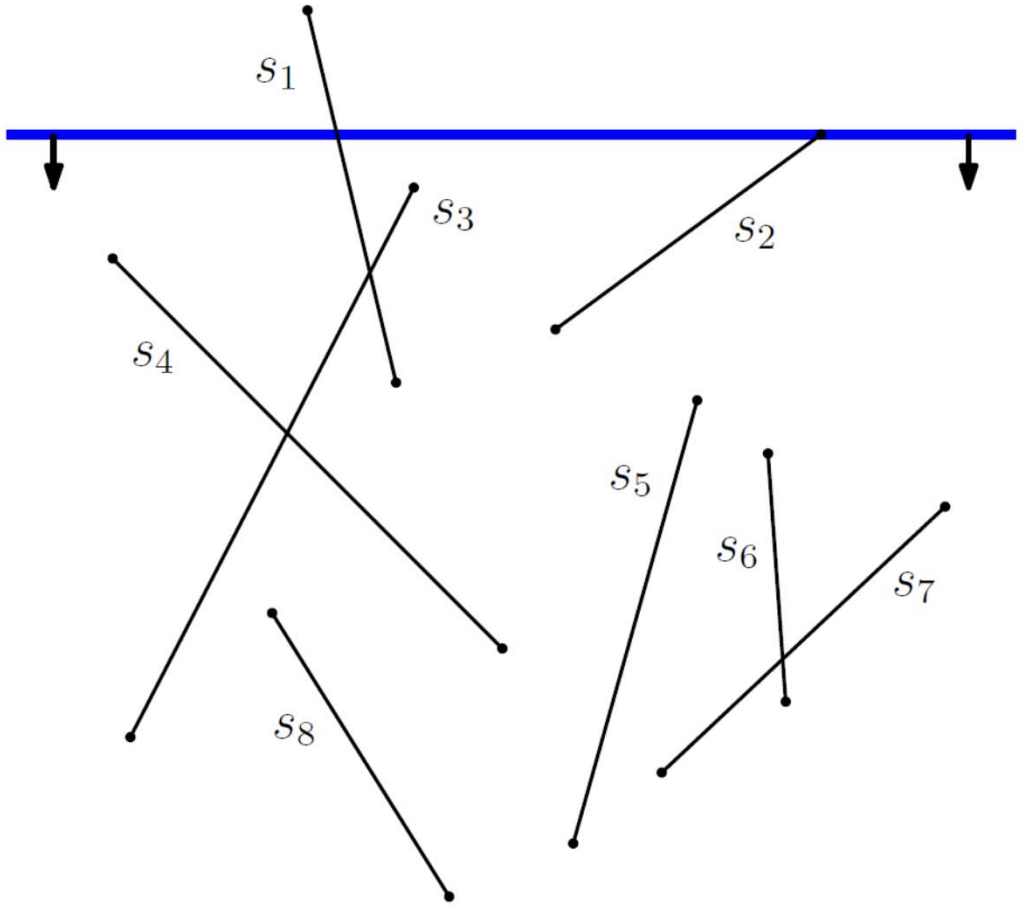
- The status of this particular plane sweep algorithm, at the current position of the sweep line, is the set of line segments intersecting the sweep line, ordered from left to right
- Events occur when the status changes, and when output is generated
  - event = interesting y-coordinate

