CS 294-74
Mesh Generation and Geometry Processing in Graphics, Engineering, and Modeling
Delaunay Triangulation Project

Due 5:30 pm, Monday, March 5
25% of final grade

You may be able to skip this project if you're interested in doing an unusually challenging second project, such as a volume mesh generator. Speak to me if you want to explore this option.

Implement the star-based two-dimensional triangulation data structure of Blandford, Blelloch, Cardoze, and Kadow (without the compression, unless you want it). You may use linked lists for the vertex links, even though it's not space efficient. (I want you to understand the data structure and its interface more than actually realize its potential for memory efficiency.) Your library should have the ability to add and remove triangles, and to query the triangle adjoining an edge (on a chosen side of the edge). Your library should require the user to address triangles by their vertices; a triangle cannot be identified by a "triangle number" or pointer.

Next, implement an algorithm (your choice) that constructs two-dimensional Delaunay triangulations. Its asymptotic running time should be faster than quadratic for random points uniformly distributed in a square. Your implementation must use your Blandford et al. implementation as the sole representation of the triangulation.

Your code might become a component of your second project, so it's worth your while to debug it well.

Algorithms. The simplest algorithms to implement are incremental insertion and gift-wrapping. If you want the fastest triangulator possible, consider the divide-and-conquer algorithm described in the paper by Guibas and Stolfi, below. (You can make it even faster by alternating between vertical and horizontal cuts.) Because they provide complete pseudocode, it's not particularly difficult to implement, though some thought will be required to translate their pseudocode to a very different data structure.

If you choose the incremental insertion algorithm, walking point location is acceptable, though a faster point location method is welcome. The combination of a BRIO, a quadtree traversal to order points within each round, and walking from the last inserted vertex can be very fast yet easy to code. The method of simply checking every triangle in the mesh is not acceptable, because it will give you a quadratic-time algorithm.

If you implement gift-wrapping, use simple bucketing (with a square grid) and a search like the spiral search described by Su and Drysdale so that the triangulation speed is roughly linear on points uniformly distributed in a squre. The number of buckets should depend on the number of input points—certainly there should never be more buckets than input points.

If you took CS 274 (Computational Geometry) from me in the past, you have already written a Delaunay triangulator. You are welcome to retrofit your old code to use the new data structure, and turn it in.

If you think your second project might be an advancing front mesh generator, the gift-wrapping algorithm would be good practice. If you think your second project might be a Delaunay mesh generator, you will be able to reuse incremental insertion code from this project.

The following paper is a good source for pseudocode for both the divide-and-conquer algorithm and the incremental insertion algorithm (with a lame version of walking point location). However, the pseudocode is written for an edge-based data structure. Converting it for the Blandford-Blelloch-Cardoze-Kadow data structure will take some effort.

Leonidas J. Guibas and Jorge Stolfi, *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams*, ACM Transactions on Graphics **4**(2):74–123, April 1985. I recommend skipping Section 3.

Language. If you write in any language other than C or Java, you are required to give me very complete instructions on how to compile and run your code. If your submission does not run on a Mac, I may need even more help, and perhaps an account on a suitable machine.

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`. 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 that comes with Triangle to view and print your input and output. You can also use Triangle to check if you're producing correct triangulations.

Geometric predicates. Insofar as possible, your algorithms should base all their arithmetic decisions on the results of "orientation" and "incircle" predicates. If you write your own, they'll sometimes give wrong answers because of roundoff error. I suggest you use 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, file reading and writing, and geometric primitives like the orientation and incircle predicates. You must write the triangulation data structure and geometric algorithms all by yourself.

Your submission. My preferred submission method is that you email me an archive of the code (in `.zip` or `.tar.gz` formats). If it's too big to email, send me the secret URL of an archive containing all your code. If your code doesn't run on a Mac, you may also need to arrange a live demo of your code to me. (Note that Macs run Unix, so if you've written your code in a reasonably portable manner on a Linux or other Unix machine, it will probably work.)

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

- Instructions for compiling and running your code. If you use Unix and C, C++, or Java, relatively rudimentary instructions will probably do. Be sure to document how to specify the input file, and whether there are any options or switches.
- Timings on uniformly random sets of 10,000, 100,000, and 1,000,000 points. If your code is too slow to finish on the largest point set, just note that. 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.)
- Descriptions of your choice of algorithm(s) and point location method(s), and any extra or unusual features.