CMSC 754: Lecture 1
Introduction to Computational Geometry

What is Computational Geometry? "Computational geometry" is a term claimed by a number of different groups. The term was coined perhaps first by Marvin Minsky in his book "Perceptrons", which was about pattern recognition, and it has also been used often to describe algorithms for manipulating curves and surfaces in solid modeling. It's most widely recognized use, however, is to describe the subfield of algorithm theory that involves the design and analysis of efficient algorithms for problems involving geometric input and output.

The field of computational geometry grew rapidly in the late 70's and through the 80's and 90's, and it is still a very active field of research. Historically, computational geometry developed as a generalization of the study of algorithms for sorting and searching in 1-dimensional space to problems involving multi-dimensional inputs. Because of its history, the field of computational geometry has focused mostly on problems in 2-dimensional space and to a lesser extent in 3-dimensional space. When problems are considered in multi-dimensional spaces, it is often assumed that the dimension of the space is a small constant (say, 10 or lower). Nonetheless, recent work in this area has considered a limited set of problems in very high dimensional spaces, particularly with respect to approximation algorithms. In this course, our focus will be largely on problems in 2-dimensional space, with occasional forays into spaces of higher dimensions.

Because the field was developed by researchers whose training was in discrete algorithms (as opposed to more continuous areas such as a numerical analysis or differential geometry) the field has also focused principally on the *discrete* aspects of geometric problem solving. The mixture of discrete and geometric elements gives rise to an abundance of interesting questions. (For example, given a collection of n points in the plane, how many pairs of points can there be that are exactly one unit distance apart from each other?) Another distinctive feature is that computational geometry primarily deals with straight or flat objects (lines, line segments, polygons, planes, and polyhedra) or simple curved objects such as circles. This is in contrast, say, to fields such as solid modeling, which focus on issues involving curves and surfaces and their representations.

Computational geometry finds applications in numerous areas of science and engineering. These include computer graphics, computer vision and image processing, robotics, computer-aided design and manufacturing, computational fluid-dynamics, and geographic information systems, to name a few. One of the goals of computational geometry is to provide the *basic geometric tools* needed from which application areas can then build algorithms and *theoretical analytic tools* needed to analyze the performance of these algorithms. There has been significant progress made towards this goal, but it is still far from being fully realized.

A Typical Problem in Computational Geometry: Here is an example of a typical problem, called the *shortest path problem*. Given a set polygonal obstacles in the plane, find the shortest obstacle-avoiding path from some given start point to a given goal point (see Fig. 1). Although it is possible to reduce this to a shortest path problem on a graph (called the *visibility graph*, which we will discuss later this semester), and then apply a nongeometric algorithm such as Dijkstra's algorithm, it seems that by solving the problem in its geometric domain it should

be possible to devise more efficient solutions. This is one of the main reasons for the growth of interest in geometric algorithms.

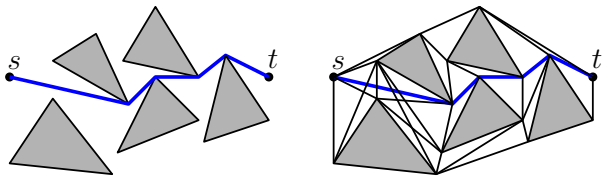


Fig. 1: Shortest path problem.

The measure of the quality of an algorithm in computational geometry has traditionally been its *asymptotic worst-case running time*. Thus, an algorithm running in $O(n)$ time is better than one running in $O(n \log n)$ time which is better than one running in $O(n^2)$ time. (This particular problem can be solved in $O(n^2 \log n)$ time by a fairly simple algorithm, in $O(n \log n)$ by a relatively complex algorithm, and it can be approximated quite well by an algorithm whose running time is $O(n \log n)$.) In some cases *average case* running time is considered instead. However, for many types of geometric inputs (this one for example) it is difficult to define input distributions that are both easy to analyze and representative of typical inputs.

Features of Computational Geometry: As is the case with any field of science, computational geometry owes its structure to the interests and expertise of its founders in combination with the gradual evolution over time from the members of this research community. Here are a number of features of the field.