Plane Sweep



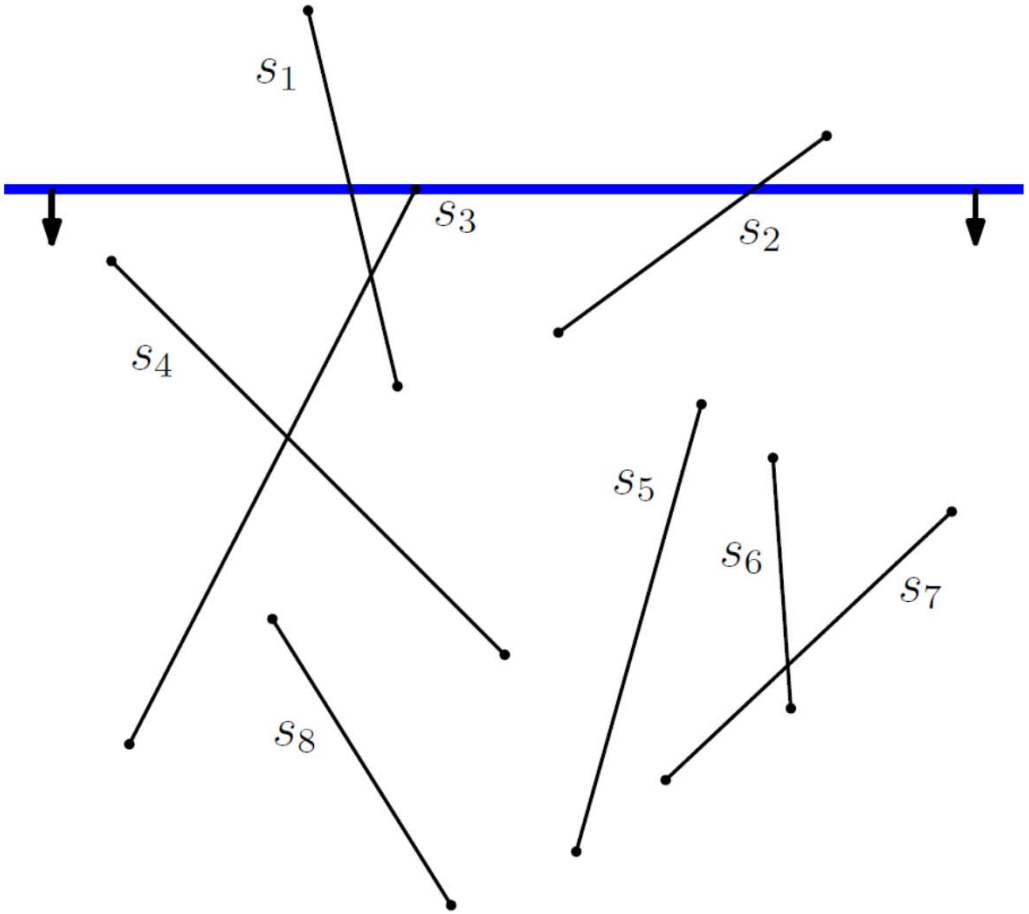
add s_1

Plane Sweep



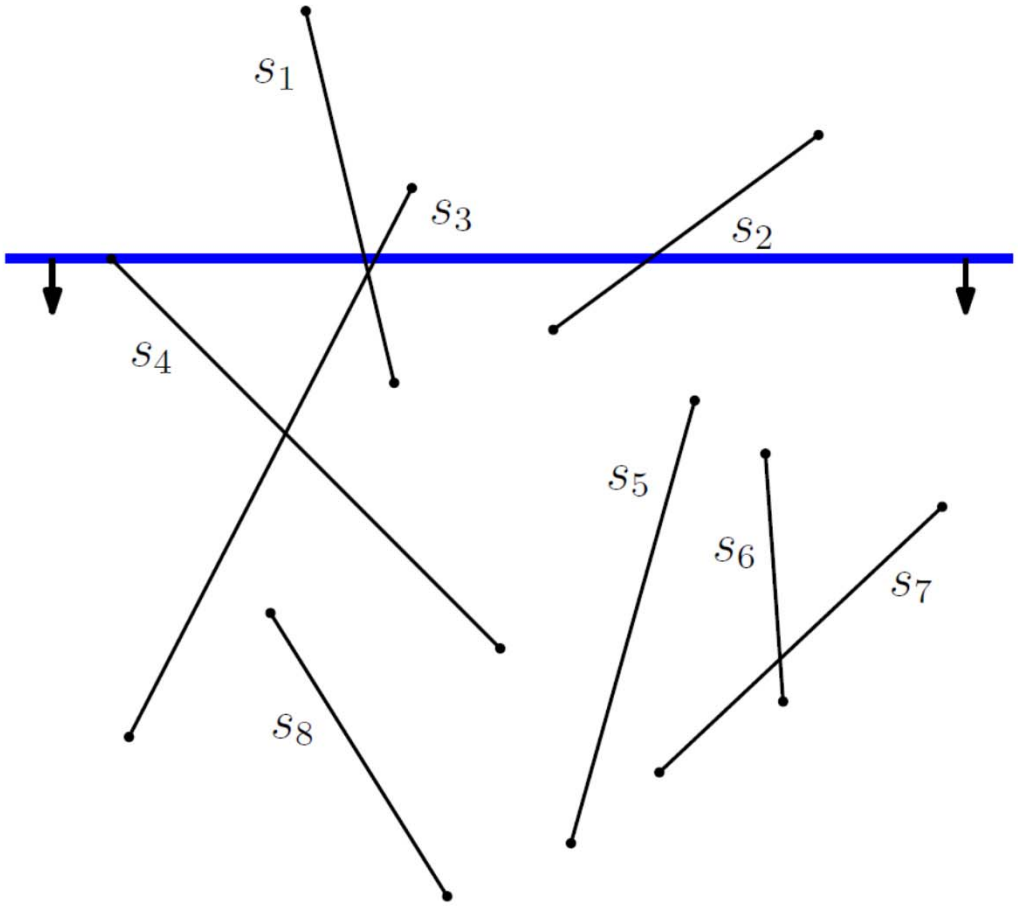
add s_2 after s_1

Plane Sweep



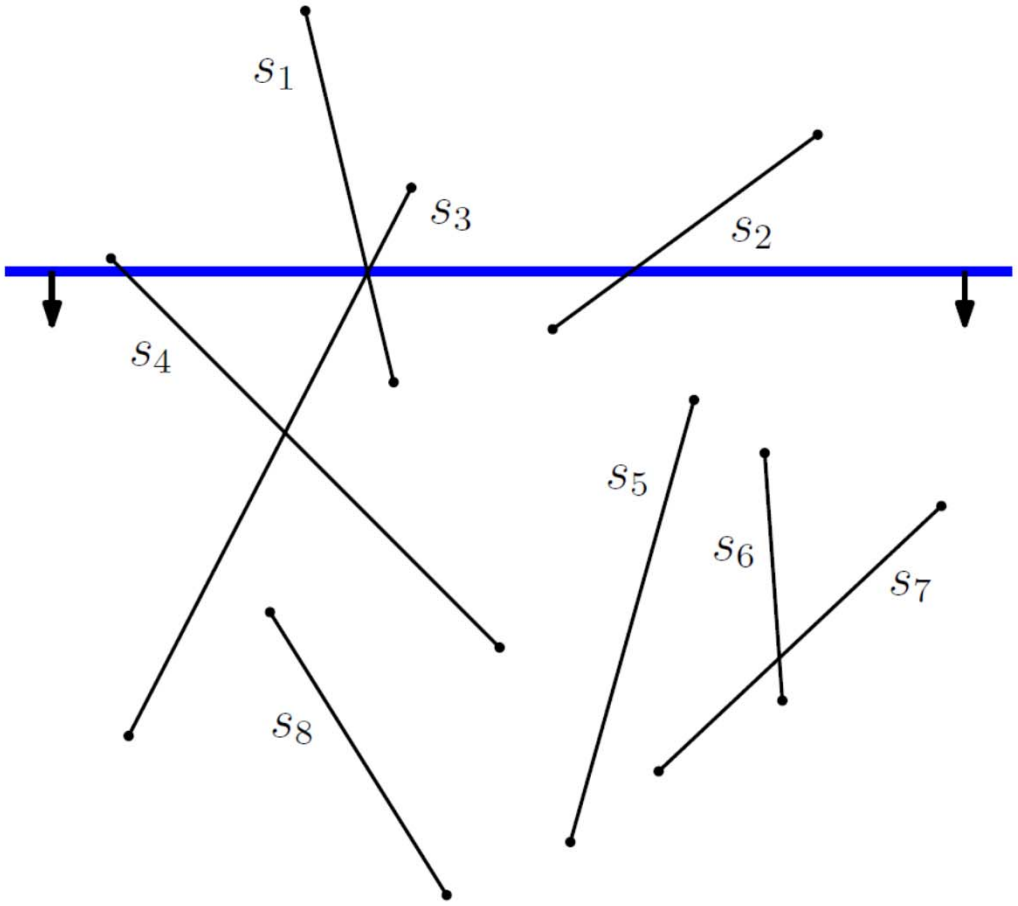
add s_3 between s_1 and s_2

Plane Sweep



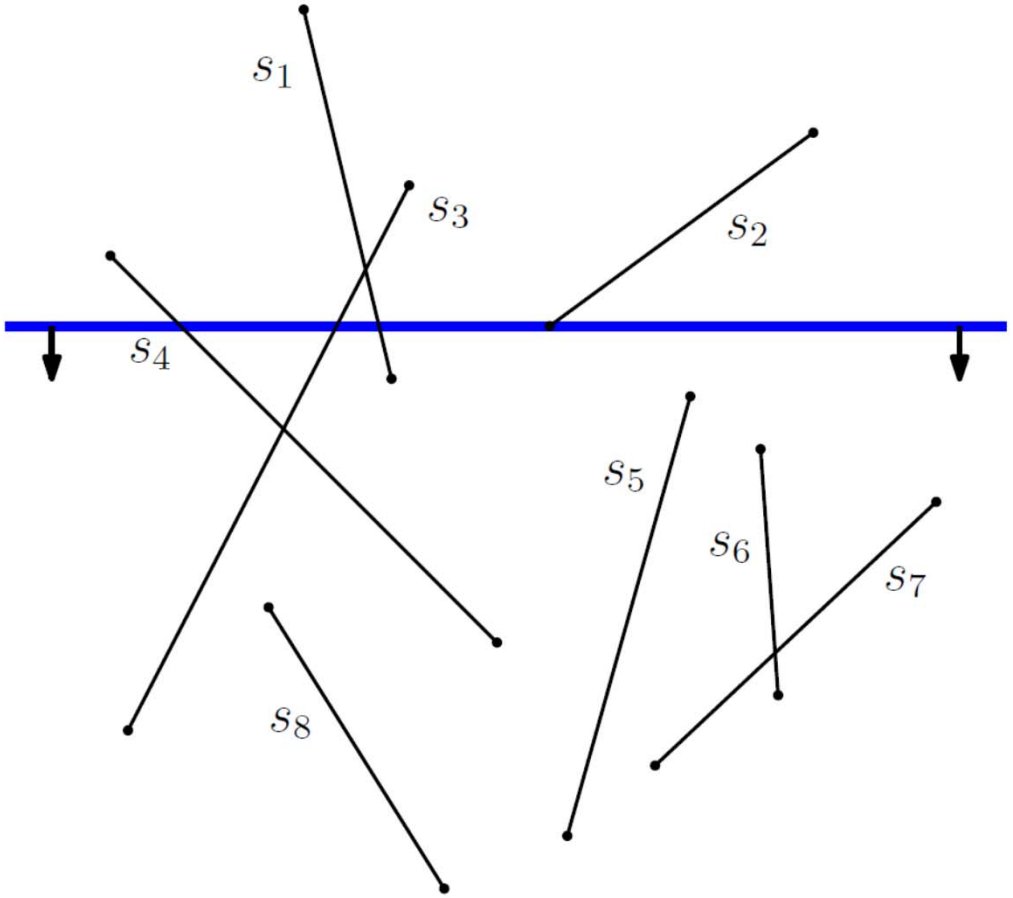
add s_4 before s_1

Plane Sweep



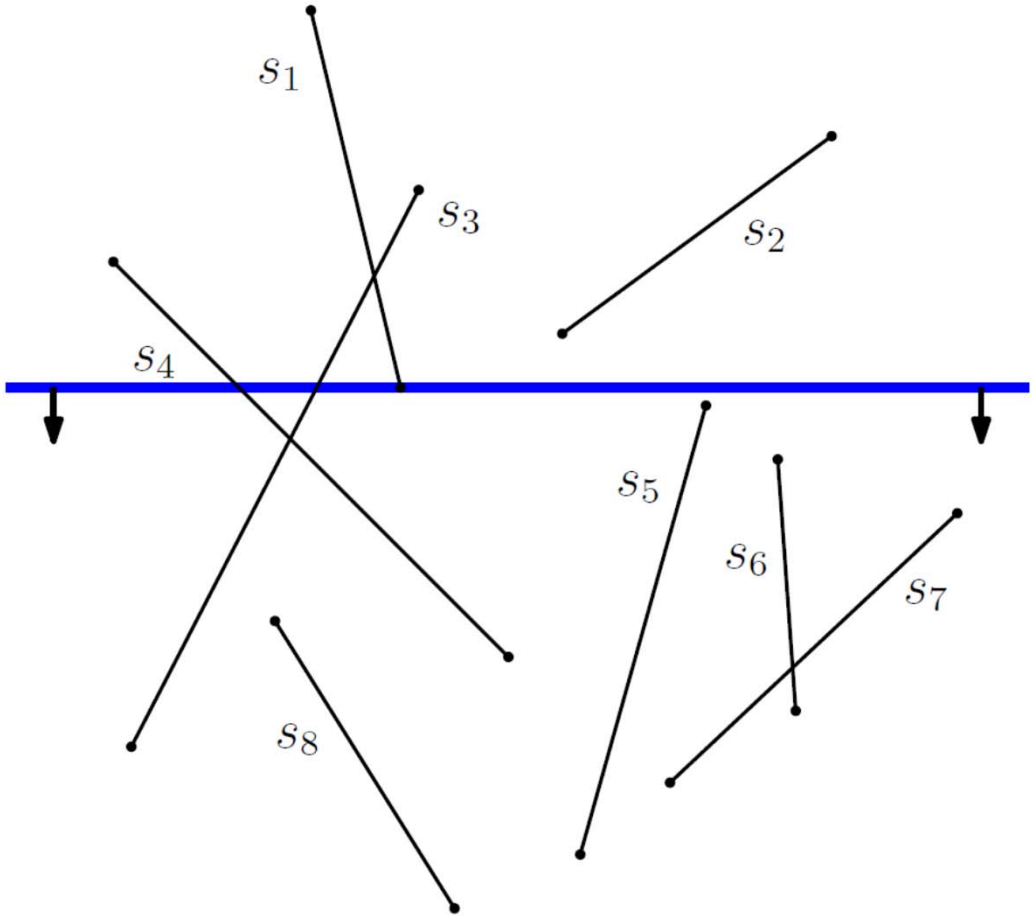
report intersection (s_1, s_2); swap s_1 and s_3

