MCS 481 / David Dumas / Spring 2014

Project 2: Polygons

Due Friday, March 7

0. Overview

For this project you have two options:

- Implement *polygon triangulation* and a set of *test cases*.
- Go "off the rails" and propose your own experimental or coding project.

The options are described in more detail below.

1. Coding option - Polygon Triangulation

Problem specification. Implement a $O(n \log(n))$ algorithm to triangulate a simple polygon without holes. For example, you could implement the plane sweep to decompose an arbitrary simple polygon into monotone pieces, followed by a stack-based algorithm to triangulate each piece. For the second step you might use the direct approach in the text or the variation involving a decomposition into "mountains" that was presented in lecture.

As part of the implementation you are expected to test your program on various polygons and examine the output. You should think about a convenient way to generate test cases and also how to visualize the triangulations produced by your program. You might also wish to compare your program to one that uses CGAL to produce a triangulation (see below).

You are also required to submit test cases with your code, and the quality of the tests (and your program's correctness when applied to them) will be considered as part of your project evaluation.

The evaluation of your project will also involve automated testing of your code. Therefore, it is very important to conform to the following input and output format specifications.

Input. Your program will read a list of points from standard input (`stdin`). Each line of input will contain the coordinates x y of one point, separated by whitespace characters (e.g. space, tab). The points will be the vertices of a simple polygon P in counterclockwise order. The first vertex in the list will have the largest y coordinate and will be leftmost among such vertices in case of a tie. For example, the following input describes a square inscribed in the unit circle:

```
0 1
-1 0
0 -1
1 0
```

The list of vertices may be followed by any number of blank lines or lines containing only whitespace characters, which your program must ignore.

Project description v1.0 (February 12, 2014)

Output. Write a list of triangles to standard output (`stdout`), one per line, defining a triangulation of P. A triangle is described by three integers, representing the zero-based indices of its vertices in the input list. On each line of output, the triangle vertices will appear in counterclockwise order. For example, the following represents one possible triangulation of the polygon described above:

```
0 1 2
0 2 3
```

In this sample output, the first line describes the triangle with vertices $(0, 1)$, $(-1, 0)$, and $(0, -1)$.

Reference implementation. A C++ program that uses CGAL to triangulate a polygon and print the result in the format described above is included with this project description. (See code listings at the end of this document.)

Keep in mind that you need to create your own triangulation program from scratch, and may not use CGAL in your project. This sample program is included in the hope that it will clarify the desired input and output format and because it might be useful for testing purposes.

Implementation advice. The textbook assumes that the input polygon is represented by a doubly-connected edge list (DCEL), but implementing a full DCEL is not necessary for this assignment. Instead of maintaining a separate vertex structure, you can build the coordinates of the origin into each half-edge record. A face structure is not needed at all. At the last step you can find cycles of half-edges and print the results.

Language choice. The same programming language policies apply as in Project 1. Contact me for approval if necessary.

Built-in functions and libraries. Your code may *not* use any built-in computational geometry functions.

Your code may use built-in data structures such as arrays, linked lists, hashes, search trees, stacks, or queues, if such structures are provided by the language or CAS you choose. If you want to use a library that implements any of these basic data structures, contact me for approval.

What to submit. Write a report on your implementation process. Start by briefly describing the triangulation problem and the algorithm. Then focus on the design decisions that were involved in your implementation.

Create test cases for your code that demonstrate its correctness for various types of input (a convex polygon, a monotone polygon, a polygon requiring several monotone pieces, etc.). Display these test polygons and the triangulations produced by your program in the report.

Submit the report, source code, and test cases by email (`ddumas@math.uic.edu`), following the standards for coding option and source code submissions from the description of Project 1.

2. Off the rails

For this option you will design your own coding or experimental project and email me a proposal (`ddumas@math.uic.edu`). Your proposal should be a few paragraphs in length, roughly similar to the kind of description given above for the polygon triangulation coding option. Your description **must** indicate:

- What final products you will submit for evaluation
- What references you will use (to show you have some idea where to start)
- What algorithms you will implement (if coding) or study (if doing experiments)

Furthermore, if choosing an experimental option, indicate what implementation(s) you will study and what questions you intend to answer about them.

The proposal is due by **Wednesday, Feb 19**. I will read your proposal and either approve it or suggest some changes. I will also indicate how your project would be evaluated. Once you receive approval and are satisfied with the proposed evaluation standards, you can begin working.

Example. For an experimental project you might choose to study the convex decomposition algorithms in CGAL by applying them to random polygons. You could use the CGAL documentation, source code, and the research papers cited therein as your primary references. Your final product could be a report describing timing studies and comparison of all three convex decomposition methods applied to several classes of polygons:

- Random n-gons
- Monotone n-gons
- n-gons in which the number of concave vertices is $\Omega(n)$

Goals for such a project might include:

- Explore the claims in the CGAL documentation claims about which "approximately optimal" method is likely to produce fewer convex pieces
- Study whether the running time behaves as claimed (e.g. $O(n^4)$ for the optimal decomposition) and whether it appears to be sensitive to the number of concave vertices.

3. Code listings

3.1. **Triangulation using CGAL.** This program reads vertices a simple polygon from standard input and writes a triangulation of the polygon to standard output. The input and output formats are as described in section 1.

```
// triangulate.cpp
// MCS 481 Spring 2014 project 2 description version 1.0
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/centroid.h>

#include <cassert>
#include <iostream>
#include <list>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_2<K>                      Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K>            Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>               TDS;
typedef CGAL::Exact_predicates_tag                                Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, TDS, Itag> CT;

typedef CGAL::Partition_traits_2<K>    Traits;
typedef CT::Point                      Point;
typedef CGAL::Triangle_2<K>            Triangle;
typedef Traits::Polygon_2              Polygon;
typedef Polygon::Edge_const_iterator   EdgeIterator;

int
main( )
{
  Polygon pgn;

  // Read vertex list from stdin and construct polygon
  Point p;
  while (std::cin >> p) {
    pgn.push_back(p);
  }

  // We ask CGAL to triangulate the entire plane so that each edge of
  // P is one of the edges of this triangulation (these are the
  // "constraints").
  CT ct;
  EdgeIterator ei;
  for (ei = pgn.edges_begin(); ei != pgn.edges_end(); ei++) {
    ct.insert_constraint( (*ei).source(), (*ei).target() );
  }
  assert(ct.is_valid());

  // The CT object maintains a triangulation of the plane at all
  // times, adapting as needed it every time we add a constraint.  So
  // after the loop above, it includes a plane triangulation with P
```

```
  // among its edges.

  // Now we determine which triangles actually lie inside P and print
  // them.

  CT::Finite_faces_iterator fi;
  for (fi = ct.finite_faces_begin(); fi != ct.finite_faces_end(); fi++) {
    Triangle t = ct.triangle(fi);
    Point m = CGAL::centroid(t);
    if (pgn.bounded_side(m) == CGAL::ON_BOUNDED_SIDE) {
      // Current triangle is inside P, print it
      for (int i=0; i<3; i++) {
        std::cout << std::distance(pgn.vertices_begin(),
                                   std::find(pgn.vertices_begin(),
                                             pgn.vertices_end(),
                                             t.vertex(i) ));
        if (i==2)
          std::cout << std::endl;
        else
          std::cout << " ";
      }
    }
  }
  return 0;
}
```