

**A WORKBENCH FOR
COMPUTATIONAL GEOMETRY
(WOCG)**

P. Epstein, A. Knight, J. May, T. Nguyen,
and J.-R. Sack

SCS-TR-180, SEPTEMBER 1990

School of Computer Science, Carleton University
Ottawa, Canada, K1S 5B6

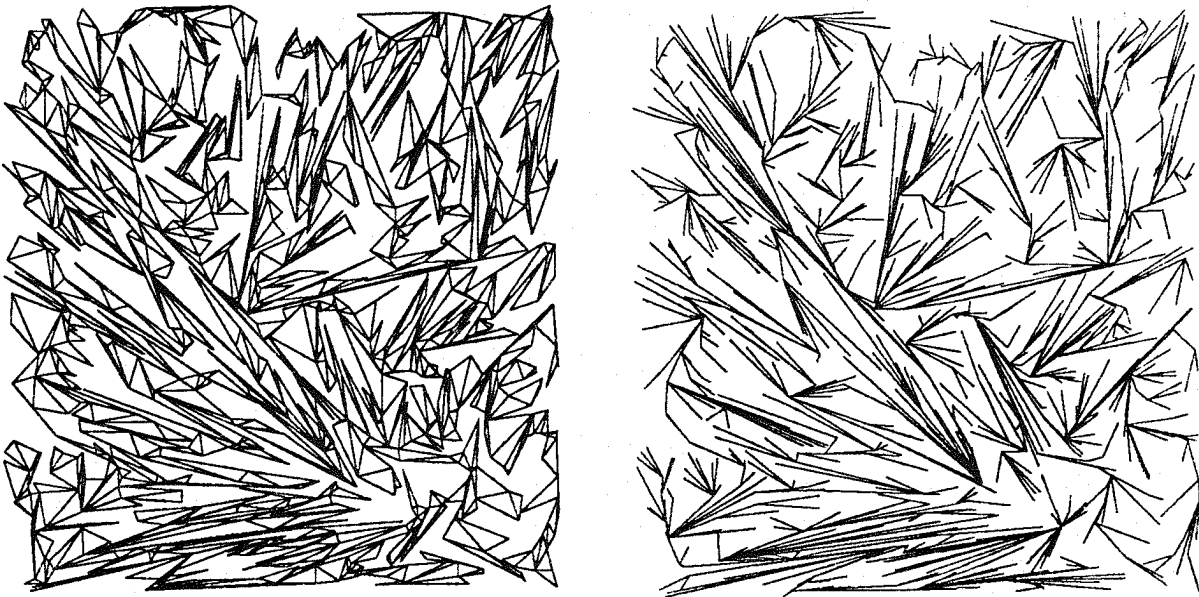
A WORKBENCH FOR COMPUTATIONAL GEOMETRY (WOCG)*

P. EPSTEIN, A. KNIGHT⁺, J. MAY, T. NGUYEN, AND
J.-R. SACK⁺

*School of Computer Science,
Carleton University, Ottawa, Canada K1S 5B6*

Abstract.

We describe the design and implementation of a workbench for computational geometry (WOCG). We discuss issues arising from this implementation, including comparison of different algorithms. The workbench is not just a library of computational geometry operations and algorithms, but is designed as a geometrical programming environment, providing tools for: creating, editing, and manipulating geometric objects; demonstrating and animating geometric algorithms; and most importantly, for implementing new algorithms. In particular, automatic garbage collection, high-level debugging facilities, and control mechanisms are provided.



A triangulated 1000 vertex polygon and the shortest path tree from a query point, generated by the workbench.

* Research partially supported by the Natural Sciences and Engineering Research Council of Canada and Carleton University.

⁺ Research was carried out in part while the authors were at the University of Passau, Germany.

INTRODUCTION

MOTIVATION

During the past decade, algorithms have been developed for solving a wide spectrum of problems in computational geometry. Research activities have focused mainly on the design and analysis of geometric algorithms. However, many of these algorithms have never been implemented. Those that have been implemented have often been treated in isolation. Constant factors, numerical stability, conversion between representations, and other implementation concerns are sometimes discussed in algorithm descriptions but the results of actual implementations are seldom made available.

As a consequence of these and other factors, much of this large body of knowledge remains inaccessible to application-oriented researchers from within the computational geometry community and from areas such as computer graphics, robotics, image processing and pattern recognition which have been the source of many geometric problems.

Forrest emphasizes the need for a "geometric computing environment" [F] which can provide a unified framework and a suite of tools to reduce difficulties in the implementation, application, and evaluation of geometric algorithms. There is also a need expressed from within the computational geometry community also for more direct comparisons of algorithms in terms not only of worst-case complexity, but of execution time and space (including constant factors), ease of implementation, robustness, handling of special cases, numerical stability and average-case performance. Information on these characteristics is important in assessing an algorithm, particularly to the implementation-oriented researcher.

We describe here the design-goals, concepts, and some implementation aspects of the geometric computing environment we have developed, and which we refer to as the Workbench for Computational Geometry (WOCG). We summarize the design and some major features, illustrate the system in operation, discuss issues arising from this implementation-oriented work, and mention empirical comparison of implemented algorithms performed using the system.

HIGH-LEVEL DESCRIPTION

For portability, the user interface and other machine-dependent components of the system are isolated as much as possible from the algorithmic components. In this paper, we will focus on the algorithmic components, illustrating aspects of the user interface briefly.