Plane Sweep



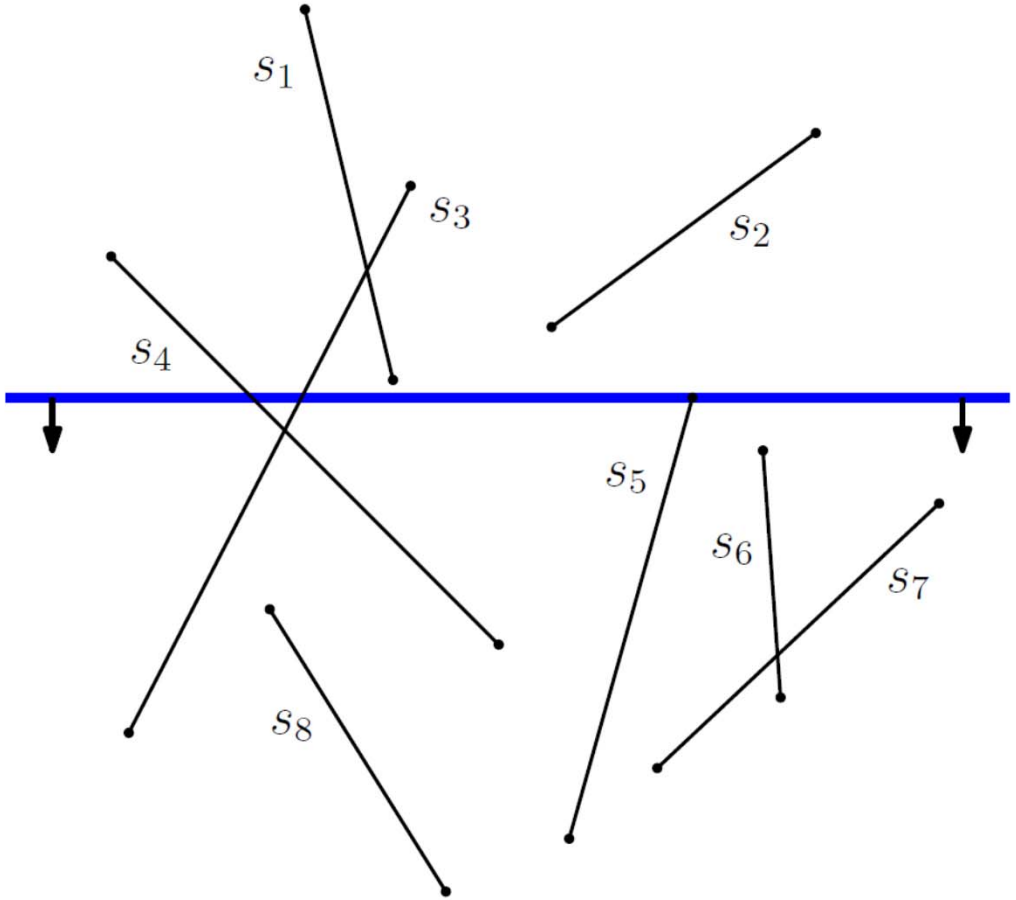
remove s_2

Plane Sweep



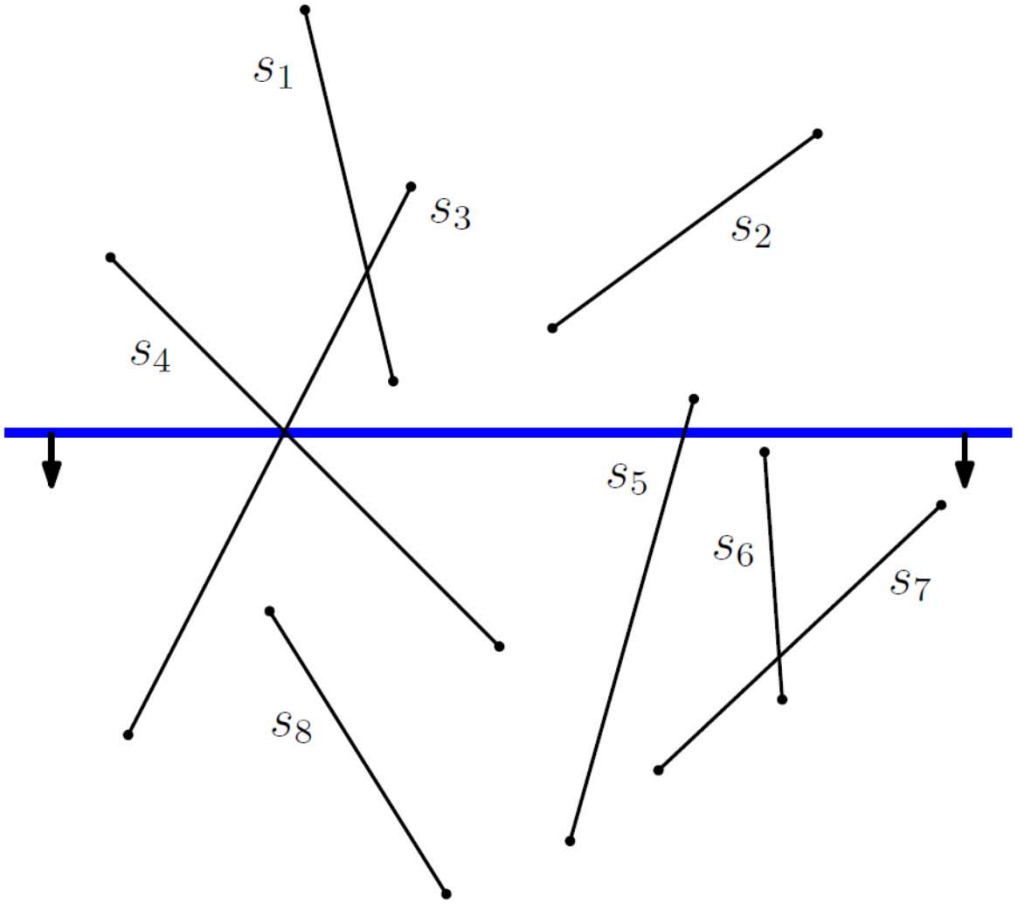
remove s_1

Plane Sweep



add s_5 after s_3

Plane Sweep



report intersection (s_3, s_4); swap s_3 and s_4

Plane Sweep

… and so on …

Events

When do the events happen? When the sweep line is at:

- an upper endpoint of a line segment
- a lower endpoint of a line segment
- an intersection point of a line segment

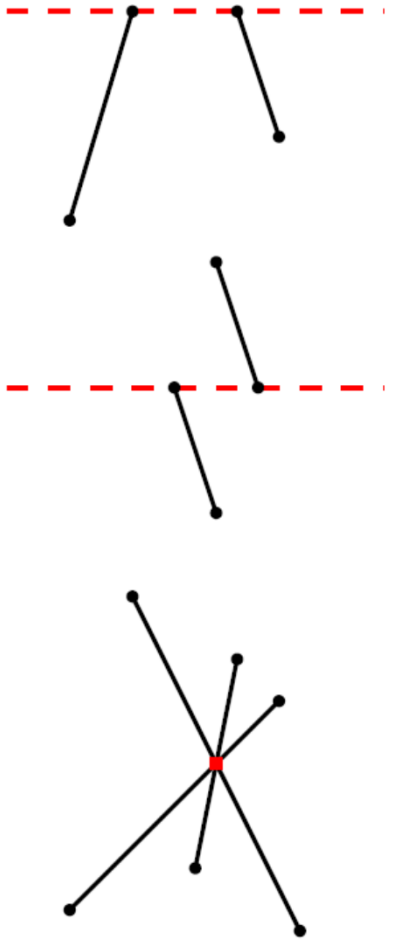
At each type, the **status** changes; at the third type **output** is found too

Assume No Degenerate Cases

We will at first exclude degenerate cases:

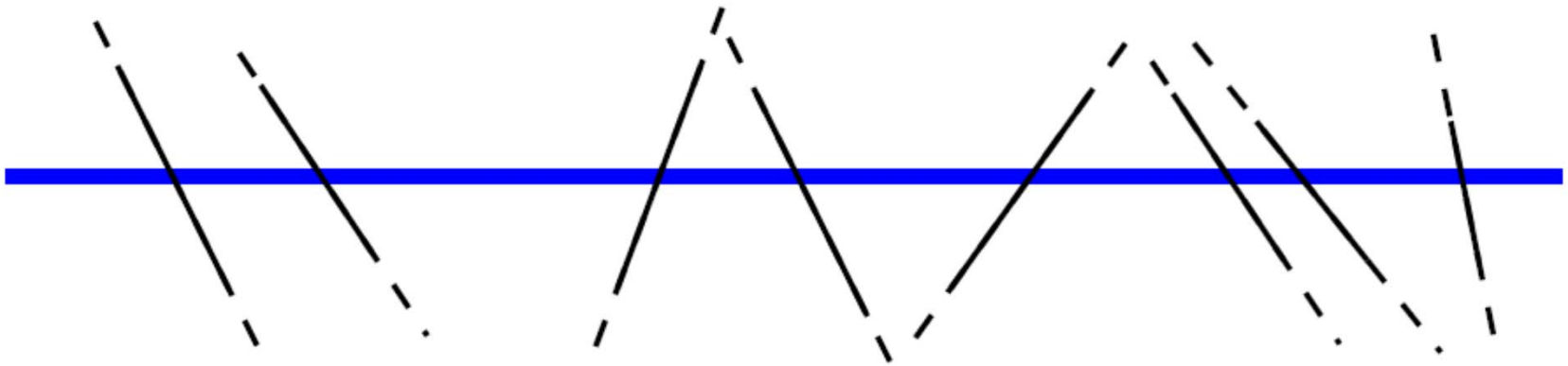
- No two endpoints have the same y-coordinate
- No more than two line segments intersect at a point
- …

Question: Are there more degenerate cases?



