Report

eXperimental geometry Zurich
Software for geometric computation

Author(s):
Nievergelt, Jürg

Publication Date:
1991

Permanent Link:

https://doi.org/10.3929/ethz-a-000597280 →

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Theoretische Informatik

Jürg Nievergelt
Peter Schorn
Michele De Lorenzi
Christoph Ammann
Adrian Brüngger

eXperimental geometrY Zurich

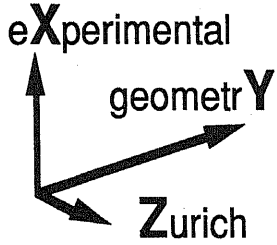
Software for Geometric Computation

July 1991

163

Authors´ address:

Institut für Theoretische Informatik
ETH-Zentrum
CH-8092 Zurich, Switzerland
e-mail: nievergelt@inf.ethz.ch



Software for Geometric Computation

Contents

Preface

An increasing number of computer applications in graphics and image processing, CAD, and geographic information systems, depend critically on a relatively small kernel of programs that perform geometric computations on spatial objects. Developing correct, robust, and efficient geometric software is a formidable endeavor that requires a lot of time and calls for specialized knowledge in disciplines that are traditionally as disjoint as computational geometry, numerical analysis, and software engineering.

Computational geometry has made impressive advances since the mid-seventies, producing a wealth of elegant and efficient algorithms. The goal of our project XYZ is to make some of these advances available to the practitioner in the form of a loosely coupled collection of carefully crafted software packages, in particular:

- The XYZ GeoBench, a programmer's workbench
- The XYZ Program Library,
  an open-ended collection of geometric algorithms
- The XYZ Grid File for managing spatial data on disk.

This report contains two papers that describe the current state of the project, and the design and implementation of the GeoBench; and a user's and programmer's manual.

The project XYZ has been underway for about 5 years, and this is our first progress report. We hope the result justifies the time invested into lengthy design deliberations such as:

- A survey of computational geometry led to the decision to base as many library programs as possible on sweep algorithms – the simplest and most efficient general class of geometric algorithms known today.

- The notorious difficulties of handling degenerate configurations correctly in the presence of round-off errors led to the decision to collect common geometric primitives and a parameterized arithmetic package in a workbench shared by all library programs.

- The desire to put the software to immediate good use in our courses on computational geometry led us to combine a programmer's workbench with an interactive algorithm animation package.

Software is never finished, and we anticipate continued development over the next few years. The GeoBench will have to accommodate a growing number of geometric data types. The Program Library now contains several dozen of the most basic geometric algorithms; we welcome well-conceived and well-tested contributions. The Grid File is under development, as are a number of applications of the XYZ Software.

I owe the realization of this challenging project to a wonderful team. Peter Schorn is responsible for the lion's share of the GeoBench and Library; Christoph Ammann, Michele De Lorenzi, and Adrian Brüngger for the extension into 3 dimensions and applications; Hans Hinterberger and Björn Beeli for the Grid File. Among a dozen students who cut their teeth on various aspects of this project we acknowledge particularly the contributions of Beat Fawer, Markus Furter, Peter Lippuner and Peter Skrotzky. This project has been supported over the years by the US National Science Foundation at the University of North Carolina and the Swiss National Foundation at ETH Zurich.

Bugs are found not only in software, but also in text, and here they are even harder to discover. We are grateful for any comments and corrections to any part of this report.

J. Nievergelt

The XYZ Software is written in THINK Pascal for Apple Macintosh computers. Source and object code are distributed for educational use only. Write to:

XYZ eXperimental geometrY Zurich
Institute for Theoretical Computer Science
Informatik, ETH
CH-8092 Zurich, Switzerland

Fax: +41-1-262-3973
E-mail: "author"@inf.ethz.ch

XYZ:
A project in experimental geometric computation

Jurg Nievergelt, Peter Schorn, Michele De Lorenzi
Christoph Ammann, Adrian Brüngger
Informatik, ETH, CH-8092 Zurich

Abstract

The project XYZ (eXperimental geometrY Zurich) aims to develop practically useful software for geometric computation, and to test it in a variety of applications. In pursuing these goals we emphasize the following points, each of which is described in one section of this paper:

1. Exploit recent progress in computational geometry through a systematic study to determine classes of algorithms that lend themselves to robust and practically efficient programs. Our program library contains many standard algorithms for 2-d problems, several for restricted 3-d problems, and a few for d-dimensional geometry.
2. Verify and evaluate algorithms experimentally: We study the problem of consistency in the presence of numerical errors, and emphasize robust programs that handle all degenerate cases; we often implement and compare different algorithms for the same problem.
3. Use state-of-the-art software engineering techniques in a workbench that supports the development of a library of production-quality programs: The XYZ GeoBench (written in Object Pascal for the Macintosh) is a loosely coupled collection of modules held together by a class hierarchy of geometric objects and common abstract data types.
4. The GeoBench is being used in education as a programming environment for rapid prototyping and visualization of geometric algorithms. Two other application projects are underway: Software for terrain modeling, and interfacing the GeoBench as a "geometry engine" to a spatial data base system.

Contents

0. Software for geometric computation, and the project XYZ

Geometry merged with automatic computation in the late fifties (during the early days of computer graphics, e.g. Sutherland's Sketchpad) to create the discipline of geometric computation. The appearance of computer-aided design (CAD) systems in the sixties greatly widened its range of applications, and gave increased importance to the nagging problem of correctly treating degenerate configurations; whereas a picture can tolerate an occasional error, an engineering design cannot. Computer scientists with a practical orientation working on graphics and CAD pushed the field forward. They developed many interesting algorithms, such as for visibility, and entire classes of related algorithms, such as scan-line algorithms. They collected experimental evidence for comparing the efficiency of different algorithms, and discovered the tantalizing and tough problems of how to compute reliably with degenerate configurations in the presence of round-off errors. Practitioners laid the foundation for a discipline of geometric computation.

In the mid seventies, led by Shamos' pioneering Ph.D. thesis, theoretically oriented researchers took over. They brought the finely honed tools of algorithm design and analysis to bear on geometric algorithms and created a new theoretical discipline of computational geometry. The well-defined, conceptually simple algorithmic problems of geometry, and the highly developed techniques of algorithm analysis, proved to be a perfect match. Computational geometry has now enjoyed a decade and a half of spectacular progress. It turned a field characterized by trial-and-error as recently as the seventies into a discipline where no programmer can work competently in ignorance of theory.

Today's research community in computational geometry still focuses most of its attention on theoretical problems. Research often stops short of investigating practical issues of implementation, so readers are left wondering whether a proposed optimal algorithm is useful in practice or not – a question that rarely has an easy answer. But this question must be answered by the computational geometry research community, and not be left to the applications programmer. It has become abundantly clear that the development of robust and efficient software for geometric computation calls for specialists with a broad range of experience that ranges from algorithm design and analysis to numerics and program optimization.

Even a prototype implementation of just one sophisticated geometric algorithm is an arduous endeavor if attempted without the right tools, such as: A library of abstract data types (e.g. dictionary, priority queue) and corresponding data structures, reliable geometric primitives (e.g. intersection of 2 line segments), and visualization aids. What the applications programmer needs, but cannot

find today, are reliable and efficient reusable software building blocks that perform the most common geometric operations. Geometric modelers, the core of CAD systems, do not address his problems – they are typically monoliths from which an applications programmer cannot extract any useful part for his own program.

We are aware of few projects whose main aim is to alleviate the problems faced by implementors of geometric algorithms. The most visible ones are the program library LEDA [MN 89] and the Workbench for Computational Geometry WOCG [ES 90]; both exhibit some similarities and some differences with our project XYZ. The goal common to all three projects is to develop practically useful software for geometric computation that is accessible to a wide range of applications programmers. Differences include: The systems software and programming language chosen as a basis of implementation, the scope of services and functions provided by the system, and range of algorithms and data structures included.

The project XYZ (eXperimental geometrY Zurich) presented here and in the companion paper [Sch 91a] aims at a broad range of goals all of which are essential for turning geometric computation from a specialty into a widely-practiced discipline:

1. Technology transfer: Exploit recent progress in computational geometry through a systematic study to determine classes of algorithms that lend themselves to robust and practically efficient programs.
2. Verify and evaluate algorithms experimentally: We study the problem of consistency in the presence of numerical errors, and emphasize robust programs that handle all degenerate cases; we implement and compare different algorithms for the same problem, and execute them using different number systems.
3. Use state-of-the-art software engineering techniques in a workbench that supports the development of a library of production-quality programs: The XYZ GeoBench (written in Object Pascal for the Macintosh) is a loosely coupled collection of modules held together by a class hierarchy of geometric objects and common abstract data types.
4. Test the software developed by exposing it to the rigors of a number of applications. The GeoBench is being used in education as a programming environment for rapid prototyping and visualization of geometric algorithms. Two other application projects are underway: Software for terrain modeling, and interfacing the GeoBench as a "geometry engine" to a spatial data base system.

This paper is an overview of the entire project, with examples of activities and results in each of these four categories, and particular emphasis on experimentation. [Sch 91a] describes the GeoBench in more detail.

1. Criteria for selection, types of algorithms, and the program library

There is no shortage of algorithms for inclusion in a program library for geometric computation. The problem is one of selection, whereby we emphasize the following criteria.

Robustness. A library routine must yield meaningful results for any geometric configuration, including highly degenerate ones. Unlike random data where degenerate configurations are rare, many practical applications generate a lot of highly degenerate configurations – degeneracy comes from the regularity that is inherent in man-made artifacts. The effort to guarantee correct results under all circumstances accounts for the lion's share of programmer time.

Practical efficiency. We strive for programs that are efficient in practice, that is, outperform competing programs on realistic input data. Example: An optimal algorithm can often be modified to run faster on a battery of test data, even though worst-case optimality is no longer guaranteed. This may occur, for example, by replacing a balanced tree implementation of a dictionary by an array implementation. The XYZ library leaves such choices of data structure to the user.

Standard problems of geometric computation. A program library is never comprehensive enough to solve most users' problems directly. We limit ourselves to basic problems that serve as building blocks for advanced geometric programs.

Well understood and elegant algorithms. We select algorithms that stand out by virtue of their elegant simplicity and can be implemented in a straightforward manner. Even if they are not asymptotically optimal, these tend to do better than their complicated counterparts with respect to robustness and practical efficiency. Some "optimal" algorithms are just too complicated for a reliable, robust implementation.

Start with 2-d geometry. The difficulties posed by 2-d problems must be solved completely before venturing into higher dimensions, so we have concentrated on accumulating a sufficient number of representative 2-d algorithms. We approach 3-d geometry by first studying restricted 3-d problems using layered objects (see also section 4). Just to show that the structure of the GeoBench is not restricted to low-dimensional space we have implemented an algorithm that computes the minimal area disk enclosing a set of points in d-space.

Types of algorithms

As a guide to selection, and in order to benefit from any similarities that might be found, we attempt to classify the multitude of published algorithms. The majority we have investigated fall into one of the following categories:

- Incremental algorithms
  - sweeps (mostly in the plane, occasionally through space)
  - in random order
- Boundary traversal
- Recursive data partitioning, i.e. divide and conquer.

These classes of algorithms differ significantly with respect to their data access patterns. Whether this access pattern is irregular, or simple and predictable, has an effect on efficiency, in particular if data is processed off disk (see also section 4). Some observations:

- Randomized incremental algorithms play an important role in theory because the assumption of randomly ordered data is favorable for average case analysis. In practice they do not appear to be superior to sweeps.
- Sweeps, on the other hand, require (lexicographically) sorted data; this orderly access pattern leads naturally to efficient implementations with simple data structures.
- Algorithms that follow boundaries (e.g. of a convex polygon) exhibit a spatial locality principle. They can be implemented efficiently in central memory using list structures, but are less efficient for data stored on disk.
- Algorithms that partition their data in recursively generate the most irregular access patterns. In section 2 we demonstrate experimentally that divide-and-conquer algorithms are typically less efficient than their plane sweep counterparts.

In conclusion, we favor plane-sweep algorithms as a simple and efficient general purpose skeleton for most 2-d geometric problems.

The 2-d algorithms currently in the library include:

- Convex hull (Graham's scan, divide and conquer)
- Diameter and intersection of convex polygons
- Tangents common to two convex polygons
- Boolean operations (union, intersection, difference) on polygons
- Contour of a set of rectangles
- Winding number
- Intersection of line segments (sweep line for the first intersection and for reporting all intersections, sweep line for the special case of horizontal and vertical line segments)
- Closest pair (sweep line, simplified sweep line, probabilistic)
- All nearest neighbors (sweep line, simplified sweep line, extraction from Voronoi diagram)

- All nearest neighbors in a sector
- Voronoi diagram (sweep line, divide and conquer)
- Euclidean minimum spanning tree (EMST)
- Traveling salesman heuristics (nearest neighbor, EMST, convex hull, tour optimizer)

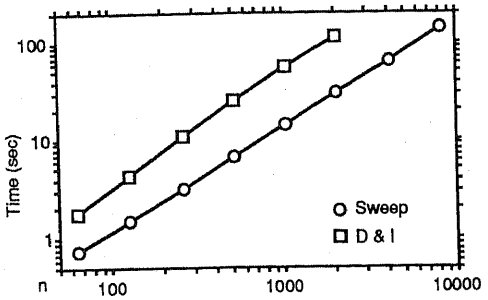
The presence of distinct algorithms for solving the same problem reflects our concern for experimental assessment and comparison, as discussed in the next section.

2. Verify and evaluate algorithms experimentally

Efficiency is at the heart of algorithm design, but it is neither easy to define nor measure. Algorithm analysis typically interprets the term 'efficiency' as asymptotic complexity, in the worst case or as an average over some data space. Although useful for mathematical analysis, this approach neglects many practical considerations such as: Simplicity and robustness of the implementation, constant factors, and whether data encountered in practice is well modeled by the randomness assumptions that go into the theory (it is usually not). For practical applications, the most promising algorithms must be implemented in a variety of ways and evaluated experimentally. We describe some of the experiments and results obtained. Time is measured in seconds as executed on a Macintosh IIfx.

Efficiency measurements. The GeoBench facilitates experimental verification and evaluation, e.g. by timing and displaying on demand each operation executed. Executing all programs on a common platform has the advantage that geometric primitives are implemented the same way in both programs, making time measurements more meaningful. For example, most algorithms for computing the convex hull are likely to use some means for detecting a 'left turn'. There are many ways to implement this primitive, and they result in different running times, but the peculiarities of this implementation is an issue of program optimization more than of algorithm design. The GeoBench provides about 20 geometric primitives such as 'left turn', and the library programs all use them to the greatest extent possible.

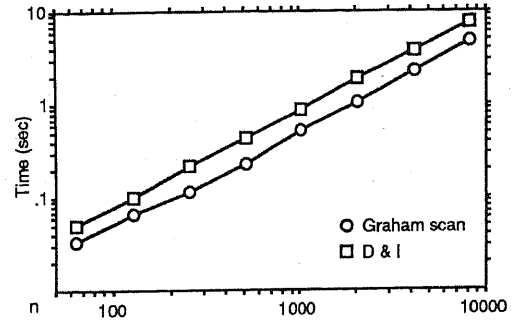
Example: Sweeps and other incremental algorithms vs. divide-and-conquer.
Fortune's sweep [Fo 87] for computing the Voronoi diagram is much easier to implement than the standard divide-and-conquer algorithm (e.g. [PS 85]). It uses memory more efficiently, enabling us to compute larger Voronoi diagrams, and is about three times faster, as the following measurements on random point sets show. This and other experiments have convinced us that the divide-and-conquer Voronoi algorithm is not competitive and does not belong in a program library.



n	Sweep	D & I
64	.733	1.767
128	1.533	4.317
256	3.2	10.85
512	6.733	25.383
1024	14.35	54.933
2048	30.283	114.133
4096	63.85	
8192	142.183	

Figure 1: Efficiency comparison of two Voronoi diagram algorithms

As a second example we consider the computation of the convex hull of a set of points in the plane. The Graham scan [Gr 7], an incremental algorithm, is about twice as fast as Preparata and Hong's [PH 77] divide-and-conquer. The divide-and-conquer code is 60% longer than the Graham scan implementation, which reflects the fact that degenerate cases are more difficult to handle.



n	Scan	D & I
64	.033	.05
128	.067	.1
256	.117	.217
512	.233	.433
1024	.533	.883
2048	1.067	1.867
4096	2.25	3.817
8192	4.8	7.7

Figure 2: Efficiency comparison of two convex hull algorithms

We conclude that incremental algorithms, in particular sweeps, are usually superior to divide and conquer algorithms. One reason is that most divide and conquer algorithms compute much information that is not part of the final solution, whereas incremental algorithms tend to compute only information that is part of the final solution. A second problem: Divide-and-conquer algorithms tend to use more memory, especially during the last merge where two large objects are combined into the final solution, an operation that can rarely be done in place.

Parametrized arithmetic. Executing a program repeatedly on the same data using different arithmetic is an effective experiment for assessing the robustness of an implementation. The GeoBench contains a floating point arithmetic package where the user specifies, at run time, radix and precision. Low precision, say 2 decimal digits in the mantissa, brings any numerical

problems quickly to the observer's attention. The program source code is independent of the choice of arithmetic.

Example: Robustness of a sweep algorithm
The figure below shows the results of executing the all-nearest-neighbors-to-the-left-sweep of [HNS 90] in a low-precision floating point arithmetic (radix 10, two-digit mantissa, equivalent to about 6 bits) and in a 32-bit floating point system. The rightmost window shows the two results superposed using XOR graphics, thus canceling the common part and showing where they differ. The comparison shows that when low precision arithmetic identifies a wrong neighbor, the distances are nevertheless close.