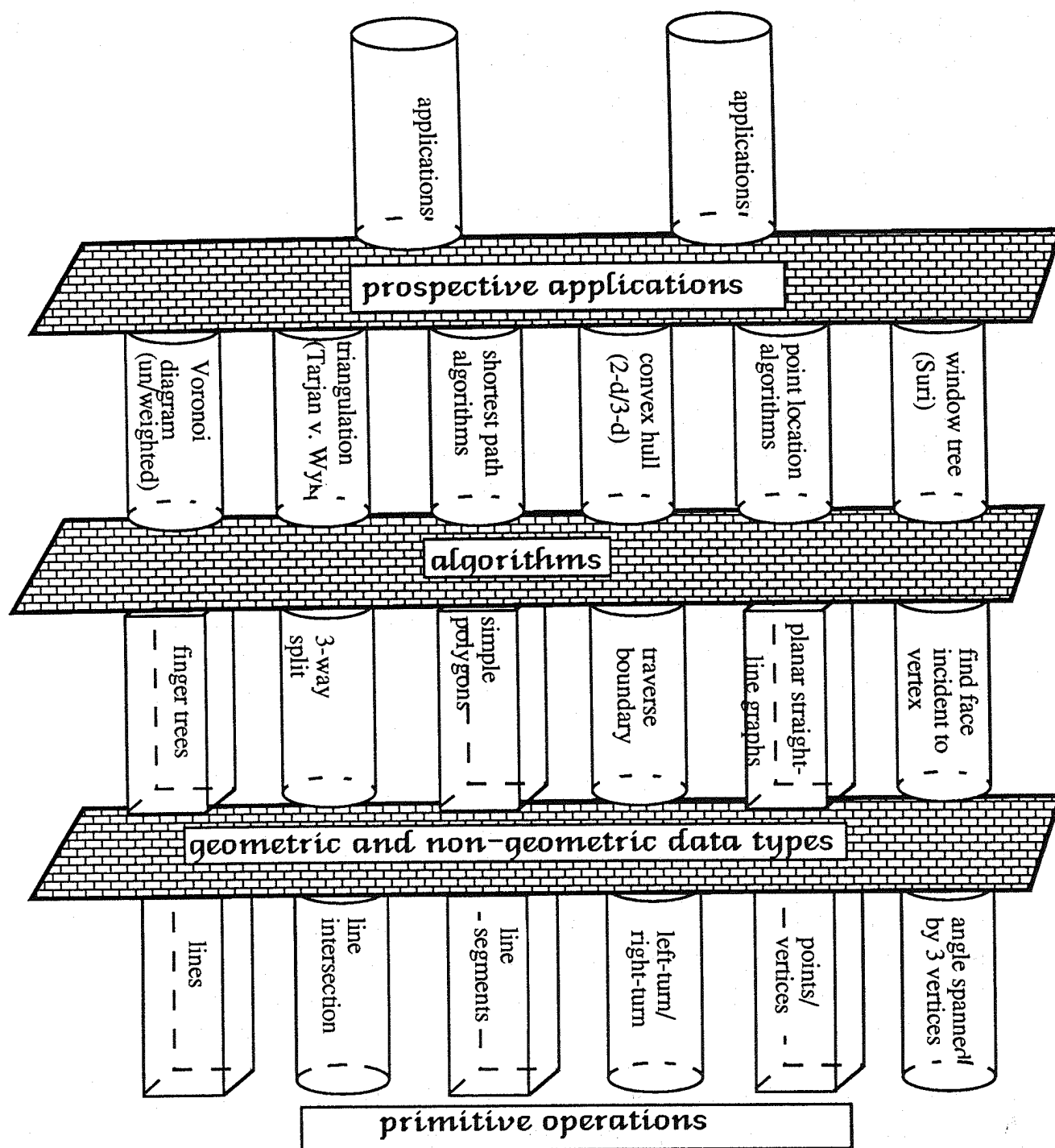



Figure 1: illustrated are the four layers of WOCG with a few sample operations, algorithms and data types. (Operations and data types are given as below:)

 <operations/algorithms>

 <data types>

The algorithmic part of the workbench provides several layers of support, covering a wide range of complexity (see Figure 1). In the uppermost layers are applications and complex algorithms, making use of other geometric algorithms and data types as "building blocks". These lower-level structures in turn make use of the elementary operations and data types which form the base layer.

In the workbench, "base layer" operations are typically simple and require constant time. Examples of such operations are functions to determine the angle spanned between three co-planar points and to detect and/or compute the intersection of two line segments. The geometric data types at this level are the primitives from which more complex objects may be built, and include, (in 2 and 3-d) points, line segments, and lines.

On the second layer are located more complex structures and operations. Operations here will often make use of lower-level primitives and require more than constant time. At this level, the operations are often subordinate to the complex data types and data structures which appear. Many such operations combine to provide the functions of a single abstract data type.

Data types at this layer include the standard non-geometric types, such as stacks, priority queues and dictionaries. For representation, these may require the availability of balanced search tree structures and their attendant operations, also at this level. Most importantly, this layer contains geometric data structures (e.g. segment trees and range trees), and data types, including graphs, planar subdivisions, polygons, and polyhedra.

An important design criteria, which will be discussed in more detail below, is the generality of operations and data types. It is extremely important for maintenance to avoid the proliferation of variations on common data structures. We attempt to do this by providing structures which are as general as possible, and are able to accommodate most variations without change. As an example, a balanced search tree should be able to accommodate any type of keys for which a total ordering is defined. This may include points, line segments, or even different types within the same tree (e.g. a tree of trapezoids with point queries).

The third layer contains algorithms, operating on the geometric data types defined in the second layer, and covering a wide range of complexities. Naturally, this separation into layers is not strict, and complex algorithms at this layer may make use of several simpler algorithms on the same layer. As an example, finding the visibility polygon from a point inside a simple polygon makes use of triangulation, itself a complex algorithm. Operations such as finding the convex hull or the Voronoi diagram are fundamental to many algorithms.

The fourth layer (currently under investigation) will contain "real" applications from outside the immediate area of computational geometry using the available algorithms on the third level.

All of these layers contain code which is independent of the particular machine and environment. Separated from this is the user-interface component of the system. A similar hierarchy might be seen in the organization of this component, with the lowest layer containing graphics primitives and the interface to the MacIntosh operating system, a second layer containing routines for editing and display of geometric objects, and third containing the algorithm animation facilities and graphical editor which are the aspect of the system most visible to the user.

DESIGN GOALS

The design phase of a system such as WOCG is crucial. At the highest level of abstraction, the system must be "useful"; this simple term encompasses many other design criteria discussed in more detail below. For example, if the system is to provide a set of tools, then these tools should be as general and as re-usable as possible. Unnecessary restrictions should be avoided, and code should be made flexible enough to allow a wide variety of uses. The system must also cater to different users who have different interests, knowledge and demands. Finally, the system should not be tied too closely to any particular operating environment.

Utility

As mentioned above, utility/usefulness is in some sense the paramount design goal for any system. We make this more precise and define a *computational geometry workbench* as a system which can provide the following:

- (1) representations of (at least) 2 and 3-dimensional geometric objects (polygons, line segments, polyhedra, etc.) with their respective operations.
- (2) geometric data types and data structures (segment tree, range tree, etc.)
- (3) non-geometric data types and data structures (priority queues, dictionaries; 2-3 trees, finger trees, splay trees, heaps, etc.)
- (4) algorithmic tools, correctly implemented, running in the specified time and space complexities, and in as general a form as possible.
- (5) numerical support. The system should be able to handle not just integer and floating point arithmetic, but also arbitrary precision and rational arithmetic, as well as infinite quantities, transparently to the user.

- (6) graphical support, including graphical display, creation, and editing of geometric objects, and animation of algorithms.

This is an important facility for a geometric environment, since geometry is a visual discipline. The ability to quickly and easily create and edit examples is extremely valuable, and algorithm animation can be an important tool for debugging, presentation and understanding of algorithms.

- (7) a good programming environment and debugging facility

A geometric workbench should allow the programmer to concentrate on geometric issues. The programmer should not be forced to deal with low-level issues, or to work without adequate support. Tools provided should include source-level debugging and allow access to the graphical representations of objects from the debugger. Graphical support as described above is important, as are tools to assist in storage management and reclamation, code browsing, and version control. It should be possible to store geometric objects to and retrieve them from secondary storage, preferably in a form which is also human-readable. Naturally, these tools must be well-integrated with each other, providing as congenial an environment as possible for geometric computing.

The workbench for computational geometry (WOCG) is implemented in an object-oriented style and environment. This is a style of programming which has proven very appropriate for graphics-oriented applications (see e.g.[G]); particularly for algorithms which operate on geometric objects.

Many concepts of computational geometry fit well into an object-oriented model. Classes of geometric objects are well defined, with a common set of operations (e.g. convex hull, triangulate) which can be applied to many such classes. There often exist natural subclassing relationships (e.g. MonotonePolygon is a subclass of SimplePolygon is a subclass of Polygon).

The workbench is implemented in Smalltalk/V, a dialect of Smalltalk similar to, but distinct from, Smalltalk-80. In addition to the advantages of object-oriented programming described above, this offers a sophisticated programming environment, a large, mature, and well-integrated library which includes source code for the windowing environment, good reflective capabilities, weak typing of variables (allowing greater flexibility of code), and automatic storage management. It has disadvantages in terms of execution time and space requirements relative to some other languages, but these were judged to be less important than the advantages in programmer productivity for a large research project with relatively few programmers.

