# An object oriented workbench for experimental geometric computation

Peter Schorn, Institut für Theoretische Informatik, ETH Zürich, CH-8092 Zürich, Switzerland

Extended Abstract

## 1. Introduction

Research in Computational Geometry has traditionally focused on the more theoretical side of geometric computation and one often wonders whether a proposed algorithm is useful in practice or not. Even a prototypical implementation can become very time consuming if it is attempted without the right tools such as sufficiently general abstract data types (e.g. dictionary, priority queue) or simple visualization aids. In order to alleviate this situation we have designed and implemented a framework for geometric algorithms which supplies the implementor with the necessary tools and allows experiments with an algorithm with little or no source code modification. For debugging and teaching we support algorithm animation. The system is implemented in an object oriented extension of Lightspeed Pascal on the Macintosh computer and consists of about 450 kByte of source code.

Similar to our efforts is the project LEDA [M 89] which also focuses on efficient reusable software. In addition to reusability and efficiency we are concerned with robustness and want to provide an interactive system for experimental geometric computation.

In the following we describe how we aid the implementor, what kind of experiments one can perform and how algorithm animation is done. We then give a short overview of the user interface and present the lessons learned from the object oriented approach before we come to the conclusions.

## 2. Components of the workbench

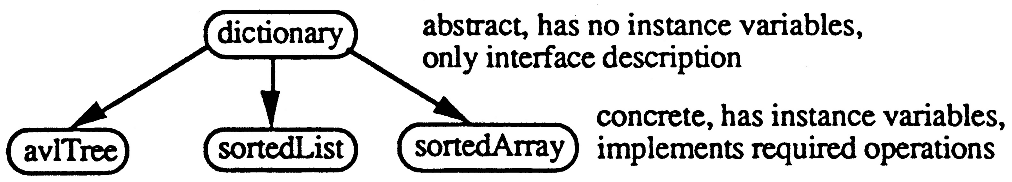
The following features have proven useful when implementing nontrivial geometric algorithms: geometric primitives, abstract data types, a rich basis of geometric algorithms, interchangeable arithmetic, input / output. We describe them in turn and show what we have to offer the implementor.

### 2.1 Geometric primitives

We have coded the most common geometric primitives such as testing on which side of a directed line segment a given points lies, testing whether two line segments intersect, computing the intersection of two line segments (can be a segment), computing the center of a circle given by three non collinear points etc. We have tried to make them numerically robust and as general as possible without sacrificing efficiency. Note also the distinction between test operations (e.g. a intersection test) that yield a boolean value and operations that actually compute geometric data (e.g. center of a circle).

### 2.2 Abstract data types

Almost all geometric algorithms rely on data structures for efficiency reasons and the data type dictionary is one of the key structures. In our setting, a dictionary is an object which allows the usual operations such as create, insert, find and delete. We can create dictionaries holding any objects as long as we supply a function to compare them. Another important characteristic is the distinction between key operations and dictionary operations: when an object is inserted into a dictionary the dictionary returns a so called *reference* which can be used later to refer to the object inside the dictionary; when the object is to be deleted this reference is supplied to the dictionary. This approach has the following advantages: key comparisons are avoided since they tend to be expensive in geometric algorithms and objects can be modified while remaining in a dictionary thus saving time consuming delete and insert operations. The following shows a small part out of the object hierarchy.


dictionary
abstract, has no instance variables, only interface description
avlTree
sortedList
sortedArray
concrete, has instance variables, implements required operations

The root object 'dictionary' only specifies the necessary operations while its descendants actually implement the specification. This ensures a common interface for different implementations of a dictionary and facilitates experiments with different data structures, if only the methods provided by 'dictionary' are used.