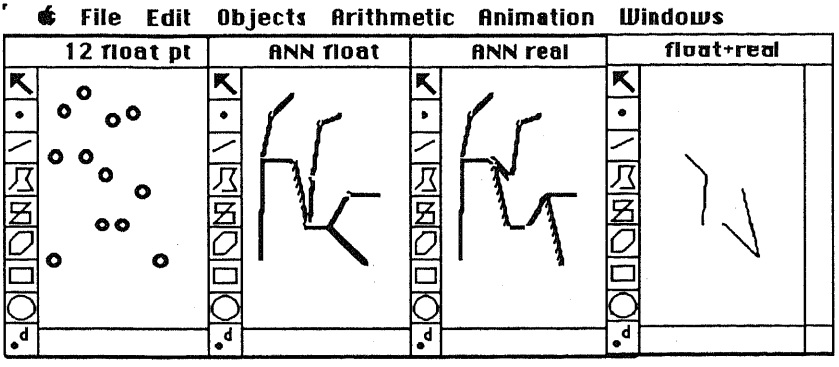


Figure 3: The effect of using different arithmetic systems

Different implementations of abstract data types, and instrumentation

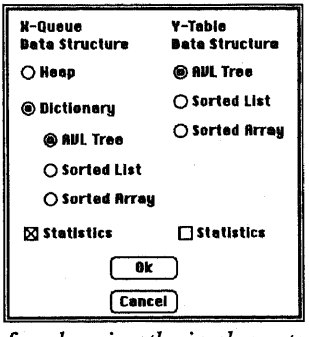
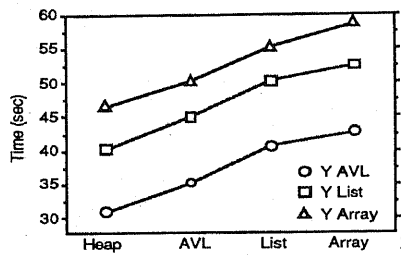


Figure 4: Dialog box for choosing the implementation of abstract data structures

As we may postpone the choice of arithmetic until run time, we may also explore the influence of different implementations of the same abstract data type on the performance of a given algorithm. The dialog box in figure 4 shows how a sweep can be tuned to use various data structures for a dictionary and priority queue. When the user selects 'Statistics', without changing a single line of source code, GeoBench records the maximal number of elements in the data structures and the number of insertions and deletions performed.

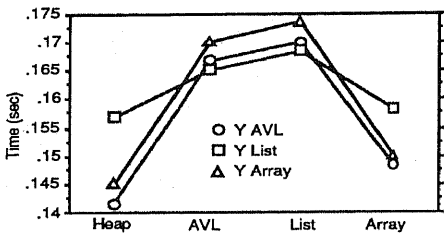
Example: Data structures in the all-nearest-neighbors sweep
Figures 5 and 6 show the influence of different implementations of the X-priority-queue and the Y-dictionary in the all-nearest-neighbors sweep [HNS 90]. The experiment involved sets of 8192 and 64 random points uniformly distributed in a square. For 8192 points, the choice of X-queue and Y-table implementations are independent: A heap is best for the X-queue, a balanced tree for the Y-table. It is perhaps surprising that the best and worst time differ by only a factor of 2. For 64 points we have a more complex picture, but the combination heap + balanced tree is still best.



X \ Y	AVL	List	Array
Heap	30.917	40.317	46.617
AVL	35.200	45.000	50.183
List	40.633	50.250	55.267
Array	42.833	52.533	58.650

	Max	Insert	Delete
X	53	13587	13587
Y	105	8192	8108

Figure 5: Efficiency comparison of different data structures (n = 8192)



X \ Y	AVL	List	Array
Heap	.142	.157	.145
AVL	.167	.165	.17
List	.17	.168	.173
Array	.148	.158	.15

	Max	Insert	Delete
X	6	51	51
Y	16	64	52

Figure 6: Efficiency comparison of different data structures (n = 64)

Test data generation. GeoBench generates two main types of test data, random and degenerate.

Random configurations: Random configurations consist of basic geometric data types such as [d-]dimensional point, line segment, polyline, polygon, convex polygon, circle and rectangle. The user specifies the number of objects

(for points, line segments, circles and rectangles) or the number of vertices (for poly lines, polygons and convex polygons). d-dimensional points can be uniformly distributed in a d-dimensional square or in a d-dimensional circle.

Degenerate configurations: Degenerate configurations include: More than two collinear points, more than three points on a circle, more than two line segments with a common intersection point, horizontal and vertical segments, point sets that lie on a rectangular grid, point sets with equal x- or y-coordinate, coinciding objects. With integer arithmetic most of these degeneracies are exact, that is, the corresponding test polynomials evaluate to zero. In the case of floating point arithmetic the test polynomials either evaluate to zero or to a value close to zero, corresponding to nearly degenerate configurations. Both cases are useful for debugging and testing.

Example: Approximate generalized Voronoi diagrams
GeoBench provides the facility to cover objects consisting of straight lines (e.g. line segments, polygons) or circles with evenly spaced points. This feature can be used in two ways: To create degenerate configurations as explained above, or to approximate linear objects by a sequence of points, as the following example illustrates.
Figure 7 shows, from left to right, line segments, their point covers, and a Voronoi diagram of these points. By omitting the many parallel lines in this diagram we obtain an approximate generalized Voronoi diagram of the original set of line segments.

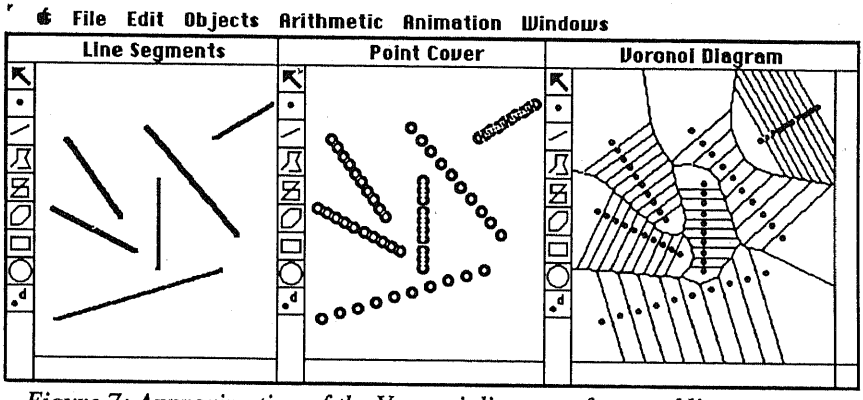


Figure 7: Approximation of the Voronoi diagram of a set of line segments

3. The XYZ software packages

The main goal of the project XYZ is to make available to the practitioner a loosely coupled collection of carefully crafted software packages, in particular:

- The XYZ GeoBench, a programmer's workbench
- The XYZ Program Library,
  an open-ended collection of geometric algorithms
- The XYZ Grid File, a package for managing spatial data on disk.

The relationship among these three packages and the class hierarchy that defines all common object types is shown in the figure below, where an arrow indicates the relationship "is_based_on".

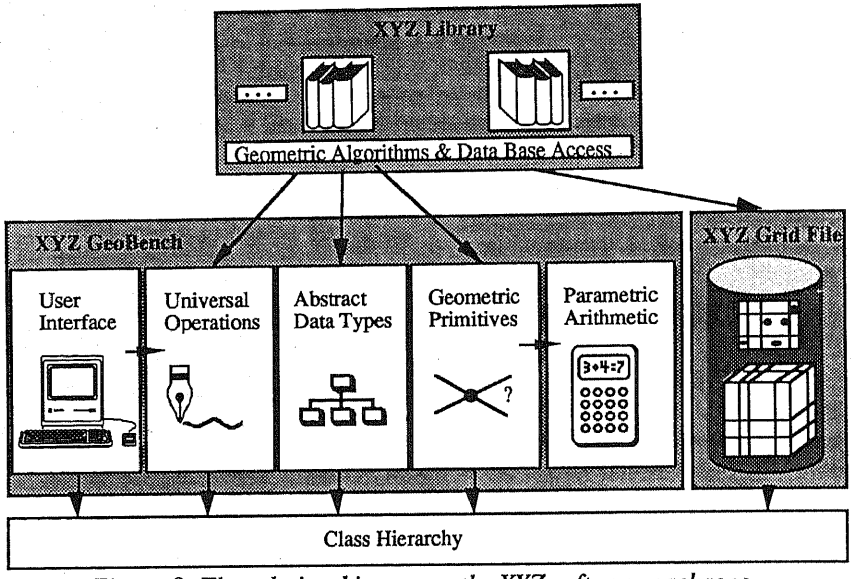


Figure 8: The relationship among the XYZ software packages

The GeoBench ([Sch 91a] and [Sch 91b]) is the programmer's workbench and run-time environment that holds all the library programs and the disk storage grid file package together, keeping them data-compatible. Its components serve the following functions. The user interface manages windows for interactive data generation and algorithm animation, as illustrated in several pictures above. A collection of the most important geometric primitives and abstract data types, and various implementations thereof, saves the programmer a lot of time-consuming detail work. A parametrized arithmetic package supports experimental program validation by providing floating point arithmetic of varying precision and base.

The library is an open-ended collection of geometric algorithms that work in central memory, and of disk access procedures that pack geometric objects into grid files and perform queries on them. The grid file disk management package provides multidimensional data access for an arbitrary number of dimensions, each of which is individually measured by one of the types integer, long-integer, or real.

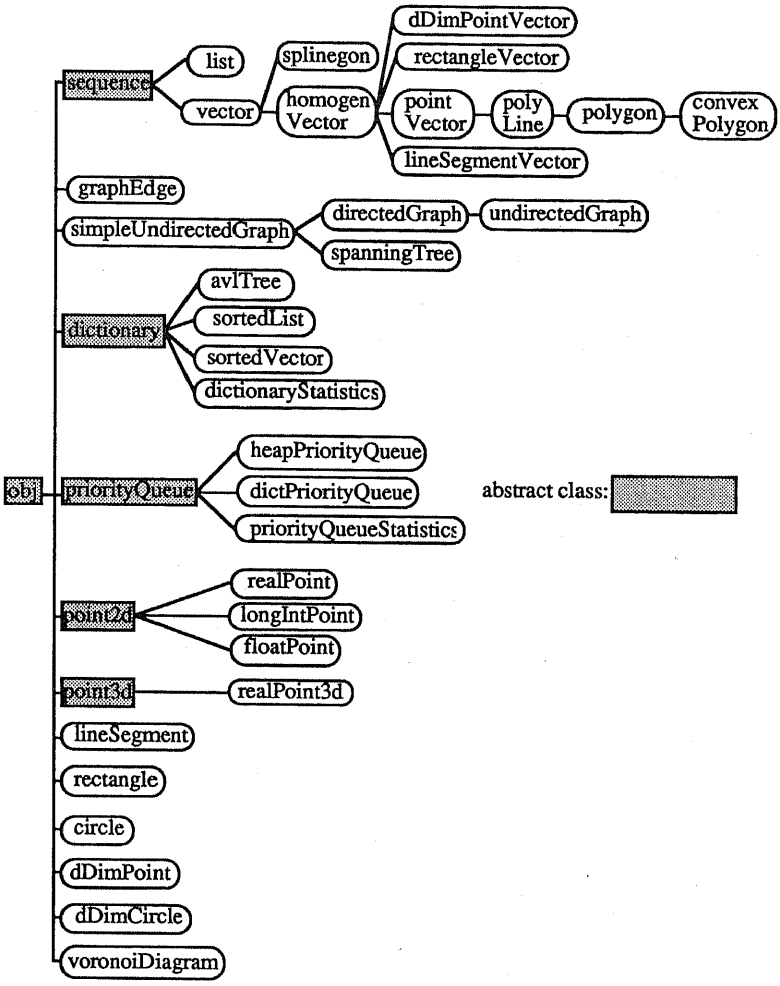


Figure 9: Class hierarchy of the XYZ GeoBench

The backbone of all this software is a class hierarchy that defines the common data types and serves as interface between all software components. The figure above shows the tree of the most important classes currently in the GeoBench

and describes the "is_a" relationships among the classes. All algorithms are methods associated with the class on which they operate. For example, the Voronoi algorithm is a method in the class 'pointVector' that yields an object of type 'voronoiDiagram'. The principle of inheritance insures that all methods for a given class are also available for their descendants. Thus the 'method' Voronoi diagram is also applicable to 'polyLine', 'polygon', and 'convexPolygon', for each of which it may have its own implementation.

4. Uses and applications

During its entire period of development, the XYZ GeoBench and program library have served as a useful demonstration package and tool for algorithm animation in various courses on algorithms and data structures. Now that the GeoBench is essentially complete, we are also using it as a programming environment for term projects and in a course on computational geometry. Work on realistic applications started only recently; we briefly describe two ongoing projects.

Layered objects and triangulated surfaces for terrain modeling.

Layered objects are an attempt to reconcile two conflicting facts of geometric computation: On the one hand, it is well known that 2-dimensional geometric algorithms are usually a lot simpler and more efficient (often O(n log n)) than their 3-d counterparts (usually $O(n^2)$ or higher); on the other hand, most applications call for 3-d geometry. Fortunately, the objects to be processed in any one application are often subject to restrictions that make it profitable to define various restricted classes of 3-d objects with special properties that yield to simpler algorithms than unrestricted 3-d objects.

Layered objects are a striking example of the benefits and limitations of this approach: A 3-d object is represented or approximated as a vector of layers (parallel slices orthogonal to the z-axis), where each layer is defined by its thickness and a 2-d contour. Important classes of real-world objects are naturally modeled as layered objects, such as terrain (using contour lines), certain semiconductor devices (perhaps using projections and cross-sections), and, in general, objects whose shape is defined by one or more functions of the type $z = f(x, y)$. Layered models are particularly appropriate in CAD systems for stereolithography, a new manufacturing technology that "grows" 3-d objects one layer at a time. Each layer is defined by tracing its outline with a laser and marking the part that is to remain; the latter hardens on top of the preceding layer when exposed to light.

Layered objects are particularly effective, as is the case with maps, when the number L of layers needed to achieve a desired accuracy is small compared to the complexity n of the 2-d figures in each layer. Operations on layered objects typically trigger a sequence of calls to 2-d algorithms, one for each of L layers, and thus work in time O(L n log n). This often compares favorably to the complexity $O(N^2)$ of a quadratic algorithm on a comparable unrestricted 3-d object of complexity N, where a fair comparison suggests $n < N < L\,n$.

Although layered objects greatly simplify the problems of processing 3-d objects, they do not eliminate the necessity to consider the 3rd dimension explicitly. When computing the visible surface, for example, we wish to look at a layered object from an arbitrarily chosen point of view (not necessarily at infinity). This results in clipping a stack of layers against a pyramid in arbitrary position, a true 3-d problem. A second example is the problem of correct treatment of all degenerate configurations: Layered objects exhibit new types of degeneracies beyond those that occur in 2-d.

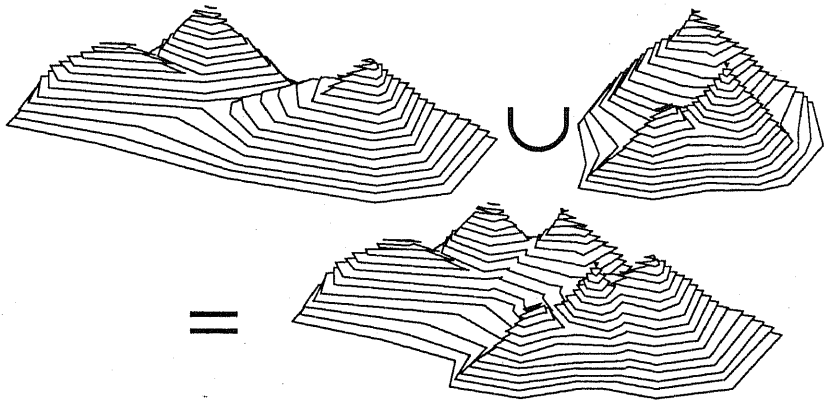


Figure 10: Union of two layered objects

In addition to visibility, we have implemented boolean or set-theoretic operations, as shown in Figure 10.

But the stair-case shape of layered objects make them unsuitable models for the graphic representation of smooth objects, so we are also introducing triangulated surfaces as an alternative 3-d model. As terrain modeling requires seemingly realistic images, we experiment with the automatic generation of synthetic images, with grey-levels or color-shading (a graphics problem rather than a geometric problem). As an example, the following terrain images are generated automatically from geographic (x, y, z)-data that represents Switzerland on a 250m × 250m grid:

Figure 11: Grey level map of the southern slope of the Alps (Ticino and Lombardia) automatically generated from geometric data

The interplay between geometric computation and spatial data bases

Practically all algorithms of computational geometry, and the corresponding complexity results, are based on the "random-access-machine model of computation". This model provides realistic performance predictions as long as all the data fits in central memory, where access time to any data element is approximately constant. When large data configurations must be processed off disk, on the other hand, disk access often becomes the bottleneck. The efficiency of computation is then determined primarily by the following issues: 1) How data is stored on disk, and 2) in what order algorithms access data, and, of course, the interplay between 1) and 2).

1) How data is stored on disk. Along with the development of computational geometry there has been growing interest in spatial data bases, for which efficient data access is perhaps the major issue. [Ni 89] and [Wi 91] are general surveys of spatial data structures. The XYZ GeoBench interfaces to a Grid File [NHS 84] as a general-purpose multidimensional data structure for storing geometric objects on disk. [NH 87] describes how a broad range of proximity queries are answered efficiently on large collections of objects stored as points in parameter space.

2) In what order algorithms access data. In section 2 we classified geometric algorithms according to their data access pattern as follows:
- Plane sweep or space sweep: data is accessed in an order (e.g. of increasing x) known a priori.
- Boundary traversal: data access follows a spatial locality principle, but is only known at run time.
- Recursive partitioning: data access is usually random.

The advantages of a predictable data access pattern, and the disadvantages of random access, are even more pronounced when the data is processed off disk. Thus the preponderance of sweep algorithms in the XYZ library, and the choice of layered objects for 3-d modeling, are a consequence of our aim at applications such as terrain modeling that require efficient processing of very large data volumes.

A geometry engine as front end to a spatial data base

The tight coupling between the XYZ GeoBench and its Grid File allows us to focus on algorithms that interact in a particularly efficient way with the grid file data structure. The typical situation in applications that process large volumes of spatial data, however, is different. Usually, the user's data is organized and stored in some commercial spatial data management system that provides a few types of spatial queries only – clearly a spatial data base system cannot anticipate the access patterns of all algorithms its users might run on its data. Thus it is an open question in spatial data base research as to how efficiently geometric algorithms interface with typical built-in queries. In order to explore this issue, we started a joint project with the database research group at ETH (H.-J. Schek) where the GeoBench is used as a front end to a spatial data base system built on DASDBS [DSW 90]. The first experiments aim to use the GeoBench as a powerful user interface for retrieving data from the data base, perform geometric operations on it, and finally store (modified) objects back in the database. In a second phase we aim at a tighter coupling based on the extension capabilities of DASDBS, i.e. the ability to manage arbitrary geometric objects provided a certain set of (geometric) operations is supplied [DSW 90].

In conclusion, we have presented an overview of a research project that attacks the practical problems of software development for geometric computation on a broad front. The XYZ GeoBench in particular has proven its usefulness in numerous implementations of geometric algorithms. Its animation capability is regularly used for demonstrating algorithms in courses at ETH. Experiments have led to some surprising insights about efficiency and robustness of well-known algorithms. Other projects have just started, in particular the interaction between a "geometry engine" and spatial data bases.