Reusability and Generality

The tools provided by a computational geometry workbench may be used for many different purposes, some far beyond the scope of their original conception. Further, many geometric algorithms require variations on standard data types and methods. Thus the tools provided should be as general and as flexible as possible, and they should be able to accommodate such variations within a single structure. Without this generality, many different versions of what is fundamentally one data type will need to be created and updated separately, causing duplication of code, confusion for the user, and difficulties in maintenance and version control. By way of illustration we provide three specific examples from the workbench. The first deals with the dictionary data type using comparable elements, commonly implemented using a balanced tree structure. The second concerns the general method of plane sweep, and the last illustrates the storage of additional information required for specific algorithms.

Generality of Data Types

A very common data type in computational geometry is the dictionary; it supports the operations:

- **Insert**(Key, Value);
- **Search**(Key);
- **Delete**(Key);

A data structure (e.g. a tree) which implements this data type must be able to provide such operation for arbitrary comparable key values. A dictionary may be required to contain points, line segments, trapezoids, polygonal chains, or even more complicated structures, perhaps simultaneously. The underlying data structure must likewise be able to contain these objects.

Furthermore, such a data structure must support the use of different comparison operators. Even in a tree containing only points, there exist at least three different, commonly used sort orders (horizontal, vertical, circular). In the Fournier and Montuno plane-sweep algorithm for triangulation of a simple polygon, the sweep structure is a balanced tree of trapezoids. However, in addition to insertion and deletion of trapezoids, the structure must respond to point location queries.

A fully general tree structure must support not only the use of arbitrary objects as keys and values but also the use of arbitrary functions for comparison of elements (including equality comparison).

The workbench implementation supports this general dictionary data type by performing all comparisons and equality tests using functions which can be overridden by the user, and by allowing any object to be used as a key or value.

It is also important that data types and their underlying representation be separated. It is tempting, given a data structure such as a splay tree, simply to provide it with dictionary operations and to use it in place of a dictionary. We believe that this is inappropriate and can lead to confusion and difficulties in maintenance. This is particularly true when, as in the case of the splay tree, multiple data types can be represented using the same structure. For example, in the Tarjan and van Wyk triangulation algorithm, finger trees (or splay trees) are used as a dictionary type (representing family lists). They are also used as an ordered collection type (representing the boundary of the polygon). The classes of operations which are used are distinct. In the first case they are **Insert**, **Delete**, **Find**, etc.; in the second case they are **Append**, **RemoveLast**, **AtIndex**, etc. Combining these classes will result in an inconsistent data structure and errors.

In order to keep data types unambiguously separated, the workbench defines classes to represent each data type with its implementation. Thus there exist classes `PriorityQueueSplayTree`, `PriorityQueueHeap`, `DictionarySplayTree`, and so on. These data types provide the mapping between the data type operations and the operations of the underlying representation, and restrict the operations available so that the data structure is always used in a consistent fashion.

Generality of Methods (Paradigms)

A common problem-solving technique in computational geometry is the plane-sweep method. In this paradigm, a line sweeps across the plane, stopping at "event-points".

"The intersection of the sweep-line with the problem data contains all the relevant information for the continuation of the sweep. Two basic data structures are used:

- 1) The event-point schedule, which is a sequence of abscissae ... which define the halting positions of the sweep-line.
- 2) The sweep-line status, which ... contains the information that is relevant to the specific application. The sweep-line status is updated at each event-point, and a suitable data structure must be chosen in each case" - [PS].

The workbench includes a general mechanism to support plane sweep operations. The implementor of a plane sweep using this mechanism provides:

- 1) The sweep ordering, which allows any direction of sweep, including circular.
- 2) A sweep status structure.
- 3) Functions for:
 - initialization of structures
 - a discriminant function for event-points (e.g. is this point the start or end of a line segment)
 - actions to be performed at each event-point, based on the result of the discriminant function (e.g. test for intersection)
 - update of the sweep status structure, also based on the discriminant function (e.g. insert or delete the line segment corresponding to this event-point)

A simple example of how a plane-sweep might be used is in solving the interval overlap detection problem. In this problem we are given a set of intervals on a line and are to determine if any pair of intervals overlaps. The following code (See Figure 2) solves this simple problem, illustrating the general mechanism by which plane sweeps may be conducted in the workbench.

```

testIntervalOverlap: aCollectionOfIntervals relativeTo: aVector

"Given a collection of intervals, to a plane sweep to determine if they
overlap in the direction specified by aVector. If they do, return the first pair of
overlapping intervals encountered, otherwise return nil"

|sweeper schedule|

sweeper := PlaneSweeper new.
schedule := OrderedCollection new.
aCollectionOfIntervals do: [:eachInterval|
    schedule add: (Vertex eventPointAt: eachInterval start from: eachInterval).
    schedule add: (Vertex eventPointAt: eachInterval end from: eachInterval)].

"Sort with respect to the given vector"
sweeper schedule: (SortedCollection sortBlock:
    [:a :b| (a dotProduct: aVector) <= (b dotProduct: aVector)]).
sweeper schedule addAll: schedule.

"The status is a single location"
sweeper status: (Array new: 1).

"Characterize each event point as the start or end of an interval"
sweeper discriminant: [:eventPoint|
    (eventPoint = eventPoint owner start) ifTrue: [#start]
    ifFalse: [#end]].

"If an event point is the start of an interval and the status already
contains something, then terminate, returning both intervals"
sweeper action: [:eventPoint :planeSweeper|
    (eventPoint type = #start) ifTrue: [
        sweeper status first isNil ifFalse: [
            sweeper result: (Array with: eventPoint owner copy with: sweeper status first copy).
            sweeper abortSweep]]].

"If an event point is the start of an interval, then put it in the status,
otherwise remove it from the status"
sweeper update: [:eventPoint :planeSweeper|
    (eventPoint type = #start) ifTrue: [
        sweeper status at: 1 put: eventPoint owner]
    ifFalse: [
        sweeper status at: 1 put: nil]].

"Perform the sweep and return the result"
sweeper sweep.
^sweeper result.

```

Figure 2: A simple illustration of the plane-sweep paradigm implementation used for detecting overlap between a pair of intervals in a set.

This mechanism has been used to implement several algorithms, including: the line segment intersection test of Hoey and Shamos; the regularization of a planar straight line

graph of Garey, Johnson, Preparata, and Tarjan; and the Fournier and Montuno algorithm for triangulation of a simple polygon.

Storage of additional information

Many algorithms, while using standard structures, also require the storage of additional information. For example, an algorithm may require that each vertex of a polygon store a pointer to the corresponding vertex of a planar straight line graph which represents the polygon. In the plane sweep paradigm described above, event-points often store the structure which contains them (line segment, polygon, etc.).

This can also lead to a situation in which many different versions of similar data structures exist, with the same difficulties described above. It can also give rise to excessive complexity and overhead for algorithms which do not require it, as one is tempted to implement as many contingencies as possible in the basic types.

The standard object-oriented approach would resolve this by subclassing. This option is available, the workbench also provides the option of using "labellings". Any geometric object can be labelled with information in addition to what is normally stored. Labels are stored in a dictionary, using the name of the label as a key. As long as the total number of labels remains bounded by a constant, this requires only a constant amount of extra time for retrieval. Responsibility for the creation and deletion of labels lies with the algorithm which uses them, and thus labels which are needed for only a short time or in limited circumstances do not affect the class adversely.

Code which does not require the use of a label need not consider, and space is required only for those labels which are currently in use. While subclassing can be used for more permanent or frequently used data, this provides a valuable alternative.

Portability

Portability is a very desirable feature in a geometric computing environment. To some extent, however, the need for portability conflicts with the need for sophisticated graphical support and an interface to system facilities. As windowing environments and graphics file formats are not standardized, a system which uses such features must be to some extent non-portable.

