An Implementation of a plane-sweep algorithm on a personal computer

Author(s):
Beretta, Giordano B.

Publication Date:
1984

Permanent Link:
https://doi.org/10.3929/ethz-a-000338864

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
AN IMPLEMENTATION OF A PLANE-SWEEP ALGORITHM ON A PERSONAL COMPUTER

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Giordano Bruno Beretta
Dipl. Math. ETH
born April 14, 1951
citizen of Lugano, Ticino

accepted on the recommendation of
Prof. Jürg Nievergelt, examiner
Prof. Walter Nef, co-examiner

1984
# Table of contents

Abstract

Zusammenfassung 4

0. Introduction

0.1. The idea of sweep algorithms 5
0.2. Objectives and organization 6
0.3. Notation 6

1. The skeleton of plane-sweep algorithms 7

1.1 Previous work 7
1.2. Terminology and assumptions 7
1.3. Description of the algorithm 9

1.3.1. Abstract data types: x-queue and y-table 9
1.3.2. Outline of the algorithm 10
1.3.3. Growth rate 10

2. Implementation of the plane-sweep skeleton 12

2.1. Overall system structure 12
2.2. Geometrical workbench 13

2.2.1. Low-level components 13
2.2.2. Graphical kernel 15
2.2.3. Kernel interface to programs 18

2.3. Plane-sweep skeleton 16

2.3.1. Application specific abstract data types 16
2.3.2. Transition procedure 17

2.4. X-queue 17

2.4.1. Element structure 17
2.4.2. Operations 18
2.4.3. Heap data structure 19
2.4.4. Bucket data structure 20

2.5. Y-table 22

2.5.1. Element structure 22
2.5.2. Operations 22
2.5.3. Data structure 22

2.6. Skeleton interface to the application programs 24

2.6.1. Information flow and usage of the module 24
2.6.2. Exported procedures 25
2.6.3. Update of the y-table 27

3. Influence of finite arithmetic 28

3.1. Statement of the problem 28
3.2. Nature of the input data 28
3.3. Values of the search keys of the x-queue 29
3.4. Criterion for the robustness of plane-sweep implementations 32
3.5. Values of the search keys of the y-table 33
3.6. Increasing the robustness of the y-table access 36

4. Efficiency analysis 37
4.1. Memory requirements 37
4.2. The constant factor of the running time 37
   4.2.1. Generation of the experiment data 37
   4.2.2. Experiments and results 39
   4.2.3. Discussion 40
4.3. An example of optimization 41
   4.3.1. Parametric representation of segments 42
   4.3.2. Segment intersection 43

5. Applications 52
   5.1. Plane-sweep tutorial 52
   5.2. Statistics 52
   5.3. Raster-scan conversion 53
      5.3.1. Introduction 53
      5.3.2. Model of the implemented algorithm 55
      5.3.3. Data structures and initialization 57
      5.3.4. Processing of a slice 58
      5.3.5. Implementation of the raster-scan conversion 60
   5.4. Area computation 61
   5.5. Maximal height of polygons 63
   5.6. Further applications 64

6. Conclusions 68

Appendix A. Results of the experiments with the heap data structure 69
Appendix B. Results of the experiments with the bucket data structure 72

Literature 75

Acknowledgements 78

Curriculum vitae 79
Abstract

Plane-sweep algorithms were introduced by Shamos to solve the intersection problem of \( n \) straightline segments in time \( O(n \log n) \). Nievergelt and Preparata have shown that regions of arbitrary shape and topology can be processed with the same asymptotic effort required for line segments.

In this paper we consider a set of polygons in the plane. The polygons are given by their edges, and each edge is given by its two endpoints. The \( x \)-axis is marked as the direction of the process which leads to a unidirectional sweep. The segment endpoints along with all the information required to process them represent a queue of tasks to be accomplished. The queue is described by the abstract data type \( x \)-queue. A parallel to the \( y \)-axis, called a cross section, is decomposed into intervals by the regions which partition the plane. The abstract data type describing the state of the current cross section is called the \( y \)-table.

A key feature of the algorithm is that the intersection points of the originally given segments need not be known in advance, but are found during the sweep and considered as forthcoming tasks. No backtracking is needed and no intersection points are left undiscovered.

This paper focuses on implementation aspects. It is shown that an application independent skeleton can be extracted from the original plane-sweep algorithm. This skeleton performs most of the work; the applications must only supply the code to process a single region. The \( x \)-queue has been implemented using two different data structures: a partially ordered tree stored in a heap, and a bucket organization. The \( y \)-table has been implemented using a height-balanced binary tree. Only 4681 words of memory are required on the \textit{Lilith} personal computer for the storage of the code of the plane-sweep skeleton.

Sweep algorithms are said to lack robustness. This is due to numerical problems caused by the fact that the mapping from real numbers to machine representable numbers is not bijective. For each floating-point number represented in a given machine there is a corresponding interval on the number line such that all numbers in the number line have the same machine representation. If the segment endpoints are assumed to have integer coordinate values in the range \([0, M]\), then at least \( 5 \log M + 4 \) bits are required for the storage of the fraction part of floating-point numbers in order for the algorithm to run correctly.

A performance analysis has been carried out based on the described implementation. Statistical experiments confirm the hypothesized result that the running time of programs using the plane-sweep skeleton is \( T(n, s) = c \ (n + s \log n) \), where \( n \) is the number of originally given segment endpoints and \( s \) is the number of segment intersection points. The constant factor has been estimated to be \( c = 0.0027 \pm 0.0003 \) seconds on the \textit{Lilith} personal computer. Thus the algorithm is fast even for small values of \( n \) and \( s \).

This paper also considers a number of applications which have been implemented using the plane-sweep skeleton, e.g., raster-scan conversion.
Zusammenfassung


Wir betrachten eine Menge von Polygonen in der Ebene, wobei die Polygone durch ihre Kanten und die Kanten durch ihre beiden Endpunkte gegeben sind. Die x-Achse ist als Richtung des Verfahrens ausgezeichnet, was zu einem gerichteten Durchlauf der Ebene führt. Die Endpunkte der Strecken mit der für ihre Verarbeitung notwendigen Information, stellen eine Warteschlange zu verarbeitender Aufgaben dar. Sie ist durch den abstrakten Datentyp X-Queue beschrieben. Die Ebene wird in Gebiete unterteilt, welche jede Parallelle zur y-Achse (genannt Querschnitt) in entsprechende Intervalle zerlegen. Der den Zustand des gegenwärtigen Querschnittes beschreibende abstrakte Datentyp wird Y-Table genannt.

Eine wesentliche Eigenschaft des Algorithmus ist, dass die Schnittpunkte nicht von vorherein bekannt sein müssen, sondern während dem Durchlauf gefunden werden. Dazu ist kein Backtracking erforderlich.


Eine Effizienzanalyse basierend auf statistischen Versuchen hat die theoretisch vorausgesagte Ausführungszeit $T(n, s) = c (n + s) \log n$ bestätigt, wobei $n$ die Anzahl der im voraus gegebenen Strecken-Endpunkte ist und $s$ die Anzahl der Strecken-Schnittpunkte. Nach einer Schätzung auf dem Lilith Arbeitsplatzrechner, ist der konstante Faktor $c = 0.0027 \pm 0.0003$ Sekunden, demzufolge ist der Algorithmus auch für kleine $n$ und $s$ schnell.

Zusätzlich wird eine Anzahl von Anwendungen diskutiert, die das Ebenen-Durchlaufs-Gerüst verwenden, speziell auch Konvertierung von Rasterbildern.
0. Introduction

0.1. The Idea of sweep algorithms

There are algorithms and formulae of classical geometry which solve geometrical problems, but it is not possible to code the algorithms and formulae of classical geometry indiscriminately in a computer language. The discipline of studying the computational aspects of geometry within the framework of analysis of algorithms is called computational geometry [Shamos 1978]. The challenge is to design algorithms and data structures that involve minimal computation complexity. Minimal computation complexity is to be understood in the sense of the Random Access Machine generalized to floating-point arithmetic and storage of floating-point variables [Aho et al. 1974].

The efficacy of sweeping a geometrical configuration in the plane is based on the transformation of a two-dimensional problem into a sequence of one-dimensional problems. The one-dimensional problem is significantly simpler than the original two-dimensional one. The projections of the line segments that define these two-dimensional objects onto a one-dimensional sweep-line can be totally ordered, permitting logarithmic access time to any object intersected by the sweep-line (figure 1). The sweep process consists of breaking down the configuration into a sequence of slices in which the order of the segment's projections onto the sweep line remains unaltered. The sequence of slices is obtained by projecting the segment's endpoints on the axis orthogonal to the sweep line. This leads to $O(n \log n)$ algorithms, where $n$ is the number of objects.

![Fig. 1. Sweep of a configuration consisting of segments in the plane. The sweep line $\gamma$ which sweeps the plane from left to right is shown at the moment when the intersection point is discovered.](image)

A sweep process handles a sequence of events, i.e. points, and all the information necessary to process them. For each event the information about the current cross section must be updated and application specific results must be accumulated. This is reflected by the fact that many sweep algorithms use three abstract data types:

- $x$-queue: a priority queue that supports the operations $\text{Min}$ and $\text{Insert}$ in time $O(\log n)$ if it
has \( n \) entries;

- \( y \)-table: a table that supports the operations Insert, Delete, Pred, Succ in time \( O(\log n) \) if it has \( n \) entries;

- An application specific structure to gather the result to be produced by the algorithm.