Rigor: Prior to computational geometry, there were many *ad hoc* solutions to geometric computational problems, some efficient, some inefficient, and some simply incorrect. From its beginnings, computational geometry has focused on mathematical rigor in its approach to problems. This field has made great strides in establishing correct, provably efficient algorithmic solutions to many fundamental geometric problems.

Correctness/Robustness: Many classical geometric software systems have been troubled by bugs arising from the confluence of the continuous nature of geometry and the discrete nature of computation. (For example, given two line segments in the plane, do they intersect? This problem is remarkably tricky to solve exactly due to floating point rounding errors.) Software that is based on discrete decisions, each of which is subject to some error, is inherently unreliable. Much effort has been devoted in computational geometry to the robust and correct computing of geometric primitives, thus forming a solid foundation for geometric computations.

Emphasis on Discrete Geometry: Many applications of geometric computing involve quantities that are naturally continuous, such as the 3-dimensional scalar field representing the air temperature in a room. Since computers and computation is inherently "finite" in nature, solving such problems requires discretizing them. Fields like *computational fluid dynamics* involve computing approximate solutions to such continuous problems. In contrast, the field of computational geometry has focused on problems that are naturally discrete in nature (e.g., involving a finite set of points) as opposed to discretizations of continuous problems.

Discrete Combinatorial Geometry: The asymptotic analysis of algorithms in discrete geometry has inspired a great deal of new work in the topic of *combinatorial geometry*. For example, consider a polygon bounded by n sides in the plane. Such a polygon might be thought of as the top-down view of the walls in an art gallery. As a function of n, how many "guarding points" suffice so that every point within the polygon can be seen by at least one of these guards. Such combinatorial questions can have profound implications on the complexity of algorithms.

Flatness: Fields such as solid modeling in industrial design deal naturally with curved objects, like the body of an automobile. In contrast, computational geometry typically deals with flat (or "affine") objects, such as points, lines, planes and hyperplanes, or subsets of these (such as line segments, polygons, and polytopes). Exceptions are usually fairly simple, such as Euclidean balls and algebraic surfaces of constant degree, such as ellipsoids.

Low Dimensionality: It is typically assumed in computational geometry that inputs are drawn from spaces of constant dimension. Many algorithms work only in 2-dimensional or 3-dimensional space. Many algorithms that work in spaces of higher dimension suffer from the so-called "curse of dimensionality," where there are factors that grow exponentially with dimension. While there are notable examples, the majority of work in the field has this limitation.

Overview of the Semester: Here are some of the topics that we will discuss this semester.

Convex Hulls: Convexity is a very important geometric property. A geometric set is *convex* if for every two points in the set, the line segment joining them is also in the set. One of the first problems identified in the field of computational geometry is that of computing the smallest convex shape, called the *convex hull*, that encloses a set of points (see Fig. 2).

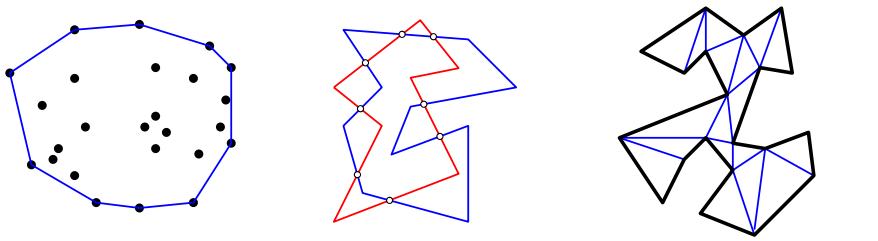


Fig. 2: Convex hulls, intersections, and polygon triangulation.

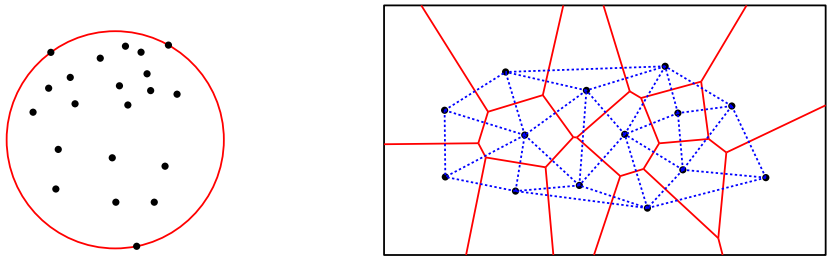
Intersections: One of the most basic geometric problems is that of determining when two sets of objects intersect one another. Determining whether complex objects intersect often reduces to determining which individual pairs of primitive entities (e.g., line segments) intersect (see Fig. 2). We will discuss efficient algorithms for computing the intersections of a set of line segments.