The approach taken in the workbench is to isolate the system-dependent code as much as possible from the algorithmic and data structures code. All geometric objects must implement a specific set of messages, which are used by the geometric object browser and other user interface tools to manipulate these objects.

These operations include displaying and moving the object as well as adding, moving, and deleting components. The operations are sufficiently general that the user interface does not need to know the type of geometric object which it is manipulating.

Further, it is worth noting that display of geometric objects is handled using the abstract notion of a *pen*. A pen is defined as an object which is capable of drawing on some medium. This medium can be the screen, a bitmap, a graphics metafile (e.g. the MacIntosh PICT format), a printer, plotter, or any other medium. The same code is used by the routine which draws the geometric object, regardless of the medium. All medium-dependent code is encapsulated within the pen.

APPLICATIONS OF THE WORKBENCH

The workbench is designed for and is, in itself, implementation-oriented algorithms research.

For the authors, the work provided an invaluable experience in the design process for such a complex system. It provided insight into what the important features of a good geometric programming environment are, where the difficulties lie in their realization and how to overcome these difficulties.

Such a system can be applied to a number of different problem areas. We provide examples of its use in problem-solving, algorithm implementation, implementation-oriented research, and algorithms research.

Users of the workbench may

- apply available algorithms to their own problem sets,
- demonstrate algorithms, possibly in animated form, for educational purposes, or
- use the geometric editor with its capabilities and algorithms for preparing documents (papers, texts, course notes, etc.).

Regarding the latter point it is particularly important that the workbench provides an interface to the MacIntosh clipboard. Any geometric object created by the workbench can produce a graphical representation in PICT format which can easily be incorporated into other application programs, such as MacWrite, Microsoft Word, or MacDraw.

Algorithm implementors using the workbench may

- wish to add implementations of new algorithms,
- incorporate other systems into the workbench, or alternately
- translate (portions of) the workbench into an existing system.

This last would be most useful where there is a large investment in an existing system (e.g. a CAD system). Translation of an existing implementation would be substantially easier and more reliable than implementing a complex algorithm independently.

Implementation-oriented researchers may wish to use the workbench to explore implementation issues such as

- implementability/difficulties in implementation
- constant factors
- comparison studies to other existing algorithms
- special case handling
- stability

Algorithms researchers may obtain feedback during the design phase of a new algorithm on implementability, completeness of description, and correctness of case analyses. The workbench can also provide a useful tool for the generation of counter-examples or for enumeration of combinatorial examples. Timing of an actual implementation may also aid in guiding the complexity analysis.

COMPARISONS

A major design goal of this project is to provide an environment in which constant factors and implementation concerns of algorithms can be compared. Here we can only sketch some of the comparisons performed (for further results see [K], [Ep]).

Attempts at empirical comparison of algorithms are always difficult. Different languages, coding styles, and hardware can greatly obscure the results. To help overcome these difficulties, the algorithms compared below have all been implemented in the same environment (WOCG), using the same primitive operations, and (in most cases) coded by the same person. Thus we can expect the resulting comparisons to reasonably reflect the efficiency of the algorithms themselves as far as this is possible.

None of these implementations have been optimized to any significant degree, and they are as close as possible to the original algorithmic description. The degree to which an algorithm admits optimization is also an important concern, but one which has not been addressed here.

Many algorithms, including the triangulation algorithm of Tarjan and van Wyk [TvW] use finger search trees. These are a complex data structure which provides good amortized time bounds. However, as Tarjan and van Wyk say "Finger search trees are sufficiently complicated that one would probably not want to use them in an actual implementation". The complications of finger trees arise not just from their structure, which in itself induces numerous special cases and exceptions, but from the operations which are to be performed on them, including very complex three-way splits.

An alternative data structure is the splay tree, a form of self-adjusting search tree [ST], which is considerably simpler, but whose worst-case complexity in this context is not proven. Tarjan and van Wyk also state that "the use of splay trees might lead to a practical implementation of our algorithm <ed.: the triangulation algorithm>, although this must be verified by experiment". We discuss these alternatives with respect to code size, ease of implementation and timing.

Table 1, below, shows approximate code sizes for the workbench implementations of finger trees, splay trees, and two different triangulation algorithms. Note that the figure for finger trees includes both the homogeneous and heterogeneous variations.

Component	Lines of Code
Finger Trees (2 kinds)	3000
Splay Trees	700
Jordan Sorting (excluding trees)	2700
Tarjan, van Wyk triangulation algorithm (excluding Jordan sorting and finger trees)	2100
Garey, Johnson, et al triangulation algorithm (including AVL trees)	1200

Table 1: Approximate code sizes for various data structures and algorithms in the computational geometry workbench.

This clearly shows the difficulty of implementing finger trees as compared to splay trees. The results in terms of implementation time were very similar, with splay trees requiring only a few days to implement, while finger trees required several weeks.

With the measured running times of the two structures, our results appear to support the dynamic optimality conjecture of Sleator and Tarjan [ST]. Several different algorithms, including Jordan sorting [HMRT], and point-to-all-vertices shortest path tree [GHLST] were run using both data structures (See Figure 3) The results show the algorithms executing in the correct $O(n)$ time complexity using either structure. In fact, surprisingly little performance difference was observed in the algorithms, even in terms of constant factors.

Performance studies using random queries were also conducted for different varieties of search trees (finger trees, splay trees, randomized search trees). Although there was significant variation with the distribution of the queries, randomized search trees were clearly the fastest. Finger trees were faster than splay trees, but by a smaller margin.

These results are quite plausible for random queries, as all of the structures should be well-balanced and the time would be dominated by the time required to update. Since splay trees restructure after every search operation, they require the most time. Finger trees need

not restructure, but their code is quite complex, and thus also requires more time than the randomized search trees. It should be emphasized, however, that these results are by no means indicative of performance under all circumstances. Since all of these structures are designed to exploit non-random access patterns, other tests, particularly those relating to the access patterns of particular algorithms, are more valuable.

With regards to triangulation, we found that for "random" polygons with less than 10000 vertices (the maximum tested) the $O(n \log \log n)$ Tarjan van Wyk algorithm proved to be slower than the $O(n \log n)$ algorithms of Fournier and Montuno or of Garey, Johnson, et al (Even at 10000 vertices no crossing of the two graphs is in sight). This may be due to the fact that the "randomly" generated simple polygons which were used tend to have few intersections with a horizontal line. If this number of intersections is $O(\log n)$ for an n -vertex polygon then the Fournier and Montuno algorithm would run in $O(n \log \log n)$ time (after sorting which is relatively fast). For star-shaped polygons, where the number of intersections is much larger, the Tarjan and van Wyk algorithm proved to be faster for polygons in excess of approximately 1500 vertices. Conversely, for y -monotone polygons, where the number of intersections is constant, the Tarjan and van Wyk algorithm was at least an order of magnitude slower than either of the other two.

As previously mentioned, algorithms were not optimized to any significant degree, as we have been primarily interested in relative speeds of algorithms following as closely as possible their descriptions in the literature. One significant optimization in this case would be to exploit a floating-point coprocessor, not currently used. This would be most likely to improve the performance of the Tarjan and van Wyk algorithm, as it performs many such operations (primarily in computing intersections). Both of the other triangulation algorithms [FM, GJPT] can avoid floating point arithmetic entirely if given vertices with integer coordinates, although this was not the case in the tests described here, as all vertices had floating point coordinates.