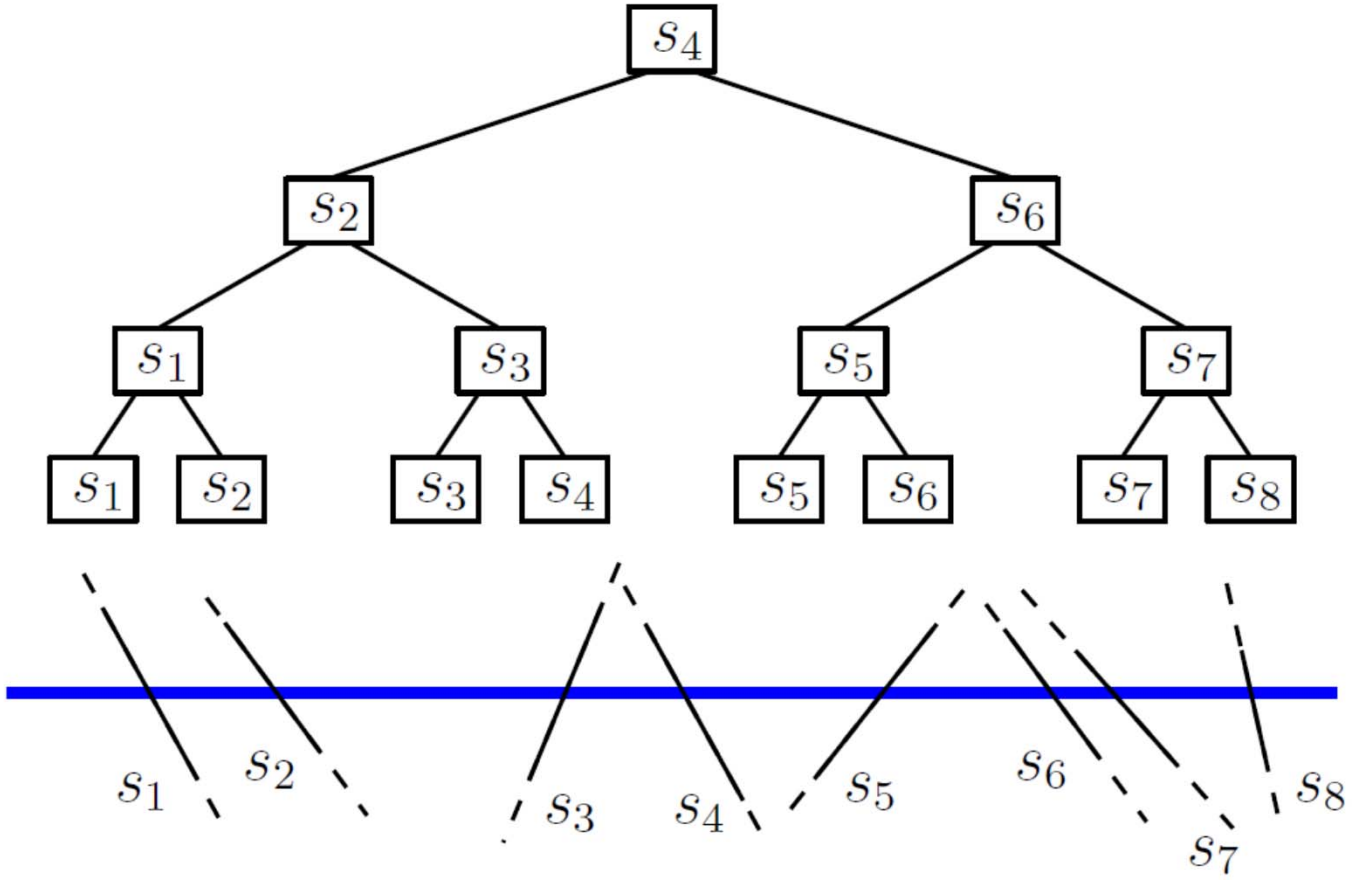
Event List and Status Structure

- The event list is an abstract data structure that stores all events in the order in which they occur
- The status structure is an abstract data structure that maintains the current status
- Here: The status is the subset of currently intersected line segments in the order of intersection by the sweep line



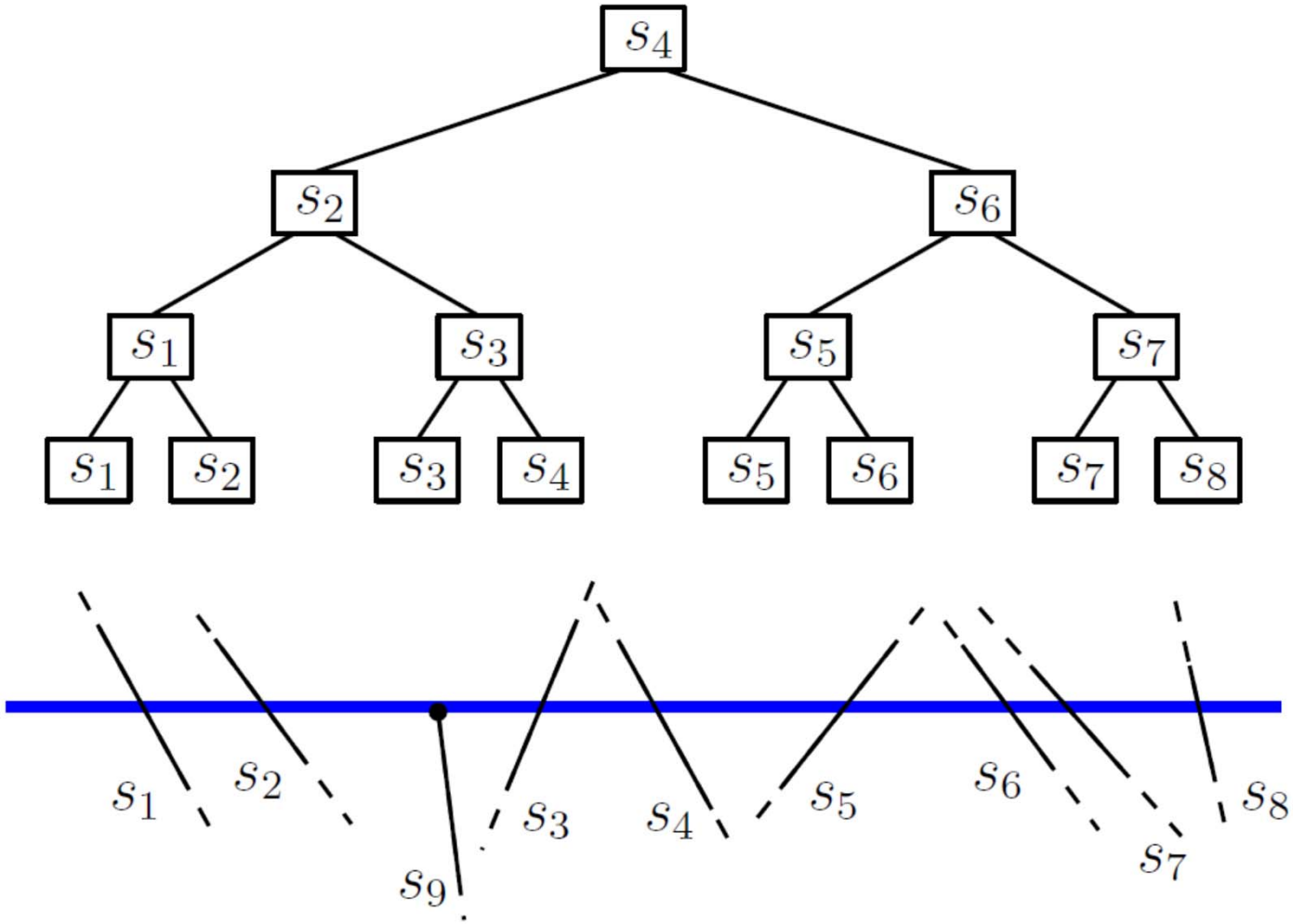
Status Structure

- We use a balanced binary search tree with the line segments in the leaves as the status structure