Triangulation and Partitioning: Triangulation is a catchword for the more general problem of subdividing a complex domain into a disjoint collection of "simple" objects (see

Fig. 2). The simplest region into which one can decompose a planar object is a triangle (a *tetrahedron* in 3-d and *simplex* in general). We will discuss how to subdivide a polygon into triangles and later in the semester discuss more general subdivisions into trapezoids.

Optimization and Linear Programming: Many optimization problems in computational geometry can be stated in the form of *linear programming*, namely, finding the extreme points (e.g. highest or lowest) that satisfies a collection of linear inequalities. Linear programming is an important problem in the combinatorial optimization, and people often need to solve such problems in hundred to perhaps thousand dimensional spaces. However there are many interesting problems that can be posed as low dimensional linear programming problems or variants thereof. One example is computing the smallest circular disk that encloses a set of points (see Fig. 3). In low-dimensional spaces, very simple efficient solutions exist.

Voronoi Diagrams and Delaunay Triangulations: Given a set S of points in space, one of the most important problems is the nearest neighbor problem. Given a point that is not in S which point of S is closest to it? One of the techniques used for solving this problem is to subdivide space into regions, according to which point is closest. This gives rise to a geometric partition of space called a *Voronoi diagram* (see Fig. 3). This geometric structure arises in many applications of geometry. The dual structure, called a *Delaunay triangulation* also has many interesting properties.



Smallest enclosing disk Voronoi diagram and Delaunay triangulation

Fig. 3: Voronoi diagram and Delaunay triangulation.

Line Arrangements and Duality: Perhaps one of the most important mathematical structures in computational geometry is that of an arrangement of lines (or generally the arrangement of curves and surfaces). Given n lines in the plane, an arrangement is just the graph formed by considering the intersection points as vertices and line segments joining them as edges (see Fig. 4). We will show that such a structure can be constructed in $O(n^2)$ time.

The reason that this structure is so important is that many problems involving points can be transformed into problems involving lines by a method of *point-line duality*. In the plane, this is a transformation that maps lines to points and points to lines (or generally, $(d-1)$-dimensional hyperplanes in dimension d to points, and vice versa). For example, suppose that you want to determine whether any three points of a planar point set are collinear. This could be determined in $O(n^3)$ time by brute-force checking

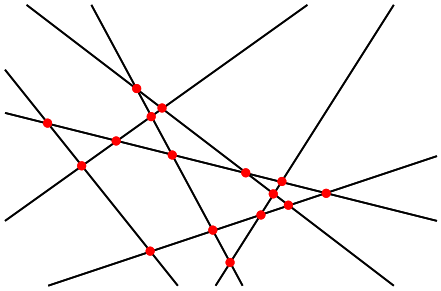


Fig. 4: An arrangement of lines in the plane.

of each triple. However, if the points are dualized into lines, then (as we will see later this semester) this reduces to the question of whether there is a vertex of degree greater than four in the arrangement.

Search: Geometric search problems are of the following general form. Given a data set (e.g. points, lines, polygons) which will not change, preprocess this data set into a data structure so that some type of query can be answered as efficiently as possible. For example, consider the following problem, called *point location*. Given a subdivision of space (e.g., a Delaunay triangulation), determine the face of the subdivision that contains a given query point. Another geometric search problem is the *nearest neighbor problem*: given a set of points, determine the point of the set that is closest to a given query point. Another example is *range searching*: given a set of points and a shape, called a range, either count of report the subset of points lie within the given region. The region may be a rectangle, disk, or polygonal shape, like a triangle.