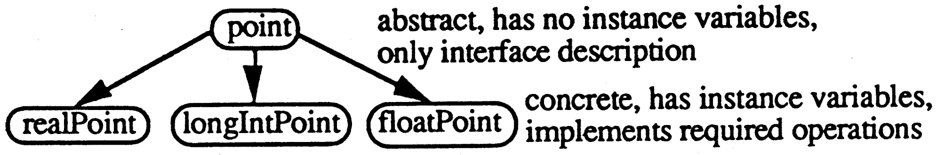
More complex objects (e.g. a priority queue) can be composed from simpler ones.

## 2.3 Geometric algorithms implemented

The workbench provides a wide variety of algorithms as a basis for further implementation efforts: convex hull (Graham's scan, divide and conquer), closest pair (sweep line, probabilistic), all-nearest-neighbors, Voronoi diagram (sweep line, divide and conquer), Euclidean minimum spanning tree, Euclidean traveling salesman (three different heuristics), intersection of (convex) polygons, etc.

## 2.4 Interchangeable arithmetic and parameterized floating point arithmetic

A very important aspect of this testbed is the ability to study the effect of various arithmetics on the outcome of a computation. In order to achieve this goal we have defined an object 'point' which has no instance variables for the coordinates but specifies an interface with access procedures to the coordinates and various geometric primitives.


point
abstract, has no instance variables, only interface description
realPoint
longIntPoint
floatPoint
concrete, has instance variables, implements required operations

From this abstract point we derive concrete point objects having instance variables and implementing the geometric primitives in their respective arithmetic. In order to study not only the built-in floating point arithmetic (as used in realPoint where the x- and y-coordinates are of type *real*) we have implemented a software floating point package with arbitrary base and precision which is used for the coordinates of the object 'floatPoint'. Algorithms using only the functions and procedure specified by the abstract type 'point' can be run in any of the three kinds of arithmetic currently supported.

## 2.5 Input / output of geometric objects

All of the geometric object types currently supported (point, line segment, rectangle, polygon, convex polygon) can be stored on secondary storage and retrieved from there. Furthermore they can be input using the mouse in an interactive fashion. On the screen the operations display, highlight (e.g. draw with thicker lines) and flash are supported. Every object provides methods to produce a textual description of itself and of its type which are used mostly for debugging.

## 3. Support for experimentation

### 3.1 Efficiency measurements

Once an algorithm is implemented our testbed enables the user to perform different kind of experiments. If two algorithms for the same problem have been implemented, timing studies can show which algorithm is better and when. Using the same testbed for both algorithms factors out common things (e.g. implementation of an AVL - tree) and makes the test result more reliable.

For example we have found that Fortune's sweep algorithm for the Voronoi diagram [F 87] is about three to four times faster than the divide and conquer version [PS 85]. On the contrary we saw that the probabilistic expected linear time algorithm for the closest pair [R 76] is slower by a factor of two than our O(n log n) sweep algorithm [HNS 88] for all sizes of sets we tested (up to $2*10^4$ points).

Timing studies are often revealing when it comes to assessing the necessity of balanced trees: we have seen implementations using an AVL - tree perform much worse than implementations using simple linear search, although there is always the possibility of a worst case scenario where the simple version will fail miserably. For example our closest pair sweep line algorithm will perform best on uniformly distributed random data for all n if the y-table is removed altogether (see [G 88] for an explanation why) while this modified version will need $\Omega(n^2)$ time if the point lie on a vertical line. Data structures can be replaced by each other either by changing the type of a variable (e.g. from 'sortedList' to 'avlTree') or creating the required object dynamically at run time.

## 3.2 Generating test data, including degenerate configurations

Since our system is interactive in nature we can easily construct degenerate configurations as test cases. These configurations can be saved and serve later as a test suite for different implementations. We also support the automatic construction of large, highly degenerate random configurations (e.g. n points on a vertical line).

## 3.3 Influence of arithmetic

If an implementation is coded in a sufficiently abstract manner (e.g. only use of the abstract object 'point'), the influence of the underlying arithmetic on the computed result can be studied without changing a line of code. The combination of the built-in zooming capability together with a low precision floating point arithmetic (e.g. base 10, 2 digits mantissa) often uncovers robustness problems and leads to new insights.