The nature of the sweep suggests using the algorithm for a wide variety of geometrical and topological problems.

0.2. Objectives and organization

The objective of this paper is to analyse the practical feasibility of the plane-sweep algorithm described by Nievergelt and Preparata. The study consists of the following stages:

- Extract an application independent skeleton of the algorithm.

- Implement the skeleton on a personal computer and study the numerical problems involved.

- Develop some applications using the skeleton to assess the generality of the algorithm.

- Perform an efficiency analysis to determine the constant factor \( c \) of the running time \( T(n, s) = c(n + s) \log n \).

Although the implementation had to be simple enough to fit into 16 Kbytes of memory, a geometrical workbench requiring only a small amount of memory has been implemented in order to perform the efficiency analysis on the Lilith personal computer.

This paper is structured as follows: The algorithm is presented in chapter 1. The implementation of the algorithm is presented in chapter 2. In chapter 3, the numerical problems encountered in the implementation of sweep algorithms are addressed. In chapter 4, experiments are used to document the memory and time requirements. Applications using the plane-sweep skeleton are presented in chapter 5.

0.3. Notation

A point \( a \) is specified by a coordinate pair \((x, y)\). A segment \( s \) is specified by the coordinates \((x_{a1}, y_{a1})\) and \((x_{b2}, y_{b2})\) of its endpoints.

In the program examples the following declarations are used:

\[
\text{TYPE POINT} = \text{RECORD} \ x, y : \text{INTEGER} \ \text{END}; \\
\text{SEGMENT} = \text{RECORD} \ a, b : \text{POINT} \ \text{END}.
\]

Therefore, the endpoints of segment \( s \) will be denoted \((s.a.x, s.a.y)\) and \((s.b.x, s.b.y)\) in the program examples.
1. The skeleton of a plane-sweep algorithm

1.1. Previous work

Plane-sweep algorithms were introduced by Shamos to solve the segment intersection problem in time \(O(n \log n)\), i.e., to decide whether or not any two of \(n\) line segments intersect [Shamos 1978; Shamos and Hoey 1976]. The \(2n\) endpoints of the given line segments are sorted lexicographically according to increasing coordinates, and a vertical cross section is moved from left to right across the plane. In the worst case, the data structure of the cross section has to be updated at each of the \(2n\) \(x\) coordinates. The updating consists of inserting or deleting a line segment from the cross section (an \(O(\log n)\) operation), and checking the adjacent line segments for intersection, an \(O(1)\) operation. The total work is bounded by \(O(n \log n)\) for the initial sorting, plus another \(O(n \log n)\) for the sweep phase.

Bentley and Ottmann extended this algorithm to report all \(s\) intersections among \(n\) line segments in time \(O((n+s) \log n)\) [Bentley and Ottmann 1979]. They treated every intersection point as a transition point. The cross section is updated at all \(2n+s\) transition points, each of which is performed in time \(O(\log n)\).

The first algorithms to handle areas, as opposed to line segments, dealt with rectangles. In Nievergelt and Preparata 1982, however, it is shown that regions of arbitrary shape (e.g., spirals) and topology (e.g., multiply connected) can be processed within the same asymptotic effort as line segments.

Nef and Bieri have developed a sweep-plane algorithm that enumerates the cells of all dimensions into which \(\mathbb{R}^d\) is partitioned with a finite set of hyperplanes [Bieri and Nef 1983]. Their algorithm is recursive with respect to the dimension of space. Nef and Bieri have used this algorithm to develop an algorithm for computing the volume of bounded polyhedra of any finite dimension [Bieri and Nef 1983], and for developing an algorithm for computing the Euler characteristic of such polyhedra [Bieri and Nef 1984].

1.2. Terminology and assumptions

Input to the plane-sweep algorithm is a set of polygons in the plane. It is called configuration and defined by a set of line segments as follows. Let \(x\) and \(y\) be the Cartesian coordinates in the plane. For a sequence of \(m\) points \(a_i = (x_i, y_i), i = 1, 2, ..., m\) in the plane, the line segments \(a_1a_2, a_2a_3, ..., a_ma_1\) define a polygon. Let \(n\) be the total number of line segments in all polygons.

These points constitute the information known for each polygon in the configuration at the start of the algorithm. Under the assumption of nondegeneracy in which no more than two segments intersect in a point, they can be classified uniquely into one of three types: start, bend, and end point.

If a polygon is self-intersecting the above line segments have \(r\) intersection points \(b_i, i = 1, 2, ..., r\), and the line segments of all polygons have \(s\) intersection points. These points are unknown at the start of the algorithm. They are determined during the sweep of the plane and stored for subsequent processing with the classification type intersection (figure 2).

In order to describe the algorithm, auxiliary concepts are introduced that reflect the dynamic
Fig. 2. A configuration consisting of a single self-intersecting polygon. γ is a cross section, and R₀, R₁, R₂ are the regions into which the plane is partitioned. The polygon points P₁, P₂, P₃, P₄, Pₑ are the given points, the intersection point P₄ is discovered when the start point Pₑ is processed.

aspects, i.e. a unidirectional sweep of the plane. The x-axis is distinguished as the sweep direction, making the sweep line orthogonal to the x-axis.

In the first stage of the algorithm, the original n segments are read. The polygon to which they belong and the classification type of their common point is determined from their sequence and orientation in the input. The information consisting of two consecutive segments, their common point with its classification type, and the polygon of which the segments are edges is called an event. Events are sorted according to the increasing x coordinate of the common point. Together with the intersections found during the sweep these points are called transition points pᵢ.

A cross section γ is a vertical line in the plane (a sweep line) and the information about which line segments it cuts in what order. A cross section that passes through a transition point is called a transition line. The open set of all (topologically equivalent) cross sections that lie between two adjacent transition points pᵢ, pᵢ₊₁ is called a slice.

The implementation is simplified so that the plane-sweep skeleton with an application fits into a minimum amount of memory on the Liliith personal computer. The following assumptions of nondegeneracy are made to accomplish this simplification:

- No more than two segments intersect in a point.
- A segment may not have a length of zero.
- Vertical segments may be tilted by incrementing by one unit the x coordinate of an endpoint.

1.3. Description of the algorithm

On any cross section of a slice, the regions into which the plane is partitioned by the polygons induce a partition of the cross section into intervals. The order of the intervals within a slice does not change, which means that the topology in each slice is fully determined by the partition of the left transition line. The geometry of the portions of the regions in a slice is determined by the two transition lines which delimit the slice.

The informal algorithm takes advantage of the total order existing on the projections of the transition points on the x-axis, and on the intervals on the transition lines. The transition lines are processed sequentially in order of the increasing x coordinate. Intersection points are discovered during the sweep without increasing the time complexity of the algorithm because intersecting segments are neighbours before they intersect. To discover them, the segment pair in the event of the transition point is intersected with its neighbours, that is the segment just above and the one just below.

1.3.1. Abstract data types: x-queue and y-table

The above allows one to specify the data types and the operations to be performed on them, i.e. the abstract data types. In the case of plane-sweep at least two abstract data types are required.

The x-queue contains the events known so far and are yet to be processed, sorted according to increasing x coordinates. The x-queue behaves like a priority queue. It must support the operations Initialize, Min and Insert. Initialize reads the n originally given segments, constructs the events, and inserts them into the queue. Min retrieves the next event, the one with the lowest x coordinate, and deletes it. Insert adds a new event in the appropriate location. The initial content consists of the events corresponding to the n originally given points. At the end of the sweep process the x-queue is empty.

The y-table contains the information about a cross section which is representative of its entire slice. It contains an entry for each interval of the slice, including the intervals that extend to \( y = -\infty \) and \( y = +\infty \), and thus it never contains more than \( n + 1 \) elements. An equivalent description is that the y-table has an entry for each line segment intersected by the slice. At the beginning and the end of the algorithm the y-table contains the single interval \((-\infty, +\infty)\).

During the sweep the algorithm must maintain the consistency of the y-table. Its contents must reflect the current transition line. This means that on the transition lines the intervals affected by the event have to be handled according to the type of event. At a transition of type start, two new segments are inserted. At an end point two segments are deleted. At a bend point a segment is replaced, and at a crossing two segments are permuted.

To find the intersection points, operations must be available to find the neighbourhood of a segment in the y-table. The y-table must support, in addition to Insert, Delete, Replace, and Permute, the operations Predecessor, and Successor.
1.3.2. Outline of the algorithm

The following simple overall structure for a plane-sweep algorithm may be found:

PROCEDURE Sweep;
    x := sorted events, constructed from the given points;
    y := (-∞, +∞);
    WHILE x ≠ Ø DO
        p := Min (x);
        Transition (p)
    END
END Sweep;

The procedure Transition is the advancing mechanism of Sweep. It encompasses all the work involved in processing the event corresponding to one transition point p and moving the transition line to the next slice. It breaks into four cases depending on the type of transition point p. By denoting with s the left (respectively the upper) and by t the other segment of which p is the common point, Transition consists of the following case statement:

CASE transition type OF
    start : h := Successor (s); t := Predecessor (t);
    Intersect (s, h); Intersect (t, l);
    Insert (s); Insert (t) |
    bend : h := Successor (s); t := Predecessor (t);
    Intersect (t, h); Intersect (t, l)
    Replace (s, t) |
    end : h := Successor (s); t := Predecessor (t);
    Intersect (h, l);
    Delete (s); Delete (t) |
    crossing: h := Successor (s); t := Predecessor (t);
    Intersect (t, h); Intersect (s, l)
    Permute (s, t)
END CASE;

1.3.3. Growth rate

The running time T of a program based on the plane-sweep algorithm depends on the input size n and the number s of intersection points. The efficiency or worst case time complexity of an algorithm is said to be $O(f(n,s))$ [Aho et al. 1983], if there is some constant c such that for sufficiently large n and s, $c f(n,s)$ is an upper bound on the number of steps taken by the algorithm on inputs of length n and with s intersections. The term steps refers to the number of high level language statement groups involved. Each takes a constant amount of time when translated into the machine language of any computer. A program whose running time is $O(f(n,s))$ is said to have growth rate $f(n,s)$.