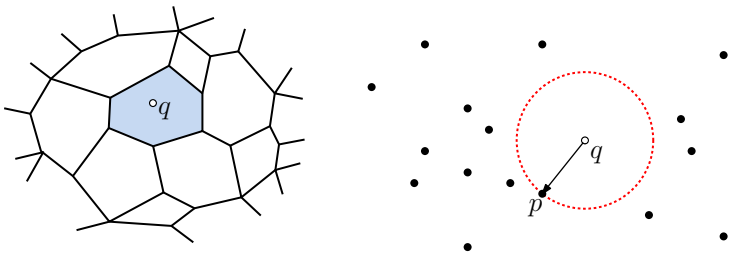


Fig. 5: Geometric search problems. The point-location query determines the triangle containing q. The nearest-neighbor query determines the point p that is closest to q.

Approximation: In many real-world applications geometric inputs are subject to measurement error. In such cases it may not be necessary to compute results exactly, since the input data itself is not exact. Often the ability to produce an approximately correct solution leads to much simpler and faster algorithmic solutions. Consider for example the problem of computing the diameter (that is, the maximum pairwise distance) among a set of n points in space. In the plane efficient solutions are known for this problem. In higher dimensions it is quite hard to solve this problem exactly in much less than the brute-force time of $O(n^2)$. It is easy to construct input instances in which many pairs of

points are very close to the diametrical distance. Suppose however that you are willing to settle for an approximation, say a pair of points at distance at least $(1-\varepsilon)\Delta$, where Δ is the diameter and $\varepsilon > 0$ is an approximation parameter set by the user. There exist algorithms whose running time is nearly linear in n, assuming that ε is a fixed constant. As ε approaches zero, the running time increases.

...and more: The above examples are just a small number of numerous types of problems that are considered in computational geometry. Throughout the semester we will be exploring these and many others.

Computational Model: (Optional) We should say a few words about the model of computation that we will be using throughout in this course. It is called the *real RAM*. The "real" refers to real numbers (not the realism of the model!) and RAM is an acronym for "random-access machine", which distinguishes it from other computational models, like Turing machines, which assume memory is stored on tape. The real RAM is a mathematical model of computation in which it is possible to perform arithmetic (addition, subtraction, multiplication, and division, and comparisons) exactly with real numbers in constant time.

Why should we care? As an example, consider a geometric version of a famous optimization problem, known as the *Traveling Salesperson Problem* (TSP). We are given a set of n points in the plane, and we wish to compute the circuit of minimum total Euclidean distance that visits all these points. Ignoring the difficulty of this combinatorial problem (which is NP-Hard), consider the (apparently) much simpler question of whether one tour is shorter than another. (Consider, for example, the blue and red tours shown in Fig. 6.) Clearly, if we are going to solve the general problem, we must be able to at least compare the relative lengths of two tours.

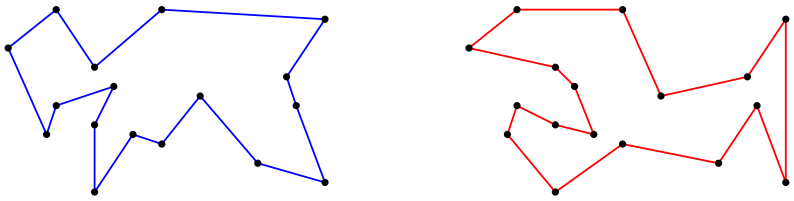


Fig. 6: Comparing the length of two TSP tours on the same point set.

While this problem can be solved approximately using floating point computations, it is quite difficult to solve the problem exactly. To see why, observe that the Euclidean distance between two points involves computing square roots. Two square roots can be compared by squaring both sides to eliminate the square root, thus reducing the computation to one involving simple arithmetic operations. Unfortunately, when you attempt this with a sequence of n numbers, each of which is a square root, many squarings upon squarings must be performed. The sizes of the numbers (in terms of the number of bits required) grows exponentially. In practice, these computations would be impossible. But in the Real RAM model, each such computation can be computed in constant time each.

In spite of the unusual power of this model, it is possible to simulate this model of computation for many common geometric computations. This is done through the use

of so-called *floating-point filters*, which dynamically determine the degree of accuracy required in order to resolve comparisons exactly. The CGAL libarary supports exact geometric computations through this mechanism.