COMPUTATIONAL GEOMETRY
Some Suggested Optional Projects

By **April 1, 2014**, you must notify me of which project you are doing and submit a brief (1-page) description of the project plan. Some projects can be done in pairs (2 people), but if more than one person is involved, the project plan must include a detailed breakdown of who will do what part of the project.

Project reports are due **May 6, 2014**; the same day, there will be in-class presentations and demos. In addition to submitting a report, you must submit the bundle of software. The report can be brief but should state clearly the problem, the approach, and give basic information about compiling/running the software, and report relevant experimental experience. Submission should be by email. If you do the project in Java, please also create an applet and webpage, so that particularly educational projects can be easily featured in future classes.

Note that some projects may lead to publishable research or to more extensive projects (e.g., masters projects (CSE 523/524)), depending on how successful the investigation is.

All software should be written in standard C, C++, or Java. (Check with me before using another language.) It should be easily compilable under Unix (e.g., a PC running Linux) or Windows. Please document your code! Also, provide a README, and bundle all files in one directory. You may find that C or Java is the most convenient: you are allowed to use C code fragments provided in the O'Rourke textbook (and available on the web, both in C and in Java – follow pointers to O'Rourke's home page, from our course home page). You may be asked to give a short demonstration of your code to me on your own computer. Try to handle degenerate cases and to make your implementation as robust as you can.

You will find it most convenient to have a graphical display to output and "animate" your algorithm. You may also want to be able to mouse in data.

(0). *Hitting Lines.* Given a set of n lines in the plane, the Hitting Lines problem asks us to find a smallest set of points so that each line is "hit" by one of the points. It is closely related to some guarding problems in polygons. Of course, the hitting points chosen may as well always be at vertices of the arrangement of the n input lines. Devise and compare heuristic methods to find small hitting sets of points for a set of n lines. ("Greedy" methods are natural, but you may think of other options too.) Key to doing experiments on this problem is figuring out a way to generate input sets of n lines that are nontrivial: If you just generate n lines "at random" (e.g., picking slope and intersept uniformly at random), the probability that 3 or more lines pass through a common point is 0. (And, for a set of lines in general position (what we call a "simple arrangement" of lines), it is trivial to find an optimal hitting set, since any vertex where lines cross is a point of crossing of just *two* lines, so we know we need exactly $\lceil n/2 \rceil$ points to hit all n lines.) Figure out a way to generate "interesting" instances, and try out your heuristic(s) on them. (Can you think of ways of computing a lower bound on the optimal number of points in a hitting set (like our notion of "witness points" in guarding problems)?) Play with this problem as much as you can.

(1). *Computing a "Central" Trajectory.* The input is a set of n (polygonal) trajectories, with each specified as a sequence of $k+1$ "points", (x, y, t), in space-time. For instance, the trajectory $\tau = ((x_0, y_0, t_0), (x_1, y_1, t_1), \ldots, (x_k, y_k, t_k))$ corresponds to a trajectory for a point that starts at (x_0, y_0) at time t_0, then travels at constant speed along a straight line segment to (x_1, y_1), where it arrives at time t_1 (the time difference, $t_1 - t_0$, and the distance from (x_0, y_0) to (x_1, y_1) determine the speed of motion). Assume that the time stamps t_0 and t_k are the same for all trajectories and that the starting point, (x_0, y_0), and destination point, (x_k, y_k), are also the same for all trajectories. Other intermediatre points/times along trajectories can be arbitrary (varying from trajectory to trajectory), except that time is monotone ($t_0 < t_1 < \cdots < t_k$) for each trajectory. Your goal is to define, compute, and compare notions of a "central trajectory" that represents the input set of

n trajectories. The idea is that the trajectory data may be noisy, and while the trajectories are all "similar" in some sense (starting and ending at the same times/places), the motion varies: Can we compute a single trajectory, τ^*, that "optimally" represents the set of n trajectories, serving as a "central trajectory". Describe your proposed definitions and try them out on trajectory data that you randomly generate in a meaningful way. How efficient are your algorithms? (For some definitions, you may implement an algorithm that is not theoretically best (maybe your implementation is even brute-force), but you should state what the best method in theory is.)

