CS 164 & CS 266:
Computational Geometry

Lecture 2

Projective geometry and line segment intersection

David Eppstein
University of California, Irvine

Fall Quarter, 2025

Projective geometry

Motivation

Cartesian coordinates — (x, y) pairs — are familiar and work well for many purposes

Sometimes other coordinates are easier to use

Example:

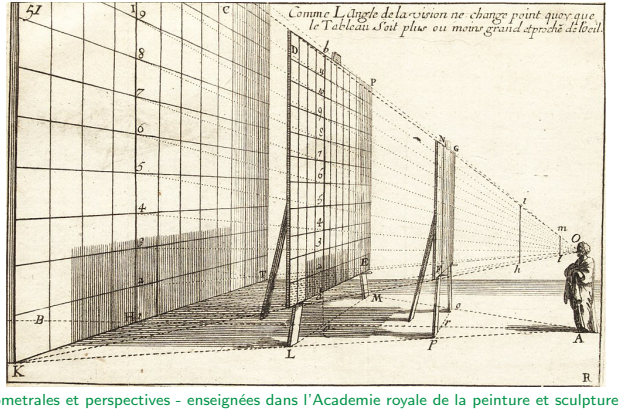
> If we represent most lines as (m, b) where $y = mx + b$, finding the line through two points uses division, problematic if we want integers
>
> And what about vertical lines $x = c$?

Projective geometry eliminates both problems

Equivalence between 2d points and 3d lines

Map plane into 3d by $(x, y) \mapsto (x, y, 1)$, view from $(0, 0, 0)$

Would get same view for all scaled embeddings (ax, ay, a)
All points (ax, ay, a) on a line through $(0, 0, 0)$ look the same!



File:Traité des pratiques geometrales et perspectives - enseignées dans l'Academie royale de la peinture et sculpture (1665) (14762910201).jpg

Projective geometry

New meaning for the words "point" and "line":

- "Point": 3d line through $(0, 0, 0)$

  Represent by 3d coordinates of any nonzero point on the line

  Many different triples all represent the same "point":
  $(x, y, z) = (0.5x, 0.5y, 0.5z) = (3x, 3y, 3z) = \cdots$

- "Line": 3d plane through $(0, 0, 0)$

Contains all points and lines of the Euclidean plane

$$(x, y) \mapsto (x, y, 1) \qquad (x, y, z) \mapsto (x/z, y/z)$$

But it also contains extra "points" (x, y, z) with $z = 0$,
called "points at infinity" (although x, y, z are all finite numbers)

Lines in projective geometry

Line: set of points (x, y, z) obeying an equation $ax + by + cz = 0$

Important requirement: at least one of a, b, or c must be $\neq 0$

(Otherwise: Every point obeys the equation.)

Coordinates of the line: the numbers a, b, c

Many different triples all represent the same line:
$(a, b, c) = (0.5a, 0.5b, 0.5c) = (3a, 3b, 3c) = \cdots$

Converting between coordinates

Euclidean to projective

Point $(x, y) \Rightarrow (x, y, 1)$

Line $(m, b) \Rightarrow (m, -1, b)$
$y = mx + b \Rightarrow mx - y + bz = 0$

Vertical line $c \Rightarrow (1, 0, -c)$
$x = c \Rightarrow x + 0y - cz = 0$

Projective to Euclidean

Point $(x, y, z \Rightarrow (x/z, y/z)$

Line $(a, b, c) \Rightarrow (-a/b, -c/b)$
$ax + by + cz = 0 \Rightarrow y = -ax/b - c/b$

Vertical line $(a, 0, c) \mapsto -c/a$
$ax + cz = 0 \mapsto x = -c/a$

But points at infinity $(x, y, 0)$ and line at infinity $(0, 0, 1)$
do not correspond to anything in Euclidean geometry!

What are these extra points?