In the following paragraphs the worst case time complexity is discussed. The constant factor c is determined experimentally for the Lilith personal computer in chapter 4.

At each transition point an element is fetched from the x-queue and, at most, two intersections are inserted. During execution a total of $n+s$ elements move through the x-queue, ensuring that the maximum number of entries at any given time is bound by $n+s$. There are data structures for priority queues (e.g., partially ordered trees) that support the operations Min and Insert within the
time bound \(O(\log n)\) when they contain \(n\) elements. In the worst case \(s = O(n^2)\), any operation on the x-queue can be done in time \(O(\log(n + s)) = O(\log n)\).

There are data structures for tables (e.g., height balanced binary trees) which support the operations \textit{Insert}, \textit{Delete}, \textit{Predecessor}, \textit{Successor}, \textit{Replace} and \textit{Replace in time} \(O(\log n)\) when the table contains \(n\) entries. From the procedure \textit{Transition} of the previous subsection it can be seen that all its operations can be executed in time \(O(\log n)\), because the y-table contains \(O(n)\) elements.

Since the plane-sweep algorithm makes \(n + s\) transitions, its worst case time complexity is \(O((n + s) \log n)\).
2. Implementation of the plane-sweep skeleton

It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming [...]. Machines are made to behave in wondrous ways, [...] but once a particular program is unmasked, [...] its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible.

Joseph Weizenbaum

The target machine Lilith [Wirth 1981] is based on four AMD 2901 bit-slice chips. The CPU operates at a basic clock cycle of 150 ns. The size of the main memory originally was 84 Kwords. Later a second bank was added for bitmap and font storage. The display is based on the memory mapped raster-scan technique. The size of the bitmap from which the display is refreshed is 28'416 or 33'280 16-bit words, depending whether a landscape (768X592 pixels) or a portrait (640X832 pixels) screen is used.

The micro-coded instruction set is designed to host the Modula-2 language. Special instructions have been provided to perform raster operations, double-precision unsigned integer arithmetic, shift and rotate operations, and to access devices and the second memory bank. The Lilith version of Modula-2 allows one to write procedures containing the octal code of these special instructions.

The implementation of an algorithm consists of using collections of variables of possibly different data types called data structures to represent an abstract data type, and of using the basic operators of the programming language to build the operations. An entry of a data structure is called an element.

The aggregating mechanisms supplied by the programming language Modula-2 are the array and the record. The elements of the data structures are built of basic Modula-2 types and aggregating types. The constructs pointer and cursor are used for this purpose. A pointer is the absolute address of an element. A cursor is the relative address of an element in a data structure stored in consecutive memory locations. In an implementation the abstract data types are represented by data structures featuring operations that can be performed within the time bounds specified in the informal algorithm.

In section 2.1 a survey of the structure of the system is presented. The environment of the plane-sweep implementation is outlined in section 2.2. 2.3 discusses an application independent skeleton of plane-sweep. The data structures are presented in sections 2.4 and 2.5. Section 2.6 contains a description of the usage of the plane-sweep skeleton.

2.1. Overall system structure

The implementation is organized as a workbench for computational geometry. For the kernel of this workbench the low-level components of Medos-2 [Knudsen 1983] and the Tree File System [Sugaya 1982] have been used. The graphical kernel is based on an early draft of the GKS standard.

The result of a plane-sweep application is obtained in three stages:

- input of a configuration with an interactive polygon editor
Fig. 3. Logical structure of the implementation. The system components are used in the sequence from left to the right.

- processing of the configuration with a plane-sweep

- output of the result

To avoid monolithic application programs, three independent small programs corresponding to the three stages are executed successively (figure 3).

The workbench manages the hardware resources, the data, and the programs. It also contains the instrumentation required to perform the efficiency analysis of the plane-sweep algorithm. Its major benefits are the very low memory space requirement for the resident part and the structure it puts on the data and the programs to provide a good man-machine interface.

Each data structure implementing an abstract data type has been encapsulated in a separate module. This allows one to experiment with data structures by switching among various implementations of the data types without any change in the module containing the plane-sweep skeleton. Two different data structures have been retained for the x-queue. The first is a partially ordered tree, organized as a heap. The second uses a bucket organization. The y-table has been implemented using a height-balanced binary tree.

2.2. Geometrical workbench

2.2.1. Low-level components

The following low-level components have been taken from the native Lilith operating system Medos-2 [Knudsen 1983]:

SYSTEM contains the machine dependent tools of the Modula-2 system. It exports the types for addresses and storage words. It also exports functions to get the address and size of
variables as well as a function to determine the amount of memory required to store variables of a given type.

\textit{D140Disk} and \textit{DiskSystem} implement files on cartridges for the \textit{Honeywell Bull D140} disk drives.

\textit{FileSystem} is used to get the status of disk operations and to read the previous system date when the machine is booted.

\textit{Program} implements the loader and the allocation of contiguous dynamic memory areas. The loader allows one to suspend the execution of the current program and to load and execute a new program. This operation can be iterated until the memory space is exhausted. The new program can access the data and execute procedures of the suspended programs. Upon termination of a program, the execution of the parent program is automatically resumed.

\textit{Monitor} hides very-low level constructs from the module \textit{Program}. It is needed by the instrumentation facilities to access the system clock.

\textit{Terminal} provides the reading of single characters from the keyboard.

\textit{DisplayDriver} delivers the address and size of the bitmap and the default font. It is used by the graphical kernel to drive the \textit{Lilith} display.

All the above modules are resident. The programs have been divided into three classes depending on the system resources they use. Before a program is loaded and executed, the kernel of the geometrical workbench loads the corresponding non-resident portion of the operating system.

The 31 programs currently available are selected from a menu organized as a tree. Whenever the system is idle the operating system loads and executes a menu program allowing one to explore the tree. Once a program is selected by the user, the memory of the menu program is released and the selection is stored in the \textit{schedule} of the operating system. The operating system first loads the appropriate non-resident portion of the operating system and then loads and executes the program itself.

When the tree of programs is explored, for each movement the current position is popped or pushed on a stack in the \textit{schedule} depending on the direction of the movement. This permits the menu program to return to its original position after the system is idle again.

The data space is also structured as a tree. For this purpose the \textit{Tree File System} available on the \textit{Lilith} personal computer is used [Sugaya 1982]. The tree structure has been chosen as a compromise between the quest for generality in representing relationships among data and the necessity of efficient realization.

The \textit{Tree File System} maintains a window into the tree: the \textit{current node} which is part of the \textit{schedule}. This window can be moved to any node in the tree, and is used to inspect and modify all information related to a node.

From the functions of the \textit{Tree File System}, only those relevant to the applications described in this work are reported.

- Change of attributes: Setting and retrieving the name and type of the current node. Reading and writing an arbitrary number of bytes at a specified node data position.
Motion: Motion relative to the neighbours of the current node. Absolute motion to a node recognized by its name or by a system-generated identification.

Tree manipulations: All structural manipulations are performed in the son generation of the current node. In this application we use only the insertion of a new node and the deletion of a subtree.

2.2.2. Graphical kernel

The workbench contains a graphical kernel. The purpose of this kernel is to provide a frame for interactive graphical data input, output, and manipulation. Several devices (called workstations in the GKS standard) are supported. They can be as simple as a bitmap display with a mouse or as complex as a graphical terminal with local intelligence. Any number of devices can receive output from the graphical kernel, but only one at a time can produce input. The restriction for the input device has been introduced to prevent application programs from using an inappropriate man-machine dialogue. Routing of output to different devices requires an additional transformation step for the coordinates. Together with the standard transformations this leads to the following sequence:

- **The modelling transformation** consists of a rotation specified by the angles with the three axes, a scaling with a different factor for every axis, a three-dimensional translation, and a projection specified by the centre of perspective.

- **The viewing transformation** consists of a window and of a viewport. A window defines the visible part of the object and is specified in world coordinates. A viewport defines the portion of the screen on which the window shall be mapped and is specified in normalized device coordinates.

- **The workstation transformation** maps the normalized viewport for every active device into the physical workstation viewport. It is also called a viewer.

To manipulate and process the data in subsequent sessions, a simple data base built upon the Tree File System has been implemented. To obtain acceptable response times the part of the data base containing displayed data is loaded into main memory when a session is started. Because of heavy memory problems at the project's start, the permanent data base is updated with the temporary data base only when there is a move in the data base and at the end of a session.