Acknowledgments

A number of students have contributed to the development of the XYZ software, in particular Beat Fawer, Markus Furter, Peter Lippuner, and Peter Skrotzky. The XYZ Grid File is based on a grid file package written by Hans Hinterberger and adapted to the GeoBench by Björn Beeli.

References

[DSW 90] G. Dröge, H.-J. Schek, A. Wolf: Erweiterbarkeit in DASDBS, Informatik Forschung und Entwicklung 5, 4 (special issue on Nicht-Standard-Datensysteme), pp. 162-176, 1990.

[ES 90] P. Epstein, A. Knight, J. May, T. Nguyen, J. Sack: A workbench for Computational Geometry (WOCG), Tech. report, Carleton University, 1990.

[Fo 87] S. Fortune: A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2, pp. 153-174, 1987.

[Gr 72] R. Graham: An efficient algorithm for determining the convex hull of a finite planar set, Information Processing Letters 1, pp. 132-133, 1972.

[HNS 88] K. Hinrichs, J. Nievergelt, P. Schorn: Plane-Sweep Solves the Closest Pair Problem Elegantly, Information Processing Letters 26, pp. 255-261, 11 Jan. 1988.

[HNS 90] K. Hinrichs, J. Nievergelt, P. Schorn: An all-round sweep algorithm for 2-dimensional nearest-neighbor problems, submitted.

[MN 89] K. Mehlhorn, S. Näher: LEDA, A Library of Efficient Data Types and Algorithms, preliminary version, Universität des Saarlandes, 1989.

[NH 87] J. Nievergelt, K.H. Hinrichs: Storage and access structures for geometric data bases. Proc. Kyoto 85 Intern. Conf. on Foundations of Data Structures (eds. Ghosh et al.), Plenum Press, pp. 441-455, NY 1987.

[NHS 84] J. Nievergelt, H. Hinterberger, K. Sevcik: The Grid File: An adaptable, symmetric multikey file structure. ACM Trans. on Database Systems, Vol. 9, No. 1, pp. 35-45, 1984.

[Ni 89] J. Nievergelt, 7 ± 2 criteria for assessing and comparing spatial data structures, in A. Buchman et al. eds.: Design and Implementation of Large Spatial Databases, invited paper at 1st Symp. SSD'89, UC Santa Barbara, Lecture Notes CS 409, Springer, pp. 3-27, 1990.

[PH 77] F. Preparata, S. Hong: Convex hulls of finite sets of points in two and three dimensions, Comm. ACM 2 (20), pp. 87-93, Feb. 1977.

[PS 85] F. Preparata, M. I. Shamos, Computational Geometry: an Introduction, Springer, 1985.

[Sch 91a] P. Schorn: The XYZ GeoBench: A programming environment for geometric algorithms, submitted.

[Sch 91b] P. Schorn: Robust Algorithms in a Program Library for Geometric Computation, ETH PhD Dissertation 9519, to appear 1991.

[We 90] E. Welzl: A fast randomized algorithm for computing the minimal area disk enclosing a set of points in d-space, presentation at the Workshop on Computational Geometry, Dagstuhl, Oct 1990.

[Wi 91] P. Widmayer: Datenstrukturen für Geodatenbanken, Tech. Report, Univ. of Freiburg, 1991.

The XYZ GeoBench: A programming environment for geometric algorithms

Peter Schorn, schorn@inf.ethz.ch
Informatik, ETH, CH-8092 Zurich

Abstract: The XYZ GeoBench (eXperimental geometrY Zurich) provides a comprehensive infrastructure for rapid prototyping of geometric algorithms and the implementation of production-quality library programs. This paper introduces the components of this programming environment and gives some implementation details. The system is implemented in an object oriented extension of Pascal on the Apple Macintosh computer. We report our experience with object oriented programming in the context of geometric algorithms and give some advice on building a programming environment for geometric computation.

Contents

1 The role of a programmer's workbench in a project for experimental geometric computation

The XYZ project [NSABD 91] has the goal to produce production-quality software for geometric computation. This paper describes the design and implementation of the XYZ GeoBench, the major development tool.

The following figure shows the GeoBench and its components, as well as its relation to the other software packages of our system.

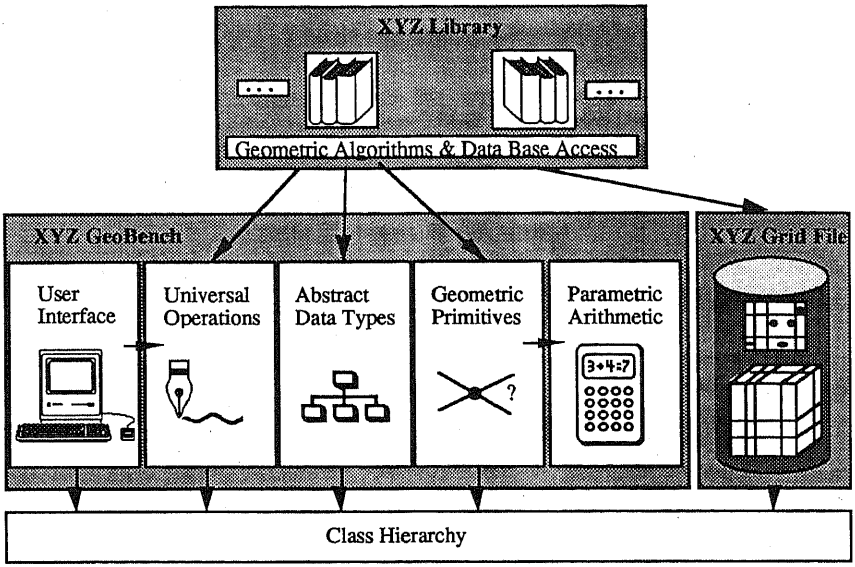


Figure 1: System architecture

The XYZ GeoBench provides the following functions:

1) A programming environment and tool kit to facilitate rapid prototyping of geometric programs
2) A run-time environment for software testing and the experimental validation of geometric algorithms
3) A tool for education, featuring algorithm animation for demonstration purposes

The XYZ GeoBench is written in Object Pascal for the Macintosh. Currently the whole system consists of more than 70 modules with a total source code size of approximately 1000 KB. The author is responsible for the design and architecture of the system and has written most of the code, although other people have contributed substantially.

The paper is organized as follows. Section 2 describes the components in more detail whereas section 3 gives a random sample of some interesting implementation details. Section 4 relates our experience with object oriented programming in the area of geometric algorithms. We assume that the reader is somewhat familiar with object oriented programming techniques in general (e.g. see [M 87] for an introduction).

2 Architecture of the XYZ GeoBench and its components

In the following sections we examine the components in turn and give important details that are useful to implementors of similar systems. We claim that the issues addressed by these components must be solved by any system that enables the user to perform research in experimental geometric computation.

2.1 User interface and algorithm animation

GeoBench uses the Macintosh conventions. The user finds 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. Selecting an object for input can either be done by using the palette attached to each geometry window or by using the *objects menu* that also allows the creation of a random instance of any of the currently directly accessible objects point, line segment, circle, rectangle, poly line, polygon, convex polygon and d-dimensional point.

Computation takes place in *geometry windows*: The user creates a new one, enters geometric objects, selects 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 which contains the result of the operation already selected for a subsequent operation. For example one could enter some points using the mouse, select them, compute the Voronoi diagram, compute a Euclidean minimum spanning tree using the Voronoi diagram and finally compute a traveling salesman tour from the spanning tree.

The geometric transformation operations translate, rotate, scale and reflect can be found in the *edit menu* which also provides commands for changing the viewing transformation: Zoom in and zoom out.

In the *animation menu* the user selects 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 in the operations menu.

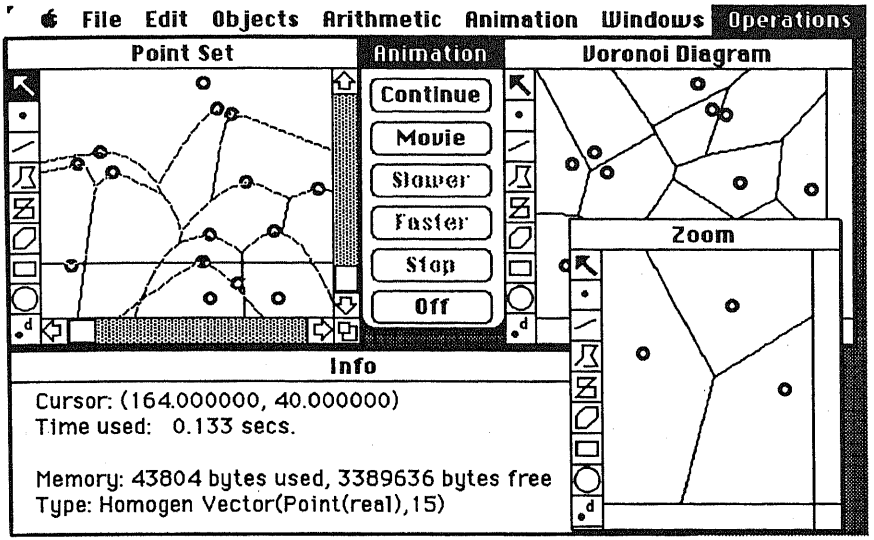


Figure 2: Screendump of the XYZ GeoBench while animating the computation of a Voronoi diagram using Fortune's sweep [F 87]

Algorithm animation

Algorithm animation is used for demonstrating and debugging. We have chosen a simple yet powerful approach to animation. 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. If yes, it updates the currently visible state of the algorithm and waits for the user to let it proceed. In situations where speed is crucial, we avoid the slight overhead of repeatedly checking whether animation is turned on by setting the appropriate compile time variable to 'false'. Animation code has the following general structure.

```
...
{ Geometric algorithm changing internal state. }
{$IFC myAlgAnim }
  if animationFlag[myAlgAnim] then
    { Update graphical state information, usually draw some objects. }
    waitForClick(animationFlag[myAlgAnim]);
    { Update graphical state information, usually erase some objects. }
  end;
{$ENDC }
...
```

The procedure 'waitForClick' provides an interface between the user and the algorithm currently animated. It supports single step mode and a movie mode

with user selectable speed (see the 'Animation' dialog box in the previous screen dump). Updating the visualization of the internal state is facilitated by the convention that all drawing on the screen is done using XOR graphics which has the benefit that erasing is the same as drawing. Animating an algorithm consists of choosing a 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. Algorithm animation is implemented for all non trivial geometric algorithms.

2.2 Geometric primitives

The type 'point2d' (section 2.3) is the basic building block of our geometry. Therefore all geometric primitives are methods in this class and are usually implemented in three different ways taking advantage of the respective arithmetic. The overwhelming part of our library is based on the following primitives.

```
function whichSide(p, q, r: point2d): (-1, 0, +1);
  (* Determines on which side of a directed line segment given by two points a third
     point lies. *)

function crossProduct (p, q, r, s: point2d): real;
  (* Computes the cross product (p - q) × (s - r). *)

function distance(p, q: point2d): real;
  (* Computes the Euclidean distance between p and q. *)

function squaredDistance (p, q: point2d): real;
  (* (p_x - q_x)^2 + (p_y - q_y)^2 *)

function squaredDx (p, q: point2d): real;
  (* (p_x - q_x)^2 *)

function squaredDy (p: point2d): real;
  (* (p_y - q_y)^2 *)

function circleCenter (p, q, r: point2d): point2d;
  (* Computes the center of the circle through p, q and r. *)

function intersectLineSegment
     (p, q, r, s: point2d): [point2d, point2d, boolean];
  (* Tests whether the segments pq and rs intersect and computes the intersection
     segment. *)

function xPlusY(p: point2d): real;
  (* p_x + p_y *)

function xMinusY(p: point2d): real;
  (* p_x - p_y *)
```

In total we need about 15 primitives for all geometric algorithms implemented so far and 'whichSide' turned out to be the most common one.

2.3 Interchangeable arithmetic and parameterized floating point arithmetic

The choice of arithmetic may have a significant impact on how an implementation of an algorithm behaves in the case of degenerate or nearly degenerate configurations. Since we want to experiment with different arithmetics, easily interchangeable arithmetic is needed. This means that in the best case no line of code of an implementation must be modified in order to try out a different model of arithmetic. Since points are a basic building block of geometry, we achieve this goal by defining an abstract 'point2d' class which has no instance variables for the coordinates but specifies an interface with access procedures to the coordinates and various geometric primitives (see also the previous section 2.2).

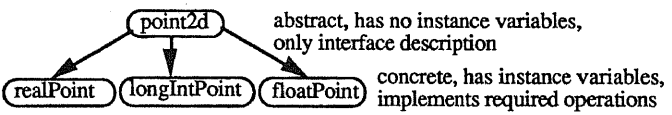


Figure 3: The abstract class 'point2d' and its descendants

From this abstract 'point2d' we derive concrete point objects having instance variables and implementing the geometric primitives in their respective arithmetic. Algorithms using only the functions and procedures specified by the abstract type 'point2d' can be run in any of the three kinds of arithmetic currently supported.

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 (including odd bases) and precision which is used for the coordinates of the object 'floatPoint'. The idea here is of course not to simulate high precision floating point arithmetic but on the contrary low precision arithmetic in order to make rounding errors and other problems of floating point arithmetic more pronounced.

2.4 Abstract data types

Efficient geometric algorithms organize data in lists, dictionaries, queues, etc. We describe design decisions and implementation details for some abstract data types.

Sequence: The abstract class 'sequence', realizing collections of arbitrary objects, is implemented as a linked list structure and as a (dynamic) array. The list implementation is useful when no a priori bound on the number of

elements is known and sequential processing of the elements is feasible. The array implementation provides a much greater flexibility and functionality.

Dictionary: In a dictionary based on the reference concept (see also section 3.1) 'find' is the only one key-based operation. 'find' takes an object *o* to be found and a key comparison function and returns, if possible, a reference to an object with the same key value. If none is found, 'find' returns a direction *d* and a reference to an object *p* such that *o* can be inserted as *p*'s direct neighbor in direction *d*. Performing the insertion yields a reference for *o*. All other operations like 'delete' or 'swap' require references as arguments.

This separation between key-based operations and operations changing the data structure has two advantages: 1) Different objects with identical key values are easier to handle (see the example in 3.1) and 2) operations just changing the data structure can be implemented more efficiently.

If a dictionary is implemented as an AVL-tree, the delete operation takes constant time whenever rebalancing is not needed. In an ordinary implementation we would have to find the object first before we can delete it, making delete an O(log *n*) operation most of the time. Other implementations realize a dictionary as a sorted list and as a sorted vector.

Priority queue: The abstract data type priority queue is implemented 1) based on a heap and 2) based on a dictionary. We describe in section 3.2 how to implement a heap with an efficient delete operation using the reference concept. In our experience, a heap is sufficient most of the time and a general 'find' operation on a priority queue is rarely needed.

2.6 Universal operations

Object oriented programming allows the specification of a common interface that is understood by all objects in the system by placing this interface at the root of the object hierarchy. All objects in our system are descendants of the root object 'obj' and therefore share a large set of common methods that we call 'universal operations'. A typical example is the ability for an object to display itself on the screen. In the following we discuss the kinds of universal operations provided.

Memory management. We provide methods for the creation, destruction and duplication of objects. When creating an object the difficulty arises that for its proper initialization additional parameters might be necessary, e.g. the length of a dynamic array. In this case we specify that the parameterless initialization method 'init', which must always be called after an object is created, may use additional global variables as implicit parameters.

A second problem is the treatment of objects that contain other *dependent* objects either explicitly as instance variables or implicitly like in a list object.

The effect of destruction or duplication of such an object is unclear since the dependent objects might be destroyed or duplicated or not. We solve this problem by explicitly stating at each object declaration for an object *o* whether the dependent objects of *o* belong to the *internal state* of *o* or not. Dependent objects belonging to the internal state are destroyed and copied just like regular instance variables whereas dependent objects that do not belong to the internal state are left intact. The following example shows the difference.

```
lineSegment = object(obj)
   (* Derived from the root class 'obj'. *)

   p, q: point
   (* 'point' is a class. 'p' and 'q' belong to the internal state of a 'lineSegment'. *)

end;


vector = object(obj)
   (* Derived from the root class 'obj'. *)

   length:   0..maxLength;
   (* Length of the vector *)

   elements: array [1..maxLength] of obj
   (* Elements of the vector do not belong to the internal state. *)

end;
```

The decision whether a dependent object belongs to the internal state or not is a purely pragmatic one and must be documented when declaring such an object in order to specify the semantics of destruction and duplication. In the case of 'lineSegment' we prefer completely independent copies whereas in the 'vector' example a (recursive) copy of the vector's elements is usually undesirable. We provide additional methods for cases where the recursive destruction or duplication of all dependent object is required.

Interactive input/output: Interactive input/output is needed for experimentation and demonstration. We support the following tasks.

- Display a geometric object on the screen. We use the XOR mode for drawing which has the advantage that drawing and erasing are the same operations.
- Display an object in a highlighted fashion which is useful for giving a visual feedback in algorithm animation or when an object has been selected by the user.
- Flash an object. This operation is used when animating an algorithm.
- Let the user interactively enter an object. This is usually done by dragging with the mouse. We have implemented a universal dragging method based on the following two primitive operations: 1) Construct the object given the location where the mouse button was first pressed and the current location of the mouse. 2) A method that displays the object in XOR mode on the screen.

File input/output: When processing geometric objects, we are interested in saving them permanently on secondary storage. File input/output becomes therefore another area where operations applicable to all objects must be provided. As file format we normally use a byte stream which contains geometry data and some type information in a prefix format. The introduction of lists and arrays makes this format recursive. A second file format provides a human readable LISP like notation which is useful for debugging or for data exchange with other programs.

Geometric transformations: We provide the following geometric transformations: Translation, rotation, scaling and mirroring of a geometric configuration with respect to a given line.

Type computations: Sometimes we wish to enquire the type of an object, e.g. when testing whether all members of a collection are of the same type. Another example is the creation of a byte, specifying the object's type for storage purposes. These and similar type computations are implemented as universal operations since they must be available for all objects in the system.

Random instances: Generation of random instances is used for the rapid creation of test data. We provide a method that changes the internal state randomly, preserving the object invariant. A global variable, usually a rectangle, specifies the boundaries in which the random change takes place.