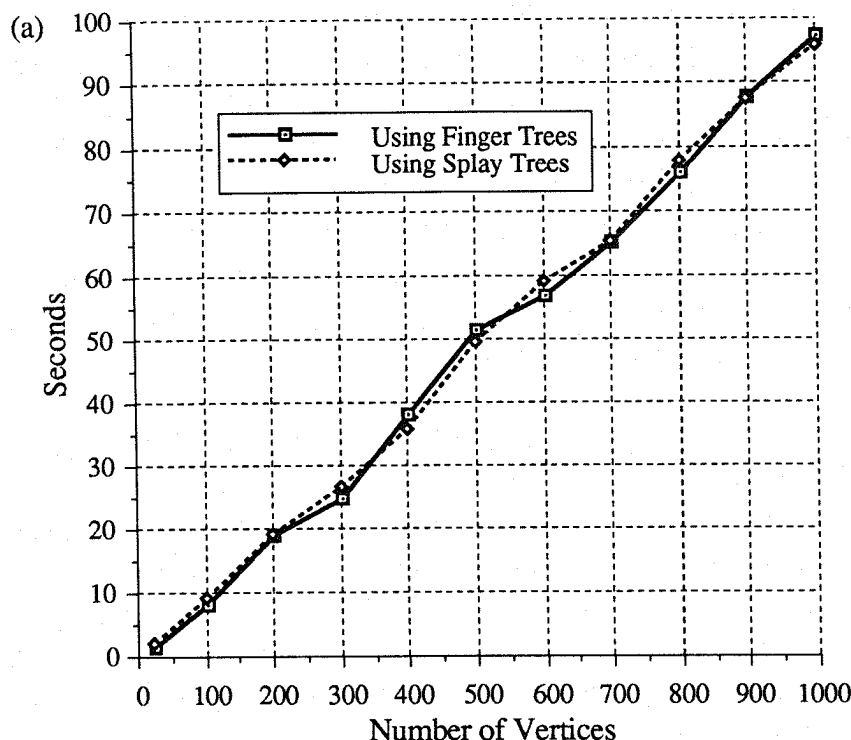


Figure 3: Performance of the point-to-all-vertices shortest path algorithm.

Another interesting point of comparison was between the linear-time algorithm for sorting Jordan sequences and simpler $O(n \log n)$ sorting methods. As might have been expected, the simpler methods (Quicksort and Heapsort) were much faster than the $O(n)$ method for all input sizes tested. It is worth mentioning, however, that these simpler methods could not be immediately used in the Tarjan and van Wyk algorithm, as they do not produce the upper and lower trees, nor can they provide the required "error correction".

Generation of Geometric Objects

To perform comparison studies we needed geometric objects with a large number of points and generated as randomly as possible. In particular, we wanted to compare Jordan sorting algorithms and triangulation algorithms for simple polygons with many edges.

To obtain timing results for the Jordan sorting algorithm it is necessary to create Jordan sequences of arbitrary length. The most direct approach is to create a polygon with a large number of edges, find the intersections of this polygon with some horizontal line, and use the Jordan sequence which is defined by these intersections. This has two serious drawbacks. First, it appears difficult to estimate the number of intersections that will be created. Second, the number of intersections generated for the "random" polygons which

we used is much smaller than the number of vertices in the polygon. For example, an 8000-vertex polygon might have only 100 intersections with a given line.

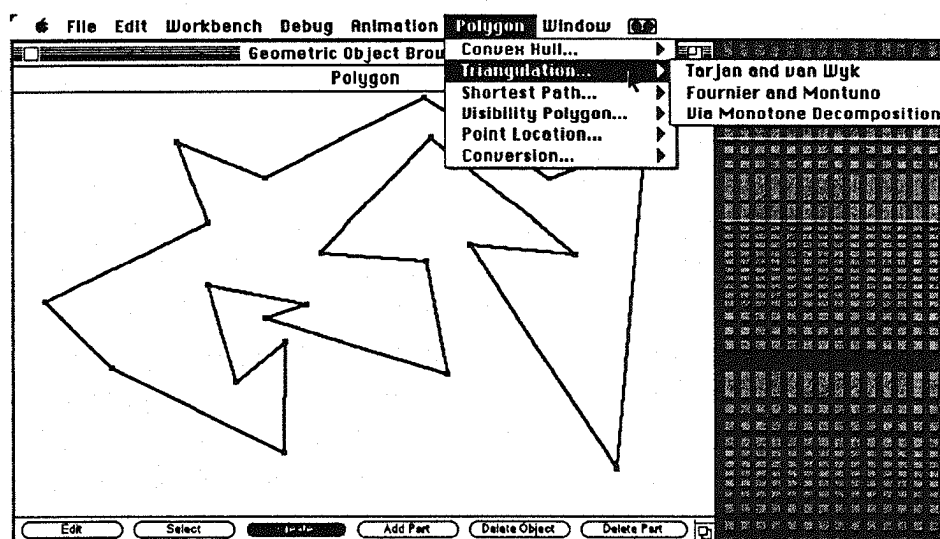
Restricted classes of polygons may admit a large number of intersections. For example, a star-shaped polygon with n vertices can be intersected with a horizontal line so as to produce a number of intersections which is a significant fraction of n . However, these intersections occur in sorted order along the line of intersection. This can be exploited by the sorting algorithm and thus does not provide an interesting test case.

We have designed two algorithms for generating Jordan sequences of arbitrary length with run-time $O(n \log n)$ and $O(n \alpha(n))$ respectively; the former has already been implemented and used for our comparison studies.

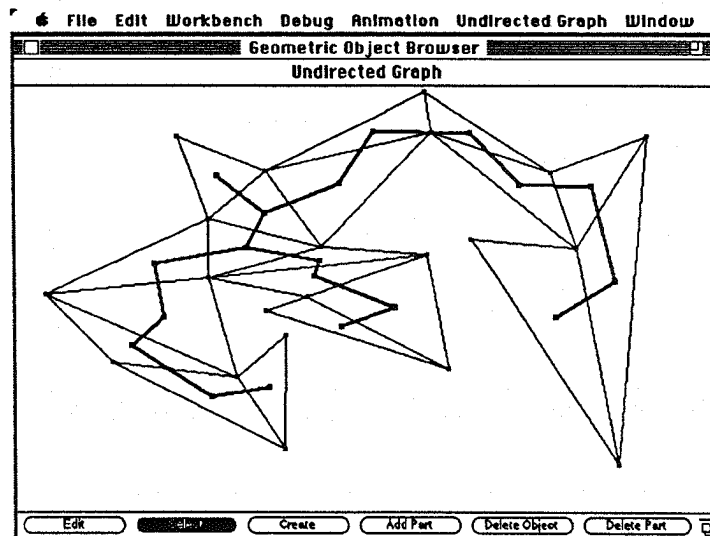
Several algorithms exist for generating simple polygons with a given number of vertices [see e.g. [C], [D], [DE], [E], [O'R], [S]). These algorithms all generate "nice" polygons as opposed to the "difficult" polygons needed to provide demanding test cases. More specifically, we are interested in generating polygon which do not possess properties which can be exploited by the algorithms under consideration. Such properties include in particular monotonicity and star-shapedness. We have designed and implemented several algorithms for generating simple polygons which do not seem to show special properties that these algorithms were able to exploit (see [K]).

