CS 294-5
Meshing and Triangulation in Graphics, Engineering, and Modeling
Delaunay Triangulation Project
(20% of final grade)

Implement two divide-and-conquer algorithms and two incremental insertion algorithms for constructing two-dimensional Delaunay triangulations. Your implementations must use Guibas and Stolfi's quad-edge data structure. You are responsible for reading and understanding the Guibas and Stolfi paper, except Section 3, which may be safely skipped. It's a long paper, so start early.

The purpose of this project is to burn an understanding of Delaunay triangulations into your brain. Your code may also become a component of your second project, so it's worthwhile to debug it well.

Divide-and-conquer algorithms: Implement the divide-and-conquer algorithm for which Guibas and Stolfi give pseudocode.

In addition, implement the variation of divide-and-conquer in which the partitioning steps alternate between horizontal and vertical cuts. You will need an $\mathcal{O}(n)$ selection algorithm, preferably quickselect, to perform the partitioning step quickly. (If you are not familiar with quickselect, read the "Medians and Order Statistics" chapter of Cormen, Leiserson, and Rivest's *Algorithms*.) Don't forget that the base cases (2 or 3 vertices) must always be sorted by x-coordinate if Guibas and Stolfi's pseudocode is to work unchanged.

If you are thoughtful, the merge step will require little modification of Guibas and Stolfi's pseudocode. Their merge code isn't direction-dependent, as long as you grab the right extremal edges of the triangulation boundary before merging.

Incremental insertion algorithms: Implement the incremental insertion algorithm for which Guibas and Stolfi give pseudocode, using their suboptimal "walking" method for point location.

In addition, implement the point location method (based on the simplified conflict graph) described in the lecture notes. Since you are using an edge-based data structure, rather than a triangle-based data structure, you will need to modify the point location data structure slightly. Each uninserted vertex should maintain an edge reference to an oriented edge of the triangle that contains the uninserted vertex. Each quad-edge has four `Data` fields. Two of these are used to reference the vertices of the quad-edge, and the other two may be used to reference the linked lists of uninserted vertices associated with the triangles on each side of the edge.

For both point location methods, there should be a command-line switch or input prompt that determines whether or not the order of the input vertices is randomized (using a permutation uniformly selected from all possible permutations).

Extras: A perfect realization of this project as described here will earn a bottom A-. Higher grades may be assigned for any particularly interesting feature added beyond those described here, especially if it makes it possible to compare the running time of an additional algorithm, variation on an algorithm, or point location method. Examples include (in increasing order of ambition): producing a Voronoi diagram as well as a Delaunay triangulation; a version of incremental insertion that doesn't use a bounding box; the gift-wrapping Delaunay triangulation algorithm; the multi-level dynamic point location method of Devillers (ask me for the paper); Chew's divide-and-conquer algorithm for constructing CDTs; and Fortune's sweepline Delaunay triangulation algorithm (ask me for the paper). In fact, if you implement Chew's algorithm, you are exempt from having to implement the divide-and-conquer algorithm with alternating cuts, and may still receive an A+. Whatever algorithm you implement should be based on quad-edges.

Language: If you write in any language other than C, you are required to give me very complete instructions on how to compile and run your code. You may also be required to obtain for me an account with which I can run your code. If your submission does not run under Unix, I may need even more help.

Interface: Your program should use the same file formats as the program Triangle (`http://www.cs.cmu.edu/~quake/triangle.html`). Specifically, it should read a file with the suffix `.node`, and write a file with the suffix `.ele`. (If you implement any CDT algorithm, your triangulator should read a file with the suffix `.poly`. If you write a Voronoi diagram, I recommend you use Triangle's file formats for that, too.) All the file formats are documented on the Triangle Web pages.

An advantage of using these file formats is that you (and I) can use the Show Me visualization program to view and print your input and output. You can also use Triangle to check if you're producing correct triangulations. (Note that Triangle uses a triangle-based data structure instead of quad-edges, and its code will probably not be as helpful to you as Guibas and Stolfi's pseudocode in producing your quad-edge-based implementation.)

You should provide command-line switches, an input prompt, or some other easy way to select: the triangulation algorithm used; whether the order of the input vertices is randomized prior to running an incremental algorithm; which implementation of the InCircle and CCW (orientation) predicates each algorithm uses. Be sure to document how these selections are made.

Geometric predicates: Insofar as possible, your algorithms should make all their arithmetic decisions based on the results of InCircle and CCW predicates. (If you are crazy enough to implement Fortune's algorithm, you will definitely need other predicates.) You should have three versions of the InCircle and CCW predicates: the less and more robust inexact versions (which will be discussed in class), and exact predicates (that do not suffer from roundoff error), like those available at one of the following.

- http://www.cs.cmu.edu/~quake/robust.html (floating-point inputs)
- http://www-sop.inria.fr/prisme/personnel/devillers/anglais/determinant (integer inputs)

Borrowed code: You are welcome to use publicly available libraries or implementations of the following, so long as none of them was produced by any of your classmates: sorting, selection, binary heaps, balanced search trees, other fundamental non-geometric data structures, command-line switch processing, and geometric primitives like the InCircle and CCW predicates. You must write the quad-edge implementation and geometric algorithms all by yourself.

Your submission: My preferred submission method is that you email me the secret URL of a file or directory containing all your code, so I can download it at my leisure. Alternatively, you could email me a uuencoded archive of your files. If your code doesn't run under Unix, special arrangements will definitely need to be made.

You should also submit a report (on paper) that includes the following:

- Instructions for compiling and running your code. If you use C and Unix, relatively rudimentary instructions will probably do. Be sure to document how to specify the input file, and how to choose between different algorithms.
- Timings for each of your algorithms and point location methods on random points sets (which I will provide) of 10,000, 100,000, and 1,000,000 points. (Don't worry if it proves to be impossible to run the "walking" method of point location on sets as large as one million points; just note that you

couldn't wait for it to terminate.) Time only the randomized versions of the incremental insertion algorithm. Use the *exact* InCircle and CCW predicates for all your timings. Try to exclude all file I/O from your timings if possible. (If using a timer within your program isn't possible, the Unix `time` command will do, although file I/O time will be included.) Why do you think the fastest algorithm is fastest?

- Create an orderly point set, like a square grid whose vertices are given in a structured order. Does failing to randomize the order of the vertices significantly change the running time of the incremental insertion algorithm (with either point location method)?
- Can you create a point set (or PSLG) for which one of the algorithms fails when the least robust predicates are used, but succeeds when exact predicates are used? (Hint: figure out a way to generate point sets with lots of nearly-collinear points.) Does the algorithm fail when you use the medium-robust predicates? Using a debugger (or other method), find out exactly why the algorithm is failing with the least robust predicates, and describe the reason.
- Descriptions of any extra or unusual features, algorithms, or point location methods you provide.

Prizes:

I'll award a $20 gift certificate (from a yet unchosen bookstore or record store) for each of the following:

- The fastest divide-and-conquer implementation.
- The fastest incremental insertion algorithm (using any point location method).
- The most ambitious extra feature (if it works and is pertinent).

For the first two awards, I will time the total running time of the program using the Unix `time` command; hence, I/O time counts. I will use a nonuniform point set for the timing, so an algorithm based on bucketing will probably not do well. If your program selects its algorithms and point location methods by using an input prompt (rather than command-line switches), you are unlikely to have the fastest timing. If I can't figure out how to time your code (which is quite likely if you're not using Unix), you're not eligible for the first two prizes.