MCS 481 / David Dumas / Spring 2011

Project 2: Polygons

Due Friday, March 4

0. Overview

For this project you have two options:

- Coding option: Implement *polygon triangulation.*
- Experimental option: Study *polygon decomposition* with CGAL.

The options are described in detail below.

1. Coding option - Polygon Triangulation

Problem specification. Implement the polygon triangulation algorithm discussed in class, i.e. a plane sweep to decompose an arbitrary simple polygon into monotone pieces, followed by a linear incremental algorithm to triangulate each piece. For the second step you can use either the stack-based monotone triangulation algorithm in the text, or the variation involving a second decomposition into "mountains" that was presented in lecture.

Your program will accept as input a list of 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 output of your program will be a list of triangles, 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, starting from the one with smallest index. 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 $(0, -1)$.

Extra credit opportunity: Instead of the triangulation algorithm discussed in class, implement a different triangulation method whose running time is also $O(n \log n)$. In this case your report should begin with a complete description of the algorithm. Before pursuing this option, contact me for approval of the algorithm you intend to implement.

Project description v1.0 (February 13, 2011)

Implementation advice. The textbook assumes that the input polygon is represented by a doubly-connected edge list (DCEL), but implementing a full DCEL is probably overkill for this assignment. You may instead want to work with simple arrays of vertices stored in CCW order, breaking the algorithm into additional steps to maintain and update these structures, i.e.

(1) Find monotone decomposition diagonals using a plane sweep
(2) Use diagonals to break the input list into vertex lists for the monotone pieces
(3) Sort the vertex list for each monotone piece by y-coordinate and triangulate.

Each vertex should be stored in a structure that includes its coordinates and its index in the input list, since these indices are needed for the output format.

Language choice. You may use any approved programming language or computer algebra system (CAS) for your implementation. The following languages and CASs are pre-approved:

```
C C++ Python Perl Java
Sage Octave Mathematica Maple Matlab
```

Contact me for approval if you want to use another language or CAS.

Built-in functions and libraries. Your code may *not* use any built-in computational geometry functions or general-purpose solvers (e.g. `Solve[]` in Mathematica).

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 in class or by email. Also email an archive containing the source code for your program and the data files for the test cases described in your report.

2. Experimental option - Polygon decomposition

Experiment 1: Random polygons. How should one generate a "random" simple polygon with n vertices? A naïve algorithm might repeatedly generate a list of n random points (in a certain planar region) until the list forms the CCW boundary of a simple polygon. CGAL provides a more sophisticated polygon generator, `CGAL::random_polygon_2()`. Compare these two methods by generating polygons with various numbers of vertices. Sample generator programs are included in the source code listings at the end of this document.

It should become apparent that the naïve algorithm is not useful for polygons with more than a dozen or so vertices. Based on your results, do you have a guess for its expected running time as a function of n?

Read the CGAL documentation to determine what method `CGAL::random_polygon_2()` uses and the expected asymptotic behavior of the running time $T(n)$ of this algorithm as a function of n. Include a brief description in your report. Do you observe the expected running time in your experiments? It may be useful to graph $\log(T(n))$ as a function of $\log(n)$; in such a graph, a running time function that behaves like Cn^α would appear as a line of slope α.

Extra credit opportunity: The naïve random polygon generator and the CGAL generator induce different probability distributions on the set of all simple n-gons in $[-1,1] \times [-1,1]$. Can you observe any differences between the polygons generated by these two algorithms? If so, can you justify your observation theoretically, in terms of the generator algorithms?

Experiment 2: Monotone decomposition. CGAL implements a plane sweep algorithm to decompose a given CCW-oriented simple polygon into monotone pieces with the function `CGAL::y_monotone_partition_2()`. The example program `y_monotone_partition_2` in `examples/Partition_2` demonstrates its use, and a modified version of this program which reads input from `stdin` is shown in the source code listings at the end of this document.

We analyzed this plane sweep algorithm in class and obtained a bound of $O(n \log n)$ for its running time. Is this the behavior that one sees in practice, or does this represent a worst-case that is very unlikely? Test the CGAL implementation with random polygon input and analyze the results. Can you distinguish linear growth from $n \log(n)$?

Be careful to measure only the time used by the monotone decomposition, and not the time necessary to generate the random polygons used as input.