SNAPSHOTS OF THE SYSTEM IN OPERATION

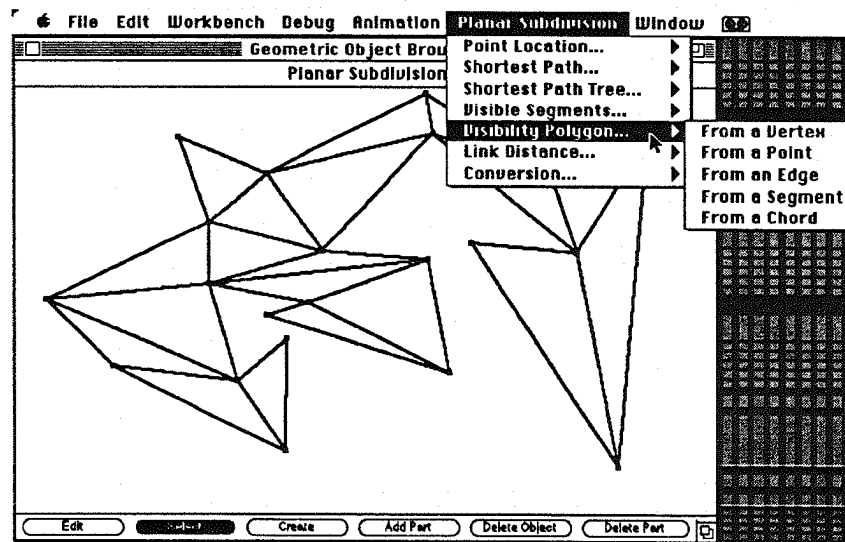
The following snapshots illustrate the system in operation and some of the algorithms available.



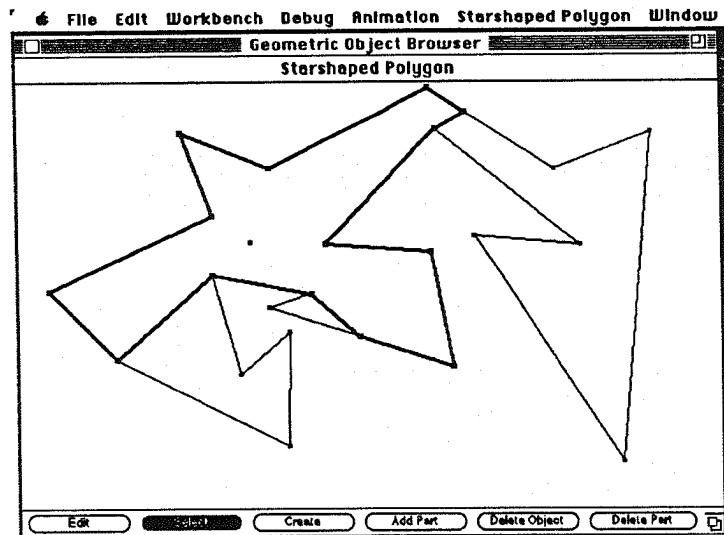
Snapshot 1: Showing some available triangulation algorithms.



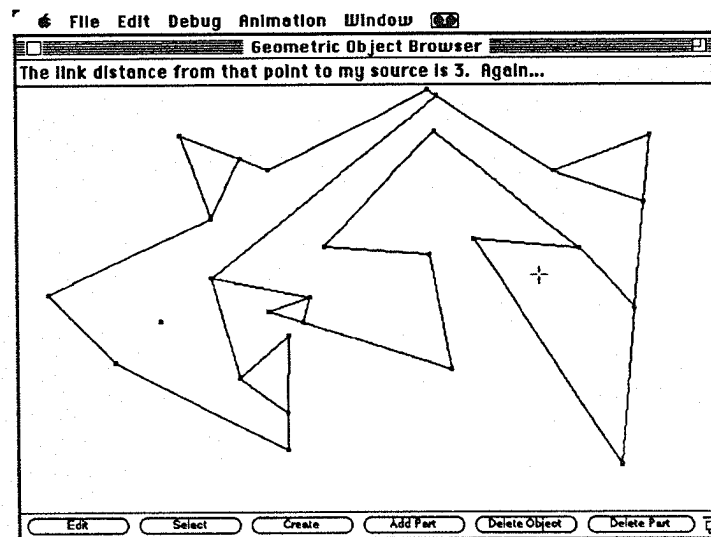
Snapshot 2: Showing a triangulation and its dual tree.



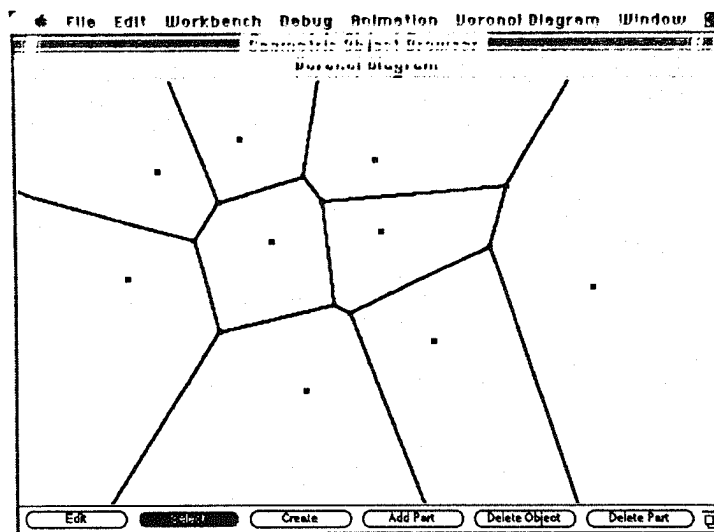
Snapshot 3: Showing linear time visibility algorithms (from triangulation).



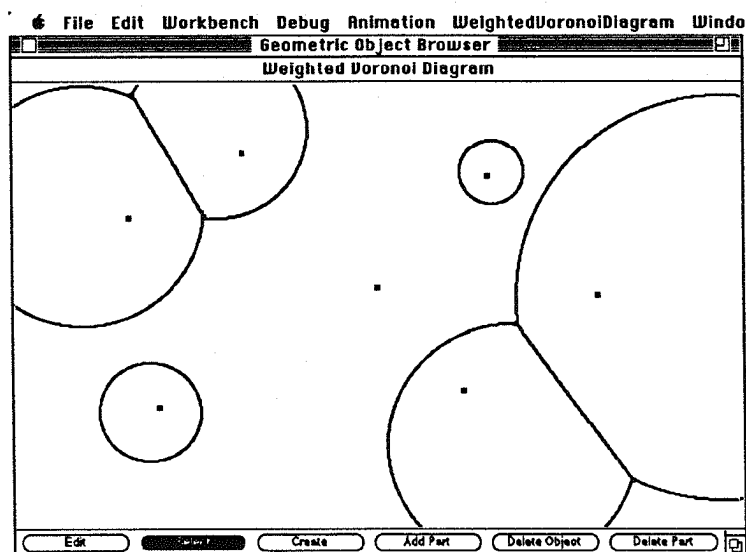
Snapshot 4: Showing the visibility polygon from the specified query point.



