An Experimental Comparison of Two-Dimensional Triangulation Packages*

Matthias Bäsken and Oliver Zlotowski

FB IV – Informatik
Universität Trier, D-54296 Trier
{zlotowsk,baesken}@informatik.uni-trier.de

Abstract

We describe a generic software package for two-dimensional triangulations that can be used with different geometry kernels and discuss some implementation aspects such as data structures, filter strategies and algorithms. In the second part we present an experimental comparison of different triangulation packages.

1 Introduction

The computation of triangulations, especially of Delaunay and simple triangulations, is a fundamental problem in computational geometry. Libraries like CGAL [3] and LEDA [7] provide algorithms for these data structures. There are also other packages such as triangle [9] or Qhull [1].

In this abstract we compare some of these packages with our implementation. This implementation is based on a prototype of a generic graph library. We discuss a number of different versions of Delaunay-flipping based implementations and describe our efforts to speed up the used geometric predicates. Finally, we present some experimental results.

2 The generic graph library

There is a strong relation between geometric algorithms and graphs because many geometric algorithms work on a graph data structure. In this section we discuss some aspects of the generic graph that is used to represent the triangulation.

LEDA contains a very powerful and flexible graph data structure that can be used for solving most standard graph problems. This type is used to represent all kinds of graphs used in the library, such as networks, planar maps or triangulations. However, it is not hard to see, that in many situations the general graph is not necessary. In practice a loss of efficiency compared to more specialized solutions can be observed.

Another approach is to use special data structures tailored for the specific problem. The memory consumption of these data structures are usually lower and the algorithms are faster on such a structure. On the other hand, specialized solutions are not well suited in situations where more than one special problems has to be solved. In such cases often a conversion to a more general data structure might be necessary which is inefficient and

*This work was supported in part by DFG-Grant Na 303/1-2, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen"

memory-consuming. Our goal is to bridge this gap by developing a generic graph library. For more details see [8].

For a lot of applications a fully dynamic data structure is not required, so it makes sense to offer static structures. In our package we provide a generic static graph with constant degree implemented by the data type *scd_graph*. This type can be parameterized with user defined node and edge types.

A graph with static constant degree is usable for storing two-dimensional triangulations. Our triangulation graph *triang_graph* is derived from the base graph *scd_graph*. It uses a special triangulation node and edge for representing the graph objects and enhances the base graph by a point container. The representation is close to the representation described by Shewchuk [9] that is used in its triangle package. A node in the triangulation graph represents a face of the triangulation. Two neighbor faces are linked by an edge.

By using a vector for storing the combinatorial structure the graph traversal can perform very efficiently. The *triang_graph* provides also operations for fast local transformations used in the algorithms to transform a triangulation into a Delaunay triangulation.

3 Algorithms for computing Delaunay triangulations

Shewchuk [9] and Su/Drysdale [10] have presented experimental comparisons of a number of different algorithms for computing the Delaunay triangulation. The conclusion of these comparisons is that divide and conquer based algorithms are the best solutions for many problem instances. Implementations of such algorithms (especially Dwyer's divide and conquer algorithm) can be found in LEDA and triangle. Another successful strategy is the combination of randomized incremental algorithms with a hierarchical data structure used for point location [2]. An implementation using hierarchies can be found in CGAL.

In our implementations we combine ideas of divide and conquer based algorithms with simple local transformation strategies. This way we achieve good performance for randomly distributed input. The reason for this observation is that the running time of Delaunay flipping algorithms transforming an input triangulation into a Delaunay triangulation depends heavily on the *quality* of the input triangulations. We measure the *quality* of an input triangulation by counting the number of local transformation operations that we have to perform. We observe that a triangulation with many triangles with large circumcircles has a bad quality. Such a triangulation for instance is constructed by the LEDA algorithm for simple triangulations.

We obtain a much better start triangulation by applying a divide and conquer step using horizontal cuts (similar to the second phase of Dwyer's algorithm) that divide our input point set into lexicographically ordered subsets. These subsets are separately triangulated using a sweep algorithm. Then we merge these triangulated parts. In the next step we perform the Delaunay flipping algorithm on the resulting triangulation and obtain a Delaunay triangulation.

Another variant performs before the merging step Delaunay flipping on the triangulated parts. Afterwards these parts are merged by an algorithm similar in kind to the merge subroutine used in Dwyer's algorithm.

4 Speeding up the predicates

For the computation of the Delaunay triangulation the following predicates are used:

- a predicate for checking whether a point p lies inside, outside or on a circle defined by three points a, b, c
- predicates *left_turn*(a, b, c) and *right_turn*(a, b, c) returning *true* if the three points a, b, c form a *left_turn* resp. a *right_turn*; these predicates are special cases of the *orientation* predicate
- predicates for comparing point coordinates

Our algorithm implementations can be parameterized by a traits class so that we can support the different kernels available in CGAL, the LEDA floating point kernel and rational kernel.

We also provide a few own implementations of point data types in our package for experimental reasons, especially points with simple Cartesian double coordinates and variable caching of an exact representation of these objects.

The usage of some of these kernels (for instance the LEDA floating point kernel or CGAL kernels using non-robust number types) is dangerous because they use inexact arithmetic. By switching from a non-robust kernel to a kernel with exact arithmetic we overcome the danger of predicates returning faulty results, but we have to pay for this by obtaining increased running times for our predicates.