(2). Given a simple polygon P, compute an "optimal" triangulation using dynamic programming. Here, "optimal" may mean (a) minimum weight triangulation (minimize the sum of the edge lengths of diagonals), (b) minimize the longest edge, (c) maximize the shortest edge, etc. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

Experimentally compare the weight of the resulting triangulation with the weight of the triangulation obtained using ear clipping (algorithm Triangulate, from O'Rourke).

(3). Use a bounding volume hierarchy (e.g., of axis-aligned bounding boxes) to perform intersection queries among a set of n line segments in the plane. Play with different methods to build the hierarchy (i.e., to partition the segments into two sets) and see what impact they have on the running time.

Try randomly generated line segments. How fast can you solve the REPORT problem to find all pairs of intersecting segments?

Another option is to do a careful experimental study of grid-based methods, quadtree-based methods, etc. Investigate the best grid sizes, considering the space-time tradeoffs.

(4). Let S be a set of n points ("sensors") in the plane. These sensors lie within a polygonal domain (polygon with holes ("obstacles")), P. We say that two points $p, q \in S$ *see* each other if the segment $pq \subset P$ (i.e., the points are line-of-sight visible to each other). The *visibility graph*, $VG(S, P)$, joins pairs of sensors that are line-of-sight visible; the nodes are the sensors S and the edges E join pairs of visible sensors.

Our goal is to find a longest (or nearly longest) edge of $VG(S, P)$, without building the entire visibility graph. Come up with some clever ideas to try, and do an experiment to compare your ideas with the naive method of just trying every pair of points from S.

(5). The k-box cover problem takes as input a set S of n points in the plane and an integer k. The output should be a set, $\{B_1, \ldots, B_k\}$, of k axis-aligned boxes (rectangles with sides that are vertical/horizontal) that cover S. The goal is that the sum of the "sizes" ("size" may be area or perimeter or diameter) of the boxes be minimized.

Design at least two distinct heuristics for k-box cover in 2D, and run comparisons between them, and compare to brute-force methods for small values of k ($k = 2, 3, 4$).

(6). Implement a method to compute a minimum-area terrain triangulation of a set of points in 3D: Each point $p_i = (x_i, y_i, z_i)$ sits above the xy-plane by distance $z_i > 0$ and our goal is to triangulate the projections of the points onto the xy-plane so that the corresponding set of triangles in 3D has the smallest possible surface area. This problem arises in surface reconstruction from point cloud data. I have some ideas for how to do this with heuristic methods – see me, and I can give details. You are encouraged to come up with your own ideas too. I think local search methods may be good.

I am also very interested in theoretical results: Can you *approximate* optimal in polynomial time? Can you show that the exact solution is NP-hard?

(7). Implement a method to place a "small" number of guards in a simple polygon, P. The goal is to design a reasonable heuristic to do this, making sure that the entire polygon is seen. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers,

separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

The implementer of this algorithm may work in partnership with the implementer of the witness point algorithm below.

(8). Implement a method to place a "large" number of independent witness points in a simple polygon. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

(9). Implement the Hertel-Mehlhorn algorithm to compute a decomposition of a simple polygon P into a small number of convex polygonal pieces. You can use O'Rourke Triangulation code to produce the initial triangulation, but you will have to put it into a DCEL (or similar) in order to do the Hertel-Mehlhorn algorithm efficiently. (It is OK if the conversion to DCEL is done naively, but the H-M algorithm should be done efficiently (linear time), after you have a DCEL triangulation.