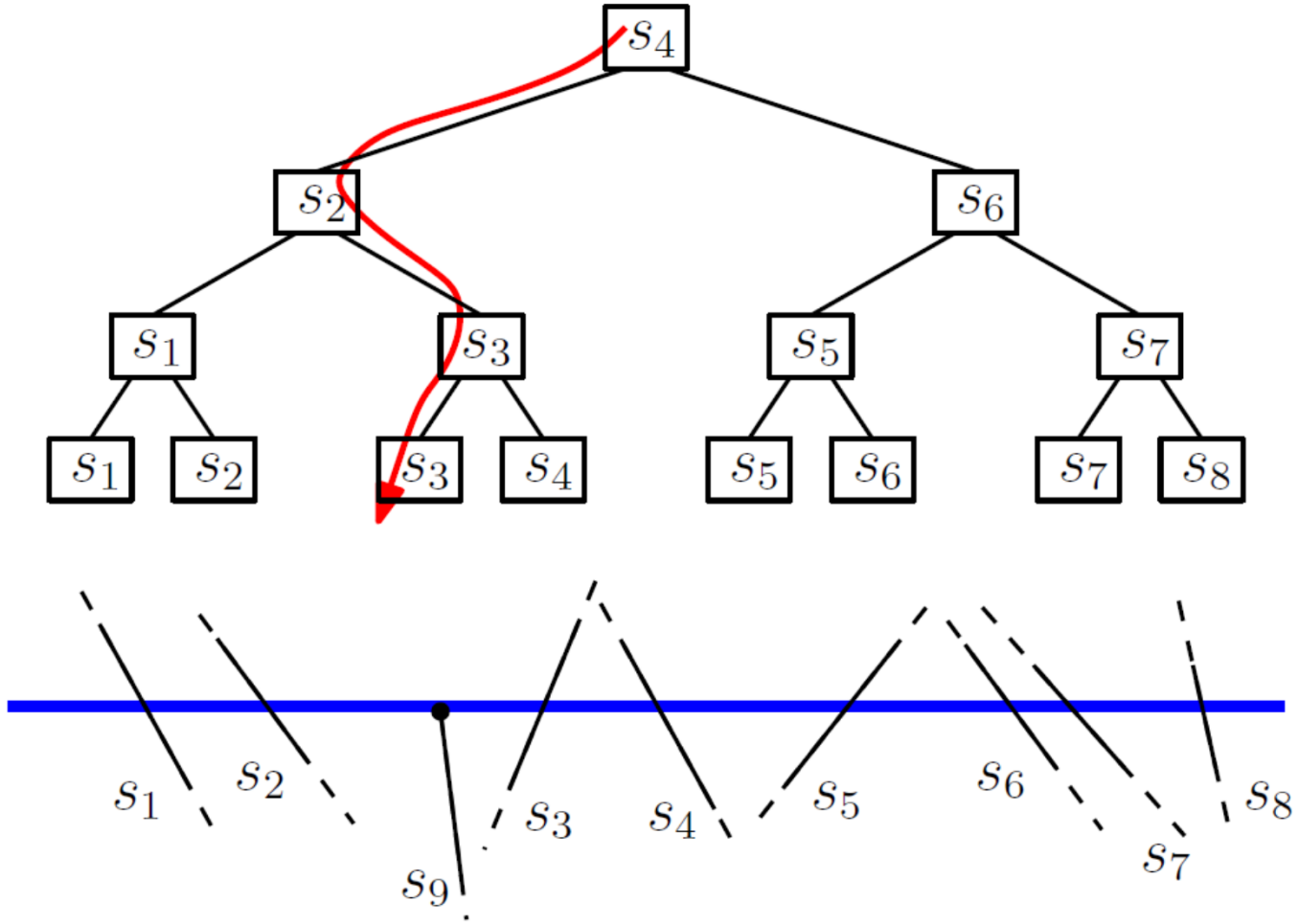
Status Structure

- Upper endpoint: search and insert



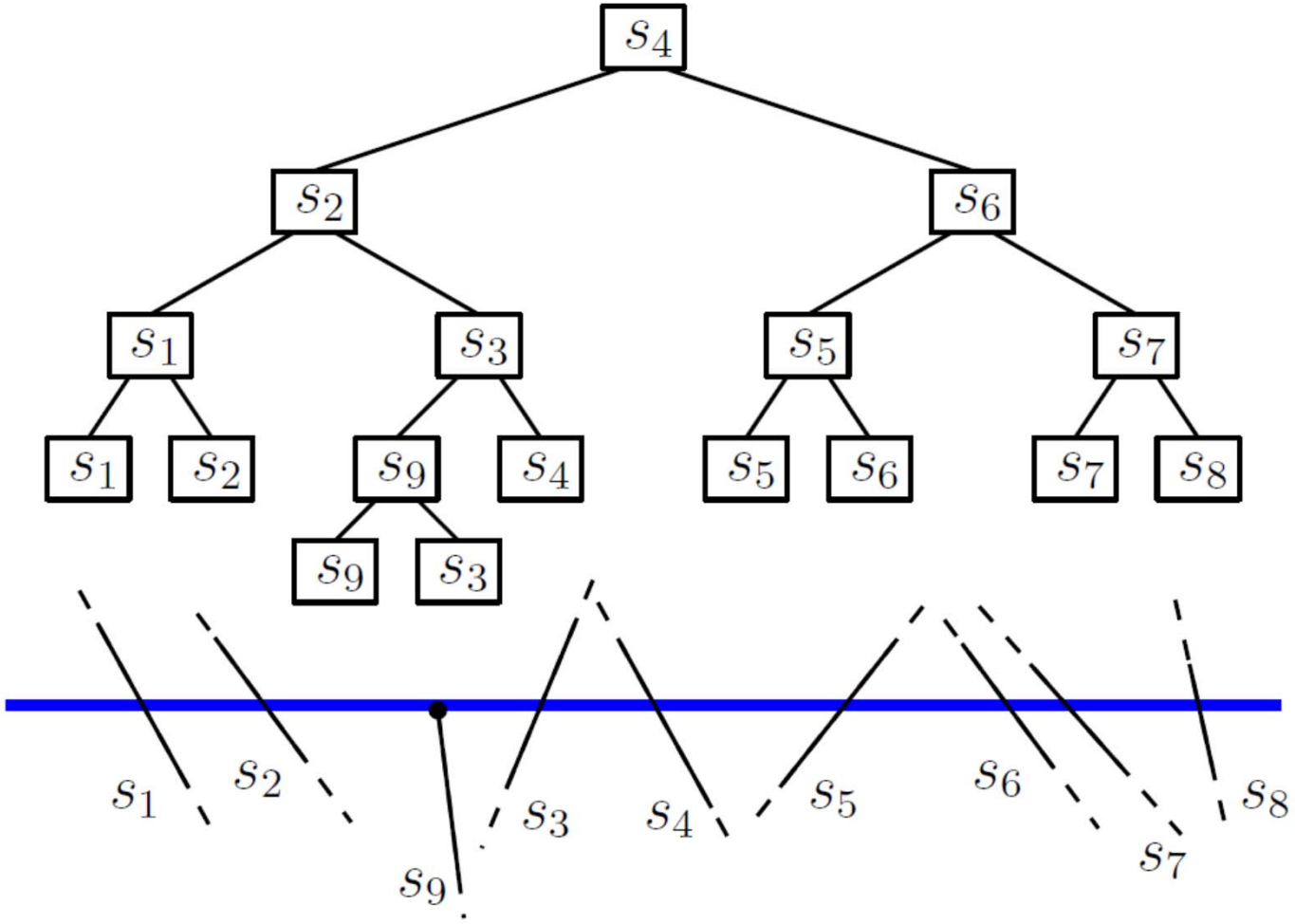
Status Structure

- Upper endpoint: search and insert



Status Structure

- Upper endpoint: search and insert



Status Structure

- Sweep line reaches lower endpoint of a line segment: delete from the status structure

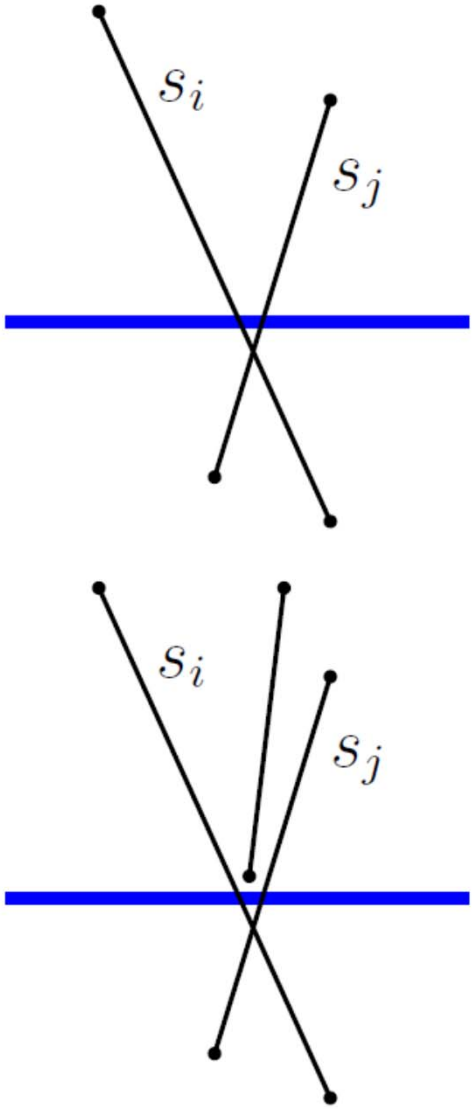
- Sweep line reaches intersection point: swap two leaves in the status structure (and update information on the search paths)

Finding Events

- Before the sweep algorithm starts, we know all upper endpoint events and all lower endpoint events
- But: How do we know intersection point events??? (these we were trying to find …)
- Recall: Two line segments can only intersect if they are horizontal neighbors

Finding Events

- Lemma: Two line segments s_i and s_j can only intersect after (= below) they have become horizontal neighbors
- Proof: Just imagine that the sweep line is ever so slightly above the intersection point of s_i and s_j, but below any other event, prove by contradiction
- Also: some earlier (= higher) event made s_i and s_j horizontally adjacent



Event List

- The event list must be a priority queue, because during the sweep, we discover new events that will happen later

- We know upper endpoint events and lower endpoint events beforehand; we find intersection point events when the involved line segments become horizontal neighbors

Plane Sweep Algorithm

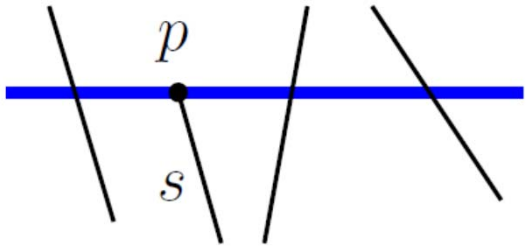
Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The intersection points of the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue Q. Insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it
2. Initialize an empty status structure T
3. **while** Q is not empty
4. **do** Determine next event point p in Q and delete it
5. HANDLEEVENTPOINT(p)

Event Handling

If the event is an upper endpoint event, and s is the line segment that starts at p:

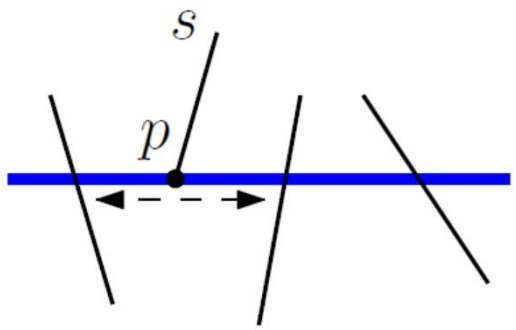
1. Search with p in T, and insert s
2. If s intersects its left neighbor in T, then determine the intersection point and insert in Q
3. If s intersects its right neighbor in T, then determine the intersection point and insert in Q



Event Handling

If the event is a lower endpoint event, and s is the line segment that ends at p:

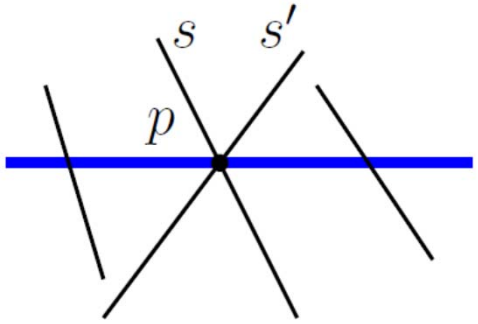
1. Search with p in T, and delete s
2. Let s_l and s_r be the left and right neighbors of s in T (before deletion). If they intersect below the sweep line, then insert their intersection point as an event in Q