The data is structured as follows. A picture is the set of objects that can be manipulated at the same time. An object is the set of items with the same set of transformations. The atomic elements are called items, which can be points, sets of consecutive straightline or curve segments, or text strings.

The following characteristics apply to the graphical kernel:

- The internal structure is hierarchical.

- The interface to the application writer is an abstract transformation processor. The application writer 'sees' only pictures, objects, or items, all in world coordinates. Where device coordinates would be required, e. g. for the specification of viewports, a normalized system is used. The abstract transformation processor is the number crunching part of the system. It performs a large number of floating-point operations and
produces a small number of results. These results are processed by the display code generator, which generates code for a normalized device and triggers the various workstation drivers. These produce the data that results in the display of a picture.

- It has a single and global variable called the state which contains the information about the state of the system and descriptors of the data. This gives a good overview of the system, reduces parameter passing in procedures to a minimum, and frees all the data space when the kernel is not active.

- To save memory, all data is stored in the dynamic memory of the machine instead of on the stack. The coordinate values are stored into dynamic arrays, implemented with the aid of descriptors. In this way only the amount of memory corresponding to the effective number of coordinate triples is physically allocated.

2.2.3. Kernel interface to programs

The kernel of the geometrical workbench supplies the following services:

- it defines the structure of the x-queue elements,
- implements the comparison operations on them,
- handles string input, output,
- handles system exceptions,
- and provides some instrumentation tools.

The procedure Handle-Exception displays meaningful error messages using a minimum of memory when exceptions occur in the basis software modules. The supported modules are: the loader, the file systems, the graphical kernel, and the plane-sweep skeleton. If a system component encounters a - possibly fatal - error, it flags the error type in the variable exception and calls Handle-Exception. As a parameter, it passes the identification of the module in which the error occurred. The geometrical workbench then loads a program relative to the problem source which reads the exception flag, displays a comprehensive error message and waits until the return-key is pressed on the keyboard. If the geometrical workbench crashes, it displays a message and reloads itself as soon as the user hits a key on the keyboard.

A set of procedures provides the instrumentation needed to perform the efficiency analysis. The instruments are: frequency counters for the data structure operations, and a timer. There is a counter for each data structure operation. Reset-Frequencies resets all counters. Upon being executed, each data structure operation calls the procedure Inc to increment its frequency counter. Application programs requiring the rate of occurrence of the operations call the procedure Frequency to read the values of the counters. The personal computer used is a single-user, mono-programming system. The execution time can be measured directly, without the need of statistical techniques, because there is no randomness. The system clock is reset by calling the procedure Start-Timer and read with Lap-Time.

2.3. Plane-sweep skeleton

2.3.1. Application specific abstract data types
In 1.3.1 we have presented the abstract data types inherent to the plane-sweep. The application specific information about a cross section usually requires an own abstract data type. For sake of generality, it cannot be integrated into the plane-sweep skeleton. It must reside in the application program.

However, the y-table and the application specific data structure both contain the information about a cross section, therefore they behave similarly. Thus the y-table should provide as much as possible for the application specific data structure while remaining general.

In our implementation the y-table is augmented by an eyelet for each interval it contains, into or among which the application can hook or thread his own data structure. In this way the plane-sweep skeleton automatically performs on the application structure the operations deriving from the change of topology from one transition to the next. If an application requires more than one data structure, it can all be hooked into the same eyelet.

The application program must be able to update its information at each transition. This requires an extension to the procedure Transition.

2.3.2. Transition procedure

The extension applied to the transition procedure breaks it down into two separate procedures. The first is called Update-Y-Structure and has the purpose of keeping the y-table consistent when the sweep line passes over a transition point. It consists of the case statement presented in section 1.3.2. All the application specific operations including the maintenance of user data structures are performed by a second procedure called Process-Slice which is supplied by the application program. Therefore the plane-sweep algorithm has the following modified structure:

```
PROCEDURE Sweep;

x := sorted events, constructed from the given points;
y := (-\infty, +\infty);
WHILE x \neq \emptyset DO
  p := Min (x);
  Update-Y-Structure (p);
  Process-Slice (p) (* supplied by the application program *)
END
END Sweep;
```

This is the plane-sweep skeleton. It accomplishes the bulk of the work, i.e. all data structure management involved in the advancement of the transition line. The flesh is supplied to the skeleton by the user in form of a parameter Process-Slice. It is a simple procedure to perform the actual application processing on the data fetched from the x-queue. For this reason the slice is always to the right of the transition line. In the example of raster-scan conversion, Process-Slice maintains depth queues and uses a simple incremental algorithm to produce the scan lines (see chapter 5).

2.4. x-queue

2.4.1. Element structure

The x-queue elements contain all the information required to accomplish the operations at a
transition point. This information consists of:

- a search key to sequence the events
- the transition point
- the segments whose intersection is the transition point
- the type of the transition point
- all the information required by application programs

This translates into the following structure of the elements of the x-queue:

\[
\text{TYPE XREC = RECORD}\\
\text{key : REAL;}\\
\text{s, t : SEGMENT;}\\
\text{type : TRANSITION-TYPE;}\\
\text{owner: CARDINAL}\] 

\[
\text{END;}
\]

where

\[
\text{TRANSITION-TYPE} = (\text{start, bend, end, crossing});
\]

\[
\text{POINT} = \text{RECORD x, y: INTEGER END};
\]

\[
\text{SEGMENT} = \text{RECORD a, b: POINT END}.
\]

The first field of the record contains the search key. The two segments incident at the transition point are stored in the fields \(s\) and \(t\). They define the transition point which needs not be stored explicitly. When the segments are filled into the x-queue, they are always stored from left to right, i.e. such that \(x_{s1} < x_{s2}\) (in section 1.2 we introduced the assumption that there are no vertical segments). This simplifies the processing of the x-queue in the transition procedures. Further \(s\) and \(t\) are ordered (by Update-Y-Structure), depending on the transition type, according to the following scheme:

- \(\text{start}\): \(s\) is the upper segment just after the transition line;
- \(\text{bend}\): \(s\) is the segment whose right endpoint is the transition point;
- \(\text{end}\): \(s\) is the upper segment just before the transition line;
- \(\text{crossing}\): \(s\) is the upper segment prior to intersection.

The fourth field of the record contains the transition point type, and the last field of the x-queue elements, \textit{owner}, is a cursor into a structure containing the information required by the application programs about the polygons in the configuration.

\section*{2.4.2. Operations}

At this level the operations are independent from the specific data structure selected to implement the x-queue.

The event fetched from the x-queue is not passed as a procedure parameter but is stored in a common buffer accessible by any module. Since a slice is defined by two transition lines, the
buffer contains two elements with the first decisive for the topology of the slice. The buffer also contains a constant sentinel, which is put into the buffer storage space of the second element when the x-queue is empty. The declarations are:

```plaintext
VAR Xrecord, lookahead: XREC;
Xsentinel: XREC (* Marks the last element in the x-queue *)
```

The parameterless procedure `Open-Xqueue` initializes the data structure and fills the originally given points into the x-queue. First it reads from the schedule (kernel of the geometrical workbench) the name of the current node in the data space. The corresponding geometrical data is read, and the events are constructed and inserted into the x-queue.

`Close-Xqueue` releases the resources allocated for the x-queue. When the data structure is in main memory, this procedure is empty because there is no system overhead.

The parameterless procedure `MinX` retrieves and deletes the next event from the x-queue data structure and stores it into the variable `lookahead`. When the x-queue is empty it moves the sentinel to `lookahead`. The move of the previous element to the `Xrecord` is accomplished by the plane-sweep skeleton.

`InsertX` adds a new record into the x-queue. The same intersection point may be found several times at different transition lines. This is because the plane-sweep algorithm finds the intersection points by intersecting the neighbouring segments at each transition. The same pair of intersecting segments may become adjacent more than once. Obviously the record should be stored only the first time, since it is a single event.

### 2.4.3. Heap data structure

The x-queue is a priority queue. Several data structures are used to implement priority queues. Linear lists, either sorted or unsorted are not suitable, because either `insert` or `Min` has a linear growth rate. From a practical point of view a bucket structure is ideal, because the average time complexity is constant and the storage requirements are small compared to a partially ordered tree organized as a heap. On the other hand the worst case time complexity of the bucket structure is linear.

A partially ordered tree (POT) is an implementation of priority queues requiring only $O(\log n)$ time for the operations when it contains $n$ elements (see e. g. [Aho et al. 1983]). The following procedures are needed to implement the operations described above:

- **Init**: the initialization routine.

- **Queue**: a new element is inserted at the leftmost free place at the lowest level $h$ of the POT and then pushed up the tree as long as the parent element has a larger priority key.

- **Dequeue**: fetches the element with the smallest key. It is removed from the root of the tree. To preserve the tree structure, the rightmost element is deleted at the lowest level $h$ and inserted at the root. This element then is sifted down the tree to restore the partial order.

To realize the x-queue, an array implementation of partially ordered trees is selected. This representation of a POT is called a heap.

The POT allows duplicates but, because of the partial order, elements with the same keys are not
neighbours. It is not possible to scan the data structure for the elements prior to inserting them, because this is an $O(n)$ operation when the POT contains $n$ elements. Therefore in our implementation duplicate events are inserted into the x-queue and are identified only when they are fetched.