(10). Implement the Bentley-Ottmann sweep to DETECT if a polygonal chain $C = (p_1, p_2, \ldots, p_n)$ is *simple* (it is not simple if it properly crosses itself, at a point interior to two edges, or if it crosses itself at a vertex, in the degenerate case). Report a witness to the crossing if the chain is not simple.

The input is assumed to be either graphical (e.g., allowing the user to mouse-click enter points) or read from a simple text file, with each line consisting of two floating point numbers (the x- and y-coordinates of a vertex) separated by spaces. You should animate the algorithm to show each step during the sweep.

Conduct an experiment to determine the running time in practice for large values of n, for randomly generated inputs.

(11). Implement an algorithm that runs in $O(n)$ time to triangulate a y-monotone simple polygon having n vertices. The input is assumed to be a text file with a clockwise listing of the (x, y) coordinates (floating point) of the vertices; each line of the text file consists of two floating point numbers, separated by a space. (Again, it will be useful to have the option of mouse-clicking input too.) You may assume that the polygon is simple and that it is y-monotone (you need not check to confirm this in your code), but you do not know which vertex comes first (e.g., it need not be the topmost or the bottommost). Again, there may be degeneracies (duplicated points, collinear points, etc).

The output should be a list of pairs of points that define the diagonals of the triangulation. (You need not build a DCEL or other data structure for the triangulation.)

Conduct experiments to determine the in-practice efficiency of the algorithm for large inputs.

(12). Implement the point location data structure (DAG) of Chapter 6. The input consists of a list of segments $S = (s_1, s_2, \ldots, s_n)$, which are to be inserted in the order given. Show graphically the trapezoidal diagram as each one is inserted. (Basic animation of the algorithm using simple graphics will greatly ease the debugging.)

(13). Implement the algorithm VisibleVertices(p, S) on page 312 of the text. You may assume that each simple polygon in the set S is given by a list of its vertices in order (ccw) around its boundary, that they are known to be pairwise disjoint (and not touching), and that p is necessarily disjoint from S. Draw the connections from p to each visible vertex.

(14). Implement either the Lawson edge swap algorithm (LegalizeTriangulation) or the randomized incremental method for Delaunay triangulations of points in the plane.

(15). Implement Timothy Chan's convex hull algorithm in 2D.

(16). A set S of n line segments in the plane is said to have a Type 1 degeneracy if all segments lie on a common line. They are said to have a Type 2 degeneracy if some subset of 3 or more endpoints are collinear. It is easy to see that a Type 1 degeneracy can be detected in $O(n)$ time. Using duality and building a line arrangement, you can see that a Type 2 degeneracy can be detected in $O(n^2)$ time. (Do you see how?)

Impelement a degeneracy-tester that takes as input a set of line segments in the plane and reports if there is a Type 1 or Type 2 degeneracy.

(17). Implement the incremental linear programming algorithm of Section 4.3 for the case of two dimensions ($d = 2$). The input is assumed to be a text file of floating numbers, with the first line being c_1 c_2 (separated by a space), then the next n lines being $a_{i,1}$ $a_{i,2}$ b_i (with spaces separating the numbers). The output should print the optimal objective function value (with the option to return "NOT FEASIBLE" or "UNBOUNDED"), along with the point (x, y) that achieves the optimal value. You should animate the algorithm, showing each constraint line as it is inserted, along with the currently optimal vertex, etc.

Apply the algorithm to the red-blue line separation problem: The input is a set R of "red" points and a set B of "blue" points, and the output should be a line ℓ that separates R and B, or the statement that no such line exists.

There is no need to randomize the order of the input; you may assume that the constraints arrive in random order already.

(18). (See Problem 2.10, Chapter 2, BKOS) Let S be a planar subdivision with n vertices, stored as a DCEL. Let P be a set of m points. Implement a plane-sweep algorithm to locate each point of P in a face of S. In order to be able to verify correctness (and to debug), you should use a graphical interface and animate the algorithm. Be creative! (It may be useful to have a mouse-based editor to input S and build a DCEL representation of it.)