Extra credit opportunities:

- Generate monotone polygons and use these to test for output-sensitive behavior in the partition algorithm.
- Generate polygons that require many pieces in the monotone decomposition and use these to test for output-sensitive behavior in the partition algorithm.
- For a given value of n, make a histogram showing the number of pieces in the monotone decomposition for a large number of random n-gons. Comment on the results.

Experiment 3: Convex decomposition. A *convex decomposition* of a polygon P is a set of diagonals that cut P into convex pieces. For example, a triangulation is a convex decomposition. A convex decomposition of P is *optimal* if there does not exist a convex decomposition with fewer pieces.

CGAL provides an algorithm to find an optimal convex decomposition of a given polygon with the function `CGAL::optimal_convex_partition_2()`. The example program `optimal_convex_partition_2` in `examples/Partition_2` demonstrates its use, and a modified version of this program which reads input from `stdin` is shown in the source code listings at the end of this document.

Read the CGAL documentation to determine the worst-case running time of this algorithm as a function of n. Experiment with random polygonal input and compare your results to this worst-case bound. Try to find a real number α so that the running time behaves like Cn^α for some $C > 0$.

Guidelines for data collection. When examining the running time of an algorithm as a function of the input size n, use several values of n distributed in either an arithmetic

progression (e.g. $100, 200, 300$) or approximate geometric progression (e.g. $2, 4, 8, 16$ or $10, 20, 50, 100, 200, 500$). For arithmetic progressions, the total range of values (i.e. $n_{\max} - n_{\min}$) should be much larger than the smallest value $n_{\min}$. Generally speaking, you should use 8 or more values of n. Start with a value of n that is large enough to exceed the overhead of initializing the data structures; failure to do so might appear as running times that are almost the same for the first few values of n.

What to submit. Write a report that describes your experiments, presents the data, and interprets the results. The report should have one section for each of the three experiments described above. It is not necessary to include tables listing all of the raw data; rather, include graphs and tables that summarize enough of the data collected to justify your conclusions.

Submit the report in class or by email. Also email an archive containing the source code for all programs and scripts used in your experiments.

3. Code listings

3.1. **Naïve random polygon.** The following program prints the CCW-oriented vertex list of a random polygon in the square $[-1, 1] \times [-1, 1]$. The polygon is generated using the naïve algorithm described above. The number of vertices can be specified as a command-line parameter.

```
// rand-ngon-naive.cpp
// MCS 481 project 2 description version 1.0
#include <iostream>
#include <cstdlib>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Polygon_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon;
typedef Polygon::Vertex_iterator VertexIterator;

using namespace std;

double dblrand()  // pseudorandom double in [0.0,1.0).
{
  return rand()/(double(RAND_MAX) + 1);
}

int main(int argc, char *argv[])
{
  Polygon pgn;
  int nv = 5;  // Default number of vertices

  // First command-line argument is number of verices
  if (argc > 1)
    nv = atoi(argv[1]);

  srand((unsigned)time(0)); // Seed the random generator with current time

  // Generate random sequences of points until the result is a simple polygon
  int n = 0;
```

```
  do {
    pgn.clear();
    for (int i=0; i<nv; i++) {
      Point p(2.0*dblrand()-1.0, 2.0*dblrand()-1.0);
      pgn.push_back(p);
    }
    n++;
  } while (!pgn.is_simple());

  // Uncomment next two lines to print number of iterations
  // cout << "Required " << n << " attempts to generate a simple "
  //      << nv << "-gon." << endl;

  // Make counterclockwise, for consistency
  if (!pgn.is_counterclockwise_oriented())
    pgn.reverse_orientation();

  // Print the vertices
  for (VertexIterator vi = pgn.vertices_begin(); vi != pgn.vertices_end(); ++vi) {
    cout << *vi << endl;
  }
  return 0;
}
```