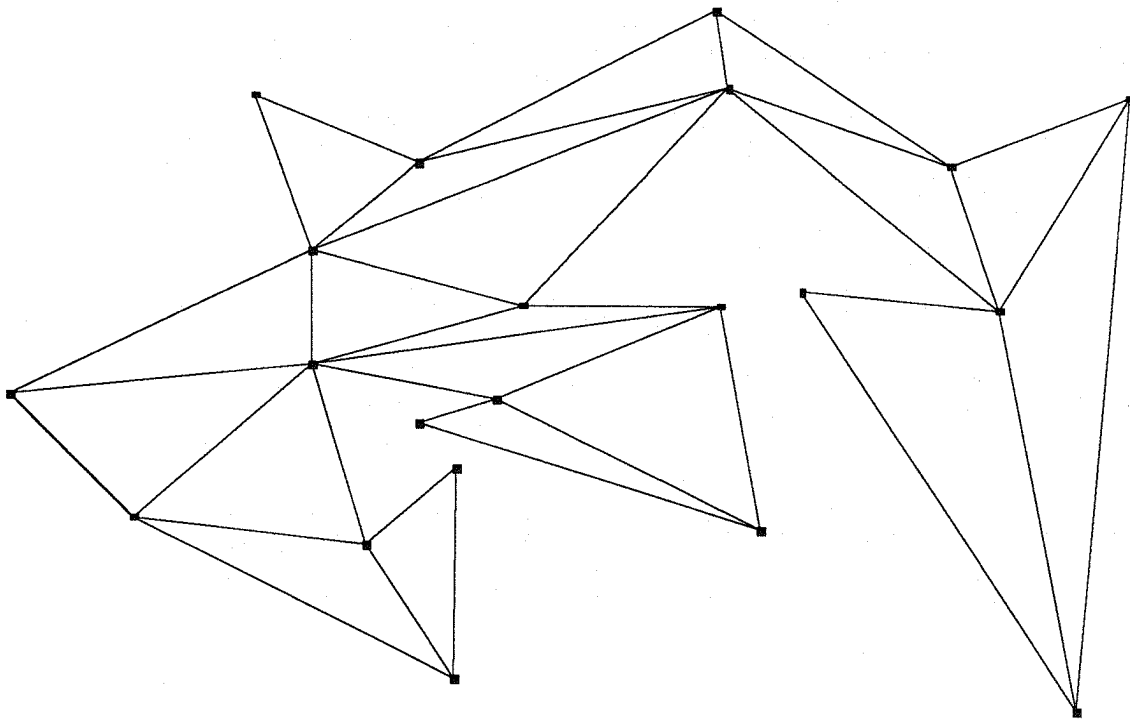
Snapshot 5: A window partition (as in Suri) with link distance query.



Snapshot 6: A Voronoi diagram created by Fortune's algorithm.



Snapshot 7: A weighted Voronoi diagram created by Aurenhammer's algorithm.



Snapshot 8: A triangulated polygon, generated by the workbench, and pasted into Microsoft Word through the MacIntosh clipboard.

CURRENT STATUS AND FUTURE WORK

The workbench now contains a large number of algorithms and data structures. For the sake of brevity we list only the most significant below:

- For convex hulls of point sets and/or simple polygons: the Graham Scan [Gr], Gift-Wrapping [J] and Melkman [M] algorithms.
- For triangulation of simple polygons: the algorithms of Tarjan and van Wyk [TvW], Fournier and Montuno [FM], and of Garey, Johnson, Preparata, and Tarjan [GJPT]
- For Voronoi diagrams, the algorithms of Fortune [F87] and Aurenhammer [Au] (weighted)
- For visibility, shortest path calculations the various linear-time algorithms of Guibas et al [GHLST]
- For window trees and link distances: the algorithms of Suri [Su]
- For point location, the chain decomposition method of Preparata [P]
- Visibility graph of a set of line segments, using the algorithm of Welzl [W]
- Testing a set of line segments for intersections [PS]

Data structures include:

- AVL trees
- 2-3 trees
- Finger search trees [TvW, HM]
- Splay trees [ST]
- Randomized search trees (treaps) [AS]
- Segment and range trees [PS]
- Growable arrays
- Linked lists, either suffix-sharing or circular doubly-linked
- Heaps

We have succeeded in creating what we believe to be a good programming environment for implementing computational geometry algorithms. The design has been tested through the implementation of a number of algorithms of different flavours and complexity of implementation from simple convex hull algorithms to the Tarjan van Wyk triangulation algorithm.

In this on-going project a variety of algorithms and geometric classes for 2-dimensional objects have been implemented and are currently being implemented. We have recently started work on providing a 3-dimensional geometric editor and algorithms.

ACKNOWLEDGEMENTS

We are grateful for the stimulating discussions with researchers of the computational geometry community during the Workshop on Data Structures and Algorithms, Ottawa, 1989 and the ACM Computational Geometry Conference, Berkeley, 1990. We would like to thank Frank Dehne, Binay Bhattacharya, John Pugh, Wilf Lalonde, and Thomas

Strothotte for their input into this project. We thank Raimund Seidel for making available his C-code for treaps to us. We also acknowledge the valuable contributions of Andrzej Bieszczad, Ralph Boland, Danny Epstein, Anne-Lise Hassenklover, Jeff McAffer, Duong Nguyen, Una-May O'Reilly, Alex Smith, James Stringham, and all other graduate and senior undergraduate students who were actively involved in this project.

REFERENCES

- [AS] C.R. Aragon, and R.G. Seidel, Randomized binazry search trees, Proc. 30th FOCS, pp. 450-455.
- [Au] F. Aurenhammer and H. Edelsbrunner, An optimal algorithm for constructing the weighted Voronoi diagram in the plane, Pattern Recognition 17, 1984, pp. 251-275.
- [C] J. Culberson and G. Rawlins, Turtlegons: generating simple polygons from sequences of angles, Proc. 1st ACM Symposium on Computational Geometry (1985), pp. 305-310.
- [D] L. Devroye, private communication, McGill University, November, 1989.
- [DE] T. Doe and S. Edwards, Course Project: Random generation of polygons, 95.508, Carleton University, 1984.
- [E] H. Edelsbrunner, Algorithms in Combinatorial Geometry, W. Brauer, G. Rozenberg, and A. Salomaa (Ed.), EATCS Monographs on theoretical computer science 10, Springer-Verlag, 1987.
- [Ep] P. Epstein, Polygon shortest path algorithms and applications, 4th Year Honours Thesis, Carleton University, May 1990.
- [F] A.R. Forrest, Computational Geometry and Software Engineering, Towards a Geometric Computing Environment, in Techniques for Computer Graphics, D.F. Rogers, R. A. Earnshaw (Eds.), Springer Verlag, pp. 23-37.
- [F87] S. Fortune, A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2 (1987), pp. 153-174.
- [FM] A.Fournier, D.Y. Montuno, Triangulating simple polygons and equivalent problems, ACM Trans. on Graphics 3 (1984), pp 153-174.
- [G] A. Goldberg and D. Robson, Smalltalk-80: The language and its implementation, Addison-Wesley (1983).
- [GHLST] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, "Linear time algorithms for visibility and shortest path problems inside simple polygons", *Algorithmica*, Vol. 2, No. 2, 1987, pp. 209-233.
- [Gr] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters*, Vol. 1, 1972, pp. 132-133.
- [HMRT] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, R. Tarjan Sorting Jordan Sequences in Linear Time using Level-Linked Search Trees, Inform. and Control, 68 (1986), pp. 170-184
- [HM] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists", *Acta Informatica*, Vol. 17, 1982, pp. 157-184.
- [J] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane", *Information Processing Letters*, Vol. 2, 1973, pp. 18-21.
- [K] A. Knight, A computational geometry workbench and its use in algorithms, M.CS. thesis, Carleton University May 1990.
- [M] A. A. Melkman, "On-line construction of the convex hull of a simple polygon", *IPL*, Vol. 25, 1987, pp. 11-12.
- [O'R] J.O'Rourke, H. Booth, R. Washington, Connect-the-dots: A new heuristic, Comput. Vision Graphics Image Process. 39, No. 2 (1987), pp 258-266.
- [P] F. P. Preparata, "A new approach to planar point location", *SIAM J. Comput.*, Vol. 10, No. 3, 1981, pp. 473-482.
- [PS] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, D. Gries (Ed.), Texts and monographs in computer science, Springer-Verlag (1985).
- [S] J.-R. Sack, Rectilinear Computational Geometry, Ph.D. thesis, McGill University, Montréal, Canada, 1984; technical report SCS-TR 54, School of Computer Science, Carleton University, 1984.
- [ST] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees, J. Assoc. Comput. Mach., 32 (1985), pp. 652-686.
- [Su] S. Suri, "Minimum link paths in polygons and related problems", PhD Thesis, The John Hopkins University, 1987.
- [TvW] R.E.Tarjan, C.J. van Wyk, An $O(n \log \log n)$ -time Algorithm for Triangulating a Simple Polygon, SIAM J. Comput. 17, No. 1 (1988), pp. 143-178.
- [W] E. Welzl, Constructing the visibility graph of n line segments in $O(n^2)$ time, *IPL*, Vol. 20, 1985, pp. 167-171