Class description and method execution: Each class should be able to describe itself to the user, meaning that given a list of arbitrary objects, a class should deliver all the methods it can execute on these objects in human readable form. Assuming a list of three points, the 'point2d' class should offer a method for creating a circle, the 'pointVector' class should offer a method for computing a closest pair and a Voronoi diagram and so on. After all classes have given such a description, it must be possible to execute a specific method of a specific class on the given list of objects.

3 Implementation aspects: A sample of interesting details

3.1 The reference concept

In the context of key-based data structures the reference concept helps us distinguish between operations that require a key (e.g. 'find') and those that change a data structure without needing a key (e.g. 'swap'). We motivate this distinction with a problem that occurs when implementing the plane sweep algorithm for finding all intersecting line segments.

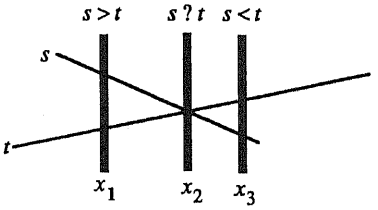


Figure 4: The difficulty of accessing line segments by their y-value

The line segments form a total order along the front and their key value is essentially their y-value when evaluated at the position of the front. At position x_2 we must exchange s and t, which is difficult using their key value: There are two objects with the same key value and we might access the same segment twice.

As a solution, we prohibit key based access in this case. Instead, we provide means for performing the required 'swap' without using key values. We associate with each object o in the data structure a unique *reference* that is supplied whenever o is known to participate in an operation. For easier understanding, imagine a reference to be a pointer into a data structure although the internal representation of a reference is hidden from the user. The idea of a reference resembles the notion of *items* introduced in LEDA [MN 89] as an abstraction of pointers and locations.

3.2 A heap with an efficient delete operation

This section demonstrates how to implement a priority queue as a heap using the reference concept such that an efficient delete operation is supported. A heap is a partially ordered binary tree such that the value associated with each internal node is surpassed or reached by any of its children. The figure below shows a heap together with its standard breadth first array representation.

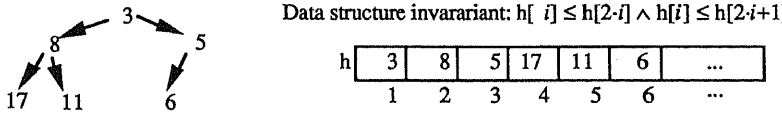


Figure 5: Tree and array representation of a heap

The textbook [AHU 83, p. 163] states that a heap does not allow an efficient delete operation and implementors of priority queues with delete have abandoned the heap and used more complicated data structures such as AVL-trees [B 81]. The reference concept is used to add an efficient delete operation to a heap.

The two key observations are: 1) Any element in a heap can efficiently be

deleted if we know its position and 2) we can fix this position as soon as an element is inserted. The following figure shows how this works.

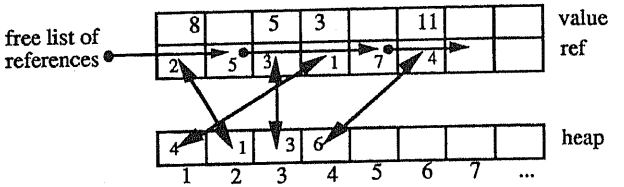


Figure 6: Implementing a heap with the reference concept

Data structure invariant:
value[heap[i]] ≤ value[heap[2·i]] ∧ value[heap[i]] ≤ value[heap[2·i+1]] ∧ ref[heap[i]] = i.

When an element is inserted into the heap we use the free list of references to assign it a location in the array 'value'. This location is the reference and is never changed. What we change instead is the appropriate double arrow between 'ref' and 'heap' which determines the element's position in the heap. The operation for insertion ('sift' or 'pushDown') is basically the same as for an ordinary heap while the 'delete' operation is a simple generalization of the standard 'deleteMin' operation: The element to be deleted is exchanged with the last element in the heap which is a leaf node. Then we perform the 'pushDown' operation on this sub-heap followed by the 'pushUp' operation reestablishing the data structure invariant. Therefore the cost of delete is O(log n) where n is the number of elements in the heap.

The code is simple.

```
type

   relation  = (less, equal, greater);
   seqIndex  = 1..maxN;
   seqIndex0 = 0..maxN;
   reference = ^integer;


   HPQ = object(obj) (* Heap Priority Queue *)

      length:     seqIndex0;      (* Number of elements *)
      freeList:   seqIndex;       (* Points to first empty element in 'ref' *)
      ref, heap:  array [seqIndex] of seqIndex0;
      value:      array [seqIndex] of obj;

      procedure init; (* Establishes the data structure invariant after creation. *)

      function insert
         (x: obj;
          function compare (a, b: obj): relation): reference;
```

```
    procedure delete
      (x: reference; function compare (a, b: obj): relation);

    procedure swap(p, q: seqIndex); (* Private methods *)

    procedure pushUp
      (i: seqIndex; function compare (a, b: obj): relation);

    procedure pushDown
      (i: seqIndex; function compare (a, b: obj): relation);

  end;

procedure HPQ.init;
begin
  length   := 0;
  freeList := 1;
  ref[1]   := 0
end;

procedure HPQ.swap(p, q: seqIndex);
  var pm, qm: seqIndex;
begin
  pm      := heap[p];     qm      := heap[q];
  ref[pm] := q;           ref[qm] := p;
  heap[p] := qm;          heap[q] := pm
end;

procedure HPQ.pushUp
  (i: seqIndex; function compare (a, b: obj): relation);
begin            (* ➡ conditional and *)
  while  (i > 1)    &
         (compare(value[heap[i]],value[heap[i div 2]])=less) do
  begin
    swap(i, i div 2);
    i := i div 2
  end
end;

procedure HPQ.pushDown
  (i: seqIndex; function compare (a, b: obj): relation);
var
  j, length2: seqIndex0;
  continue:   boolean;
begin
  continue := true;
  length2  := length div 2;
  while continue and (i <= length2) do
  begin
    j := 2 * i;  (* ➡ conditional and *)
    if (j <> length) &
      (compare(value[heap[j]], value[heap[j + 1]]) > less)
      then
      j := j + 1;
    if compare(value[heap[i]], value[heap[j]]) = greater then
    begin
```

```
      swap(i, j);
      i := j
    end
    else
      continue := false
  end
end;

function HPQ.insert
  (x: obj; function compare(a, b: obj): relation): reference;
var
  i: seqIndex0;
begin
  length := length + 1;
  i := freeList;
  if ref[freeList] = 0 then
  begin
    freeList := freeList + 1;
    ref[freeList] := 0
  end
  else
    freeList := ref[freeList];
  insert       := reference(i);
  ref[i]       := length;
  value[i]     := x;
  heap[length] := i;
  pushUp(length, compare)
end;

procedure HPQ.delete
  (x: reference; function compare (a, b: obj): relation);
var
  startPosition: seqIndex0;
begin
  startPosition    := ref[seqIndex(x)];
  swap(startPosition, length);
  ref[seqIndex(x)] := freeList;
  freeList         := seqIndex(x);
  length           := length - 1;
  if startPosition <= length then
  begin
    pushDown(startPosition,compare);
    pushUp(startPosition,compare)
  end
end;
```

More careful analysis shows that 'pushUp' and 'pushDown' can be implemented more efficiently by remembering the element that is common to consecutive swap operations. In [S 91] and [NSABD 91] we present experimental evidence that confirms the superior efficiency of this implementation of a priority queue as compared to an AVL-tree implementation.

A caveat as a final remark: This kind of priority queue cannot be used when the element to be deleted is only known by its key value. In this case we know

no efficient way to locate the element in a heap. In our experience implementing plane sweep algorithms however, this kind of priority queue was sufficient in all cases.

3.3 Implementing plane sweep in an object oriented fashion

Another advantage of object oriented design is the uniform treatment of plane sweep algorithms. When implementing plane sweep object oriented, we model events as objects, derived from an abstract event class that provides an execute method. The typical structure is given by the following declarations.

This decomposition has the advantage that all code for handling a certain event is concentrated in one place, making the implementation clearer and easier to understand. Furthermore the main program needs no change in the case new events become necessary (for example when we transform the sweep for testing whether two line segments intersect to the sweep that additionally finds all such intersections).

```
program genericPlaneSweep;

var
   xQueue: priorityQueue;
   yTable: dictionary;

type
   event = object
      procedure process          (* Abstract method *)
   end;

   event1 = object(event)        (* Derive the interface from the abstract class *)
      procedure process; override  (* Real implementation *)
   end;

   event2 = object(event)        (* Derive the interface from the abstract class *)
      procedure process; override  (* Real implementation *)
   end;

   (* Other event types *)

begin
   initialize xQueue; initialize yTable;
   while not xQueue.isEmpty do
      xQueue.extractMinimum.process (* Let the event process itself. *)
end;
```

3.4 Helpful hints for increasing the reliability of a geometric library

In order to create a reliable system we have tried to catch programming errors early. We recommend the following methods which were used successfully.

Show the dynamic memory allocated. The 'info' window contains the number of bytes currently allocated on the heap for dynamic variables (mostly objects). This number gives a first hint whether memory management is working correctly. A constantly growing number of allocated bytes is usually something to worry about.

Use assertions. Especially geometric algorithms often depend on the truth of certain assertions (e.g. the value of some denominator should be different from zero) and a defensive programmer introduces checks (assertions) at appropriate places which give warnings when the assertion fails. Even errors in algorithms can be detected earlier this way.

Data structures should have a method checking their invariant. This is especially helpful while testing non trivial data structures like AVL trees or even a heap. One tests the data structure by different operations, and after each operation the invariant is checked which increases the confidence in its correctness. Having program code actually check the invariant (as opposed to doing it by hand) also makes automatic testing feasible: One could randomly insert or delete a random element in a dictionary and check the invariant after each operation.

Write and keep test programs for the central data structures. We have written test programs to test the abstract data type 'dictionary' which has the additional advantage that it can be used for all concrete implementations of the type 'dictionary'. The alternative to test programs is to actually use the data structure in an algorithm which has the two disadvantages that 1) in the case of an error one does not know precisely the source of the error and 2)one algorithm rarely uses all methods offered by a data structure designed for universal applicability.

Create geometric test data and keep it for reference in a test suite. Creating good test data is not trivial and the results of this effort should be kept. A new program solving the same problem might be written, or more often the current implementation changes. In both cases one increases the confidence in the reliability by checking with the test suite. It should contain configurations ranging from no degeneracies at all to multiple ones.

Use algorithm animation for geometric algorithms. Algorithm animation should be used to present an algorithm's essential state information in a graphical way. In the debugging phase, this facilitates the detection of inconsistencies before an incorrect result occurs.

Do not remove the code used for debugging. The code used for debugging should be left in the programs since future modifications might benefit from it. Using conditional compilation removes any run time overhead if necessary.

4 Experience with object oriented programming

Techniques from object oriented programming have proven to be useful for implementing a library for geometric computation with extensive support for experimentation. We use the concept of abstract classes together with dynamic binding in order to implement interchangeable arithmetic and data structures. Furthermore the tree structure of the class hierarchy serves as an aid in structuring the whole system in an understandable way. Polymorphism helps in implementing universal data structures. Inheritance is mostly used in the form of interface inheritance; actual code is almost only inherited as methods in abstract classes that can be written in terms of other methods. In the following we discuss the points mentioned in the previous summary in more detail.

4.1 Dynamic binding and abstract classes

An abstract class is a class of objects with the property that no instances of the class itself are created, but only instances of derived classes. Dynamic binding means that the type of an object and the methods to be executed are determined at run time. Consider the example of the class 'point2d' which has descendants 'realPoint', 'floatPoint' and 'longIntPoint' (see also section 2.3). 'point2d' defines a common interface implemented in three different ways using three different kinds of arithmetic. Programs using only variables of the abstract class 'point2d' can be instantiated at run time to work with all three different concrete implementations.

We have used abstract classes and dynamic binding for two purposes: 1) Offering interchangeable arithmetic and interchangeable implementations of abstract data types and 2) factoring out common code. Factoring out common code is best illustrated by the universal dragging routine mentioned in section 2.5. An abstract class provides a method M_0 which is solely implemented in terms of other methods M_1, M_2, ..., M_k of the abstract class. As soon as M_1, M_2, ..., M_k are implemented in a derived class, a working implementation of M_0 is available. The abstract implementation of M_0 factors out the common code.

4.2 The class hierarchy and inheritance

Object oriented design allows us to take a class and derive new classes with enriched functionality from it, modeling the 'is_a' relation. This relationship is rare among geometric objects, causing the class hierarchy to be relatively flat in most places. The most prominent counterexample is the sequence 'convexPolygon' is_a 'polygon' is_a 'polyLine' is_a 'pointVector'.

Inheritance goes hand in hand with derivation: A method M defined in a class is automatically available for each of its subclasses and ideally needs not to be implemented in a different way ('overridden'). This is rarely the case in

geometry: The best way for solving the same problem for a class and a derived class can be totally different as the trivial example of computing the convex hull of a polygon and a convex polygon shows.

Having a class hierarchy with a single root has the advantage that universal data types can be created. For example we need a dictionary that can hold any other object. Even objects of different types should be admissible as long as we can define an order relation. The requirement that a dictionary can hold objects of the type of the root class achieves this goal.

As a conclusion, the concept of a class hierarchy is a useful tool for structuring the library and creating universal abstract data types. It is of less use for saving implementation effort.

4.3 Design concepts for classes

Deriving a new class from an existing one should model the 'is_a' relation since otherwise the structure of the class hierarchy would be confusing and the semantics of inherited operations could become unclear. This rule determines where to place a new class in the hierarchy. For example the class 'lineSegment' should not be derived from 'point2d' by adding another point because a line segment is not a point, but is composed of two points. In this case composition is the more appropriate construction principle.

During the evolution of a class hierarchy one often finds two classes which contain some similar methods. An example were the class for linked lists and the class for vectors for which various methods like displaying itself on the creen were similar. In this case we factor out common methods by ıtroducing a common abstract ancestor class which realizes the common ehavior.

5 Conclusion

We have described the basic building blocks that are available to the implementor of geometric algorithms working in the XYZ GeoBench programming environment.

A comprehensive set of universal operations applicable to all objects in the system covers a wide range of tasks: Memory management, interactive input/output, binary and text file I/O, geometric transformations, type computations, creation of random instances, class description and method execution.

Geometric algorithms can be implemented in an arithmetic independent way. This allows the user to experiment with different kinds of arithmetic:

Ordinary built-in floating point arithmetic, a floating point arithmetic featuring arbitrary basis and precision and standard integer arithmetic.

A set of reliable geometric primitives and common universal data structures like sequence, dictionary and priority queue facilitate the implementation effort. The reference concept on which dictionaries and priority queues are based allows efficient delete operations in AVL trees and heaps.

A variety of already implemented geometric algorithms covers a wide range of 2-dimensional Computational Geometry and serves as the building block for further implementation efforts.

The XYZ GeoBench serves also as an interactive front end to the XYZ Library of geometric algorithms. A smooth Macintosh style user interface is used to access most implemented algorithms, often with animation of their execution, and to perform a wide variety of experiments. Typical examples are efficiency measurements, testing the influence of different kinds of arithmetic, testing the effect of different implementations of abstract data types, running an algorithm on random or degenerate geometric configurations, etc.

Currently the most active application area of the XYZ GeoBench is in education. On the one hand we can demonstrate a wide variety of geometric algorithms in the class room due to the built-in animation capabilities and on the other hand we give our students the opportunity to implement geometric algorithms for themselves. The latter is greatly facilitated by the programming environment our workbench provides.

Acknowledgements

I thank J. Nievergelt for his support of the XYZ project and for commenting on an earlier draft of this paper. I am grateful to C. Ammann and M. Furter for their dedication in improving the user interface.

References

[AHU 83] A. Aho, J. Hopcroft, J. Ullman: Data Structures and Algorithms, Addison Wesley, 1983.

[B 81] K. Brown: Comments on "Algorithms for reporting and counting geometric intersections", IEEE Trans. Comput. vol. C-30, pp. 147-148, Feb. 1981.

[F 87] S. Fortune: A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2, pp. 153-174, 1987.

[NSABD 91] J. Nievergelt, P. Schorn, C. Ammann, A. Brüngger, M. De Lorenzi: XYZ: A project in experimental geometric computation, submitted, 1991.

[M 87] B. Meyer: Object-Oriented Software Construction, Prentice Hall, 1987.

[MN 89] K. Mehlhorn, S. Näher: LEDA, A Library of Efficient Data Types and Algorithms, preliminary version, Universität des Saarlandes, 1989.

[S 91] P. Schorn: Robust Algorithms in a Program Library for Geometric Computation, ETH PhD Dissertation 9519, to appear 1991.

[W 90] E. Welzl: A fast randomized algorithm for computing the minimal area disk enclosing a set of points in d-space, presentation at the Workshop on Computational Geometry, Dagstuhl, Oct 1990.

XYZ GeoBench Manual

Peter Schorn, schorn@inf.ethz.ch
Informatik, ETH, CH-8092 Zurich

Abstract

The XYZ GeoBench is a software system for experimental geometric computation. The User's Manual explains the Macintosh application GeoBench, an interactive front end to the XYZ Library of geometric algorithms. The Programmer's Manual specifies the interfaces of the object oriented programming environment supplied.

Contents

0 The XYZ GeoBench Software

The XYZ GeoBench software consists of two parts: 1) the Macintosh application 'GeoBench' and 2) the THINK Pascal sources together with the necessary project files.

The GeoBench is a Macintosh stand-alone application and is used for demonstrating the geometric algorithms of the XYZ Library and experimenting with them. It requires a Mac II with at least 2 MB of memory and, preferably, a two-page monitor. We describe the 'GeoBench' user interface, which adheres closely to Macintosh standards, in the User's Manual.

The 'GeoBench' and the XYZ Library are written in THINK Pascal 3.0, an object oriented extension of Pascal. Currently the system consists of 78 modules with about 1000 kB of source code. We describe the programming interfaces necessary for extending the GeoBench in the Programmer's Manual. We assume a basic understanding of the concepts *object*, *class*, *abstract class*, *method*, *inheritance* and *overriding of methods*.

An overview of the XYZ project is given in [NB 91] while [S 91b] emphasizes the programming environment aspect of the GeoBench. A detailed description of the design and implementation of the GeoBench is given in [S 91a] together with mathematical methods for constructing provably robust geometric programs. The diploma thesis [Brü 91] describes the realization of layered objects in the GeoBench.