We can think of:

- The plane as a surface that we're viewing from slightly above it
- The line at infinity is the horizon
- The points at infinity are the vanishing points where parallel lines meet

Point-on-line primitive and projective duality

To test if point (x, y, z) is on line (a, b, c):
compute dot product $(x, y, z) \cdot (a, b, c)$, check if zero

Unaffected by multiplying the coordinates of either by a scalar

Unaffected by which triple is a "point" and which we call a "line"

So if we reinterpret all lines in projective geometry as being points, and all points as being lines, it doesn't affect any properties defined using point–line intersection tests!

The space of points and lines formed by renaming the lines and points of projective geometry in this way is the projective dual

Algorithmic applications of projective duality

Whenever we have a valid mathematical statement about points, lines, and point-line incidence in projective geometry, we can change all points to lines and all lines to points in the statement and get another valid statement.

Example 1: every two points have a line that touches both of them

Dual 1: every two lines have a point that touches both of them

Not true of Euclidean parallel lines!

Example 2: f is a subroutine that takes as input two distinct points and outputs the line through them (the line that touches both points)

Dual 2: f is a subroutine that takes as input two distinct lines and outputs the point where they cross (the point that touches both lines)

The same subroutine does two different things!

To find line through two points / point on two lines

Math version:

Line through (x_1, y_1, z_1) and (x_2, y_2, z_2) is the set of points (x, y, z) for which

$$\det \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x & y & z \end{pmatrix} = 0$$

Because this equation is linear in x, y, z and is true of the two given points

Computer science version:

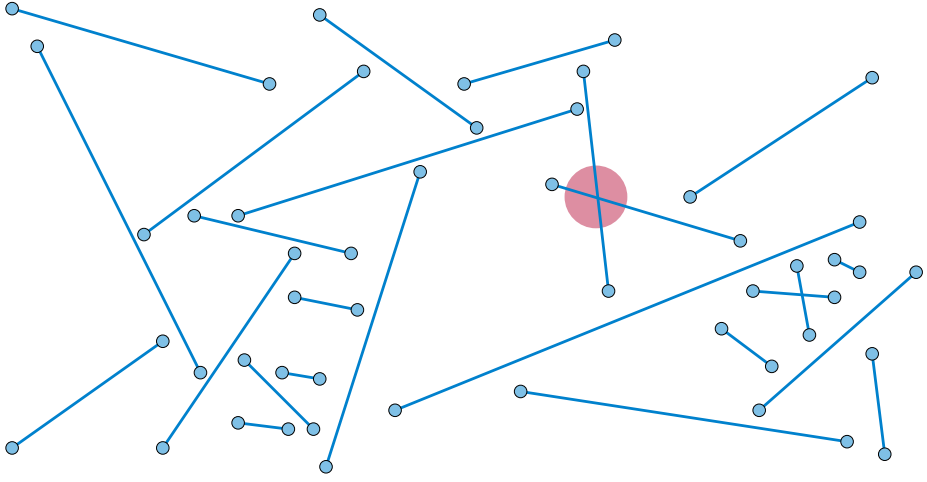
```
def thru2(p,q)
    x1,y1,z1 = p
    x2,y2,z2 = q
    x = y1*z2-y2*z1
    y = z1*x2-z2*x1
    z = x1*y2-x2*y1
    return (x,y,z)
```

Using projective geometry has given us two useful primitives!
Even when we have Cartesian coordinates we can use these by converting to projective

Intersection detection and crossing listing

Intersection detection problem

Input: A list of line segments (each one: Cartesian coords of 2 endpoints)



Output: Do any two cross? If so report a crossing (or maybe all)

Application: Test validity of geographic data or circuit designs

Naïve solution

For each pair of input segments:
test whether they cross

$O(n^2)$

Using the primitives we've seen:

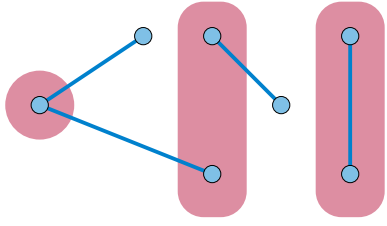
- Test for crossing using four left-right tests on triples of points
- Find line through two points, and point where lines cross, via projective geometry

$O(1)$

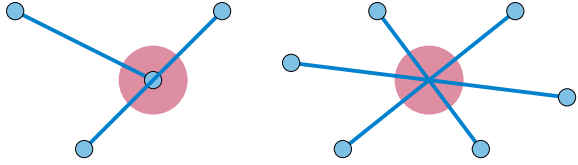
But can we do better?