3.2. **CGAL random polygon.** The following program uses CGAL to generate an n-gon in the square $[-1, 1] \times [-1, 1]$. The number of vertices can be specified as a command-line parameter.

```
// rand-ngon-cgal.cpp
// MCS 481 project 2 description version 1.0
#include <iostream>
#include <cstdlib>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/random_polygon_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon;
typedef CGAL::Creator_uniform_2<double, Point> Creator;
typedef CGAL::Random_points_in_square_2<Point, Creator> Point_generator;
typedef Polygon::Vertex_iterator VertexIterator;

using namespace std;

int main(int argc, char *argv[])
{
  Polygon pgn;
  int nv = 5;  // Default number of vertices

  // First command-line argument is number of verices
  if (argc > 1)
    nv = atoi(argv[1]);

  CGAL::random_polygon_2(nv, std::back_inserter(pgn), Point_generator(1.0));
```

```
  // Print the vertices
  for (VertexIterator vi = pgn.vertices_begin(); vi != pgn.vertices_end(); ++vi) {
    cout << *vi << endl;
  }
  return 0;
}
```

3.3. **Monotone partition.** The following program reads the vertex list of a simple polygon from `stdin` and uses CGAL to decompose this polygon into monotone pieces. The number of pieces and the vertex list for each piece are written to `stdout`.

```
// monotone-partition-cin.cpp
// Modified version of a CGAL demo (examples/Partition_2/y_monotone_partition_2.cpp)
// MCS 481 project 2 description version 1.0
#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <list>
using namespace std;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point;
typedef Traits::Polygon_2                                   Polygon;
typedef list<Polygon>                                       PolygonList;
typedef Polygon::Vertex_iterator                            VertexIterator;
typedef list<Polygon>::const_iterator                       PolygonIterator;

int main( )
{
  Polygon    pgn;
  PolygonList monotone_pieces;

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

  // CGAL requires CCW orientation for monotone partition
  // If the input polygon is CW oriented, we reverse it
  if (!pgn.is_counterclockwise_oriented())
    pgn.reverse_orientation();

  // Compute y-monotone partition and store results in monotone_pieces
  CGAL::y_monotone_partition_2(pgn.vertices_begin(),
                               pgn.vertices_end(),
                               back_inserter(monotone_pieces));

  // Print number of polygons in the decomposition
  cout << "Number␣of␣monotone␣pieces:␣" << monotone_pieces.size() << endl;

  PolygonIterator pi;
```

```
  int n=1;
  for (pi = monotone_pieces.begin(); pi != monotone_pieces.end(); pi++) {
    cout << "Piece " << n++ << ":" << endl;
    VertexIterator vi;
    for (vi = (*pi).vertices_begin(); vi != (*pi).vertices_end(); vi++) {
      cout << "\t" << *vi << endl;
    }
  }
  return 0;
}
```

3.4. **Convex partition.** The following program reads the vertex list of a simple polygon from `stdin` and uses CGAL to optimally decompose this polygon into convex pieces. The number of pieces and the vertex list for each piece are written to `stdout`.

```
// convex-partition-cin.cpp
// Modified version of a CGAL demo (examples/Partition_2/optimal_convex_partition_2.cpp)
// MCS 481 project 2 description version 1.0
#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <list>
using namespace std;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K>                         Traits;
typedef Traits::Point_2                                     Point;
typedef Traits::Polygon_2                                   Polygon;
typedef list<Polygon>                                       PolygonList;
typedef Polygon::Vertex_iterator                            VertexIterator;
typedef list<Polygon>::const_iterator                       PolygonIterator;

int main( )
{
  Polygon    pgn;
  PolygonList convex_pieces;

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

  // CGAL requires CCW orientation for monotone partition
  // If the input polygon is CW oriented, we reverse it
  if (!pgn.is_counterclockwise_oriented())
    pgn.reverse_orientation();

  // Compute optimal convex partition and store results in convex_pieces
  CGAL::optimal_convex_partition_2(pgn.vertices_begin(),
                                   pgn.vertices_end(),
                                   back_inserter(convex_pieces));

  // Print number of polygons in the decomposition
```

```
  cout << "Number␣of␣convex␣pieces:␣" << convex_pieces.size() << endl;

  PolygonIterator pi;
  int n=1;
  for (pi = convex_pieces.begin(); pi != convex_pieces.end(); pi++) {
    cout << "Piece␣" << n++ << ":" << endl;
    VertexIterator vi;
    for (vi = (*pi).vertices_begin(); vi != (*pi).vertices_end(); vi++) {
      cout << "\t" << *vi << endl;
    }
  }
  return 0;
}
```