We use the given key - the $x$ coordinate - and an auxiliary list. Each time the elements dequeued from the heap have a new search key, the list is initialized. As long as successive elements have the same key they are compared with the other elements in the list. If there is a match, the element is discarded and a new one is fetched. Otherwise the element is put both into the common buffer and into the list. When the key changes the list is removed.

Brown has proposed a modified intersection algorithm that solves this problem [Brown 1981]. His solution consists of storing only one intersection for each segment at any one time. Implementing the queue with a balanced tree, and introducing an array of intersection points for each segment, Brown proceeds as follows: If the intersection point of two segments is to the right of the next intersection for both segments, the new intersection point is ignored. Otherwise it is inserted. At the same time any other intersection points of one of the segments which is to the right of the new intersection is deleted from the queue. The storage is only $O(n)$, and the time complexity of the algorithm is not affected because in a balanced tree the required operations are performed in time $O(\log n)$.

2.4.4. Bucket data structure

A characteristic of bucket data structures is that on the average they require constant time for the Insert, Delete and Search operations. This allows an efficient solution to the duplicate problem. In the worst case these operations require time proportional to the number of elements they contain. A further advantage over the heap is that it requires physical storage space proportional to the number of elements it contains.

It consists of an array called the bucket table, indexed by a bucket number. Each field of the array is the head of a list of x-queue elements. Each bucket is an equivalence class, i.e. the elements of each list are those whose key is the same bucket number. Figure 4 illustrates these concepts.

Chapter 3 describes the necessity of knowing the range of the coordinate values in order to judge the robustness of the implementation of sweep algorithms. Knowing the coordinate values allows one to use the rounded value of the transition point $x$ coordinate as the bucket number.

In order to simplify the code of the operations and hence reduce the running time of the program, each list is ordered by the unrounded transition point $x$ coordinate. The heads of the list are stored in the lists rather than in the bucket table, and the lists are provided by a tail which is used as a sentinel. With this scheme a new element can be inserted into the data structure with a simple while-loop.

Thus the data structure can be declared as follows:

```
TYPE LINK = POINTER TO EL;
EL = RECORD
  data: XREC;
  next: LINK
END;
LIST = RECORD
  a, z: LINK (* head and tail *)
```

```
Fig. 4. The bucket method: a bucket table whose elements point to the list of the elements with an equivalent key.

VAR \text{Xqueue}: \text{ARRAY} \ [\text{minX} .. \ \text{maxX}] \ \text{OF} \ \text{POINTER TO LIST};
\text{current-Bucket}: \text{INTEGER} \ \text{(cursor \ *)}.

The operations are implemented as follows:

- The initialization routine sets the cursor \text{current-Bucket} to \text{minX}.

- The internal procedure \text{Initialize-List} allocates a new list. It creates the head \text{a} and the tail \text{z}. Then the sentinel is put into the tail and the pointers of \text{a} and \text{z} are made to point to each other.

- The insertion routine allocates a new list if the bucket is empty. The key of the new value is then put into the tail of the list as a sentinel. The ordered list is traversed to find a place. If the new element is not yet in the list, it is inserted.

- The procedure for fetching an element is straightforward. When a list becomes empty, it can be fully released because the plane-sweep algorithm performs no backtracking. Experiments have shown that the consequent prevention of memory fragmentation significantly improves the asymptotic running time of programs which are based on the plane-sweep algorithm.
2.5. y-table

2.5.1. Element structure

The y-table is the monotone sequence of the intervals of a cross section. The intervals are stored by storing the delimiting segments. Only the segment with a smaller y coordinate at the transition point is stored. Hence the elements have the following structure:

```plaintext
TYPE YREC = RECORD
  S: SEGMENT;
  eyelet: POINTER
END.
```

In addition to the segment, each element of the y-table contains an eyelet into which application specific data structures are hooked or threaded (see subsection 2.3.1). Physically an eyelet is an address field. To retrieve the data, the type conversion (recast) artifact available in many programming languages must be used.

2.5.2. Operations

The following table operations must be provided:

- **Insert**: given a line segment s, insert it at the proper place determined by the y value of s at the current x value.
- **Delete**: given a line segment s, delete it from the table.
- **Successor, Predecessor**: given segment s and the current x value, return the neighbouring segment above or below s.

The elements of the y-table do not contain a static key. The data structure is accessed by the y coordinate of the segments in a sufficiently small interval around the abscissa of the transition point. When the operations traverse the data structure, the keys are dynamically computed by evaluating the equations of the segments.

2.5.3. Data structure

For the organization of the data in the y-table, the selected data structure is a height-balanced binary tree [Adelson-Velskii and Landis 1962]. The elements are called nodes and have the following structure:

```plaintext
TYPE TABLE = POINTER TO NODE;
NODE = RECORD
  S : SEGMENT;
  eyelet : ADDRESS;
  left, right: TABLE;
  balance : (¬, ↔, ∨)
END.
```

The data and eyelet fields are the plane-sweep data. The other three fields are specific to
height-balanced binary trees.

Left and right are the pointers to the left and the right subtree.

The balance field is required to rebalance the tree after an insertion or a deletion. It takes the values $\checkmark$, $\blacksquare$, $\checkmark$. The value $\checkmark$ (resp. $\blacksquare$) stands for 'before insertion the left (resp. right) subtree of this node was higher than the right (resp. left) subtree' and the value $\blacksquare$ stands for 'equally high subtrees'. The insert operation initializes this field to $\blacksquare$, because just after the insertion, a new node is always a leaf.

The term TABLE can indicate a pointer to the whole y-table, to a part of it (subtree), or to an individual element.

Now that the element structure has been presented the implementation of the operations will be described.

The initialization routine is straightforward: it assigns the value $\lambda$ to the root.

The insertion procedure is programmed recursively, operating in three stages. It traverses the tree along a root to leaf path evaluating the equations of the segments to determine whether or not the segment is already in the tree. If this is not the case, the element is inserted as a new leaf. In the final stage, the path from the root to the new leaf is traced backwards. At each node the balance field is examined to check whether the tree must be rebalanced.

In height-balanced trees the deletion operation is more complex than the insertion operation. While an insertion requires at most one transformation to rebalance the tree, a deletion may require a transformation at each node along the path from the deleted node to the root of the tree. However, this does not affect the logarithmic time complexity.

There are two ways to implement the Successor and the Predecessor operations. The first is to extend the nodes to contain a pointer to the father. The second is that the Search operation can store in global variables the pointers to the current node, to its father, to a right-Mark and to a left-Mark. It also stores in a global variable the direction of the last move. During tree traversal the Search procedure always updates the pointers in the variables for the current and the father node. If the traversal direction changes, the pointer to the current node is stored into the variable left-Mark or right-Mark depending on which side the direction changes.

For the y-table the second method has been choosen. The global variables are:

VAR current, father, left-Mark, right-Mark: TABLE;
last-Move: (LEFT, RIGHT, UNDEFINED).

The procedure for implementing the search operation is straightforward:

PROCEDURE Search (x: SEGMENT; VAR t: TABLE; VAR found: BOOLEAN);
IF t = $\lambda$ THEN found := FALSE
(• In this case the current node is the father ! •)
ELSE
   father := current; current := t;
   IF x below t+.data THEN
      last-Move := LEFT; left-Mark := t;
      Search (x, t+.left, found)
   ELSE IF x above t+.data THEN
last-Move := RIGHT; right-Mark := t;
            Search (x, tt.right, found)  
ELSE IF y = tt.data THEN found := TRUE
END
END
END Search.

For future extensions these procedures are implemented in a more general form than would be required for the maintenance of the y-table.

PROCEDURE Successor(probe: SEGMENT; VAR above: SEGMENT;
                        VAR t: TABLE; VAR done: BOOLEAN);
VAR hit: BOOLEAN; i: TABLE;
current := left-Mark := A; last-Move := UNDEFINED;
done := TRUE;
Search (probe, t, hit);
IF hit THEN
  IF last-Move = LEFT THEN
    IF currentt.right = A THEN above := fathert.data
    ELSE
      i := currentt.right;
      WHILE i.t.left <> A DO i := i.t.left END;
      above := i.t.data
    END
  ELSE (not Nt.)
    IF last-Move = LEFT THEN above := currentt.data
    ELSE
      IF left-Mark = A THEN done := FALSE
      ELSE above := left-Markt.data
    END
  END
END
END Successor.

2.6. Skeleton Interface to the application programs

2.6.1. Information flow and usage of the module

The only system component the application writer needs to know is the definition module Plane-Sweep. This module accepts the application specific initialization and transition routines and the hooks of the user data structures. The plane-sweep module advances the transition line and maintains the data structures, x-queue and y-table. The result of the application is produced by the user-supplied transition routine Process-Slice.

According to this scheme the objects of the plane-sweep interface can be subdivided into two classes: those supplied by the application writer using the plane-sweep skeleton and those supplied by the skeleton (figure 5).

Before starting the procedure Sweep, the user supplies the plane-sweep skeleton with a
procedure to initialize the application specific data structures and one to perform the application processing in each slice. To pass the initialization procedure, the skeleton exports the procedure `Set-Sweep-Initialization`. The initialization procedure has two parameters: the next event (an x-queue record) and the y-table. The procedure for performing the application processing on a slice is passed with `Set-Transition`. The application routine also has two parameters, one for the two x-queue records defining a slice and one for the y-table.

If the application program uses a specific data structure, the procedure `Hook-In-Attributes` is called to hook it into the eyelets of the y-table at the transition of type `start`. At transition points of the other types the application program uses the procedure `Inquire-Attributes` to access its specific data structure. Usually the application routine uses the procedure `Traverse-Table` to update its data structures and to produce the local result portion of the current slice.

Examples will be given in chapter 5.

2.6.2. Exported procedures

The following two procedures are used by the application program to pass the initialization routine and the procedure `Process-Slice` to the plane-sweep skeleton:

