Homework #2: Voronoi and Delaunay diagrams [80 points]
Due Date: Monday, 7 November 2016

- **The Theory Problems**

Problem 1. [5 points]

In the plane the edges of both the Delaunay and Voronoi diagrams are line segments. Give a simple necessary and sufficient condition on a pair of sites A and B so that AB is a Delaunay edge *and* AB intersects its dual Voronoi edge.

Problem 2. [15 points]

Delaunay's theorem states that the triangulation of a set S on n sites in the plane is a Delaunay triangulation if and only if every edge passes the `InCircle` test with respect to its two adjacent triangles. This gives a linear-time algorithm to verify that a triangulation is Delaunay, and it also suggests the following algorithm to fix it up, if it is not: Start with any triangulation of the n sites. If an edge fails the `InCircle` test, then swap it with the other diagonal of the quadrilateral formed by the two adjacent triangles (the new edge must pass this local test). Continue in this fashion, until all the edges pass the `InCircle` test. Make this idea into a rigorous algorithm and prove its correctness. For example, is such a swap always feasible? Prove that your algorithm always terminates in $O(n^2)$ steps [*Hint:* define some quantity that monotonically decreases after each flip].

Problem 3. [15 points]

Davenport-Schinzel sequences of order 2 and triangulations:
Let P be a convex polygon with n vertices. A triangulation of P is a collection of $n-3$ non-intersecting diagonals connecting pairs of vertices of P and partitioning P into $n-2$ triangles. Set up a correspondence between such triangulations and $DS(n-1,2)$ sequences (Davenport-Schinzel sequences), as follows. Number the vertices $1, 2, \ldots, n$ in their order along the boundary ∂P. Let T be a given triangulation. Include in T the edges of P too. For each vertex i, let $T(i)$ be the sequence of vertices $j < i$ connected to i in T and arranged in *decreasing* order, and let U_T be the concatenation of $T(2), T(3), \ldots, T(n)$.

(a) Show that U_T is a $DS(n-1,2)$ sequence of maximum length.

(b) Show that any $DS(n-1,2)$-sequence of maximum length can be realized in this manner, perhaps with an appropriate renumbering of its symbols.

(c) Use (a) and (b) to show that the number of different $DS(n-1,2)$ sequences of maximum length is $\frac{1}{n-1}\binom{2n-4}{n-2}$ (where two sequences are different if one cannot obtain one sequence from the other by renumbering its symbols).

Problem 4. [10 points]

We discussed in class the lifting map $\lambda(x,y) : (x,y) \mapsto (x,y,x^2+y^2)$ from points in the xy-plane to points on the paraboloid of revolution $z = x^2 + y^2$. As we mentioned, the downwards-looking faces of the convex hull of the lifted images of a collection of sites in the xy-plane correspond to the Delaunay diagram of the sites. In this problem we will investigate what the upward-looking faces correspond to. These define planes that have all all other lifted sites not above but *below* them, implying that the corresponding triangles in the plane have circumcircles that contain all the other sites inside.

(a) Prove that these triangles form a triangulation of the convex hull of the sites; in other words, prove that the sites that are included in the triangulation are exactly the sites on the convex hull.

(b) The dual of this triangulation defines a new Voronoi diagram (partition of the plane) of the original sites. Show that this diagram is actually the *furthest point Voronoi diagram* of the sites, where the Voronoi region of a point p is defined to be the set of all points that are *farther* from p than any other point in the set. Use this to show that sites not on the convex hull have empty Voronoi regions in this diagram.

(c) How fast can the furthest point Voronoi diagram be computed?

- **The Programming Problem**

Problem 5. [35 points]

Kinetic Delaunay Triangulation:

The goal of the problem is to produce an animation of the Delaunay triangulation among moving points in the plane. The points will follow linearly parametrized motions and will be confined to move within a specified square in the plane. The latter constraint will be enforced by having the point bounce (reflect) off the walls of the square container. When they do so, their entire trajectory is reflected around the bounding wall. The points themselves have a 'non-zero size' (think of small disks of a fixed radius), and so they bounce off each other when their disks collide.