Acknowledgements

I thank the following people: Jürg Nievergelt leads the XYZ project. Christoph Ammann, Michele De Lorenzi and Adrian Brüngger extended the GeoBench to include layered objects. Christoph Ammann implemented algorithms for splinegons, wrote the code for boolean operations on polygons and created color maps from geographic data. Michele De Lorenzi wrote the networking code. Christoph Ammann and Markus Furter improved the user interface considerably. Beat Fawer implemented two algorithms for computing the Voronoi diagram. Martin Müller wrote code for the Traveling Salesman Problem. Peter Lippuner and Peter Skrotzky wrote an earlier version of this manual and contributed a library routine for computing the contour of a set of rectangles. Björn Beeli adapted a grid file package written by Hans Hinterberger and Lise Pfau.

1 User's Manual

1.1 General

The standard menus 'File' and 'Edit' have their usual semantics. Clicking (i.e. pointing or selecting) and dragging with the mouse are supported where appropriate. The following discusses the menus available.

1.2 File Menu

New	⌘ N	Creates an empty geometry window.
Open...	⌘ O	Opens an existing geometry document containing geometric objects and displays it in a geometry window.
Close	⌘ W	Closes a geometry window. Holding down the shift key suppresses the saving dialog box.
Save	⌘ S	Saves a geometry window on the disk.
Save As...		Saves a geometry window on the disk under a new name.
Write Text...		Saves the contents of a geometry window in textual format. The grammar is given in Appendix A of the Programmer's Manual.
Read Text...		Reads geometric objects specified in a textual format.
Quit	⌘ Q	Terminates the program. Holding down the shift key suppresses the saving dialog box.

1.3 Edit Menu

Undo	⌘ Z	Undoes the effect of the most recent operation. This works only for dangerous or destructive operations.
Cut	⌘ X	Removes the selected objects from the geometry window and puts them into the clipboard. Other

applications like MacDraw can access the clipboard.

Copy ⌘C — Copies the selected objects into the clipboard. Other applications like MacDraw can access the clipboard.

Paste ⌘V — Pastes the geometric objects from the clipboard into a geometry window. Objects created with MacDraw can also be pasted.

Cut Last — Like 'Cut' but only the object which was selected most recently is moved to the clipboard.

Copy Last — Like 'Copy' but only the object which was selected most recently is copied to the clipboard.

Clear All — Deletes all geometric objects in a geometry window.

Select All ⌘A — Selects all geometric objects in a geometry window.

Redraw — Draws all objects again.

Zoom In... ⌘E — Enlarges a part of the window. After choosing this command the desired rectangular part of the window is selected by dragging with the mouse.

Zoom Out ⌘F — Undoes the most recent 'Zoom In' operation.

Transformations

Translate... — Moves the selected objects. In the dialog box the quantities Δx and Δy determine the translation vector. Note that the origin is usually in the top left corner of the geometry window. Objects can also be translated by dragging. Note however that dragging near a vertex (e.g. of a polygon) results in moving just that vertex and not the whole object. If a whole polygon is to be translated by dragging, one should grab it at an edge.

Rotate... — Rotates the selected objects around a point. First the point is selected and then the objects to be rotated (while holding the Shift key down). The angle is specified using the dialog box.

Scale... The selected objects are scaled by a given factor using a user specified origin.

Reflect... Reflects the selected objects with respect to a user specified line.

3d Projection... Opens a dialog box that lets the user specify the eye point and the projection point for the view of layer objects (see [Brü 91] for further details). In addition, the dialog box contains a check box that determines whether hidden line elimination should be performed.

3d Transformation... Opens a dialog box that lets the user apply transformations such as translation, rotation or scaling to the view of layer objects (see [Brü 91] for further details). For convenience this dialog box contains a check box for hidden line elimination.

1.4 Objects Menu

Select Lets the user select objects as opposed to enter objects.

Point Lets the user enter points by clicking with the mouse.

Line Segment Click at the start point and drag the segment.

Poly Line Click at the start point, drag additional segments and double click when done.

Polygon Like 'Poly Line'.

Convex Polygon Like 'Polygon'. The convex hull of the polygon entered is used if the polygon is not convex.

Rectangle Click at a corner and drag the rectangle.

Circle Click at the center and drag the circle.

D Point Create a d-dimensional point by clicking at a location. This sets the x- and the y-coordinate while the other d–2 coordinates (if present, see the 'Dimension' command) are set to zero.

Dimension...	⌘D	Specifies the dimension of subsequently created d-dimensional points.
Group	⌘G	Groups the selected objects together in an object of type 'vector'.
Ungroup	⌘H	If the selected object is a 'vector' it is ungrouped. Only one level of ungrouping is performed.
Random...	⌘R	Generates a number of randomly located objects of the same type. The type is specified either using the 'Objects' menu or by clicking at an icon in the palette at the left border of each geometry window.
Randomize		Sets the initial value of the random number generator to some arbitrary value.
Reset Random		Resets the initial value of the random number generator to the value present at program start time.

1.5 Arithmetic Menu

Real	Sets the type of the coordinates of subsequently created objects to 'Real'.
Float	Sets the type of the coordinates of subsequently created objects to 'Float', a floating point number system realized in software. The base and precision (number of digits in the mantissa) can be set using the two following commands.
Base...	Sets the base of the 'Float' number system.
Precision...	Sets the number of digits in the mantissa.
Longint	Sets the type of the coordinates of subsequently created objects to 'Longint'.

Objects of the same type (e.g. point) that were created with different settings of the arithmetic menu are treated as belonging to different types. Changing the setting of the arithmetic menu affects only subsequently created objects.

1.6 Animation Menu

Algorithm animation can be selectively turned on and off. One item activates the animation of all algorithms solving the same problem (e.g. turning on the animation for 'Closest Pair' animates all algorithms solving the closest pair problem).

1.7 Windows Menu

Hide		Hides the window that lies on top. By selecting its name in the 'Windows' menu it becomes visible again.
Info	⌘ I	The 'Info' window becomes the topmost visible window.
Geo-##		The specified geometry window becomes the topmost visible window.

1.8 Operations Menu

The 'Operations' menu contains all the operations that are applicable to the selected objects. Choosing an operation from this menu performs the operation and creates a new geometry window containing the result of the operation. For a list of implemented algorithms see section 2.9 of the Programmer's Manual.

2 Programmer's Manual

2.1 Organization

This manual is organized as follows. Section 2.2 describes the class hierarchy. Section 2.3 describes methods applicable to all objects in the system. Section 2.4 explains the basic ways of building collections of objects. Section 2.5 introduces the geometric primitives. Section 2.6 explains the most common abstract data types, dictionary and priority queue. Section 2.7 describes the graph oriented part of the GeoBench. Section 2.8 explains animation and how to interactively obtain parameters. Section 2.9 is a reference section on the implemented algorithms. Appendix A specifies the grammar of the textual I/O format. Appendix B describes the changes that we did in TransSkel, a public domain application module written by Paul DuBois [DB 89].

2.2 The class hierarchy

The class hierarchy is the glue holding the system together. All classes are descendants of a single abstract root class 'obj' which provides a set of universal operations applicable to all objects. The classes 'sequence', 'point2d', 'dictionary' and 'priorityQueue' are also abstract classes which should be used to postpone implementation decisions.

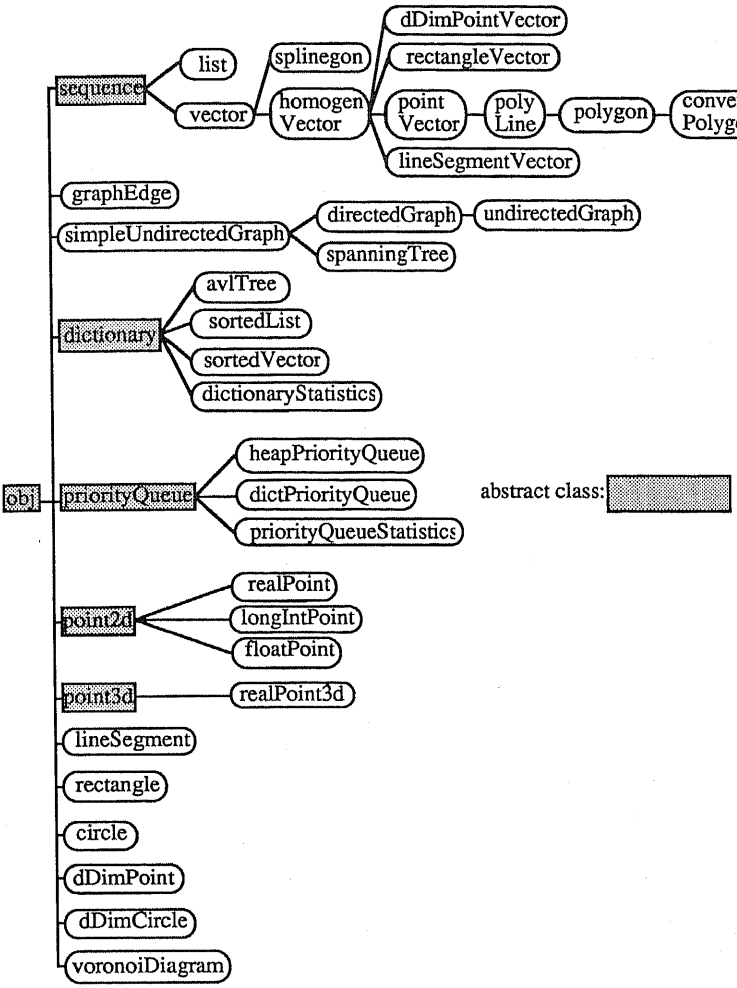
A geometric algorithm operating on an object of type T1 and producing an object of type T2 is a function method of class T1 producing an object of type T2.

Implementing a new geometric algorithm consists of

1) finding the appropriate place for the new method in the class hierarchy depending on the input data. Assume we implement a method for class T.

2) implementing the algorithm (with animation, if possible) using the methods from sections 2.3 – 2.9.

3) changing the methods 'describe' and 'execute' (see section 2.3) of class T such that the interactive front end of the XYZ GeoBench 'knows' about the newly implemented algorithm.



2.3 The common behavior of all objects: the root class 'obj'

This section describes the interface common to all objects. When implementing a new descendant of 'obj', the methods described in this section must be implemented appropriately, taking the default implementation into account.

```
type
   objectType = (intType, stringsType, point2dType,
   lineSegmentType, rectangleType, listType, vectorType,
   homogenVectorType, pointVectorType, convexPolygonType,
   voronoiEdgeType, directedGraphType, undirectedGraphType,
```

```
  graphEdgeType, voronoiDiagramType, polygon2dType,
  markedRealPointType, simpleUndirectedGraphType,
  spanningTreeType, rectangleVectorType, dDimPointType,
  dDimCircleType, dDimPointVectorType, realPointType,
  floatPointType, longIntPointType, sequenceType, avlTreeType,
  sortedListType, sortedVectorType, lineSegmentVectorType,
  dictPriorityQueueType, heapPriorityQueueType, circleType,
  splinegonType, spline2Type, straightEdgeType, polyLineType,
  layerType, polyLayerType, layerVectorType,
  polyLayerVectorType, QDPictureType, ColorQDPictureType,
  objType);

relationObj = (lessThanObj, equalObj, greaterThanObj);
```

obj = object

(* Memory management *)

```
procedure init;
```
(* Initializes the internal state of an object, i.e. all instance variables are set to a defined value. This method should be called after creating an object with 'new'. An object might *depend* on other objects, either directly like on an instance variable or indirectly like in a linked list. Dependent objects are initialized iff they belong to the internal state which is stated explicitly. 'init' may use global variables defined in 'obj' to determine types or other values. The appropriate variables are specified in the respective class interface. After 'init' has been called the object is ready to use. The default implementation does nothing. *)

```
procedure free;
```
(* Reclaims the memory belonging to the object's internal state (the reverse operation to 'init'). Dependent objects not belonging to the internal state are not destroyed. The object is removed; no dispose is allowed. The default implementation calls 'ShallowFree' which works correctly for all objects without dependent objects. *)

```
procedure freeAll;
```
(* Like 'free', but all dependent objects are also reclaimed recursively. The default implementation calls 'free' which is correct for objects that do not have dependent objects. *)

```
function duplicate: obj;
```
(* Produces a copy of the object's internal state. The default implementation calls 'ShallowDuplicate' which works correctly for all objects that do not have dependent objects. *)

```
function duplicateAll: obj;
```
(* Like 'duplicate', but all dependent objects are copied recursively. The default implementation calls 'duplicate' which is correct for objects without dependent objects. *)

```
procedure ShallowFree;
```
(* Low level method to free an object, should not be overridden. *)

```
function ShallowDuplicate: obj;
```
(* Low level method to duplicate an object, should not be overridden. *)

(* Interactive I/O *)

```
procedure interactiveOutput;
```
(* Draws the object. Because of the XOR mode, an object can be removed from the window by a second call to 'interactiveOutput' which is useful in algorithm animation. The world coordinate system is used and the standard implementation raises an error condition.
Warning: 'interactiveOutput' is also used to draw into the external scrapbook, e.g., if the user selected 'Copy' from the 'Edit' menu and leaves the application. Do *not* assume that you are necessarily drawing into a window when your 'interactiveOutput' method is called. However, you may use 'thePort^.portRect' to determine the outline of the 'window' you are drawing in. If you really need to know whether you are drawing or just writing to the scrapbook, use 'thePort^.pnVis' ('true' means you are drawing in a window). See the implementation of 'voronoiDiagram.interactiveOutput' as an example.*)

```
procedure interactiveHighlight;
```
(* Draws the object in the current window in a highlighted fashion. The world coordinate system is used. The standard implementation enlarges the pen size and calls 'interactiveOutput'. *)

```
procedure interactiveFlash;
```
(* Let the object flash in the current window. The object is not drawn. The world coordinate system is used and the default implementation is implemented completely in terms of 'interactiveHighlight'. *)

```
procedure interactiveInput (px, py: integer);
```
(* Like 'windowInput' but the world coordinate system is used. The default implementation is based completely on 'windowInput' and 'windowToWorld'. *)

```
procedure windowOutput;
```
(* Draws the internal state on the current window. The window coordinate system is used and the default implementation is based completely on 'interactiveOutput' although this is inefficient in most cases. *)

```
procedure windowInput (px, py: integer);
```
(* Like 'init', but the internal state is read using the mouse. It gives visual feedback but leaves the drawing window unchanged after completion (XOR graphics). '*px*' and '*py*' are the coordinates of the first mouse click. The window coordinate system is used. The default implementation of 'windowInput' is based completely on 'construct' and 'windowOutput'. *)

```
procedure construct (px, py, newX, newY: integer);
```
(* This method changes an existing object on which 'init' has been called according to the four parameters formed by the first click location ('*px*', '*py*') and the current click location ('*newX*', '*newY*'). The window coordinate system is used. This method is only called by the default implementation of 'windowInput' and need not be implemented if 'windowInput' is overridden. The default implementation raises an error condition. *)

(* Binary and text file I/O *)

```
procedure fileOutput;
```
(* The whole object is written to '*currentStream*' but no type identifier is written. The default implementation raises an error condition. *)

```
procedure fileOutputTag;
```
(* Like 'fileOutput', but a type identifier is written first. The function 'fileGetType' in module 'obj' can be used to retrieve the type. The default implementation is based completely on 'getType' and 'fileOutput'. *)

```
procedure fileInput;
```
(* The complete object is read from '*currentStream*', the first byte on '*currentStream*' is the first byte of the internal state. The default implementation raises an error condition. *)

```
procedure textFileOutput;
```
(* Writes the internal state in textual form to '*currentStream*'. The first token written is always the type of the object. The grammar is given in appendix A. The default implementation raises an error condition. *)

```
procedure textFileInput;
```
(* Reads the internal state in textual form from '*currentStream*'. The first token is always the type of the object. The grammar is given in appendix A. The default implementation raises an error condition. *)

(* **Description / execution.** Whenever objects are selected by the user, the interactive front end of the XYZ GeoBench calls for each class in the system the respective 'describe' method, thereby asking each class whether an operation of this class can be executed on the selected objects. GeoBench determines this way which operations are appropriate for which objects. Whenever a new method should be made available to the end user the pair 'describe' / 'execute' must be changed to reflect the new algorithm. *)

```
procedure describe;
```
(* Computes the (possibly hierarchical) menu entries of operations applicable to the list of objects in the global variable '*currentArguments*'. The procedures 'menuEntry' (plus 'beginSubMenu' and 'endSubMenu' for hierarchical menus) from the module 'sequence' are to be used. The default implementation does nothing. *)

```
procedure execute (item: integer);
```
(* Executes the selected method '*item*' with arguments in '*currentArguments*' and appends the result to the global variable '*currentResult*'. The default implementation raises an error condition. *)

(* **Geometric transformations** *)

```
procedure translate (dx, dy: extended);
```
(* Translates an object by ('*dx*','*dy*'). The default implementation raises an error condition. *)

```
procedure rotate (xOrigin, yOrigin, alpha: extended);
```
(* Rotates the object by '*alpha*' (radians) around ('*xOrigin*', '*yOrigin*'). The default implementation raises an error condition. *)

```
procedure scale (lambda: extended);
```
(* Scales the object by '*lambda*'. The default implementation raises an error condition. *)

```
procedure reflect (a, b, c: extended);
```
(* Mirrors the object at the line $a \cdot x + b \cdot y = c$, $a^2 + b^2 \neq 0$. The default implementation raises an error condition. *)

(* Other transformations *)

```
procedure randomChange;
```
(* Changes the internal state randomly constrained by '*currentRandomConstraint*'. This global variable is usually interpreted as a rectangle describing the boundaries in which the random change takes place. The default implementation raises an error condition. *)

```
procedure windowToWorld;
```
(* Transforms the object to world coordinates. The default implementation raises an error condition. *)

(* Type computation and comparison *)

```
function getType: objectType;
```
(* Get 'self''s type. The default implementation raises an error condition and returns 'objType'. *)

```
function getTextType: str255;
```
(* Computes a textual description of 'self''s type. The default implementation raises an error condition and returns the string 'Obj'. *)

```
function memberType (o: obj): boolean;
```
(* Tests whether '*o*' is a descendant of 'self'. The default implementation raises an error condition and returns 'true'. *)

```
function convertToType (t: pointTypeRange): obj;
```
(* Produces a copy of 'self' based on type '*t*'. The default implementation raises an error condition and returns the result of 'duplicateAll'. *)

```
function equal (o: obj): boolean;
```
(* Determines whether 'self' is identical to '*o*'. The default implementation raises an error condition and returns 'false'. *)

```
function genericLessThan (o: obj): boolean;
```
(* Determines whether 'self' is less than '*o*' where 'o.getType = self.getType'. This function imposes a canonical order on the objects of a type and is used to eliminate duplicates by sorting. The default implementation raises an error condition and returns 'true' $\Leftrightarrow$ '*o*' and 'self' are different. *)