## 4. Algorithm animation

We have chosen a simple yet powerful approach to animation. First of all there is only one version of an implementation into which code pertaining to the animation is included via conditional compilation. This code checks whether animation for this particular algorithm is turned on and if yes it updates the currently visible state of the program in execution and waits for the user to let it proceed. Updating the visualization of the internal state is facilitated by the convention that all drawing on the screen is done using XOR which has the benefit that erasing is the same as drawing. Animating an algorithm now consists of choosing a sensible representation of the internal state (e.g. position of the sweep line, objects in the y-table, deactivated objects, etc.) and determining appropriate locations in the program where this information needs to be updated.

The screen dump in the appendix shows an animation of Fortune's plane sweep algorithm for computing the Voronoi diagram in progress. All of our non trivial algorithms can be animated, a feature that is regularly used in the classroom for teaching or similar demonstration purposes.

## 5. The user interface

Like any Macintosh application the geometry workbench makes heavy use of menus and windows. The user will find an *objects window* displaying the geometric object currently selected for input and an *info window* containing useful information, such as available memory, the coordinates of the cursor, time taken by the last operation and the type of the currently selected object.

Computation takes place in *geometry windows*: the user creates a new one, interactively enters geometric objects, selects some or all of them and chooses the desired operation from the *operations menu*. The *operations menu* shows only operations which are legal for the selected objects. Performing an operation creates a new geometry window containing the result of the operation which is already selected for a possible subsequent operation.

In the *animation menu* the user can select which algorithms to animate while the *arithmetic menu* governs which kind of arithmetic to use for newly created objects. Since arithmetic is bound to objects and not to operations, various kinds of conversion operations are available.

## 6. The object oriented approach: evaluation and experience

Implementing geometric algorithms using an object oriented language turned out to be successful. Especially the uniformity of interfaces (e.g. every object can display itself) and the ease of implementing abstract data types proved to be very useful. Memory management in the absence of a garbage collector turned out to be the biggest difficulty. We solved this problem by a strict discipline which asks, whenever an object is created, where and when it will be destroyed. Some of the more general lessons learned are:

• Move functionality to the top of the object hierarchy
Having a uniform interface reduces the number of methods a new implementor has to know. Generality is important for reusable components.

• Move instance variables down the object hierarchy
Objects without instance variables are more general, since they can be implemented in different ways,

• Use composition instead of inheritance
For example a line segment consists of two points, it is not derived from a single point by adding a second one. Inheritance should always describe the 'is_a' relationship.