```plaintext
PROCEDURE Set-Sweep-Initialization (Init: INITIALIZE);
PROCEDURE Set-Transition (Process-Slice: TRANSITION).
```

The initialization routine takes the form:

```plaintext
PROCEDURE Init (x: XREC; y-table: TABLE).
```

The transition routine passed to the plane-sweep skeleton with the procedure `Set-Transition` has the following structure:
PROCEDURE Process-Slice (y-left, y-right: XREC; y-table: TABLE).

The first two parameters define the slice by means of two transition lines. The last parameter is the
y-table valid for the entire slice. This procedure usually consists of two parts. The first part is a
case statement for the maintenance of the application specific data structure. If there is no such
data structure, this part is omitted. The second part computes and accumulates the application
specific results.

The procedures Hook-In-Attributes and Inquire-Attributes are straightforward.

To process a slice, most application programs must traverse the y-table. To hide the
implementation of the organization of the y-table, a procedure is supplied, which processes a
cross section. It traverses the table 'in order' and executes an Action for each interval in the
y-table. Thus the processing of a slice is simple from the point of view of the application program.

PROCEDURE Traverse-Table (table: TABLE; Action: ACTION);
  IF table ≠ A THEN
    Traverse-Table (lower-table, Action);
    Action (table);
    Traverse-Table (upper-table, Action)
  END
END Traverse-Table;

where the procedure for gathering the plane-sweep application result is in the form:

TYPE ACTION = PROCEDURE (TABLE).

The procedure Sweep performs the entire plane-sweep process. This is the usual way to use the
plane-sweep algorithm. Alternatively it is possible to call the three procedures used internally by
Sweep. This consists of calling the procedure Open-Sweep, to loop on the procedure Sweep-Step
until the x-queue is empty and to finally call the procedure Close-Sweep. Thus Sweep has the
following structure:

PROCEDURE Sweep;
  Open-Sweep;
  WHILE x ≠ ∅ DO
    Sweep-Step
  END;
  Close-Sweep
END Sweep;

Open-Sweep starts by calling the procedure Open-Xqueue from the module X-queue to initialize
the x-queue using the input. After creating the y-table and the setting the transition line to x =
-∞, the two sentinels are inserted into the y-table. Its contents is the interval (-∞, +∞). Finally
the application initialization routine, passed with the procedure Set-Sweep-Initialization, is
executed.

The procedure Sweep-Step fetches the next event from the x-queue by calling MinX. After
computation of the bias values for access to the y-table (see section 3.5), it updates the y-table.
Finally it executes the application specific routine passed with the procedure Set-Transition.

PROCEDURE Sweep-Step;
  Xrecord := lookahead; MinX;
compute values for y-table access (* see chapter 3 *)
Update-Y-Structure;
Process-Slice (* supplied by the application program *)
END Sweep-Step.

Close-Sweep closes the x-queue.

2.6.3. Update of the y-table

The internal procedure Update-Y-Structure first orders the two segments it gets from the x-queue in such a way that the first segment is the upper segment, respectively the leftmost in the case of a transition of type bend. The remainder of the procedure consists of the case statement presented in section 1.3.2. Prior to the y-table operations the appropriated bias value $\epsilon$ is selected (see section 3.5). After the y-table operations, possible exception conditions are handled. See the following example of the case of a start point:

CASE start: IF $y(s, x+\epsilon) < y(t, x+\epsilon)$ THEN Swap $(s, t)$ END;
( * Now $s$ is the upper segment *)
SetX (present); (* bias for y-table access *)
Successor $(s, h, Ytable, ok)$;
IF NOT ok THEN Handle-Exception (7) END;
Predecessor $(t, l, Ytable, ok)$;
IF NOT ok THEN Handle-Exception (8) END;
IF $h.a.y < +\infty$ THEN Intersect $(h, s)$ END;
IF $l.a.y > -\infty$ THEN Intersect $(t, l)$ END;
Insert $(s, Ytable, h, ok)$;
IF NOT ok THEN Handle-Exception (9) END;
SetX (future); (* bias for y-table access *)
Insert $(t, Ytable, h, ok)$;
IF NOT ok THEN Handle-Exception (10) END.

The procedure Intersect uses an algorithm to infer the intersection of two segments. If they intersect, the intersection point is computed and an x-queue element is constructed. If the intersection point is to the right of the transition line and different from the left transition line of the current slice, then it is inserted into the x-queue. This algorithm will be presented in section 4.3.
3. Influence of finite arithmetic

3.1. Statement of the problem

In the previous chapters, no assumptions have been made about the arithmetic of the target machine. For each transition point it has been assumed that the search keys for the x-queue and the y coordinates of the segments in the y-table are represented exactly in the machine. On a computer it is possible to implement ordered data structures because the representation of real numbers as floating-point numbers has the following monotonicity property [Rutishauser 1976]. If \( a, \beta \in \mathbb{R} \) and \( a, b \) are their floating-point representation, and if \( a \) and \( \beta \) are not in the overflow range of the machine, then:

\[
\begin{align*}
    a < \beta & \Rightarrow a \leq b, \\
    a = \beta & \Rightarrow a = b, \\
    a > \beta & \Rightarrow a \geq b.
\end{align*}
\]

Different real numbers are mapped on the same floating-point number because the continuum \( \mathbb{R} \) is mapped on the discrete set of machine numbers. In the following we use the property that the pre-image of a floating-point number is a connected set of \( \mathbb{R} \).

Searching involves the comparison of record keys. If two keys are identical then the records are considered equivalent. If two different keys have the same floating-point representation, then an insert operation could misplace the corresponding records. For the x-queue e.g., this means that the program no longer performs a unidirectional sweep of the plane.

Example 1: Consider the configuration shown in figure 6. At the first four transitions the content of the y-table is:

1: s
2: s t
3: s t v
4: s t u v

The boldface segments indicates where the binary search starts. If the x coordinates of the fifth and the sixth transition have the same floating-point representation, the corresponding events might be fetched in the wrong sequence from the x-table. If the two transitions are transposed, the content of the y-table after processing the point of type crossing is:

6: v t u s

If processing the point of type end we try to delete segment u, we do not find it, because u is above t and above s, but s is the last element of the y-table.

The objective of this chapter is to specify when strict monotonicity \( (a < \beta \Rightarrow a < b) \) can be preserved.

3.2. Nature of the Input data

In all potential applications of plane-sweep, the input data is discrete and finite. In the example of
If the floating-point representation of the search key for the transitions 5 and 6 are the same and the events are processed in the wrong order, the \( y \)-table no longer is ordered and the deletion of segment \( u \) in the transition 5 fails.

VLSI design, the components are characterized by the feature size \( \lambda \) and the chips are delimited by the wafer size. In typography there is a raster on which all points lie. In computer graphics, the common input devices - tablets, joysticks, mice, light pens, tracker balls - deliver integer coordinates, delimited by the tablet or screen size.

Note. The output is not integer valued in all applications. Because of arbitrary rotations, the geometrical modelling programs used in CAD create figures described by floating-point coordinate values. In the applications presented in chapter 5 the output is scaled in order to get a sufficiently precise representation by rounded integer coordinate values.

For ease of exposition it is assumed that the values of the input coordinates are integers \( x, y \in [0, M] \). The set of all points, whose coordinates with respect to an orthogonal system are both integers, forms the plane unit lattice. Its points are called lattice points. In the following lattice will refer to the square \( [0, M] \times [0, M] \).

### 3.3. Values of the search keys of the x-queue

The plane-sweep algorithm finds polygon intersections by analysing the intersections of their edges. It is clear that the intersection point of two edges is no longer a point on the lattice. It can be computed using an elementary rule if it is assumed that the segments under consideration intersect exactly at one point. This assumption is restricted to this section.
The rule for the computation of the intersection point delivers the coordinates in the form of a quotient. This allows us to express the x coordinates of intersection points as rational numbers \( x = N(x) / D(x) \). The first straightline segment \( s \) is given by the points \((x_{s1}, y_{s1}), (x_{s2}, y_{s2})\), and the second segment, \( t \), by \((x_{t1}, y_{t1}), (x_{t2}, y_{t2})\). The intersection point is the solution of the system

\[
\begin{align*}
(y - y_{s1}) / (x - x_{s1}) &= (y_{s2} - y_{s1}) / (x_{s2} - x_{s1}) \\
(y - y_{t1}) / (x - x_{t1}) &= (y_{t2} - y_{t1}) / (x_{t2} - x_{t1})
\end{align*}
\]

Using an elementary rule for the solution of a linear system of two equations, the denominator of the coordinates of the intersection point is

\[ D(x) = (y_{s1} - y_{s2}) (x_{t2} - x_{t1}) - (y_{t1} - y_{t2}) (x_{s2} - x_{s1}) \]