Since the motion of the points will be simple and predictable in the short term, this is an ideal setting for using the framework of *Kinetic Data Structures*, which are event-driven structures that allow the efficient maintenance of geometric attributes. We remarked in class that a Delaunay triangulation can be certified locally: if every edge of

the triangulation is locally Delaunay (passes its `InCircle` test), then the whole triangulation is globally Delaunay. This implies that the points can move and, as long as all the edge `InCircle` tests remain valid, the triangulation is still Delaunay. Since we know the motion of the points, we can predict when these `InCircle` certificates will fail. If we put these failure times into an event queue, we can then allow the motions to proceed until the first certificate fails. At that instant the Delaunay triangulation can be repaired with a single operation, the *edge-flip* we discussed in class. The new edge must then add the failure time of its `InCircle` certificate to the the event queue, and the simulation can proceed.

The above is a very simple example of *kinetic proof repair* for the certification of a geometric structure. More information about this approach and several worked out examples can be found in handout 4 and the following two papers, which can be accessed on from the "Lecture Notes" part of the class web page.

1. *Kinetic data structures — a state of the art report*, L. Guibas. Proc. 3rd Workshop on Algorithmic Foundations of Robotics, Houston, TX, 191–209, 1998.

2. *Data structures for mobile data*, J. Basch, L. Guibas, and J. Hershberger. J. Algorithms, **31**, 1–28 (1999).

Download the zip file for the assignment from the Handouts page on the course web site. This zip file contains starter code in Java and README file that explains how to setup Java programming environment. Feel free to contact the CA if you have any problem starting the homework.

The code uses the *half-edge* data structure for storing triangulations – you may read more about this data structure here `http://cs184.eecs.berkeley.edu/cs184_sp16/article/7`.

You have been provided with starter code that, when run, looks just like what your code should look like when it is complete; it is simply (far) less efficient than what we are aiming for. In particular, the starter code does the following:

- It reads in the input files to load the points and their initial trajectories. It then computes an initial Delaunay triangulation of the point set. This section is fine as is. Note that the triangulation includes an additional enclosing triangle for convenience, as discussed in class. This means that changes to the convex hull of the point set are simply changes to Delaunay edges involving this enclosing triangle, and so a single edge flip event type suffices to handle all changes to the triangulation.

- It provides a half-edge data structure (with all half-edges oriented counterclockwise) for your use in manipulating the triangulation. Edge flipping is already implemented; all you need to do is call it at the appropriate times. If you require assistance traversing the half-edge data structure, please come to office hours.

- It handles the real-time rendering and updating of the point set. You do not need

to touch the rendering code; you will be focusing on when to insert and remove events.

- Every time it renders the triangulation, it first recomputes the triangulation from scratch, using the current position of the points. This is clearly way too slow, but it allows you to see what your output needs to look like.
- It maintains an event queue of all potential reflection and collision events, between all points and the walls, and between all pairs of points. This gives you an idea of how events work in kinetic data structures; however, it is currently slower than our goal, since it maintains more potential events than is necessary.
- It provides a variety of utility methods that you will find useful; the most important one is the one that computes the failure time of an edge in order to generate an edge flip event. Note that these events currently do nothing, nor are they inserted into the event queue.

While it is entirely possible to do this assignment without opening several of the starter code files, we encourage you to briefly read over all of the files to understand how they fit together. Obviously, you should carefully read the files you need to modify.

The starter code contains a few TODOs marked in two files: `Manager.java` and `EdgeFlipEvent.java`. The `Manager.java` file handles the bulk of the work, including the management of the event queue. The `EdgeFlipEvent.java` file handles the processing of edge flip events in particular. You must modify these two files to complete the following 3 (interconnected) tasks:

- You must not recompute the triangulation; instead, you must update it on the fly.
- You must handle edge flip events. This includes not only flipping the edge that fails, but also inserting additional edge flip events and updating any associated reflection and collision events. Furthermore, be warned that the processing of reflection and collision events can affect the edge flip events of incident edges.
- You must be more efficient with reflection and collision events. In particular, only points currently on the convex hull of the point set should have their reflection events tracked, and only pairs of points sharing a Delaunay edge should have their collision events tracked.

For up to 5 points extra credit, you may augment your code to support constant acceleration on any or all of the points.

Submission instructions: Pack all of your source files into a single zip file and email the file to the CA (mhsung@cs.stanford.edu). Include a README file with the names of all team members, and what augmentations (if any) you make to the input format to handle the extra credit.