(* Selecting and dragging *)

```
function isSelected (where: obj): boolean;
```
(* Determines whether the object is selected by the given rectangle '*where*'. The default implementation raises an error condition and returns 'false'. *)

```
function dragVertex (where: obj): boolean;
```
(* Tests whether the user attempted to drag a point (vertex) inside the rectangle '*where*'. If a vertex is to be dragged, the function does all the dragging until the mouse button is released, updates the object and returns 'true'. The default implementation returns 'false'. *)

```
(* Miscellaneous *)

procedure textOutput;
   (* Outputs the object in textual form in the THINK Pascal text window. Used for
      debugging. The default implementation raises an error condition. *)

procedure centerOfGravity (var cx, cy: extended);
   (* Returns a point ('cx', 'cy') close to the object. The center is used for drawing
      graphs. The default implementation raises an error condition and returns the
      point (0, 0). *)

function pointCover (n: longInt): obj;
   (* Covers the object with points, 'n' point for each atomic part. The result is a
      'pointVector'. The default implementation raises an error condition and returns
      'nil'. *)

end; (* obj *)
```

2.4 Sequences, lists and dynamic arrays: The classes 'sequence', 'list', 'vector' and 'homogenVector'

This section describes the basic classes for building collections of objects. The abstract class 'sequence' factors out the common behavior of lists and arrays. The class 'list' realizes single linked lists whereas the class 'vector' realizes dynamic arrays. All elements in a 'homogenVector' have the same type.

```
type
   seqIndex  = 1..maxN;
   seqIndex0 = 0..maxN;

sequence = object(obj)

   length: seqIndex0;
      (* Number of elements in the sequence. *)

   procedure forAll (procedure whatToDo (x: obj));
      (* Performs the procedure 'whatToDo' on all objects. *)

   procedure append (x: obj);
      (* Appends 'x' at the end of the sequence. *)

   procedure removeLast;
      (* Removes the last element from the sequence without destroying it. *)

   procedure appendNonNil (x: obj);
      (* Appends 'x' if 'x' ≠ 'nil'. *)

   function sameType: boolean;
      (* Tests whether all elements of the sequence are of the same type. *)
```

```
function ith (i: seqIndex): obj;
   (* Produces the i-th element of the sequence. Although this method is also available for
      vectors we recommend for efficiency reasons the use of 'elements^^[i]' instead of
      the more expensive method call. *)

end; (* sequence *)


objArrayH = ^^ array [seqIndex] of obj;
   (* Handles (i.e. pointers to pointers) are more efficient than pointers for the Macintosh
      memory manager. They can be moved after they have been allocated while pointers stay
      fixed. *)

vector = object (sequence)

   elements: objArrayH;
      (* Handle to an array containing the elements. The objects in the array do not belong to
         the internal state but the array does. *)

   (* Memory management *)

   procedure init; override;
      (* The global variable 'currentVectorLength' determines how many elements are
         allocated and must be set before 'init' is called. *)

   procedure initFromList (l: list);
      (* Creates a vector from the elements in the list 'l'. No call to 'init' is necessary. *)

   procedure allocateElements (number: seqIndex0);
      (* Reserves space for at most 'number' elements. This method is used to dynamically
         extend or shrink a vector although it should be used cautiously because it involves
         copying. *)

   function allocatedElements: seqIndex0;
      (* Returns the number of currently allocated elements. *)

   (* Rearranging the elements *)

   procedure revert;
      (* Reverts the order of the elements in the vector. *)

   procedure swap (i, j: seqIndex);
      (* Exchanges the 'i'th and the 'j'th element. *)

   procedure randomShuffle (n, l: seqIndex0);
      (* Chooses randomly 'n' elements from the elements in 1..'l' and places them
         into 1..'n'. *)
```

(* Sorting and searching *)

```
procedure sort
      (function lessThan (p, q: obj): boolean;
       l: seqIndex; r: seqIndex0);
   (* Sorts 'elements' between 'l' and 'r' using the comparison function 'lessThan'. *)

procedure sortAll
      (function lessThan (p, q: obj): boolean);
   (* Sorts 'elements' between 1 and 'length' using the comparison function 'lessThan'.*)

function binarySearch
      (x: obj; function compare (a, b: obj): relationObj;
       l: seqIndex; r: seqIndex0): seqIndex0;
   (* Finds 'x' between 'l' and 'r' in the vector which is sorted in ascending order.
      'binarySearch = 0' means that 'x' was not found, otherwise 'binarySearch' gives the
      location of 'x' in the vector. *)

function findExtreme
      (function lessThan (p, q: obj): boolean): seqIndex0;
   (* Finds the smallest object according to the comparison function 'lessThan'. *)
```

(* Heap operations *)

```
procedure heapify (function lessThan (p, q: obj): boolean);
   (* Produces a heap using 'lessThan'. The first element is the minimum. *)

procedure sift
      (function lessThan (p, q: obj): boolean;
       l, r: seqIndex);
   (* Sifts element 'l' into the heap which ranges from 'l' + 1 to 'r'. *)

procedure nextMin (function lessThan (p, q: obj): boolean);
   (* Creates a new heap by removing the first element. *)

procedure insert
      (x: obj; function lessThan (p, q: obj): boolean);
   (* Inserts an element into the heap. *)

procedure heapSort
      (function greaterThan (p, q: obj): boolean);
   (* Sorts the whole vector using heapsort into ascending order. Note that the usual
      lessThan function will sort the wrong way! *)

function isHeap
      (root: seqIndex;
       function lessThan (p, q: obj): boolean): boolean;
   (* Tests whether the vector with the given 'root' (usually 1) fulfills the heap property.*)
```

(* Miscellaneous *)

```
procedure appendSafe (x: obj; increment: seqIndex0);
   (* Like append, but the vector is extended by 'increment' if there is not sufficient
      space. *)
```

```
  procedure preset (number: seqIndex0; t: objectType);
    (* Creates a vector of 'number' objects of type 't'. No call to init is necessary. *)

  procedure eliminateDuplicates;
    (* Removes from 'self' all duplicated elements and destroys them with 'freeAll'. *)

  procedure forAllPermutations
        (n: seqIndex0; procedure whatToDo (v: vector));
    (* Executes the procedure 'whatToDo' for all permutations of the first 'n' elements. *)

  procedure readLength;
    (* Reads the length of the vector from 'currentStream.' *)

end;  (* vector *)

homogenVector = object(vector)

  (* All objects are of the same type and the operations are the same as for a 'vector'. *)

end;  (* homogenVector *)


elementH = ^^ record
  value: obj;          (* The object. *)
  next : elementH;     (* The successor. *)
end; (* elementH *)

list = object(sequence)    (* linked list *)

  first, last: elementH;
    (* The first and the last element in the linked list. *)

  procedure concatenate (x: list);
    (* Appends the list 'x' at the end of the list. Note: No duplication is taking place and the
       list 'x' is left intact. *)

  procedure flatten;
    (* Appends to the list recursively all elements which are member of a sequence which is
       a member of 'self'. The member sequences of 'self' are destroyed. *)

  procedure push (x: obj);
    (* Appends the element 'x' at the front of the list. *)

  procedure pop;
    (* Removes the first element from the list. *)

end;  (* list *)
```

2.5 Arithmetic independent geometric primitives: The basic geometric classes 'point2d', 'lineSegment', 'rectangle' and 'circle'

In this section we describe the elementary geometric objects and the geometric primitives available. We recommend to use the abstract class 'point2d' whenever possible in order to write code that is arithmetic independent.

```
type
  signType = -1..1;
```

```
point2d = object(obj)

  procedure randomChange; override;
    (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

  (* Coordinate manipulation *)

  function getX: extended;
    (* Gets the x-coordinate. *)

  procedure setX (newX: extended);
    (* Sets the x-coordinate. *)

  function getY: extended;
    (* Gets the y-coordinate. *)

  procedure setY (newY: extended);
    (* Sets the y-coordinate. *)

  function xLessThan (p: point2d): boolean;
    (* Is 'self.getX < p.getX' ? *)

  function xEqual (p: point2d): boolean;
    (* Is 'self.getX = p.getX' ? *)

  function yLessThan (p: point2d): boolean;
    (* Is 'self.getY < p.getY' ? *)

  function yEqual (p: point2d): boolean;
    (* Is 'self.getY = p.getY' ? *)

  (* Drawing *)

  procedure drawLine (p: point2d);
    (* Draws a line from 'self' to 'p'. *)

  procedure labelPoint (l: str255);
    (* Draws the string 'l' below 'self'. *)
```

(* Geometric primitives *)

```
function whichSideX (p, q: point2d): extended;
```
(* Determines on which side of the directed line segment from '*p*' to '*q*' the point 'self' lies.
< 0: 'self' lies to the left of the directed line segment '*pq*'
= 0: 'self' lies on the directed line segment '*pq*'
> 0: 'self' lies to the right of the directed line segment '*pq*'
'whichSideX' is also twice the signed area of the triangle with vertices 'self', '*p*' and '*q*'. *)

```
function whichSideSign (p, q: point2d): signType;
```
(* Computes 'sign(whichSideX)' and rounds it to '0' if the exact result cannot be determined. *)

```
function crossProductX (p, q, r: point2d): extended;
```
(* Computes the cross product ('self' – '*p*') × ('*r*' – '*q*'). *)

```
function distanceX (p: point2d): extended;
```
(* Computes the Euclidean distance between 'self' and '*p*'. *)

```
function squaredDistanceX (p: point2d): extended;
```
(* Computes the squared Euclidean distance between 'self' and '*p*'. *)

```
function squaredDxX (p: point2d): extended;
```
(* Computes the squared difference in x-coordinates between 'self' and '*p*'. *)

```
function squaredDyX (p: point2d): extended;
```
(* Computes the squared difference in y-coordinates between 'self' and '*p*'. *)

```
procedure circleCenterX
    (p, q: point2d; var cx, cy: extended);
```
(* Computes the point ('*cx*', '*cy*'), the center of the circle through 'self', '*p*' and '*q*'. *)

```
function intersectLineSegmentX
    (q, r, s: point2d;
     left, right: point2d): boolean;
```
(* 'true' ⇔ the segment 'self *q*' and '*rs*' intersect. The left intersection point is '*left*', the right intersection point is '*right*' (lexicographically). '*l.p*' = '*l.q*' is possible and common. *)

```
function xPlusYX: extended;
```
(* Computes '*x*' + '*y*'. *)

```
function xMinusYX: extended;
```
(* Computes '*x*' – '*y*'. *)

```
function rayEvalX
    (p: point2d; cosSlope, sinSlope: extended): extended;
```
(* Evaluates the positive ray emanating from '*p*' with the given slope at 'self.getX'. *)

```
function bisectorEvalX (p, q: point2d): extended;
```
(* Evaluates the bisector of '*p*' and '*q*' at 'self.getX'. *)

```
function intersectRayBisectorX
      (cosSlope, sinSlope: extended;
       p, q, intersection: point2d): boolean;
   (* Tests whether the positive ray emanating from 'self' with the given slope intersects
      the bisector given by 'p' and 'q'. *)

function intersectBisectorBisectorX
      (p, q, r, intersection: point2d): boolean;
   (* Tests whether the bisector('self','p') intersects bisector('q','r') where
      |{'self', 'p', 'q', 'r'}| = 3. *)

function intersectRayRayX
      (cosLower, sinLower: extended;
       p: point2d; cosUpper, sinUpper: extended;
       intersection: point2d): boolean;
   (* Tests whether the rays ('self', 'cosLower', 'sinLower') and ('p', 'cosUpper',
      'sinUpper') intersect. *)

function middle (p: point2d): point2d;
   (* Computes the point ('self.getX + p.getX', 'self.getY + p.getY') / 2. *)

function createLineSegment (p: point2d): lineSegment;
   (* Creates a line segment from 'self' to 'p'. *)

function createCircle (p, q: point2d): circle;
   (* Creates the circle through 'self', 'p' and 'q' if they are not collinear. *)

end; (* point2d *)

lineSegment = object(obj)

p, q: point2d;
   (* Points 'p' and 'q' belong to the internal state. *)

procedure init; override;
   (* The global variable 'currentPointType' determines which kind of points are used. *)

procedure randomChange; override;
   (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

function whichSideX (r: point2d): extended;
   (* Determines on which side of the directed (from 'p' to 'q') line segment 'self'
      the point 'r' lies.
      < 0: 'r' lies to the left of the directed line segment 'self'
      = 0: 'r' lies on the directed line segment 'self'
      > 0: 'r' lies to the right of the directed line segment 'self' *)
      'whichSideX' is also twice the signed area of the triangle with vertices 'r', 'self.p'
      and 'self.q'. This function is similar to 'point2d.whichSideX' and introduced here
      for convenience. *)

function intersectLineSegmentX (r, l: lineSegment): boolean;
   (* 'true' ⇔ the segment 'self' and 'r' intersect. The intersection point is 'I', where 'l.p'
      ≤ 'l.q' lexicographically ('l.p' = 'l.q' is possible and common). *)
```

```
  function length: real;
    (* Computes the length of the segment. *)

  function collinear (l: lineSegment): boolean;
    (* Tests whether 'self' and 'l' are parallel. *)

end; (* lineSegment *)

circle = object(obj)

  x, y, radius: real;

  procedure randomChange; override;
    (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

  function getRadius: markedRealPoint;
    (* Creates a 'markedRealPoint' marked with the radius. *)

  function inCircle (p: point2d): boolean;
    (* Tests whether 'p' is in the circle. *)

  procedure makeFrom2points (p, q: point2d);
    (* Creates the smallest circle through the two points 'p' and 'q'. *)

  procedure makeFrom3points (p, q, r: point2d);
    (* Creates the smallest circle through the three points 'p', 'q' and 'r'. *)

  procedure makeDisk (l: list);
    (* Creates the smallest circle through the points in 'l'. *)

  procedure makeFromPointCircle (p: point2d; c: circle);
  (* Creates the smallest circle through 'p' and 'c' with 'not inCircle()'. *)

end; (* circle *)

rectangle = object(obj)

  left, bottom, right, top: real;
    (* (left ≤ right) ∧ (bottom ≤ top) *)

  procedure randomChange; override;
    (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

end; (* rectangle *)

(* Comparison functions for points *)

function pointLessThanXY (p, q: point2d): boolean;
  (* Lexicographical comparison, first the x-coordinate then the y-coordinate. *)
```

```
function pointLessThanYX (p, q: point2d): boolean;
   (* Lexicographical comparison, first the y-coordinate then the x-coordinate. *)

function pointGreaterThanXY (p, q: point2d): boolean;

function pointGreaterThanYX (p, q: point2d): boolean;

function comparePointXY (p, q: point2d): relationObj;
   (* Determines which relation holds between 'p' and 'q'. *)

function comparePointYX (p, q: point2d): relationObj;
```

2.6 Common universal abstract data types: the classes 'dictionary' and 'priorityQueue'

The most common abstract data types encountered in geometric algorithms are the dictionary and the priority queue. Both data types are implemented in the XYZ GeoBench in a general way based on the *reference* concept. A reference is the abstraction of location in a data structure and can be viewed as a pointer into the data structure. For example, when a data item is inserted into a dictionary, the dictionary returns a reference which can be used to delete the data item or to change its value.

```
type
   reference     = ^integer;          (* Any pointer suffices. *)
   directionType = (left, right);
```

```
dictionary = object(obj)

   numberOfElements: longInt;

   function getObject (where: reference): obj;
      (* Retrieves the object referenced by 'where'. *)

   procedure setObject (where: reference; newValue: obj);
      (* Changes the object referenced by 'where' into 'newValue'. *)

   procedure sequenceToDictionary
         (s: vector; l: seqIndex1; r: seqIndex0);
      (* Equivalent to inserting the sorted vector elements between 'l' and 'r' into an empty
         dictionary. *)

   function find
         (x: obj; function compare (a, b: obj): relationObj;
          var where: reference;
          var direction: directionType): boolean;
      (* If 'find' = 'true' then 'where' points to some element 'e' with 'compare(e, x) =
         equalObj'. If 'find' = 'false' then either 'where' = 'nil' (dictionary empty) or 'x' is a
         direct neighbor of 'where' in direction 'direction'. *)
```

```
function member
     (x: obj;
      function compare (a, b: obj): relationObj): boolean;
   (* Tests whether 'x' is in the dictionary or not. *)

function insert
     (x: obj; where: reference;
      direction: directionType): reference;
   (* If 'where' = 'nil' then insert 'x' into the dictionary which must be empty. If 'where'
      ≠ 'nil' then insert 'x' before ('direction' = 'left') or after ('direction' = 'right')
      'where'. *)

function insertObj
     (x: obj; function compare (a, b: obj): relationObj;
      var found: boolean): reference;
   (* Inserts 'x' into the dictionary. 'found' = 'false' ⇔ 'x' is unique. *)

procedure insertNewObj
     (x: obj; function compare (a, b: obj): relationObj);
   (* Inserts 'x' if it is not a member of the dictionary, otherwise this is an error. *)

procedure delete (x: reference);
   (* Removes the element referenced by 'x' from the dictionary. *)

procedure swap (p, q: reference);
   (* Exchanges the elements referenced by 'p' and 'q' in the dictionary. *)

procedure rangeSwap (p, q: reference);
   (* Exchanges the elements between 'p' and 'q' in the dictionary ('compare(p, q) =
      lessThanObj' must hold and is not checked). *)

function next
     (x: reference; direction: directionType): reference;
   (* Find the predecessor ('direction' = 'left') or the successor ('direction' = 'right') of
      'x'. *)

function extreme (direction: directionType): reference;
   (* Find the leftmost ('direction' = 'left') or the rightmost ('direction' = 'right') value. *)

function isEmpty: boolean;
   (* Determines whether the dictionary is empty. *)

procedure forAll
     (procedure whatToDo (x: obj); direction: directionType);
   (* Performs 'whatToDo' on all elements 'x' of the dictionary, starting at the location
      most in direction 'direction'. *)

procedure rangeForAll
     (leftBound, rightBound: obj;
      function compare (a, b: obj): relationObj;
      procedure whatToDo (x: obj); direction: directionType);
   (* Performs 'whatToDo' on all elements 'x' of the dictionary that are between
      'leftBound' and 'rightBound' (including), starting at the location most in direction
      'direction'. The value 'nil' serves as -∞ for 'leftBound' and as +∞ for
      'rightBound'. *)
```

```
function compareReference (p, q: reference): relationObj;
   (* Determines the order of the elements referenced by 'p' and 'q' in the dictionary
      (without key operations!). *)

function infoString: str255;
   (* Computes information about the dictionary (this only makes sense for a
      'dictionaryStatistics'). *)

end; (* dictionary *)
```