Event Handling

If the event is an intersection point event where s and s' intersect at p:

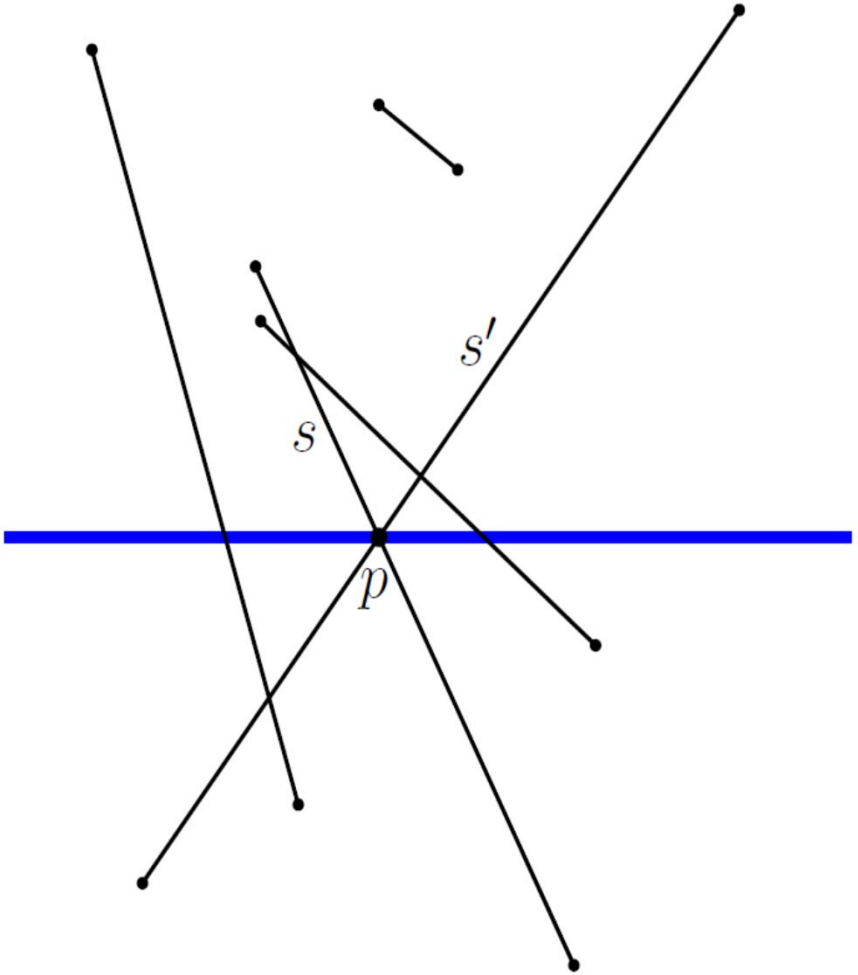
1. Exchange s and s' in T
2. If s' and its new left neighbor in T intersect below the sweep line, then insert this intersection point in Q
3. If s and its new right neighbor in T intersect below the sweep line, then insert this intersection point in Q
4. Report the intersection point



Event Handling

Is it possible that new horizontal neighbors already intersected above the sweep line?

Is it possible that we insert a newly detected intersection point event, but it already occurs in Q?



Efficiency

- How much time to handle an event?
- At most one search in T and/or one insertion, deletion, or swap
- At most twice finding a neighbor in T
- At most one deletion from and two insertions in Q
- Since T and Q are a balanced binary search tree and a priority queue, handling an event takes only O(logn) time

Efficiency

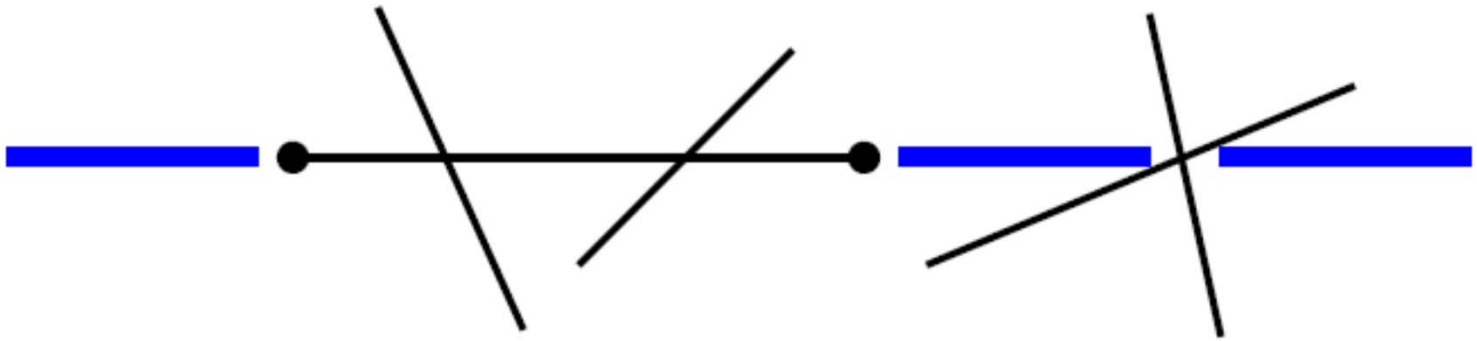
- How many events?
  - 2n for the upper and lower endpoints
  - k for the intersection points, if there are k of them
- In total: O(n+k) events

Efficiency

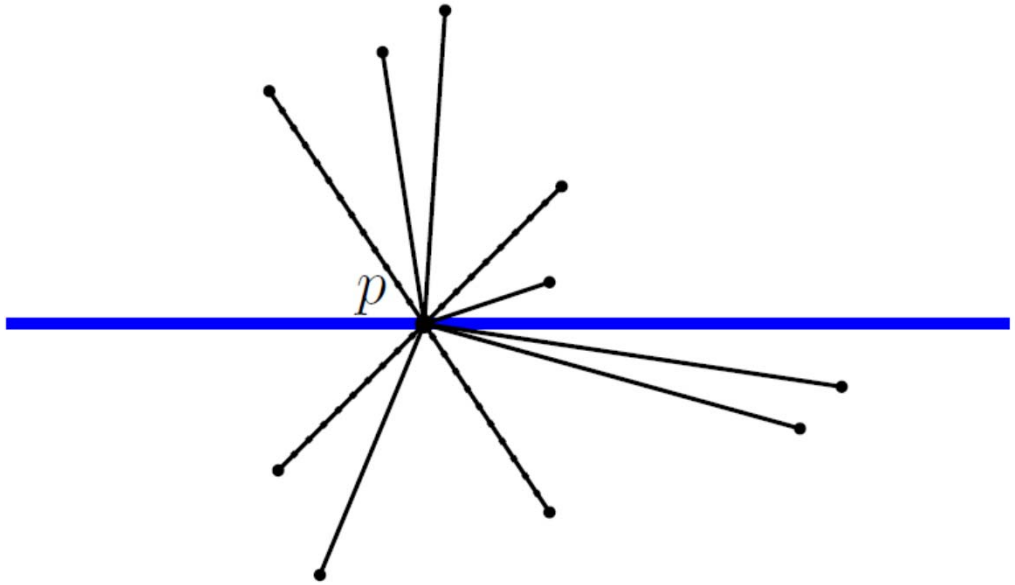
- Initialization takes O(nlogn) time (to put all upper and lower endpoint events in Q)
- Each of the O(n+k) events takes O(logn) time
- The algorithm takes O(nlogn+k logn) time
- If k = O(n), then this is O(nlogn)
- Note that if k is really large, the brute force $O(n^2)$ time algorithm is more efficient

Degenerate Cases

- How do we deal with degenerate cases?
- For two different events with the same y-coordinate, we treat them from left to right => the “upper" endpoint of a horizontal line segment is its left endpoint



Degenerate Cases



- How about multiply coinciding event points?
- Let U(p) and L(p) be the line segments that have p as upper and lower endpoint, and C(p) the ones that contain p

Degenerate Cases

- How efficiently is a multiply coinciding event point handled?
- If $|U(p)|+|L(p)|+|C(p)| = m$, then the event takes $O(m \log n)$ time
- What do we report?
  - The intersection point itself
  - Every pair of intersecting line segments
  - The intersection point and every line segment involved
- The output size for this one event is then $O(1)$, $O(m^2)$, or $O(m)$, respectively

General Sweep Algorithms

For every sweep algorithm:

- Define the status
- Choose the status structure and the event list
- Figure out how events must be handled (with sketches!)
- To analyze, determine the number of events and how much time they take