(Unrealistic) general position assumptions

- No two segment endpoints have the same x-coordinate
  (Implies: No vertical segments)
- No endpoint lies on another segment
- No three segments cross at a single point



likely to occur
probably ok

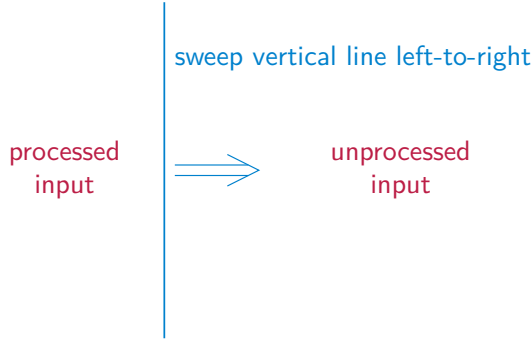


should be reported
as being crossings

Plane sweep algorithms

Plane sweep

General approach for designing geometric algorithms



The combinatorial structure of the result will only change at a finite set of discrete "event points" — process these in left-to-right order

Crossed segment data structure

As sweep line sweeps left-to-right across the input, maintain:

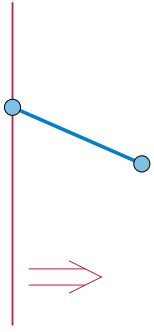
- Set of line segments that cross it
- Vertical ordering of these line segments in a binary search tree

Vertical ordering will help us find crossings when we sweep over them, because it changes at those points

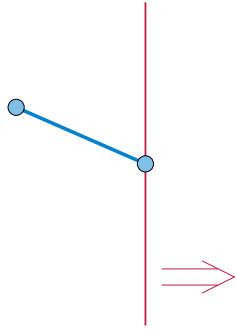
We can maintain it efficiently using a balanced binary search tree

Search tree operations only need to know the ordering of the segments, not the precise coordinates of the points where they cross the sweep line

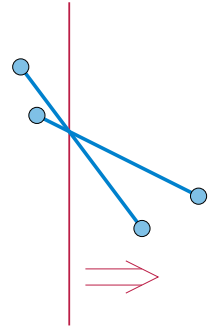
How the vertical ordering can change



cross left endpoint
of segment —
add to set of segments
crossed by sweep line



cross right endpoint
of segment —
remove from segments
crossed by sweep line



sweep over
crossing point —
swap in vertical order
of crossed segments

Event queue data structure

Keep track of future event points (where vertical ordering changes) in a priority queue, prioritized by x-coordinate (position in the left-to-right ordering used by the sweep line)

For crossing detection, we only need sorted list of segment endpoints

For listing all crossings, we also include the crossings we have found so far

In some algorithms we may also include "potential" events that we think might happen, but that can be removed from the event queue before they actually happen

Crossing detection pseudocode

- Initialize T to an empty binary search tree
- Initialize Q to be a sorted list of segment endpoints
- For each point p in Q:
  - If p is a left endpoint of a segment S: Add S to T, and check for crossings between S and its neighbors above and below it in the vertical crossing order (found using T)
  - Else p is a right endpoint of a segment S; remove S from T, and check for crossings between the two segments that were above and below it in the vertical crossing order

If we ever find a crossing, stop the whole algorithm and report it

If any two segments cross, they will be adjacent just before the sweep line sweeps over the crossing, and this algorithm will check them and discover the crossing

Listing all crossings

- Initialize T to an empty binary search tree
- Initialize Q to a priority queue of points, prioritized by x-coordinate, initially containing all segment endpoints
- While Q is non-empty:
  - Find the minimum-priority point p in Q and remove it from Q
  - If p is a left endpoint of a segment S: Add S to T, and check for crossings between S and its neighbors above and below it in the vertical crossing order (found using T)
  - Else if p is a right endpoint of a segment S; remove S from T, and check for crossings between the two segments that were above and below it in the vertical crossing order
  - Else p is a crossing point of two segments; swap the segments in T, and check for crossings between them and the two segments above and below them in T

Whenever we find a new crossing point, just insert it into Q

Analysis (of both algorithms)

Let n be the number of segments (so there are $2n$ endpoints) Let k be the number of crossing points; $0 \leq k \leq \binom{n}{2}$.

The detection algorithm has $2n$ events; the crossing listing algorithm has $2n + k$

Each event is performed using a constant number of operations in binary search trees and priority queues

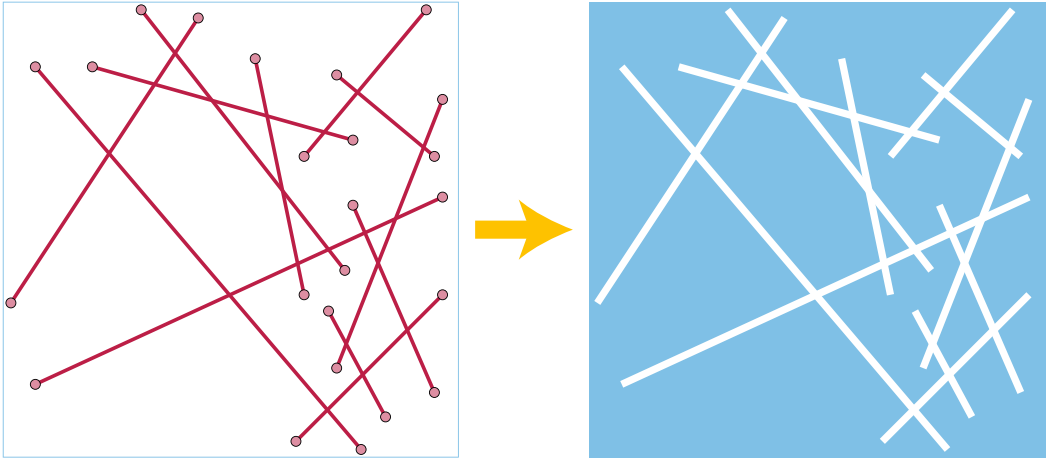
Total time: $O(n \log n)$ for detection, $O((n + k) \log n)$ for crossing listing

Arrangements and their representation

Arrangement

Think of any system of segments or curves as barriers to motion

"Face": 2d region within which you can get between any two points



How to find and represent this system of faces and their boundaries?

Some terminology

Face

2d connected region

Edge

1d boundary of two faces, separated by part of a segment

Same face can be on both sides

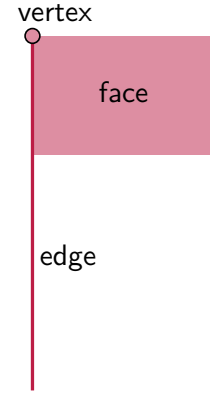
Vertex

An endpoint of a segment, or crossing point of segments

At an intersection point, multiple edges come together

Flag

A vertex, edge, and face that all touch each other



Representation issues

Most representations are centered on the edges of an arrangement

Each edge touches two vertices at its ends,
and two faces on its two sides

There are representations with:

one object per edge (pointing to all four of these things)

two objects per edge (one for each of its two sides)

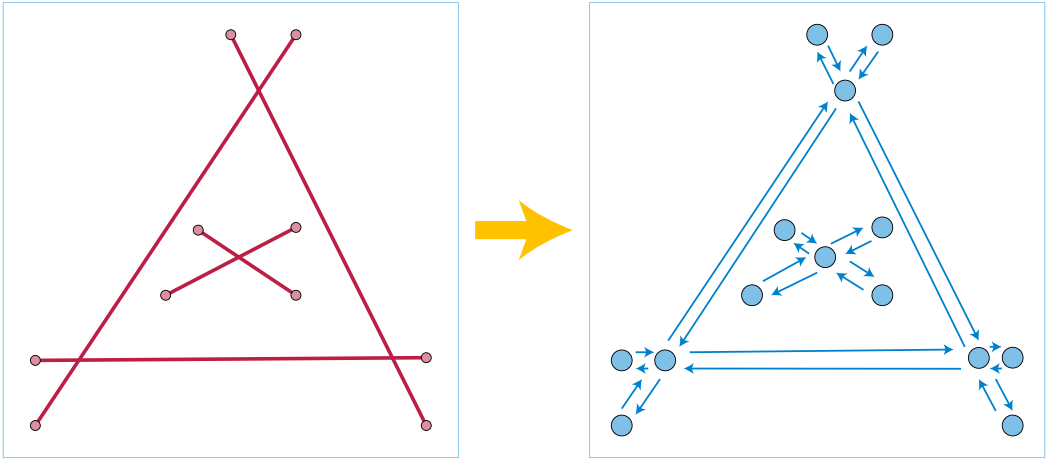
four objects per edge (one per flag)

Structure from our text: two objects per edge

Half-edges

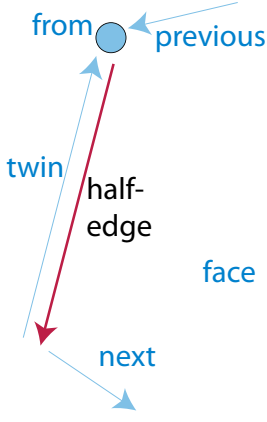
Represent each edge of the arrangement by two directed edges ("half-edges"), like the two directions of a two-way road

(But like in England or Japan where they drive on the left)



Doubly-connected edge list

Object-oriented, with objects for vertices, half-edges, and faces



Each half-edge stores:

- Pointer to twin half-edge from same edge
- The vertex it comes from (Can find other vertex from twin)
- The face on its side of the edge
- The next and previous half-edges in the cycle around its face

Each vertex stores:

- Its coordinates
- One of the half-edges it touches

Each face stores:

- A half-edge on its outer boundary
- A list of half-edges, one for each internal boundary

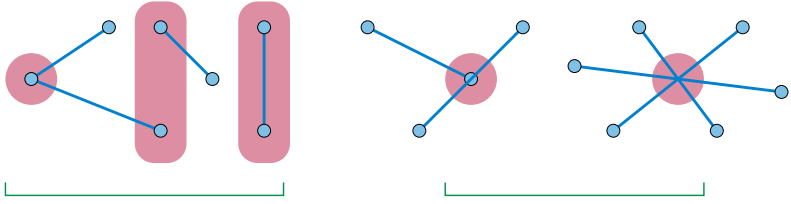
Constructing the arrangement of line segments

Same plane sweep algorithm, maintaining DCEL of the part of the arrangement to the left of the sweep line

Updates to DCEL at sweep events:

- When we sweep over an endpoint of one or more line segments:
  - Use search tree of segments crossed by sweep line to find its face(s)
  - If it is not a right endpoint, start new internal boundary in current face; otherwise, close off half-edges for which it is a right endpoint, and faces between them
  - If it is not a left endpoint, merge two faces or close off internal boundary; otherwise, start new half-edges for which it is a left endpoint, and faces between them
- When we sweep over a crossing, split its segments at that point and treat it as an endpoint of four line segments

Handling inputs that are not in general position



Event points have the same x-coordinate

- Break ties by y-coordinate
- Treat bottom endpoint of a vertical segment as left, and top endpoint as right

Endpoints on segments

- Report as a crossing?
- More cases for how to update DCEL

Multiple segments cross at one point

- Use only one event point, labeled by all the segments that cross there
- When processing event, reorder crossing segments in search tree