The questions is now: How can we get exact and fast predicates? An answer to this question is the usage of floating point filters [4]. The idea of floating point filters is to calculate geometric predicates using floating point arithmetic, but only if we are sure that the result is correct. Otherwise, we use exact arithmetic instead. There are several implementations for different filter techniques available. CGAL provides a template data type *Filtered_exact* that is easy to use and very flexible. The ideas suggested by [4] are also used in the LEDA rational kernel. The fact is used that in the predicates we only want to know the sign of an arithmetic expression E, not the value. Fortunately it is often possible to compute the sign using doubles. In a first step we compute an approximation $\tilde{E}$ of the arithmetic expression using double arithmetic.

In a second step we have to compute an error bound B that must be compared with $\tilde{E}$. The result of this comparison will tell us whether $\tilde{E}$ is reliable or not. If $\tilde{E}$ is reliable, we have the result of the predicate. Otherwise we have to switch to the slower exact arithmetic. The presented version of arithmetic filtering in the LEDA kernel is called *semi-dynamic filtering*. We compute a part of the error bound B on the fly. The other part is precomputed.

Another method for filtering is *static filtering* introduced in [5]. In this case the error bound B is entirely precomputed. To be able to do that we must have an assumption about the length of the input coordinates. We can save time because we do not need to compute (parts of) the error bound on the fly. On the other hand, this kind of filter might be less precise. To overcome this problem, we mix the semi-dynamic filter with the static approach. We add another step to our filtering scheme.

For this step we compute the highest absolute coordinate values x_{max}, y_{max} and w_{max} of the input points of our algorithm. These values are used for computing an upper bound B_{max} for all values of B that might arise in our algorithm for these input points. Now we use this upper bound B_{max} before the computation of B for a comparison with $\tilde{E}$. In many cases this will help us to avoid the computation of B. Note that we compute the approximation $\tilde{E}$ only once, so mixing static and the old semi-dynamic filtering is very easy.

Another improvement we use is filtered sorting using structural filtering [6] in the sorting step of the computation. We sort our points in two steps: in the first step we use fast but inexact predicates. In a second step filtered exact predicates are used. Usually a special sorting algorithm is applied in the second step.

5 Some experimental results

We compare our package with implementations of Delaunay triangulations in CGAL (using the "normal" and the hierarchical Delaunay triangulation), LEDA (Dwyer's divide and conquer), triangle (divide and conquer) and Qhull (lifting). All experiments were executed on a Sun UltraSPARC IIi 440 MHz with 256 Mbytes of main memory running Solaris 2.7. The test programs have been compiled with gcc-3.0.3. We used the current beta versions of LEDA-4.4 and CGAL-2.4.

The simple point (simple double in the table) is a point data type with Cartesian double coordinates. It does not use the handle type mechanism. The simple point exact (simple exact) adds evaluation of the error of the floating point computations to the simple point and switches to exact computation if necessary.

All CGAL tests were performed using the simple Cartesian kernel. In the table FE means Filtered-exact parameterized with double and LEDA real. Note that the generated input points were passed to the corresponding functions without permuting them. In case of the lattice the CGAL hierarchy works very bad on non-permuted input (the output of the lattice generator are points in sorted order). The permutation of the input point set makes the hierarchy run 10 times faster (for the square and disc input the permutation has no influence on the running time). Maybe it would be a good idea to use always permutation in the CGAL hierarchy to avoid these cases.

Our results show that we obtain good performance on randomly generated input, sometimes even better than triangle which is regarded as a fast package.

Num. of points	50.000			100.000			200.000		
Distribution Algorithm	Random in square	Random in disc	Lattice	Random in square	Random in disc	Lattice	Random in square	Random in disc	Lattice
Qhull	3.80	3.81	6.48	8.06	8.16	13.72	17.32	17.82	28.98
triangle	0.64	0.69	1.37	1.52	1.52	1.86	3.60	3.55	4.14
CGAL									
FE normal	35.90	32.07	48.02	102.47	91.59	139.65	287.29	261.31	—
FE hierarchy	7.25	6.98	53.79	14.07	14.00	153.84	30.70	30.01	—
double norm.	8.54	7.70	11.48	27.00	23.90	33.27	78.70	71.90	105.24
double hier.	1.97	1.91	12.97	4.05	3.97	36.87	8.89	8.76	—
LEDA Dwyer									
rat. kernel	2.01	2.04	2.81	4.18	4.21	6.04	8.91	8.71	12.23
float kernel	1.47	1.47	1.19	3.15	3.15	2.59	6.43	6.52	5.55
Our package									
CGAL FE	3.19	3.20	14.44	6.39	6.53	39.32	13.00	13.15	110.00
CGAL double	0.72	0.70	2.68	1.45	1.49	7.27	3.00	3.02	20.41
LEDA rat. ker.	1.58	1.61	5.74	3.27	3.33	14.52	6.94	6.86	37.73
LEDA float ker.	0.81	0.79	2.58	1.75	1.76	7.22	3.73	3.71	20.87
simple exact	0.63	0.65	6.12	1.33	1.33	13.91	2.74	2.76	33.52
simple double	0.60	0.61	2.11	1.25	1.28	5.83	2.60	2.60	15.92

Table 1: Experimental results for computing Delaunay triangulations.

References

[1] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, Dec. 1996.

[2] O. Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[3] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[4] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for comutational geometry. manuscript, 1992.

[5] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[6] S. Funke. Combinatorial curve reconstruction and the efficient exact implementation of geometric algorithms. 2001.

[7] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambrige University Press, 1999.

[8] S. Näher and O. Zlotowski. Design and Implementation of Efficient Data Types for Static Graphs. 2002. submitted to the 9th Annual European Symposium on Algorithms.

[9] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[10] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 61–70, 1995.