Dictionaries come in four different flavors, differing mainly in their internal realizations and the corresponding initialization routines.

```
type

avlTree = object(dictionary)

   (* Implementation as an AVL tree with optimal insertion / deletion costs. *)

   root: avlNodeH;
      (* No global variables are needed for the initialization procedure 'init'. *)

end; (* avlTree *)

sortedList = object(dictionary)

   (* Implementation as a sorted list. Insertion / deletion is expensive. *)

   minimum, maximum: sortedListNodeH;
      (* No global variables are needed for the initialization procedure 'init'. *)

end; (* sortedList *)

sortedVector = object(dictionary)

   (* Implementation as a sorted array. Insertion / deletion is expensive. *)

   procedure init; override;
      (* The global variable 'currentVectorLength' determines how many elements are
         allocated. *)

end; (* sortedVector *)
```

```
dictionaryStatistics =
     object(dictionary)

  d: dictionary;
     (* Actual dictionary that is used. *)

  maxLength,
     (* Maximal number of elements in the dictionary. *)

  insertions,
     (* Number of insertions into the dictionary. *)

  deletions: longInt;
     (* Number of deletions from the dictionary. *)

  procedure init; override;
     (* The global variable 'currentDictionaryType' determines which kind of dictionary is
        used for 'd' and initializes it. Note that the initialization of 'd' might require the
        correct setting of additional global variables. *)

end; (* dictionaryStatistics *)
```

Besides the abstract data type dictionary, we support the priority queue. Again, priority queue is an abstract class realized either as a heap or a dictionary. In the first case an efficient find operation is not possible while in the second case we can guarantee a logarithmic time for find.

```
type

priorityQueue = object(obj)

  function getObject (where: reference): obj;
     (* Retrieves the object referenced by 'where'. *)

  function insert
       (x: obj;
        function compare (a, b: obj): relationObj): reference;
     (* Inserts 'x' into the priority queue using 'compare' giving a reference for
        later removal. *)

  procedure delete
       (x: reference;
        function compare (a, b: obj): relationObj);
     (* Deletes the object referenced by 'x' from the priority queue. 'compare' may be used
        for restructuring. *)

  function isEmpty: boolean;
     (* Tests whether the queue is empty. *)

  function minimum: reference;
     (* Retrieves the minimum without deleting it. *)
```

```
procedure forAll (procedure whatToDo (x: obj));
   (* Performs the procedure 'whatToDo' on all elements 'x' of the priority queue. *)

function infoString: str255;
   (* Computes information about the dictionary (this only makes sense for a
      'priorityQueueStatistics'. *)

end; (* priorityQueue *)
```

The abstract class 'priorityQueue' is realized in three different ways.

```
type
dictPriorityQueue =
     object (priorityQueue)

   (*Priority queue implemented with a dictionary with an efficient find and next operation. *)

   d: dictionary;

   procedure init; override;
      (* The global variable 'currentDictionaryType' determines which kind of dictionary is
         used for 'd' and initializes it. Note that the initialization of 'd' might require the
         correct setting of additional global variables. *)

   function find
        (x: obj; function compare (a, b: obj): relationObj;
         var where: reference;
         var direction: directionType): boolean;
      (* If 'find' = 'true' then 'where' points to some element 'e' with 'compare(e, x) =
         equalObj'. If 'find' = 'false' then either 'where' = 'nil' (dictionary empty) or 'x' is a
         direct neighbor of 'where' in direction 'direction'. *)

   function next (x: reference;
        direction: directionType): reference;
      (* Finds the predecessor ('direction' = 'left') or the successor ('direction' = 'right')
         of 'x'*)

end; (* dictPriorityQueue *)

heapPriorityQueue =
     object (priorityQueue)

   (* Implements the priority queue as a heap. *)

   procedure init; override;
      (* The global variable 'currentVectorLength' determines how many elements
         are allocated. *)

end; (* heapPriorityQueue *)
```

```
priorityQueueStatistics =
    object(priorityQueue)

  (* Instruments a priority queue. *)

  currentLength,
    (* The actual number of elements in the priority queue. *)

  maxLength,
    (* Maximal number of elements in the priority queue. *)

  insertions,
    (* Number of insertions into the priority queue. *)

  deletions: longInt;
    (* Number of deletions from the priority queue. *)

  p: priorityQueue;
    (* Actual priority queue that is used. *)

  procedure init; override;
    (* The global variable 'currentPriorityQueueType' determines which kind of dictionary
       is used for 'p' and initializes it. Note that the initialization of 'p' might require the
       correct setting of additional global variables. *)

end; (* priorityQueueStatistics *)
```

We describe in section 2.8 a convenient procedure that lets the user choose between different implementations of dictionaries and priority queues (function 'getXY').

2.7 Graphs: the classes 'graphEdge', 'simpleUndirectedGraph', 'undirectedGraph', 'directedGraph' and 'spanningTree'

The GeoBench is primarily designed for geometric computation. Nevertheless we support graphs in a limited way.

```
type

graphEdge = object(obj)

  startVertex, endVertex: seqIndex;
    (* Pointers to the corresponding vector of vertices. *)

  procedure drawEdge
      (g: simpleUndirectedGraph; directed: boolean);
    (* Displays the edge which is part of graph 'g'. 'directed' specifies whether the edge
       should be drawn as a directed edge or not. *)

end; (* graphEdge *)
```

```
simpleUndirectedGraph = object(obj)

   vertices: vector;
      (* Specifies the nodes. *)

   edges: homogenVector;
      (* A vector of 'graphEdge'. *)

   procedure init; override;
      (* The global variables 'currentVertices' and 'currentEdges' determine how many
         vertices and edges are allocated. *)

   procedure addEdge (e: graphEdge);
      (* Adds the edge 'e' to the graph. *)

   function minimumSpanningTree: spanningTree;
      (* Computes a minimum spanning tree under the assumption that the edges are sorted
         by length into ascending order. *)

   function perimeter: real;
      (* Computes the total length of all edges in the graph under the assumption that the
         vertices are points (member of the 'point2d' class). *)

end; (* simpleUndirectedGraph *)

spanningTree =
      object(simpleUndirectedGraph)

   function TSPEMST: vector;
      (* Traverses the spanning tree and produces in the Euclidean case as result a traveling
         salesman tour that is at most twice as long as the optimal tour. *)

end; (* spanningTree *)

directedGraph =
      object(simpleUndirectedGraph)

   adjacency: homogenVector;
      (* Of list of graphEdge. *)

   procedure init; override;
      (* The global variables 'currentVertices' and 'currentEdges' determine how many
         vertices and edges are allocated. *)

   procedure initAdjacency;
      (* Constructs the adjacency lists for the already existing graph. This method can be
         used to transform a simple undirected graph into a directed graph. *)

end; (* directedGraph *)
```

```
undirectedGraph = object(directedGraph)
    (* The undirected graph has the same operations as an directed graph. *)
end; (* undirectedGraph *)
```

2.8 Support for animation, user interaction and error checking

Animation

Algorithm animation is primarily used for two purposes: Demonstrating algorithms and debugging them. In order to animate an algorithm the implementor chooses a graphical representation of the program state and decides when and where this information needs to be updated. The typical code looks as follows:

```
...
(* Geometric algorithm changing internal state. *)
(*$IFC AnimationEnabled AND myAlgAnim *)
   if animationFlag[myAlgAnimItem] then
      (* Update graphical state information. Often: show a picture. *)
      waitForClick(animationFlag[myAlgAnimItem]);
      (* Update graphical state information. Often: hide a picture. *)
   end;
(*$ENDC *)
...
```

The conditional compilation variable '*AnimationEnabled*' serves as a global flag for enabling animation for the whole system while '*myAlgAnim*' is a local flag enabling or disabling animation for a specific algorithm.

The procedure 'waitForClick' stops the algorithm and lets the user choose what to do next. For the choice of 'myAlgAnimItem' we distinguish two cases: 1) An appropriate flag is already defined since there are already implementations solving the problem (e.g. 'myAlgAnimItem = AconvexHullItem', if we implement another algorithm for the convex hull) or 2) there is no such flag. In case 1) nothing needs to be done while in case 2) we define another constant, say 'myAlgAnimItem', in module 'GeoBenchUtility' and update the constant 'maxAnimation' accordingly. In procedure 'initAnimation' in module 'GeoBenchUtility' we add the line

```
menuEntry(myAlgAnimItem, 'My algorithm');
```

and algorithm animation is possible.

In addition to visually animating a program, one can give feedback on whether a program is executing or the machine hangs. This is done by advancing the hands of the stopwatch. The procedure 'advanceStopWatchHands' from the module 'GeoBenchUtility' should be called sufficiently often.

```
procedure advanceStopWatchHands;
   (* Advances the hands of the stop watch. *)
```

Parameter input

Sometimes an algorithm needs additional input from the user. We provide in the module 'getUserParameter' three different kinds of interactive input procedures.

1) Get one to three numerical values

```
function getUserParameter1
     (title: str255; default: extended; var value: extended;
      function check (x: extended): boolean): boolean;

function getUserParameter2
     (title1, title2: str255;
      default1, default2: extended;
      var value1, value2: extended;
      function check1 (x: extended): boolean;
      function check2 (x: extended): boolean): boolean;

function getUserParameter3
     (title1: str255; default1: extended;
      var value1: extended;
      function check1 (x: extended): boolean;
      title2: str255; default2: extended;
      var value2: extended;
      function check2 (x: extended): boolean;
      title3: str255; default3: extended;
      var value3: extended;
      function check3 (x: extended): boolean): boolean;
   (* The string 'title' specifies the appropriate title for the numerical value, 'default' gives
      the default value and function 'check' tests whether the user supplied value is
      acceptable. The boolean result 'true' indicates that the operation was performed
      successfully while 'false' indicates that the operation was canceled or some
      numerical value was not accepted by the 'check' function(s). *)

function noConstraint (x: extended): boolean;
   (* This function returns always 'true' and can be used if a numerical value is
      unconstrained. *)
```

2) Get one or two string values

```
function getUserString1
     (title: str255; var value: str255): boolean;
```

```
function getUserString2
     (title1, title2: str255;
      var value1, value2: str255): boolean;
```
(* The string '*title*' specifies the appropriate title for the string value. '*value*' contains at entry the default value and at exit the user supplied value. The boolean result 'true' indicates that the operation was performed successfully while 'false' indicates that the operation was canceled. *)

3) Ask the user for the appropriate data structures

```
function getXY
     (xLength: seqIndex0; mustBeDictPriorityQueue: boolean;
      var xQueue: priorityQueue;
      yLength: seqIndex0; var yTable: dictionary): boolean;
```
(* If '*xLength*' > 0 the user is asked to choose the data type of the priority queue '*xQueue*'. The value of '*xLength*' indicates the maximal number of entries in the priority queue. The value of '*mustBeDictPriorityQueue*' specifies whether the priority queue must be realized as a dictionary ('*mustBeDictPriorityQueue*' = 'true') or whether a heap implementation is admissible ('*mustBeDictPriorityQueue*' = 'false'). If '*yLength*' > 0 the user is asked to choose the data type of the dictionary '*yTable*'. The value of '*yLength*' indicates the maximal number of entries in the dictionary. '*getXY*' = 'false' ⇔ the operation was canceled by the user. *)

Display of additional information

Algorithms that need to display information (e.g. when statistics information is collected by various abstract data types) can display an info string using the procedure 'displayInfoString' from the module 'infoWindow'.

```
procedure displayInfoString (info: str255);
```
(* Displays the string '*info*' in the info window of the XYZ GeoBench. *)

Error checking

We advocate the use of assertions to check invariants. The module 'GeoBenchUtility' provides the procedure assert.

```
procedure assert (condition: boolean; t: str255);
```
(* If '*condition* = false' the user gets the warning that the assertion '*t*' has failed. The user can abort the program or continue. *)

2.9 Implemented geometric algorithms

This section describes the implemented geometric algorithms and their interfaces. Many algorithms contain a boolean parameter '*ask*' which determines whether the user is asked to choose how certain data structures are implemented at run time.

```
type
lineSegmentVector =
     object (homogenVector)

  (* A collection of line segments. *)

  (* Intersection routines *)

  function simpleFirstIntersect: obj;
    (* Computes the first intersection using the trivial method. The result can be a 'point2d'
       object or a 'lineSegment' object or 'nil' if no intersection exists. *)

  function planeSweepFirstIntersect (ask: boolean): obj;
    (* Computes the first intersection using a plane sweep. The result can be a 'point2d'
       object or a 'lineSegment' object or 'nil' if no intersection exists. *)

  function simpleAllIntersect: vector;
    (* Computes all pairwise intersections with the trivial algorithm. The result is a vector
       of points and line segments or 'nil' if no intersection exists. *)

  function planeSweepAllIntersect (ask: boolean): vector;
    (* Computes all intersections using a plane sweep algorithm. The result is a vector of
       points and line segments or 'nil' if no intersection exists. If 'ask = true' the user is
       asked to choose how the x-queue and the y-table should be implemented. *)

  function hvPlaneSweepAllIntersect (ask: boolean): vector;
    (* Like 'planeSweepAllIntersect' but all line segments must be either horizontal or
       vertical. *)

  (* Projection routines *)

  function projectOnX: lineSegmentVector;
    (* Projects all line segments on the x-axis and produces a new 'lineSegmentVector'. *)

  function projectOnY: lineSegmentVector;
    (* Projects all line segments on the y-axis and produces a new 'lineSegmentVector'. *)

  function projectOnXY: lineSegmentVector;
    (* Projects all line segments on the axis that results in a shorter segment and produces a
       new 'lineSegmentVector'. *)

  (* Miscellaneous *)

  function isHorizontalVertical: boolean;
    (* Tests whether all line segments are either horizontal or vertical. *)

  function eliminateZeroLengthSegments: lineSegmentVector;
    (* Eliminates all line segments where the start point and the end point coincide and
       produces a new 'lineSegmentVector'. *)

end; (* lineSegmentVector *)
```

```
pointVector = object(homogenVector)
```

(* **Convex hull** *)

```
procedure convexHull
      (eliminate: boolean; var hullLength: seqIndex0);
   (* Computes in place the convex hull of the given points using the Graham scan. The
      elements from 1 to 'hullLength' constitute a convex polygon whereas the elements
      from 'hullLength + 1' to 'self.length' are the inner points. *)

function convexHullDivideAndConquer: convexPolygon;
   (* Computes the convex hull using the divide and conquer algorithm. *)
```

(* **Closest pair** *)

```
function closestPairHeuristic: lineSegment;
   (* Computes the closest pair using the heuristic method which sweeps only in x-
      direction. The resulting 'lineSegment' is formed by the closest pair and might have
      zero length. The order of the input data points might change. *)

function closestPair (ask: boolean): lineSegment;
   (* Computes the closest pair using the plane sweep method. The resulting
      'lineSegment' is formed by the closest pair and might have zero length. The order of
      the input data points might change. *)

function closestPairProbabilistic
      (ask: boolean): lineSegment;
   (* Computes the closest pair using Rabin's probabilistic algorithm. The resulting
      'lineSegment' is formed by the closest pair and might have zero length. The order of
      the input data points might change. 'ask = true' asks the user for the size of the hash
      table, otherwise the size is determined by the algorithm. This is useful for
      demonstration purposes where giving a large hash table causes few or no
      collisions. *)

function closestPairNewHeuristic: lineSegment;
   (* Computes the closest pair using the heuristic method which sweeps simultaneously
      in x- and y-direction. The resulting 'lineSegment' is formed by the closest pair and
      might have zero length. The order of the input data points might change. *)

function closestPairN2: lineSegment;
   (* Computes the closest pair using the trivial quadratic method. The resulting
      'lineSegment' is formed by the closest pair and might have zero length. The order of
      the input data points does not change. *)
```

(* **All nearest neighbors to the left** *)

```
function leftANN (ask: boolean): homogenVector;
   (* Computes all nearest neighbors to the left using the plane sweep method. The result
      is a homogeneous vector of line segments where the start point is a given point and
      the end point is the start point's nearest neighbor to the left. The left most point gets
      itself as nearest neighbor to the left. The order of the input data points might
      change. *)
```

```
function leftANNHeuristic: homogenVector;
```
(* Computes all nearest neighbors to the left using the projection-on-x-only method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

```
function leftANNNewHeuristic: homogenVector;
```
(* Computes all nearest neighbors to the left using the projection method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

```
function leftANNInSector
    (ask: boolean;
     cosLower, sinLower, cosUpper, sinUpper: extended):
     homogenVector;
```
(* Computes all nearest neighbors to the left in the sector that is bounded by the rays with slope 'cosLower / sinLower' and 'cosUpper / sinUpper' using the plane sweep method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left in the specified sector. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

(* Voronoi diagram *)

```
function voronoiDiagramSweepLine
    (ask: boolean): voronoiDiagram;
```
(* Computes the Voronoi diagram using the plane sweep method. The order of the input data points does not change. *)

```
function voronoiDiagramDivideAndConquer: voronoiDiagram;
```
(* Computes the Voronoi diagram using the divide and conquer method. The order of the input data points does not change. *)

(* Euclidean minimum spanning tree *)

```
function EMST: spanningTree;
```
(* Computes a Euclidean minimum spanning tree of the point set using a quadratic algorithm. The order of the input data points does not change. *)

(* Traveling salesman *)

```
function TSPEMST: polygon2d;
```
(* Computes a tour of the traveling salesman through the given points with the Euclidean minimum spanning tree heuristic. The order of the input data points does not change. *)

```
function TSPNN: polygon2d;
```
(* Computes a tour of the traveling salesman through the given points with the nearest neighbor heuristic. The order of the input data points does not change. *)

```
function TSPConvexHull: polygon2d;
  (* Computes a tour of the traveling salesman through the given points with the convex
     hull heuristic. The order of the input data points does not change. *)
```

(* Minimal area disk *)

```
function minimalDisk: circle;
  (* Computes the smallest circle which contains the given points using a randomized
     algorithm. The order of the input data points might change. *)

function simpleMinimalDisk: circle;
  (* Computes an approximation to the smallest circle which contains the given points
     using a heuristic algorithm. The order of the input data points might change. *)

function containedInCircle (c: circle): boolean;
  (* Tests whether all the given points lie in circle 'c'. The order of the input data points
     does not change. *)
```

(* Projection *)

```
function projectOnX: pointVector;
  (* Projects the points on the x-axis and produces a new 'pointVector'. The order of the
     input data points does not change. *)

function projectOnY: pointVector;
  (* Projects the points on the y-axis and produces a new 'pointVector'. The order of the
     input data points does not change. *)
```