(19). Implement kd-trees for points in the plane. The input should be a set of points, $S = \{p_1, p_2, \ldots, p_n\}$, given either by mouse clicks or by reading a text file, with each line consisting of floating point numbers x_i and then y_i, separated by a space. (The first line of the file should be the number, n, of points.) The program should then allow one to specify a rectangle, $[x, x'] \times [y, y']$, by text input from the user and/or mouse input from the user. The output, then, should be a listing of the points that are within the query box. (If you use graphical output, change the color of the points that lie within the query box.)

Perform the following experiment with your code. For various values of n (e.g., $n = 100, 200, 300, \ldots, 1000$, or other interesting range of values), do the following. Generate a set S of n random points in the unit square (each x_i and y_i is uniformly distributed between 0 and 1, using the random number generator of the language in which you wrote the code). Build the kd-tree for each S, and compute the average time, over, say, 20 samples, it takes to answer a query Q given by a box that is, say, 0.1-by-0.1. (A simple way to do this is to generate the query box as $[x, x + 0.1] \times [y, y + 0.1]$, where x and y are chosen uniformly at random from the interval (0,0.9).) Compare the speed of the kd-tree method to the "brute force" method of simply testing all n points, one by one, to see which ones are in the query box. It would be nice to see a plot of the two times, as a function of n. For what value of n is it worth using the more sophisticated data structure of the kd-tree?

(20). This project is just like project (19), except that instead of using a kd-tree to do orthogonal range queries, use a 2d range tree. (You may do it without the use of fractional cascading, if you want.)

(21). Let $S = \{p_1, \ldots, p_n\}$ be a set of n points within the unit square, $U = \{(x, y) : 0 \leq x \leq 1, 0 \leq y \leq 1\}$, in the plane. Assume that $(0, 0) \in S$ (i.e., the origin is one point of S). A rectangle R_i is *valid* if (a) p_i is the bottom left corner of R_i, and (b) no other point $p_j \in S$ lies interior to R_i. Our goal is to find a set of valid rectangles that forms a "packing" (which means that no two rectangles overlap – they can touch on their boundaries, but have no overlap of their interiors) of maximum possible area.

Devise a heuristic (or exact) algorithm to find a good set of rectangles, using concepts from CG. Try out your idea on sets of random points S in U. Compute the areas you get. (Are they always over 1/2? It is conjectured that one can always achieve area at least 1/2; can you prove it? Can you prove that one can always achieve area at least ϵ, for any positive ϵ?)

(22). *Placing glasses on a tray.* You serve drinks in a restaurant and must carry a tray with one hand. As you load glasses onto the tray, the tray can become unstable – your hand must stay under the center of mass of the glasses. There are particular spots on the tray where the glasses must be placed, but you are free to place the glasses in any order you want. How should you do it to make sure the tray stays balanced? Let's model this problem as follows, thinking of the glasses as "points" in the plane.

Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in the plane (e.g., entered by mouse clicks or randomly generated). Let $c(X)$ denote the center of mass of the set $X \subseteq S$ of points. Our goal is to find a "good" ordering (permutation π) of the points S, $(p_{\pi_1}, p_{\pi_2}, \ldots, p_{\pi_n})$, so that the center of mass, $c_j = c(\{p_{\pi_1}, \ldots, p_{\pi_j}\})$, of the first j points ($j = 1, 2, \ldots, n$) in the order does not "move around" too much. The "score" for a given ordering π might be (a) the smallest radius r of a disk centered at $c_n = c(S)$ that contains all points c_j; (b) the area of the convex hull of the centers of mass c_j (for $j = 1, 2, \ldots, n$); or (c) the total length of the chain $(c_1, c_2, c_3, \ldots, c_n)$ (i.e., the total distance traveled by the center of mass as the points were added in the order π.