**School of Computer Science, Carleton University
Bibliography of Technical Reports**

- SCS-TR-138 **On Generating Random Permutations with Arbitrary Distributions**
B. J. Oommen and D.T.H. Ng, June 1988.
-
- SCS-TR-139 **The Theory and Application of Uni-Dimensional Random Races With Probabilistic Handicaps**
D.T.H. Ng, B.J. Oommen and E.R. Hansen, June 1988.
-
- SCS-TR-140 **Computing the Configuration Space of a Robot on a Mesh-of-Processors**
F. Dehne, A.-L. Hassenklover and J.-R. Sack, June 1988.
-
- SCS-TR-141 **Graphically Defining Simulation Models of Concurrent Systems**
H. Glenn Brauen and John Neilson, September 1988
-
- SCS-TR-142 **An Algorithm for Distributed Mutual Exclusion on Arbitrary Networks**
H. Glenn Brauen and John E. Neilson, September 1988
-
- SCS-TR-143 to 146 are unavailable.
- SCS-TR-147 **On Transparently Modifying Users' Query Distributions**
B.J. Oommen and D.T.H. Ng, November 1988
-
- SCS-TR-148 **An $O(N \log N)$ Algorithm for Computing a Link Center in a Simple Polygon**
H.N. Djidjev, A. Lingas and J.-R. Sack, July 1988
Available in STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Lecture Notes in Computer Science, Springer-Verlag No. 349
-
- SCS-TR-149 **Smallscript: A User Programmable Framework Based on Smalltalk and Postscript**
Kevin Haaland and Dave Thomas, November 1988
-
- SCS-TR-150 **A General Design Methodology for Dictionary Machines**
Frank Dehne and Nicola Santoro, February 1989
-
- SCS-TR-151 **On Doubly Linked List ReOrganizing Heuristics**
D.T.H. Ng and B. John Oommen, February 1989
-
- SCS-TR-152 **Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry**
Frank Dehne and Andrew Rau-Chaplin, March 1989
-
- SCS-TR-153 **The Use of Chi-Squared Statistics in Determining Dependence Trees**
R.S. Valiveti and B.J. Oommen, March 1989
-
- SCS-TR-154 **Ideal List Organization for Stationary Environments**
B. John Oommen and David T.H. Ng, March 1989
-
- SCS-TR-155 **Hot-Spot Contention in Binary Hypercube Networks**
Sivarama P. Dandamudi and Derek L. Eager, April 89
-
- SCS-TR-156 **Some Issues in Hierarchical Interconnection Network Design**
Sivarama P. Dandamudi and Derek L. Eager, April 1989
-
- SCS-TR-157 **Discretized Pursuit Linear Reward-Inaction Automata**
B.J. Oommen and Joseph K. Lanctot, April 1989
-
- SCS-TR-158 **Parallel Fractional Cascading on a Hypercube Multiprocessor**
(revised) Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, May 1989 (Revised April 1990)
-
- SCS-TR-159 **Epsilon-Optimal Stubborn Learning Mechanisms**
J.P.R. Christensen and B.J. Oommen, June 1989
-
- SCS-TR-160 **Disassembling Two-Dimensional Composite Parts Via Translations**
Doron Nussbaum and Jörg-R. Sack, June 1989
-

- SCS-TR-161 (revised) **Recognizing Sources of Random Strings**
R.S. Valiveti and B.J. Oommen, January 1990
Revised version of SCS-TR-161 "On the Data Analysis of Random Permutations and its Application to Source Recognition", published June 1989
-
- SCS-TR-162 **An Adaptive Learning Solution to the Keyboard Optimization Problem**
B.J. Oommen, R.S. Valiveti and J. Zgierski, October 1989
-
- SCS-TR-163 **Finding a Central Link Segment of a Simple Polygon in $O(N \log N)$ Time**
L.G. Alexandrov, H.N. Djidjev, J.-R. Sack, October 1989
-
- SCS-TR-164 **A Survey of Algorithms for Handling Permutation Groups**
M.D. Atkinson, January 1990
-
- SCS-TR-165 **Key Exchange Using Chebychev Polynomials**
M.D. Atkinson and Vincenzo Acciari, January 1990
-
- SCS-TR-166 **Efficient Concurrency Control Protocols for B-tree Indexes**
Ekow J. Otoo, January 1990
-
- SCS-TR-167 **A Hierarchical Stochastic Automaton Solution to the Object Partitioning Problem**
B.J. Oommen, January 1990
-
- SCS-TR-168 **Adaptive List Organizing for Non-stationary Query Distributions. Part I: The Move-to-Front Rule**
R.S. Valiveti and B.J. Oommen, January 1990
-
- SCS-TR-169 **Trade-Offs in Non-Reversing Diameter**
Hans L. Bodlaender, Gerard Tel and Nicola Santoro, February 1990
-
- SCS-TR-170 **A Massively Parallel Knowledge-Base Server using a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990
-
- SCS-TR-171 **Parallel Processing of Quad Trees on the Hypercube (and PRAM)**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990
-
- SCS-TR-172 **A Note on the Load Balancing Problem for Coarse Grained Hypercube Dictionary Machines**
Frank Dehne and Michel Gastaldo, May 1990
-
- SCS-TR-173 **Self-Organizing Doubly-Linked Lists**
R.S. Valiveti and B.J. Oommen, May 1990
-
- SCS-TR-174 **A Presortedness Metric for Ensembles of Data Sequences**
R.S. Valiveti and B.J. Oommen, May 1990
-
- SCS-TR-175 **Separation of Graphs of Bounded Genus**
Ljudmil G. Aleksandrov and Hristo N. Djidjev, May 1990
-
- SCS-TR-176 **Edge Separators of Planar and Outerplanar Graphs with Applications**
Krzysztof Diks, Hristo N. Djidjev, Ondrej Sykora and Imrich Vrto, May 1990
-
- SCS-TR-177 **Representing Partial Orders by Polygons and Circles in the Plane**
Jeffrey B. Sidney and Stuart J. Sidney, July 1990
-
- SCS-TR-178 **Determining Stochastic Dependence for Normally Distributed Vectors Using the Chi-squared Metric**
R.S. Valivetei and B.J. Oommen, July 1990
-
- SCS-TR-179 **Parallel Algorithms for Determining K-width- Connectivity in Binary Images**
Frank Dehne and Susanne E. Hambrusch, September 1990
-
- SCS-TR-180 **A Workbench for Computational Geometry (WOCG)**
P. Epstein, A. Knight, J. May, T. Nguyen, and J.-R. Sack, September 1990
-