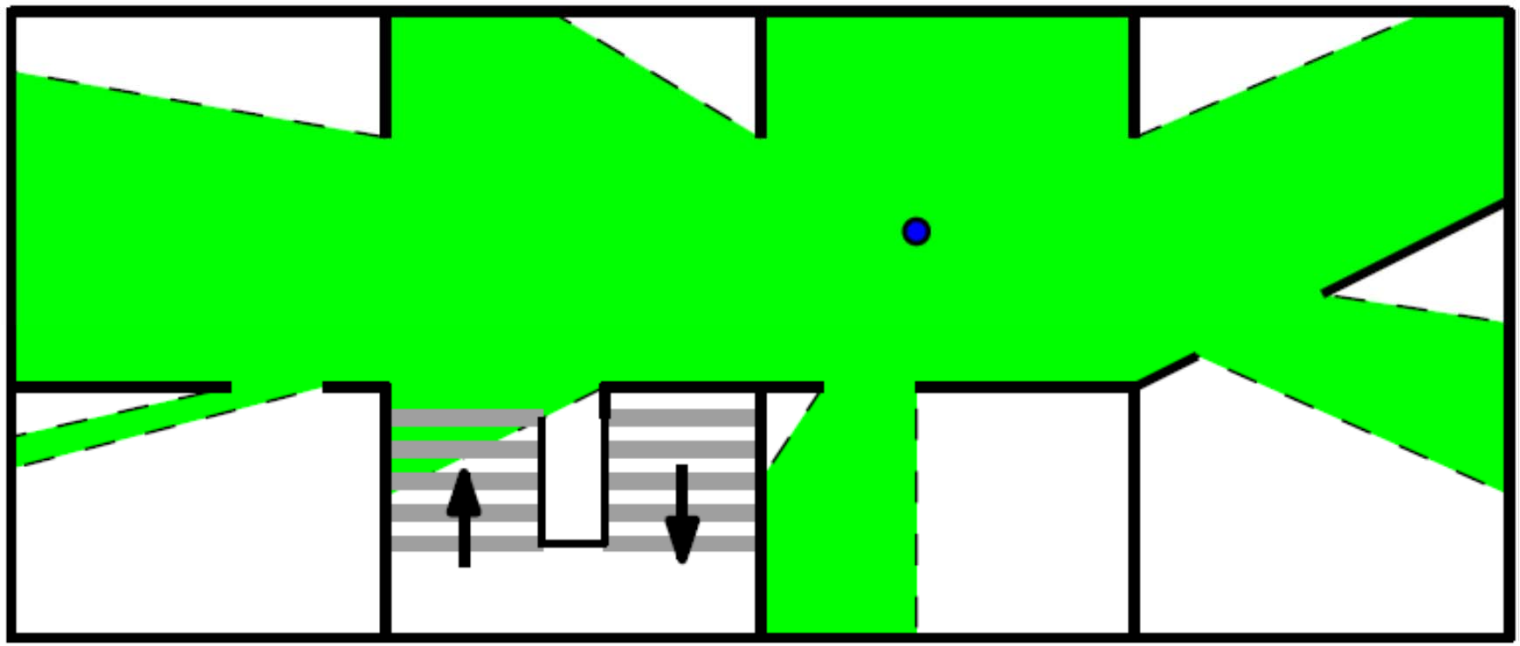
Then deal with degeneracies

Triangulating a Polygon (intro)

Slides by M. van Kreveld

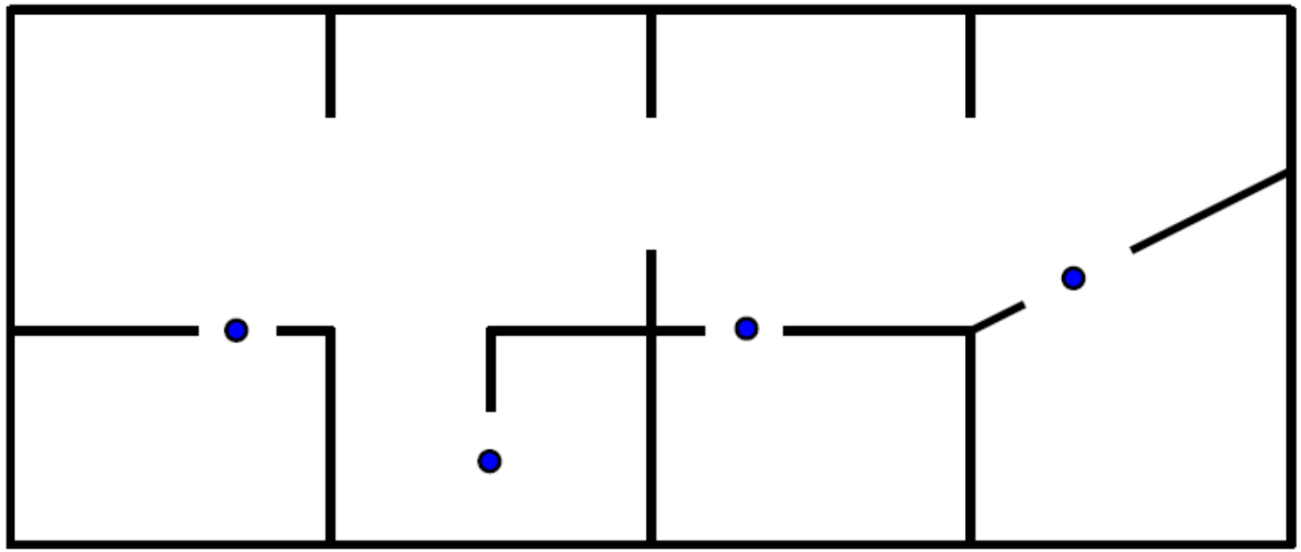
Polygons and Visibility

- Two points in a simple polygon can see each other if their connecting line segment is in the polygon



The Art Gallery Problem

- How many cameras are needed to guard a given art gallery so that every point is seen?

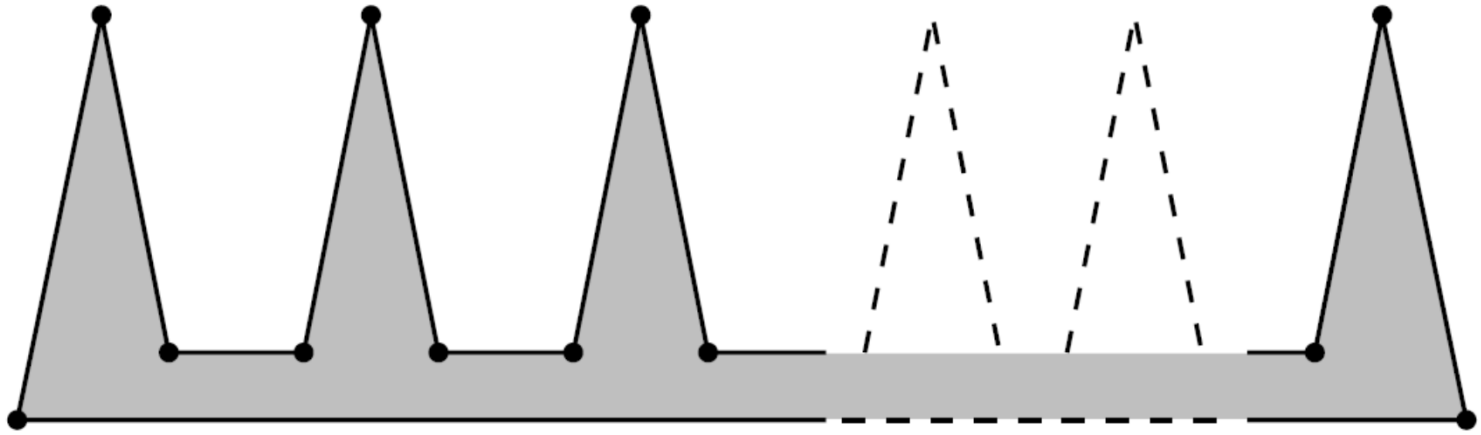


The Art Gallery Problem

- In geometry terminology: How many points are needed in a simple polygon with n vertices so that every point in the polygon is seen?
- The optimization problem is computationally difficult
- Art Gallery Theorem: $\lfloor n/3 \rfloor$ cameras are occasionally necessary but always sufficient

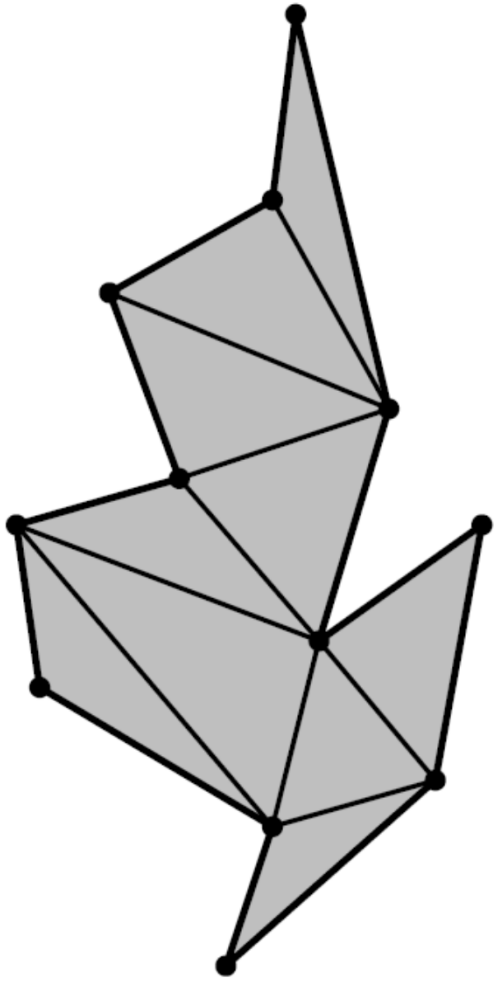
The Art Gallery Problem

- Art Gallery Theorem: $\lfloor n/3 \rfloor$ cameras are occasionally necessary but always sufficient



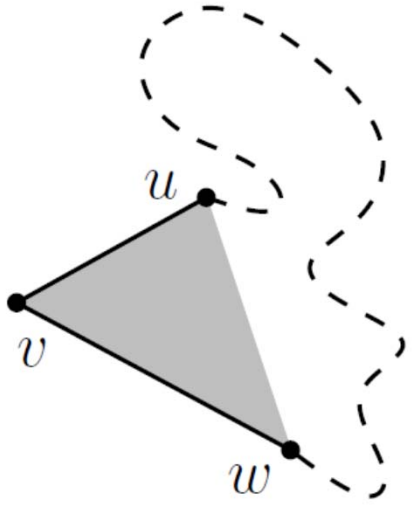
Diagonals

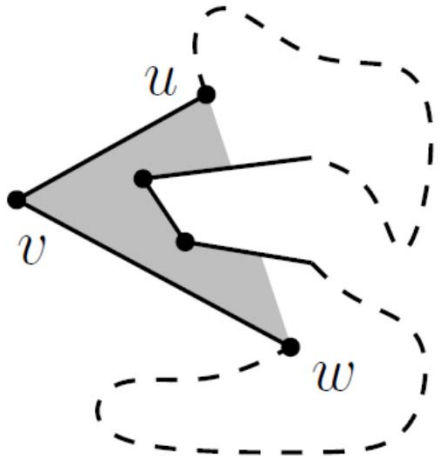
- Why are $\lfloor n/3 \rfloor$ cameras always enough?
- Assume polygon P is triangulated: a decomposition of P into disjoint triangles by a maximal set of non-intersecting diagonals
- Diagonal of P: open line segment that connects two vertices of P and fully lies in the interior of P