## 7. Conclusion

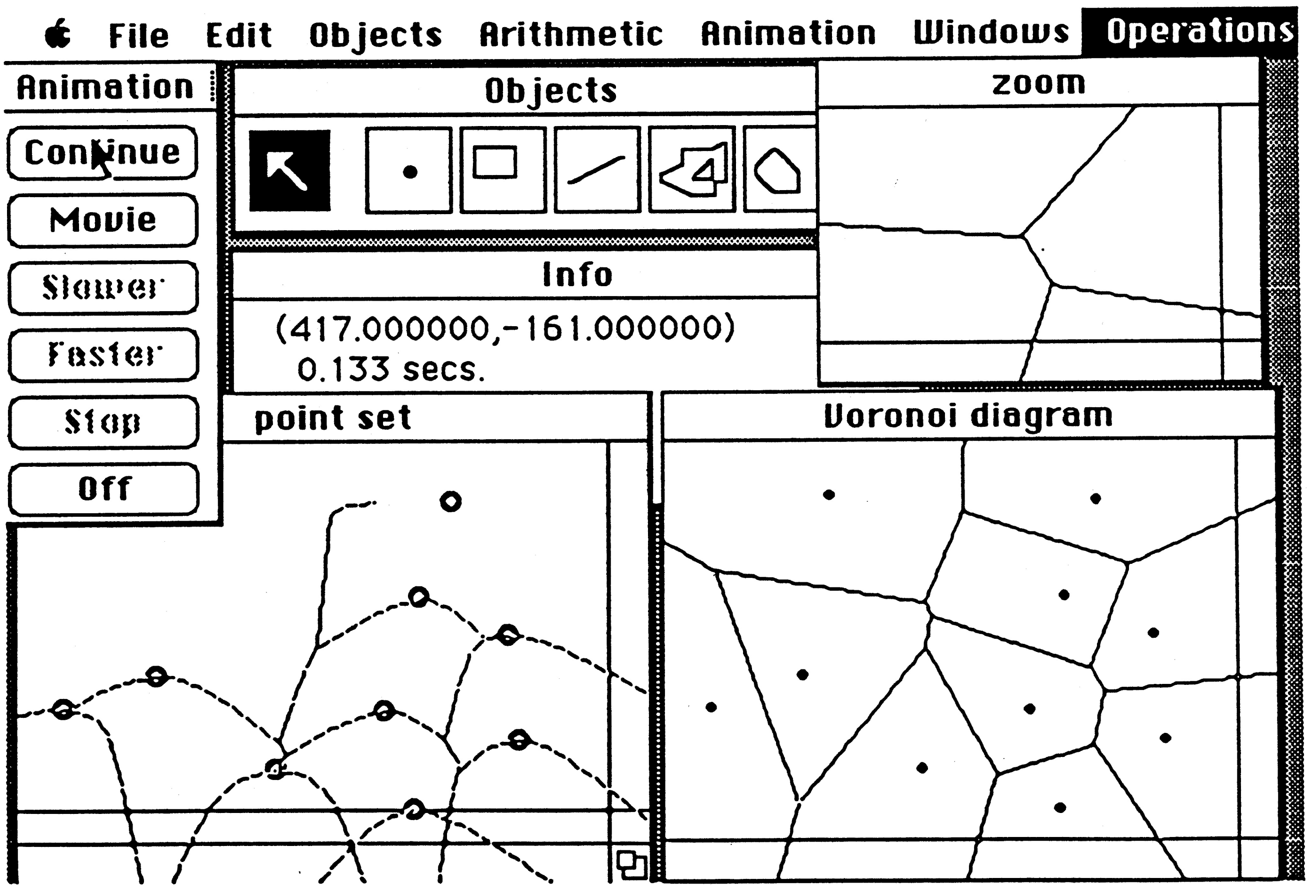
We have presented an interactive, object-oriented workbench for geometric computation. It has proven its usefulness in numerous implementations of geometric algorithms. The animation capability is used for demonstrating algorithms in courses about computational geometry here at ETH. Experiments have led to sometimes surprising insights about efficiency and robustness of geometric algorithms.

**Acknowledgements**
I am grateful to J. Nievergelt for valuable comments regarding style and presentation.

## References

[F 87] S. Fortune: A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2 (1987), p. 153 - 174.
[G 88] M. Golin, Probabilistic Analysis of a Closest Pair Algorithm, Princeton University Computer Science Dept. Technical Report TR-194-88, December 1988.
[HNS 88] K. Hinrichs, J. Nievergelt, P. Schorn: Plane-Sweep Solves the Closest Pair Problem Elegantly, Information Processing Letters 26 (11 Jan. 1988), p. 255 - 261.
[M 89] K. Mehlhorn, S. Näher: LEDA, A Library of Efficient Data Types and Algorithms, preliminary version, Universität des Saarlandes, 1989.
[PS 85] F. Preparata, M. I. Shamos, Computational Geometry: an Introduction, Springer - Verlag, 1985.
[R 76] M. Rabin: Probabilistic Algorithms, in: J. Traub, ed., Algorithms and Complexity (Academic Press, New York, 1976), p. 21 - 39.

## Appendix



```
(417.000000,-161.000000)
 0.133 secs.
```

Screen dump of a session while animating the computation of the Voronoi diagram of a point set