and the numerator is

\[ N(x) = (x_{s2} y_{s1} - x_{s1} y_{s2}) (x_{t2} - x_{t1}) - (x_{t2} y_{t1} - x_{t1} y_{t2}) (x_{s2} - x_{s1}) \]

Thus the first component of the intersection point is:

\[
x = N(x) / D(x) = \frac{[(x_{s2} y_{s1} - x_{s1} y_{s2}) (x_{t2} - x_{t1}) - (x_{t2} y_{t1} - x_{t1} y_{t2}) (x_{s2} - x_{s1})]}{[(y_{s1} - y_{s2}) (x_{t2} - x_{t1}) - (y_{t1} - y_{t2}) (x_{s2} - x_{s1})]}.
\]

In the worst case \( D(x) = M^2 + 1 \). In order to distinguish different computed intersection points, the arithmetic of the machine should allow to compare x coordinates with values on a finer lattice.

We want to consider a pair of segments while we introduce a variation of the endpoint coordinates. The new endpoints are still be lattice points. Therefore the segment pair is such that a change of the endpoint coordinates gives a small change in the coordinate of the intersection point. The slope of the tangent function is maximal when the angle of the first segment pair is \( \pi / 2 \), and the effect of a variation is maximal when the segments have maximal length. The original segment pair should be the diagonals of the square with edge length \( M \).

The solution formula in the form \( \star \) is inadequate for finding the worst case of two almost equal search keys in the x-queue because it contains mixed terms in the numerator. A more convenient form is expressed as the sum of the coordinates of the left endpoints of the segments. The terms of the sum have coefficients containing the relative slopes of the segments. The geometrical idea of fixing the left endpoints and slightly tilting the segments to obtain the closest intersection point is reflected by a variation of the coefficients. The new segment's endpoints must then be adjusted to become lattice points with their corresponding segments intersecting.

In the following discussion of \( \star \), the following abbreviations are used:

\[
\begin{align*}
m_s &= -(y_{s1} - y_{s2}) / (x_{s2} - x_{s1}) \\
m_t &= -(y_{t1} - y_{t2}) / (x_{t2} - x_{t1})
\end{align*}
\]

The mixed term of \( N(x) \) then becomes:

\[ x_{s2} y_{s1} - x_{s1} y_{s2} = y_{s1} (x_{s2} - x_{s1}) + x_{s1} (y_{s1} - y_{s2}). \]
And symmetrically for \( r \). Inserting the mixed term of \( N(x) \) into \( \star \):

\[
x = N(x) / D(x) = x_{s1} [m_s / (m_s - m_1)] + x_{t1} [m_t / (m_1 - m_s)] + y_{s1} [1 / (m_1 - m_s)] + y_{t1} [1 / (m_s - m_t)].
\]

Thus the \( x \) coordinate of the intersection point is expressed as the sum of the coordinates of one endpoint of each segment, each with a 'weight' derived from the difference of the slopes of the segments. Because of the assumptions of no verticals and exactly one intersection point, the formula always holds.

To find the worst case when accessing the \( x \)-queue, that is the transition points with the closest \( x \) coordinates, a segment pair close to the diagonal of the square with edge length \( M \) is considered. The values to be selected stem from formula \( \star \star \). The first segment pair is given by \( (x_{s1}, y_{s1}) = (0, 1) \), \( (x_2, y_2) = (M, M) \), resp. \( (x_{t1}, y_{t1}) = (1, M) \), \( (x_{t2}, y_{t2}) = (M, 0) \). The second segment pair is \( (x_{s1}, y_{s1}) = (0, 0) \), \( (x_2, y_2) = (M-1, M-2) \), resp. \( (x_{t1}, y_{t1}) = (1, M-1) \), \( (x_{t2}, y_{t2}) = (M-1, 0) \).

The slopes of the segments are:

<table>
<thead>
<tr>
<th></th>
<th>first segment pair</th>
<th>second segment pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_s )</td>
<td>( (M-1) / M )</td>
<td>( (M-2) / (M-1) )</td>
</tr>
<tr>
<td>( m_t )</td>
<td>( -M / (M-1) )</td>
<td>( -(M-1) / (M-2) )</td>
</tr>
</tbody>
</table>

The coefficients for the computation of the \( x \) coordinate of the two intersection points are:

<table>
<thead>
<tr>
<th></th>
<th>first segment pair</th>
<th>second segment pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{s1} )</td>
<td>( (M-1)^2 / [(M-1)^2 + M^2] )</td>
<td>( (M-2)^2 / [(M-2)^2 + (M-1)^2] )</td>
</tr>
<tr>
<td>( x_{t1} )</td>
<td>( M^2 / [(M-1)^2 + M^2] )</td>
<td>( (M-1)^2 / [(M-2)^2 + (M-1)^2] )</td>
</tr>
<tr>
<td>( y_{s1} )</td>
<td>( -[M(M-1)] / [(M-1)^2 + M^2] )</td>
<td>( -(M-2)(M-1) / [(M-2)^2 + (M-1)^2] )</td>
</tr>
<tr>
<td>( y_{t1} )</td>
<td>( [M(M-1)] / [(M-1)^2 + M^2] )</td>
<td>( (M-2)(M-1) / [(M-2)^2 + (M-1)^2] )</td>
</tr>
</tbody>
</table>

The \( x \) coordinates of the intersection points are:

<table>
<thead>
<tr>
<th></th>
<th>first segment pair</th>
<th>second segment pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( [M(M^2-M+1)] / [(M-1)^2 + M^2] )</td>
<td>( (M-1)^3 / [(M-1)^2 + (M-2)^2] )</td>
</tr>
</tbody>
</table>

The resolution required from the arithmetic of the machine is the distance between the two \( x \) coordinates of the intersection points. Omitting the lengthy calculation, the result is:

\[
1 / (4 M^4 - 16 M^3 + 24 M^2 - 16 M + 5).
\]

This result has been confirmed experimentally with the aid of a simulation program [Fink 1984].
3.4 Criterion for the robustness of plane-sweep implementations

In this section, an approximate criterion for the number of binary digits required for the storage of the fraction of floating-point numbers is derived. The criterion can be used to decide if on a given machine, the strict monotonic order of the x-queue keys can be retained. The endpoints of the segments in the configuration are assumed to lie on a unit lattice with \( x, y \in [0, M] \).

It is assumed that the machine uses a floating-point system with base 2, and \( \log \) denotes the logarithm to the base 2. A floating-point number \( x \) is represented as two quantities: an exponent part \( e = \text{Floor}(\log |x|) + 1 \) and a signed fraction part \( f \). The floating-point numbers are assumed to be normalized, which means that \( \frac{1}{2} \leq |f| < 1 \). Let \( b \) be the number of bits used for the fraction part.

Information can be lost because of the finite number of binary digits for the representation of a normalized number. This situation is called cancellation.

An example of cancellation is:

Example 2: Suppose a machine with decimal arithmetic and three decimal digits for the storage of numbers.

\[
10^2 + 10^{-1} = 10^2 + 0.0102 = 10^2.
\]

The sum is incorrect because of cancellation.

Cancellation is not a problem when the values of two keys are far apart, that is if their floating-point representations have different exponents. More formally, two floating-point numbers are said to be almost equal if they have the same exponent. Two almost equal search keys \( k_i, k_j \) with \( k_j < k_i \) have different floating-point representations if there is at least one bit of difference in their fraction, that is, when one abbreviates \( e(k_i) \) by \( e_i \),

\[
e_i - e_j < b.
\]

Inserting the worst case calculated in the previous section:

\[
\{\text{Floor} \left[ \log \left( \frac{M(M^2 - M + 1)}{(M - 1)^2 + M^2} \right) \right] + 1 \} + \\
- \{\text{Floor} \left[ \log \left( \frac{1}{4 M^4 - 16 M^3 + 24 M^2 - 16 M + 5} \right) \right] + 1 \} < b.
\]

Hence the criterion for the resolution of the arithmetic of the machine necessary and sufficient for representing different search keys by different floating-point values is:

\[
b > \text{Floor} \left[ \log \left( M(M^2 - M + 1) \right) - \log \left( (M-1)^2 + M^2 \right) \right] + \\
- \text{Floor} \left( -\log \left( 4 M^4 - 18 M^3 + 24 M^2 - 16 M + 5 \right) \right).
\]

Such a criterion is not convenient for practical use. Therefore the right side of the formula is increased to obtain a simpler expression. The resulting approximate criterion is sufficient but not necessary.

The following properties of the floor function are used to increase the right side of ↑:
a > 0 \Rightarrow - \text{Floor}(-a) \leq \text{Floor}(a) + 1
\quad a, b > 0, \quad a > b \Rightarrow \text{Floor}(a b) \leq \text{a Floor}(b) + (a - 1).

Thus \( \uparrow \) is simplified to

\[
\begin{align*}
& \quad b > \text{Floor}(\log M^3 - \log M^2) - \text{Floor}(-\log M^4) \\
& \quad b > \text{Floor}(\log M) + \text{Floor}(4 \log M) + 1 \\
& \quad b > \text{Floor}(\log M) + 4 \text{Floor}(\log M) + 4 \\
& \quad b > 5 \text{Floor}(\log M) + 4.
\end{align*}
\]

Further enlarging the right side by eliminating the floor function leads to the following approximate criterion for the number of bits required in the fraction part of floating-point numbers in order to access the x-queue correctly:

\[
\begin{align*}
& \quad b > 5 \log M + 4.
\end{align*}
\]

This simplified criterion is tested in the following example.

Example 3: If the plane-sweep skeleton is used to produce an 8\( \frac{1}{2} \times 11 \) inch raster with a resolution of 200 dots per inch (M = 2112), then the machine should represent floating-point numbers with at least 58 bits in the fraction. To compute \( \uparrow \) on a 60-bit machine \( \uparrow \) must first be written in a more stable form from a numerical point of view.

\[
\begin{align*}
& \quad \text{Floor} \left\{ \log [M(M^2 - M + 1)] - \log [M - 1)^2 + M^2] + \\
& \quad \quad - \text{Floor} [-\log [4 M^4 - 16 M^3 + 24 M^2 - 18 M + 5]] = \\
& \quad = \text{Floor} \left\{ \log [M(M - 1) + 1)] - \log [(M - 1)^2 + M^2] + \\
& \quad \quad - \text{Floor} [-\log [4 M (M (M - 4) + 6) - 4] + 5]].
\end{align*}
\]

The numerical result is

\[
\begin{align*}
& \quad b > \text{Floor}(10.04440) - \text{Floor}(-48.17484) = 10 + 47 = 57.
\end{align*}
\]

Computing \( \uparrow \uparrow \) on the same machine yields

\[
\begin{align*}
& \quad b > 59.22197.\,\text{J}
\end{align*}
\]