(* Test data generation *)

```
function createGrid (yCoordinates: pointVector): pointVector;
  (* Computes a grid G of points of size 'self.length · yCoordinates.length' where G =
     {p: ∃ u ∈ 'self.elements': ∃ v ∈ 'yCoordinates.elements': p_x = u_x ∧ p_y = v_y }. The
     order of the input data points does not change. *)

function createLineSegmentVector
    (q: pointVector): lineSegmentVector;
  (* Creates a vector of line segments where 'self' provides the starting points and 'q' the
     end points. Both must have the same number of elements. The order of the input
     data points does not change. *)
```

(* Conversions *)

```
function makePolygon: polygon2d;
  (* Produces a polygon from the given points. The order of the input data points does
     not change. *)

function makePolyLine: polyLine;
  (* Produces a poly line from the given points. The order of the input data points does
     not change. *)
```

```
function makeSimplePolygon: polygon2d;
   (* Produces a simple polygon using a modification of Graham's scan. The order of the
      input data points does not change. *)

function makeStarShapedPolygon: polygon2d;
   (* Produces a star-shaped polygon. A rotational sweep around the center of gravity of
      the first three points in the input 'pointVector' 'self' is used. The order of the input
      data points does not change. *)

end; (* pointVector *)

polyLine = object(pointVector)

function toLineSegmentVector: lineSegmentVector;
   (* Computes a vector of line segments that corresponds to the edges of the poly line or
      polygon. *)

function perimeter: real;
   (* Computes the sum of the length of all edges. *)

end; (* polyLine *)


polRange = 0..maxPolysM1;
polSet   = set of polRange;

polygon2d = object(polyLine)

function windingNumber
      (r: point2d; var onBoundary: boolean): longInt;
   (* Computes the winding number of 'r' around the polygon. 'onBoundary = true' ⇔
      the point 'r' lies on the polygon's boundary. *)

function TSPOptimize
      (whichN: longInt; ask: boolean): polygon2d;
   (* Tries to shorten a given traveling salesman tour. The parameters 'whichN' and 'ask'
      determine what kind of optimization is tried. The reader is referred to the source code
      for their precise meaning. *)

function clip (ls: lineSegment): polygon2d;
   (* Clips 'self' on the given lineSegment. Everything on the right side (the line segment
      'ls' itself excluded) of the directed line segment 'ls' is assumed to be visible.
      Algorithm: Sutherland-Hodgman. *)
```

```
procedure intersection
      (pols: sequence; function zoneQ (s: polSet): boolean;
       result: list; ask: boolean);
   (* Computes boolean operations on polygons using the sweepline algorithm of
      Nievergelt and Preparata and adds the resulting polygonal pieces to the list 'result'.
      'pols' is a sequence of polygons enumerated from 1 to 'pols.length'. The function
      'zoneQ' determines which areas belong to the result. To get the union of all polygons
      use 'zoneQ := polSet ≠ []'. To get the intersection, use
      'zoneQ := polSet = [1..pols.length]'. In particular, a single polygon can be
      decomposed into simple parts by using the function 'zoneQ := polSet ≠ []' (menu
      entry 'Decompose'). To divide a polygon into its 'simple hull' and zero or more
      simple polygons inside, use 'zoneQ := polSet = []' (menu entry 'Simplify'). *)

function simpleSelfIntersect: vector;
   (* Computes all intersections of edges of a polygon that do not occur at vertices using
      the trivial algorithm. *)

function selfIntersect: vector;
   (* Computes all intersections of edges of a polygon that do not occur at vertices using a
      boundary traversal algorithm. This algorithm is efficient for the class of polygons
      that have a left turn in each vertex. *)

end; (* polygon2d *)

convexPolygon = object(polygon2d)

function intersect (x: convexPolygon): convexPolygon;
   (* Computes the intersection of the two convex polygons using the boundary traversal
      method. *)

procedure tangent
      (c: convexPolygon;
       var upperP, upperQ, lowerP, lowerQ: seqIndex);
   (* Computes the two common outer tangents of the two convex polygons. The lower
      tangent is given by index 'lowerP' in 'self' and 'lowerQ' in 'c' and the upper
      tangent is given by index 'upperP' in 'self' and 'upperQ' in 'c'. *)

procedure diameter (var i, j: seqIndex0);
   (* Computes the indices 'i' and 'j' of the two points determining the diameter of the
      convex polygon. *)

function inside (p: point2d): boolean;
   (* Tests whether point 'p' is inside the convex polygon or not. The boundary belongs
      per definition to the inside. *)

end; (* convexPolygon *)
```

```
voronoiDiagram = object(obj)

  function delaunayTriangulation: simpleUndirectedGraph;
    (* Computes the Delaunay triangulation, the dual to the Voronoi diagram. *)

  function EMST: spanningTree;
    (* Computes a Euclidean minimum spanning tree from the Voronoi Diagram. *)

  function leftANN: homogenVector;
    (* Computes a vector of line segments such that each point of the Voronoi diagram
       occurs as a start point of a segment and the end point of the segment is its nearest
       neighbor to the left. The leftmost point does not occur in this collection (in contrast
       to the nearest neighbor to the left algorithms that work directly on point sets. *)

end; (* voronoiDiagram *)

rectangleVector = object(homogenVector)

  function boundingBox: rectangle;
    (* Computes the smallest rectangle that contains all the given rectangles. *)

  function contourOfUnionOfRectangles: lineSegmentVector;
    (* Computes a set of horizontal and vertical line segments that determines the contour
       of the given rectangles using a plane sweep algorithm. *)

end; (* rectangleVector *)

dDimPoint = object(obj)

  procedure randomChange; override;
    (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

  function distance (p: dDimPoint): extended;
    (* Computes the Euclidean distance between the d-dimensional points 'self' and 'p'. *)

end; (* dDimPoint *)

dDimCircle = object(obj)

  procedure randomChange; override;
    (* The global variable 'currentRandomConstraint' is interpreted as a rectangle. *)

  function inCircle (p: dDimPoint): boolean;
    (* Tests whether the d-dimensional point 'p' is inside the d-dimensional circle 'self'. *)

  function inCircleEps
       (p: dDimPoint; tolerance: extended): boolean;
    (* Tests whether the d-dimensional point 'p' is in the d-dimensional circle 'self' whose
       radius is enlarged by a factor of (1 + 'tolerance'). *)
```

```
procedure makeDisk (l: list);
   (* Creates the smallest disk in d-space having the d-space points from 'l' on the
      boundary. *)

procedure makeFromPointCircle (p: dDimPoint; c: dDimCircle);
   (* Creates the smallest circle through 'p' and 'c' with 'not inCircle(p)'. *)

function randomPoint: dDimPoint;
   (* Creates a uniformly distributed random point inside the disk. *)

end; (* dDimCircle *)

dDimPointVector = object (homogenVector)

function minimalDisk
      (tolerance: extended; initialize: boolean): dDimCircle;
   (* Computes the minimal area disk that contains the given points. The radius is correct
      within a factor of (1 + 'tolerance'), i.e. the radius is exact if 'tolerance = 0'. If
      'initialize = true' the algorithm is initialized with the pair of points that have the
      largest coordinate difference in any direction. *)

function simpleMinimalDisk
      (shuffle, initialize: boolean): dDimCircle;
   (* Computes a disk containing all given points using a heuristic that is guaranteed to
      produce a circle whose radius is at most twice as large as the radius of the minimal
      area disk. If 'shuffle = true' the points are randomly shuffled before the algorithm
      starts since it depends of the order of the input data points. The parameter 'tolerance'
      has the same meaning as in the previous method. *)

function worstSimpleMinimalDisk: dDimCircle;
   (* Rearranges the points in such a way that the worst case order for the previous
      enclosing disk algorithm is achieved. This algorithm has running time proportional
      to the factorial of the number of given points. *)

function containedInCircle (c: dDimCircle): boolean;
   (* Tests whether all given points lie inside the d-dimensional circle 'c'. *)

function projectOnXY: pointVector;
   (* Projects the given points on the x-y-plane and produces a collection of 2-dimensional
      points (type 'point2d'). *)

end; (* dDimPointVector *)
```

Appendix A: Syntax of the textual I/O format

Axiom of the grammar is 'List', i.e. GeoBench expects a list of objects as textual input.

```
Digit                 = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Natural               = Digit { Digit }
Integer               = [ '-' ] Natural
Real                  = Integer [ '.' Natural ] [ 'E' [ '+' | '-' ] Natural ]
Float                 = Natural Natural Real
                        (* base precision value *)
String                = '"' Char { Char } '"'
                        (* Char ≠ '"' *)

Circle                = '(' 'CIR'    Real Real Real                    ')'
                                     (* x y radius *)
ConvexPolygon         = '(' 'CPL'    PointList                         ')'
ColorQuickDrawPicture = '(' 'CQP'    Real Real Real Real               ')'
                                     (* left bottom right top *)
DDimCircle            = '(' 'DCI'    CoordinateList Real               ')'
                                     (* coordinates radius *)
DDimPoint             = '(' 'DPT'    CoordinateList                    ')'
                                     (* coordinates *)
DDimPointVector       = '(' 'DPV'    DDimPointList                     ')'
DirectedGraph         = '(' 'DGR'    Vector HomogenVector              ')'
                                     (* vertices edges *)
FloatPoint            = '(' 'FPT'    Float Float                       ')'
                                     (* x y *)
GraphEdge             = '(' 'GED'    Integer Integer                   ')'
                                     (* startVertex endVertex *)
HomogenVector         = '(' 'HVC'    HomogenObjectList                 ')'
Int                   = '(' 'INT'    Integer                           ')'
IntegerPoint          = '(' 'IPT'    Integer Integer                   ')'
                                     (* x y *)
Layer                 = '(' 'LAY'    Real Real Vector                  ')'
                                     (* z thickness objects *)
LayerVector           = '(' 'LAV'    Real Vector                       ')'
                                     (* dz objects *)
LineSegment           = '(' 'LSG'    Point Point                       ')'
LineSegmentVector     = '(' 'LVC'    LineSegmentList                   ')'
List                  = '(' 'LST'    ObjectList                        ')'
MarkedRealPoint       = '(' 'MRP'    Real Real Str                     ')'
                                     (* x y mark *)
Polygon               = '(' 'POL'    PointList                         ')'
PolyLayer             = '(' 'PLA'    Real Real Vector                  ')'
                                     (* z thickness objects *)
PolyLayerVector       = '(' 'PLV'    Real Vector                       ')'
                                     (* dz objects *)
PolyLine              = '(' 'PLI'    PointList                         ')'
PointVector           = '(' 'PVC'    PointList                         ')'
QuickDrawPicture      = '(' 'QDP'    Real Real Real Real               ')'
                                     (* left bottom right top *)
RealPoint             = '(' 'RPT'    Real Real                         ')'
                                     (* x y *)
Rectangle             = '(' 'REC'    Real Real Real Real               ')'
                                     (* left bottom right top *)
```

```
RectangleVector          = '(' 'RVC'   RectangleList                          ')'
SimpleUndirectedGraph    = '(' 'SUG'   VectorHomogenVector                    ')'
                                       (* vertices edges *)
SpanningTree             = '(' 'SPT'   VectorHomogenVector                    ')'
                                       (* vertices edges *)
Spline2                  = '(' 'SP2'   RealPoint RealPoint                    ')'
Splinegon                = '(' 'SPL'   ObjectList                             ')'
Str                      = '(' 'STR'   String                                 ')'
StraightEdge             = '(' 'SED'   RealPoint RealPoint                    ')'
UndirectedGraph          = '(' 'UDG'   VectorHomogenVector                    ')'
                                       (* vertices edges *)
Vector                   = '(' 'VEC'   ObjectList                             ')'
VoronoiEdge              = '(' 'VED'   Integer Point Point [ Point ] [ Point ] ')'
                                       (* edgeType p1 p2 v1 v2 *)
VoronoiDiagram           = '(' 'VDG'   PointVectorListList                    ')'
                                       (* points edges neighbors *)

CoordinateList           = Integer { Integer }
                           (* dimension coordinates *)
DDimPointList            = Integer { DDimPoint }
                           (* length elements *)
HomogenObjectList        = Integer { Object }
                           (* length elements *)
LineSegmentList          = Integer { LineSegment }
                           (* length elements *)
ObjectList               = Integer { Object }
                           (* length elements *)
Point                    = RealPoint | FloatPoint | IntegerPoint
PointList                = Integer ( { RealPoint } | { FloatPoint } | { IntegerPoint } )
                           (* length elements *)
RectangleList            = Integer { Rectangle }
                           (* length elements *)

Object                   = Circle | ConvexPolygon | ColorQuickDrawPicture | DDimCircle |
                           DDimPoint | DDimPointVector | DirectedGraph | FloatPoint |
                           GraphEdge | HomogenVector | Int | IntegerPoint | Layer |
                           LayerVector | LineSegment | LineSegmentVector | List |
                           MarkedRealPoint | Polygon | PolyLayer | PolyLayerVector |
                           PolyLine | PointVector | QuickDrawPicture | RealPoint |
                           Rectangle | RectangleVector | SimpleUndirectedGraph |
                           SpanningTree | Spline2 | Splinegon | Str | StraightEdge |
                           UndirectedGraph | Vector | VoronoiEdge | VoronoiDiagram
```

Appendix B: Changes to TransSkel V2.02 in GeoBench

The Version of TransSkel V2.02 [DB 89] used by GeoBench was extended as follows:

1. Hierarchical Menus

```
function SkelMenu
   (theMenu: MenuHandle;
    pSelect, pClobber: ProcPtr;
    drawBar, hierarchical: boolean): boolean;
```

If *'hierarchical'* is false, the function behaves as in the original TransSkel. If *'hierarchical'* is true, the menu is inserted in the hierarchical portion of the menu list. These menus can be used as hierarchical or pop-up menus (the *'beforeID'* parameter to *'InsertMenu'* is -1, see Inside Macintosh, Vol. V, p. 236).

2. Delayed scrap operations

```
procedure SkelApple
   (aboutTitle: str255;
    aboutProc, deskAccProc,   resumeProc: ProcPtr);
```

The two new procedure parameters serve to control delayed execution of the standard *Cut* and *Copy* commands of the *Edit* menu. If the two new parameters are **'nil'**, the behavior is as in the original version of TransSkel. The *'deskAccProc'* of the form

```
procedure myDeskAccOpenHandler;
```

is called whenever a desk accessory is called under the old finder. The internal scrapbook should be written to the external scrapbook in this case (In older Mac OS versions this procedure is also to inform when the user switched applications in multifinder).

The *'resumeProc'* of the form

```
procedure myResumeHandle (resume: boolean);
```

is called whenever another application is activated under System 7. The *'resume'* flag tells you whether your application has been resumed or suspended. The flag 'acceptSuspendResumeEvents' of the 'SIZE' resource should be set in order to get the corresponding events from the system. See Inside Macintosh Vol. VI, p. 5-14 and 5-19 for more information.
The procedures 'SkelSetWindResume' and 'SkelGetWindowResume' have been

removed because they were buggy and because the suspend / resume event is actually not window-related.

3. Window growing

```
function SkelWindow
   (theWind: WindowPtr;
    pMouse, pKey, pUpdate, pGrow, pActivate,
    pClose, pClobber, pIdle: ProcPtr;
    frontOnly: boolean): boolean;
```

The procedure *'pGrow'* is called after the user has resized the window. The boolean parameter to the *'pUpdate'* procedure has been removed. ***Warning***: Do ***not*** use update procedures which still have this parameter! Annoying, unrelated crashes are the result. Instead, use procedures of the following style:

```
procedure myUpdateHandler;
procedure myGrowHandler;
```

In the original version of TransSkel, the whole window was redrawn after resizing a window. Now, only the part which was not visible before sizing is redrawn. To have the whole window redrawn, just insert the statement 'InvalRect(thePort^.portRect)' in your grow handler.

References

[Brü 91] A. Brüngger: Schichtenmodelle in der XYZ GeoBench, Diploma Thesis, ETH Zürich, February 1991.

[DB 89] TransSkel version 2.02 – Transportable application skeleton, written by Paul DuBois, Wisconsin Regional Primate Research Center, 1220 Capital Court, Madison WI 53706 USA, e-mail: dubois@rhesus.primate.wisc.edu

[NB 91] J. Nievergelt, P. Schorn, M. De Lorenzi, C. Ammann, A. Brüngger: XYZ: A project in experimental geometric computation, submitted, May 1991.

[S 91a] P. Schorn: Robust Algorithms in a Program Library for Geometric Computation, ETH PhD Dissertation 9519, 1991.

[S 91b] P. Schorn: The XYZ GeoBench: A programming environment for geometric algorithms, submitted, May 1991.

Gelbe Berichte des Departements Informatik

145	J. Mössenböck	She: A Simple Hypertext Editor for Programs (vergriffen)
146	H. E. Meier	Schriftgestaltung mit Hilfe des Computers Typographische Grundregeln (vergriffen)
147	G. Weikum, P. Zabback, P. Scheuermann	Dynamic File Allocation in Disk Arrays (vergriffen)
148	D. Degiorgi	A New Linear Algorithm to Detect a Line Graph and Output its Root Graph
149	A. Moenkeberg, G. Weikum	Conflict-Driven Load Control for the Avoidance of Data-Contention Thrashing
150	M.H. Scholl, Ch. Laasch M. Tresch	Updatable Views in Object-Oriented Databases (vergriffen)
151	C. Szyperski	Write - An extensible Text Editor for the Oberon System
152	M. Bronstein	On Solutions of Linear Ordinary Difierential Equations in their Coefficient Field
153	G.H. Gonnet, D.W. Gruntz	Algebraic Manipulation: Systems (vergriffen)
154	G.H. Gonnet, St.A. Benner	Computational Biochemistry Research at ETH
155	D. Crippa	A Special Case of the Dynamization Problem for Least Cost Paths
156	R. Griesemer	On the Linearization of Graphs and Writing Symbol Files
	C. Pfister (ed.), B. Heeb, J. Templ	Oberon Technical Notes
157	T. Weibel, G. Gonnet	An Algebra of Properties
158	M. Scholl (ed.)	Grundlagen von Datenbanken (Kurfassungen des 3.GI-Workshops, Volkse, 21. - 24.5.91)
159	K. Gates	Using Inverse Iteration to Improve the Divide and Conquer Algorithm
160	H. Mössenböck	Differences between Oberon and Oberon-2 The programming Language Oberon-2
161	S. Lalis	XNet: Supporting Distributed Programming in the Oberon Environment
162	G. Weikum, C. Hasse	Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism