Implementing Geometric Algorithms

Wouter van Toll
June 20, 2017
Introduction
Focus of this course

- Create algorithms to solve geometric problems
  - Prove geometric properties and correctness
  - Analyze complexity: running time, storage, $O/\Omega/\Theta$

- No programming: why?

“The course was pretty theoretical. It might be a nice idea to have students design and implement a geometric algorithm to keep the course more interesting.”

-- a student, 2016
Focus of this lecture

- Robust implementations are **very** difficult to write
  - What makes it so difficult?
  - *(What *is* a robust implementation?)*
  - How can we deal with it?
  - Case study: Voronoi diagrams
Recap: Degenerate cases

- Cases in which a definition/algorithm is slightly unusual

- Degeneracies by **definition**
  - Voronoi diagram: ≥4 points on a circle

- Degeneracies for an **algorithm**
  - Sweep line: events at equal height

- Often require extra (pseudo)code
  - Unfortunate, but finite and doable
public List<Point> ConvexHull(List<Point> P)
{
    List<Point> P_sorted = sort(P);
    List<Point> L_upper = { P_sorted[0], P_sorted[1] };
    for (int i=2; i<n; i++)
    {
        L_upper.Append(P_sorted[i]);
        int k = L_upper.Count;
        while (k > 2 && !IsPointRightOfLine(L_upper[k-1], L_upper[k-3], L_upper[k-2]))
        {
            L_upper[k-2] = L_upper[k-1];
            L_upper.RemoveLast();
            k--;
        }
    }
    //...
}
Problems in 3, 2, 1...

- **Theory**: basic questions can be answered correctly
  - Is a point [left of / right of / on] a line?
  - Is a point [inside / outside / on] a circle?
  - Where is the intersection of two (non-parallel) lines?
  - Where is the intersection of two line *segments*, if any?
  - ...

- In a computer program...
  - ...answers can be **incorrect**
  - ...answers can be **inconsistent**
  - If the algorithm relies on correctness:
The reason

- **Inexact number representations**
  - Infinitely many numbers in $\mathbb{R}$
  - Finite #bytes in a float, double, ...
  - Imprecise numbers *and* results of operations

- Good enough for many applications
- **Very** difficult for geometric algorithms
Floating-point numbers
Real numbers in bytes

- “What Every Computer Scientist Should Know About Floating-Point Arithmetic” (D. Goldberg, 1991)
  https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

- Floating-Point Numbers 101: An Introduction
  - Base $\beta$, precision $p$ (number of digits), min/max exponent
  - A number has the form $\pm d_0. d_1 d_2... d_{p-1} \times \beta^e$
  - **Normalized** if $d_0$ is not 0
  - Examples for the real number 0.1
    - $\beta=10, p=3$: $0.1 = 1.00 \times 10^{-1}$
    - $\beta=2, p=24$: $0.1 \approx 1.10011001100110011001101 \times 2^{-4}$ (!!)
Real numbers in bytes

- Not all numbers can be represented
  - For each exponent: equally many fractions available
    → “more precision” closer to 0
  - Rounding error = |real number – FPN|
  - Relative error = rounding error / real number

- Operations lead to new FPNs
  - Errors can propagate

Dealing with inexactness

Applied to geometry
Inexact numbers and geometry

- A geometric algorithm has *numerical* and *combinatorial* aspects
  - **Numerical** = actual numbers
    - Input: coordinates
    - Algorithm: basic arithmetic
    - Output: coordinates
  - **Combinatorial** = order in which things happen
    - Input: assignment of coordinates to points
    - Algorithm: e.g. order of events in a sweep
    - Output: e.g. cyclic order of convex hull, topology of subdivision
Inexact numbers and geometry

- Numerical errors affect combinatorial properties
- Basic geometric questions: large error propagation

Possible problems
- Near-degenerate cases recognized wrongly
- Events processed in the wrong order
- Assumptions no longer hold later in the algorithm
- ...

https://en.wikipedia.org/wiki/Binary_expression_tree
Robustness

- **Wikipedia:** “(...) the ability of a computer system to cope with errors during execution and cope with erroneous input.”

- “Your program should never crash”

- Geometric algorithms: broader, vague boundary
  - Degenerate cases are handled
  - Output is correct “for some perturbation of the input”
  \(\rightarrow\) **Combinatorial output is the same as with real numbers**
  (and numerical error is negligible)

- How to write a robust implementation of a GA?
Idea 0: Ignore the problem

- Assume that floating-point precision is high enough
- Your code will work well most of the time

Problems?

- Rotation / scaling / ... can break it
- Impossible to cover all cases, unless you know / test all possible inputs

- Sometimes good enough, e.g. if you only compare input coordinates
  - kd-trees, orthogonal range queries

6/20/2017 Lecture: Implementing Geometric Algorithms
Idea 1: Build in error margins

- Treat **almost true** as **true**
  - dot product $< \varepsilon \Rightarrow$ points are collinear
  - distance $< \varepsilon \Rightarrow$ points are equal

- Effect: you accept slightly incorrect answers
  - OK if you know the consequences

- Example: convex hull
  - e.g. accept *slightly left* turns as a right turn
  - Not all points contained?
  - Not entirely convex?
Idea 1: Build in error margins

**Problems?**
- “Epsilon tweaking”
- What if you scale everything by 1000 or 0.001?
- Inconsistency: $A = B$ and $B = C$, but $A \neq C$
- Errors can propagate

- Remember this throughout the whole program
- Other programmers may not know / like your choices

**Some tricks can help**
- e.g. finding nearest answer vs. checking for equality
Idea 2: Fixed-point arithmetic

- Input coordinates are integers in a known range
  - (or floats in a known range with maximum #decimals)

- Problems?
  - Same property may not hold for the output
    (e.g. intersection of line segments)
  - Solve with snapping?
  - Rounding can change properties:
    angles, collinearity,
    existence of intersections, ...
Idea 3: Use robust predicates

- "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates" (J.R. Shewchuk, 1997).
  
  http://www.cs.cmu.edu/~quake/robust.html
  
  - Check if a determinant is 0, <0, or >0
  - Steps of increasingly smaller error (and increasing running time)
  - Guaranteed correctness

- C code in public domain

- Used in Triangle library
  
  - Delaunay triangulations
  
  - Related structures
Idea 3: Use robust predicates

- **Problems?**
  - Not everything can be expressed in *these* predicates
  - Not everything can be expressed in *predicates alone* (e.g. line segment intersection)
Idea 4: Use exact number types

- Avoid floating-point arithmetic completely
- e.g. store a number as an expression of integers
  - Fractions
  - Results of + - * / ...
- Takes away all FPN concerns: guaranteed correctness

Problems?
- (Takes some getting used to)
- Representation can become very complex → memory usage
- Expressions need to be evaluated → high running times
Idea 4: Use exact number types

- **BigNum**
  - Integers that can become as large as you need

- **LEDA**
  - Many types of numbers and arithmetic

- **CGAL**
  - The Computational Geometry Algorithms Library
  - Specifically for geometric algorithms
Idea 4: Use exact number types

https://www.cgal.org/
Ideas 5 to n-1
Examples: What would you do?

- Construct a kd-tree of a point set 0
- Check if a point is inside a triangle 1/3
- Check if two line segments intersect 1/2/3
- Compute the intersection of two line segments 1/2/3/4
- Compute a Voronoi diagram of points 2/3
- Compute the convex hull of a point set 0/1/3
- Compute a Euclidean Minimum Spanning Tree 3
- Compute a Voronoi diagram of line segments 4?
Many ways to tackle an implementation
- Consequences for precision, robustness, efficiency
- Best choice depends on your goals

“Ignore the problem” is only recommended if any (other) part of your program does not rely on correctness
Case study

Voronoi diagrams
Construction algorithms

- Divide-and-conquer (Shamos and Hoey, 1975)
- Plane sweep (Fortune, 1987, also Chapter 7)
- Incremental (Green and Sibson, 1978)
  - Randomized (Guibas et al., 1992, also Chapter 9)
  - Considered easiest to implement; lots of work on robustness
Point sites $\rightarrow$ Line segment sites

- Duality with Delaunay only for **point sites**
  - Compute DT, then convert
    (usually easier)

- More difficult for **segment sites**
  - All three VD construction algorithms can be extended
  - Discovered later, or considered “trivial”
Robust implementation

- “Topology-Oriented Implementation—An Approach to Robust Geometric Algorithms” (Sugihara et al., 2000)
  - Incremental construction
  - Floating-point numbers
  - Adaptive epsilons
  - Lots of **domain knowledge**: we know what the VD is supposed to look like
Implementation: Boost

- C++ library with many components
- Voronoi: Plane sweep algorithm (vertical sweep line)
  - Input coordinates are integers
  - Output coordinates are doubles
  - Adaptive-precision predicates
  - Topologically correct
  - Assumes non-intersecting line segments
Implementation: Vroni

- M. Held, since ~2000
  - Written in C with floating-point numbers
  - Topology-oriented incremental approach
  - Restarts whenever segments intersect
  - Also supports circular arcs as sites

https://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html
M. Karavelas, since ~2004

http://doc.cgal.org/latest/Voronoi_diagram_2/index.html

- Incremental algorithm
- Integrates well with Boost data structures + rest of CGAL

- Guaranteed to work well with CGAL’s exact numbers
  (but otherwise...)

Implementation: CGAL
My experience: UUCS framework

- Path planning and crowd simulation
  - Retraction and the medial axis

- Wrappers for Vroni and Boost
  - Both: rounded input coordinates, pre-processing
  - (Tried CGAL ~6 years ago, worth retrying)
  - (GPU implementation: unreliable)

If a usable implementation exists, use it
UUCS and the multi-layered MA

- MSc / PhD research: **multi-layered medial axis**
  - Multi-layered environment (MLE): 2D *layers* and *connections*
  - MA defined with projected distances

- Implementation in framework
  - Cannot fully rely on existing libraries
  - Performance is important
Computing the MA of an MLE:

- Compute MA of each layer, then *open* the connections 1 by 1
- Opening ≈ *deleting* a line segment site, but slightly weirder

- Opening takes $O(m \log m)$ time; maybe $O(m)$?
- Overall, for $n$ boundary vertices and $k$ connections:
  $O(n \log n \log k)$ time; maybe $O(n \log n)$?
UUCS and the multi-layered MA

- The “each layer” part: use **existing libraries**
  - Vroni: faster, captures more details
  - Boost: parallel computing, thread-safety

- The “open the connections” part: **custom code**
  - Floating-point numbers and epsilons
  - Difficult to get 100% right
  - Assumptions about input can help a lot
UUCS and the multi-layered MA
Closing comments
Summary

- Geometric algorithms are difficult to implement
  - Numerical errors affect combinatorial properties
- Floating-point numbers
  - Sometimes good enough (but not in the general case)
  - Tricks: epsilons, rounding, robust predicates
  - Domain-specific solutions (e.g. Voronoi)
- Exact numbers
  - Theoretically solves all problems
  - Resource-intensive
- Robustness/implementation is a whole research area
Take-home advice

- If you want to implement a geometric algorithm:
  1. **Study the problem well**
  2. Decide how precise / robust / fast your solution should be
  3. Check if (auxiliary) software is available
  4. Choose a style of dealing with imprecision
  5. Make clear assumptions about input
     + ensure that they are met (or return an error)


“Precision and Robustness in Geometric Computations” (S. Schirra) Chapter in LNCS Tutorial on Algorithmic Foundations of Geographic Information Systems, 1997


CGAL, Boost (Voronoi), Vroni