A Triangulation Always Exists

- Lemma: A simple polygon with n vertices can always be triangulated, and always with n-2 triangles
- Proof: Induction on n. If n = 3, it is trivial
- Assume n > 3. Consider the leftmost vertex v and its two neighbors u and w.
- Either uw is a diagonal (case 1), or part of the boundary of P is in Δuvw (case 2)
- Case 2: choose the vertex t in Δuvw farthest from the line through u and w, then vt must be a diagonal





A Triangulation Always Exists

- In case 1, uw cuts the polygon into a triangle and a simple polygon with n-1 vertices, and we apply induction

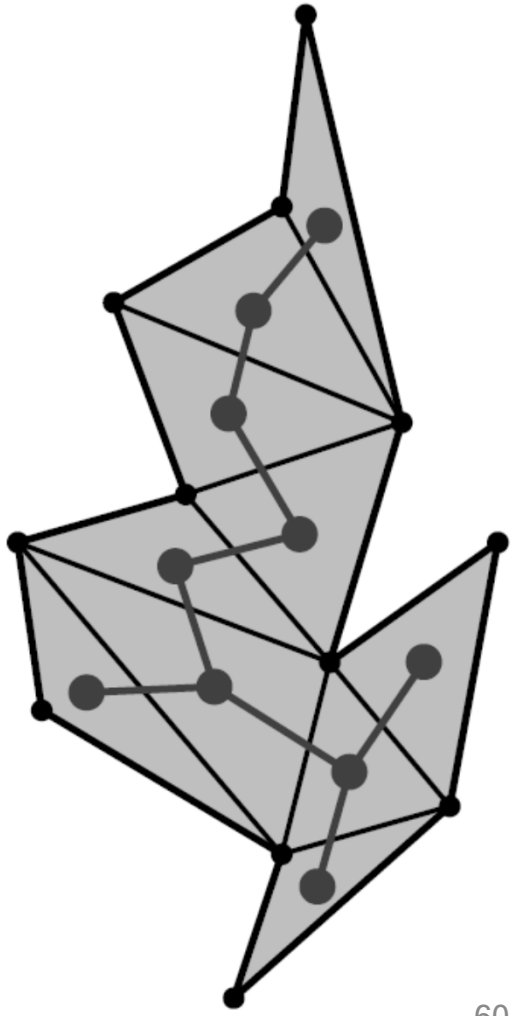
- In case 2, vt cuts the polygon into two simple polygons with m and n - m + 2 vertices, $3 \leq m \leq n - 1$, and we also apply induction

- By induction, the two polygons can be triangulated using m - 2 and n - m + 2 - 2 = n - m triangles. So the original polygon is triangulated using m - 2 + n - m = n – 2 triangles

A 3-coloring Always Exists

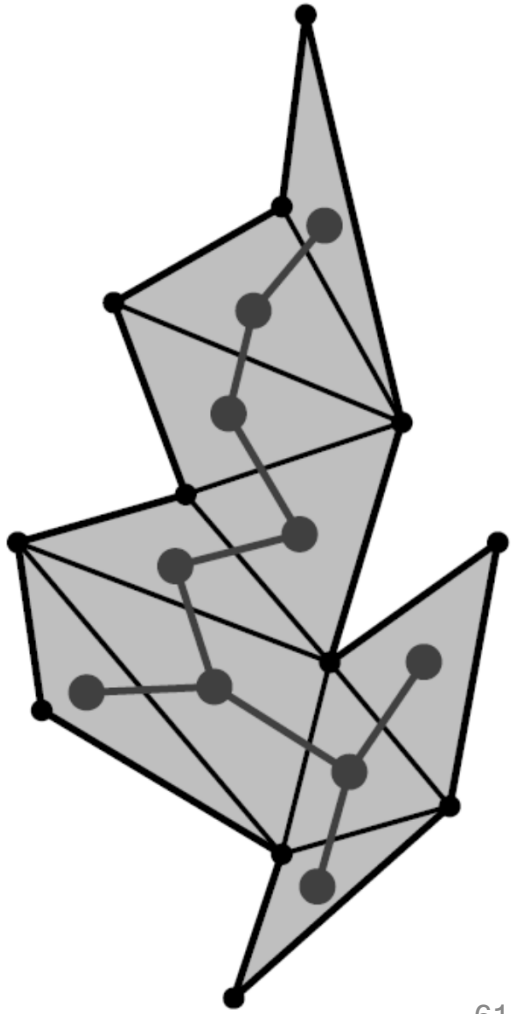
- Observe: the dual graph of a triangulated simple polygon is a tree

- Dual graph: each face gives a node; two nodes are connected if the faces are adjacent

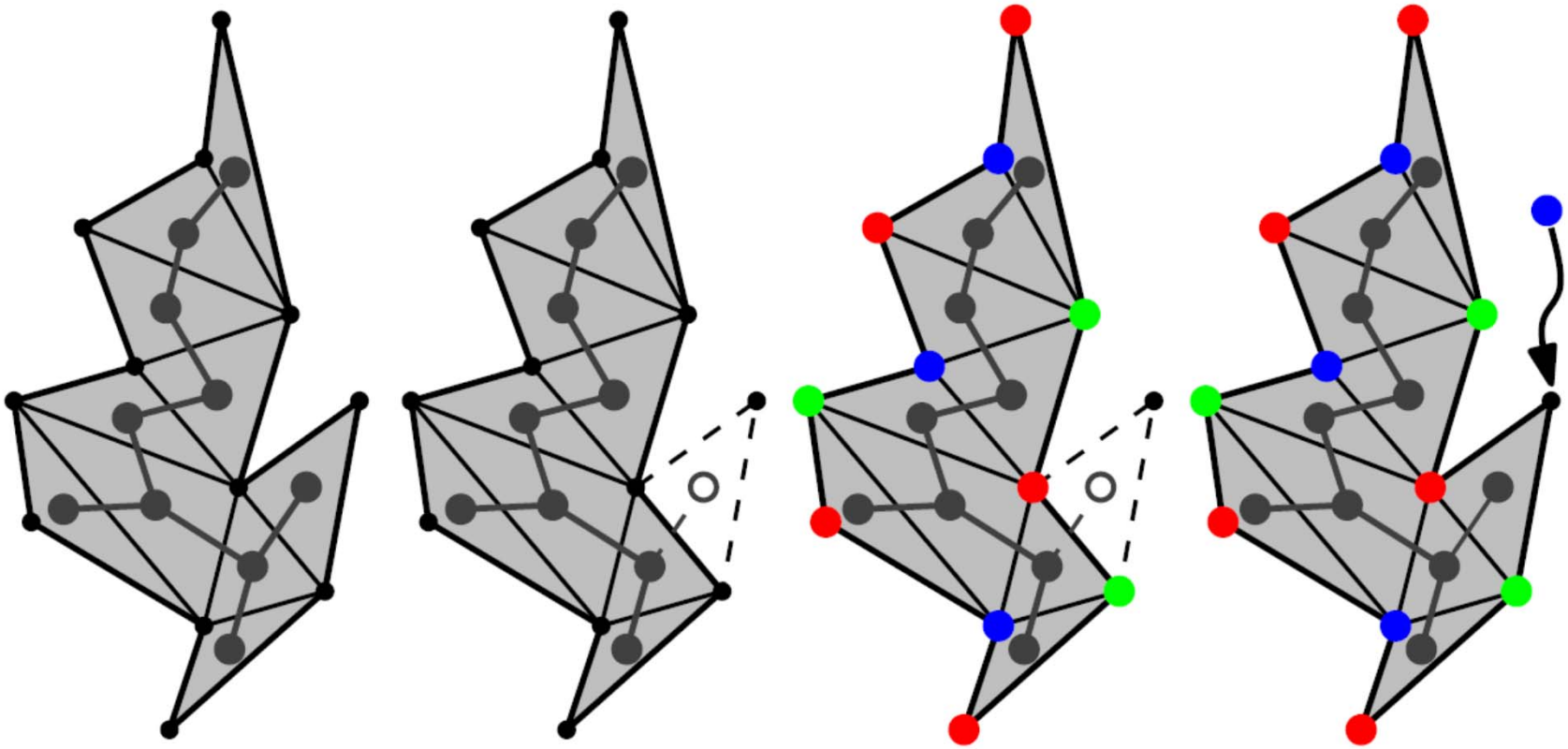


A 3-coloring Always Exists

- Lemma: The vertices of a triangulated simple polygon can always be 3-colored
- Proof: Induction on the number of triangles in the triangulation. Base case: True for a triangle
- Every tree has a leaf. Remove the corresponding triangle from the triangulated polygon, color its vertices, add the triangle back, and let the extra vertex of the neighboring triangle have the color that is not present at its neighbors



A 3-coloring Always Exists



$\lfloor n/3 \rfloor$ Cameras are Enough

- For a 3-colored, triangulated simple polygon, one of the color classes is used by at most $\lfloor n/3 \rfloor$ colors.
  - Place the cameras at these vertices
- This argument is called the pigeon-hole principle
- Why does the proof fail when the polygon has holes?