The criterion is based on worst cases. In practical applications such as drafting in document processing, the worst case for the x-queue is infrequent. At the transition points, the plane-sweep algorithm depends only on the neighbourhood of the vertices fetched from the x-queue. If two events misplaced in the x-queue have non-intersecting neighbourhoods, the plane-sweep works correctly. Otherwise a failure will occur at a later point because the y-table has become inconsistent.

### 3.5. Values of the search keys of the y-table

As has been seen in section 2.6.2, the y-table is accessed by evaluating the linear equations of the segments it contains, to obtain their y coordinate on the transition line. This value is used as a dynamic search key with the segment with the greater y value lying above. In other words the order of two segments in the y-table is computed with the help of a signed distance on the transition line.
In the neighbourhood of points of the types start, end, or crossing, the distance between two segments becomes arbitrarily small. On the transition line the segments of the current event intersect. In our implementation we use the property that the topology of the configuration swept does not change inside a slice. According to the definition of the slices, there are no intersecting segments inside a slice.

The implemented solution consists of a (signed) bias $\varepsilon$ to evaluate all straightline equations at $x = x_y + \varepsilon$ (see figure 7) where $x_y$ is the abscissa of the vertical transition line. The order of two edges is inferred too by computing the signed distance $\delta$.

Note. Since $\delta = \varepsilon (\tan \alpha - \tan \beta)$ the distance is not invariant under rotation. Therefore the sweep of a configuration may fail if the configuration is rotated according to an arbitrary angle with a subsequent rounding of the coordinate values to integers.

The numerical problems occurring when the y-table is accessed stem from the computation of the distance. The ordinate of a segment point is the sum of the ordinate of the left endpoint and the slope multiplied by the abscissa less the x coordinate of the left endpoint. If the segment is almost horizontal and the y coordinates of the endpoints are large, then cancellation occurs. In the machine representation, the segment appears to be horizontal. If this phenomenon occurs with another segment which is very close, the segments cannot be distinguished.
The informal concepts 'almost horizontal' and 'very close' are now discussed in more detail. From figure 7 it can be deduced that the worst case is given if one segment is horizontal and the other has the slope \(1/M\). The order of magnitude of the distance is illustrated in the following practical case:

Example 4. Suppose \(M\) is even and consider the segments with endpoints \((M/2, M/2), (M, M/2)\), and with endpoints \((M/2, M/2), (M, M/2-1)\) \((\alpha = 0\) and \(\tan \beta = -2/M\)). This is a point of type start, and for \(y\)-table access the keys are calculated at \(x = M/2 + \varepsilon\). Since the order can change outside the current slice, \(\varepsilon < 1 / (4M^4 - 16M^3 + 24M^2 - 16M + 5)\) should hold. For simplicity of exposition the simplified case \(\varepsilon = M^{-4}\) is considered. Introducing these values into the formula \(\delta = \varepsilon (\tan \alpha - \tan \beta)\) yields a minimum distance of the order of magnitude of \(2/M^5\).

Comparing the result of the example with the approximative criterion \(\varepsilon\), the number of bits required for the representation of the fraction part of floating-point numbers becomes of the order of

\[b > 6 \log M + 4.\]

Example 4 is not an artificially constructed worst case. It occurs frequently. When the angle of two intersecting almost horizontal segments is small, the machine representation of the distance \(\delta\) may disappear. Therefore it seems difficult to achieve a robust implementation on a personal computer without multiple precision arithmetic. Fortunately, this problem can be solved for the \(y\)-table by using simple methodology from numerical mathematics.

3.6. Increasing the robustness of the \(y\)-table access

The first measure to take is to arrange properly the order of the operands in the computations [Henrici 1980]. Because of cancellation, first the small and then the large operands should be summed. The slopes \(m_8\) respectively \(m_1\) of the two segments are computed. Then the quantities \(u_8 := m_8(x - x_8)\) and \(u_1 := m_1(x - x_1)\) are computed. The distance is given by

\[\delta = (u_8 - u_1) + (y_8 - y_1).\]

With this method almost all computations deliver a correct result. If cancellation still occurs, that is if the segments seem still to be superimposed, a collinear translation is applied to one of the segments until one of its extrema has the same \(x\) coordinate as an extremum of the other segment. Because the segments are on a resolvable lattice, the order can now be established because integer arithmetic is used.

Note. In the trace of the sweep of several critical configurations this case has never occurred. This method is of interest in the related problem of braiding lines.

The second measure is to choose the largest possible bias \(\varepsilon\). The farther the computations of the distance are away from transition lines, the 'safer' they are. Instead of using a fixed bias, a bias that is adaptable to the slice being processed is used. Since by definition there are no topological changes inside a slice, the distance is computed in the middle of the slice. That is, the bias chosen is half of the slice size.

Note. In section 2.4.1 it has been noted that the procedure Update-Y-Structure orders the two segments that give
origin to the transition event such that the first segment is the upper segment. To improve program readability the bias has been implemented as a set of three values: ±\( \varepsilon \) and 0. At the transitions of the type start, the bias is \( +\varepsilon \), at crossing and end: \( -\varepsilon \), and at bend: 0.

However, this measure introduces two new problems. Since multiple events with the same key are allowed, a slice may have the size zero. A further problem with this effect is cancellation. When a slice is so thin that the criterion \( \uparrow \uparrow \) does not hold, the distance between the two transition lines is zero in floating-point representation.

The adopted solution is to apply a threshold \( \theta \) to the value of the variable bias \( \varepsilon \): if \( \varepsilon < \theta \) then \( \varepsilon = \theta \). The correct choice is to set the threshold to half of the smallest possible difference between the two \( x \) coordinates. If this value is not represented in the machine, the criterion of section 3.4 can be used to select a suitable value.

Example 5. For \( M = 2^{112} \) (see example 3) the mathematically correct value of \( \theta < 2^{-59} \) leads to cancellation on a machine with 24 bits in the fractional part. To retain at least one bit in the floating-point representation, for \( M = 2^{112} \), the threshold must be \( \theta = 2^{-12} = 1/4096 \). This guarantees that \( M + \theta \neq M \).
4. Efficiency analysis

4.1. Memory requirements

On the Lilith personal computer, the program code occupies the following storage space (1 word = 2 bytes):

- Plane-sweep skeleton (x-queue with heap organization) 7669 words
- Plane-sweep skeleton (x-queue with bucket organization) 4681 words.

In addition, the data structure of the y-table requires space for a stack for the recursive execution of the procedures Insert, Delete, Search and Traverse-Table.

For the actual data, the data structures require the following storage space:

- x-queue with heap organization 12 words for each element
- x-queue with bucket organization 13 words for each element
- y-table 8 words for each element.

A typical application such as raster-scan conversion (section 5.3) requires an additional 2040 words of storage space. The application program which computes the area of the polygons in a configuration (section 5.4) requires 883 words of storage space. These figures do not include the storage space required by the data in the application specific data structures.

Note. The geometrical workbench (operating system) requires 17068 words of storage space.

4.2. The constant factor of the running time

The running time of a program based on the plane-sweep algorithm is \( T(n,s) = c (n + s) \log n \) (see subsection 1.3.3). The constant factor \( c \) has been estimated for the Lilith personal computer through experimentation. This section describes how configurations are generated for the experiments, the experiment data, and the results.

4.2.1. Generation of the experiment data

To produce a large number of configurations a program based on a pseudo-random number generator is used for efficiency analysis. The following configuration classes are provided for:

- **Squares** generates a set of squares of equal size (see figure 8);
- **Rectangles** generates rectangle pairs, where each pair consists of a rectangle with the longer edges parallel to the y-axis and a rectangle with the longer edges parallel to the x-axis (see figure 9);
- **Polygons** generates fully random polygons.

The pseudo-random number generator RMLM is based on the algorithm of Lehmer for creating a
linear congruential sequence, defined by the recursion formula

\[ x[n+1] = (a \times x[n] + c) \mod m \]

The modulus \( m \) has been chosen as the maximal machine number plus one \( (m = 2^{16}) \). To fulfill the conditions

\[ a \equiv 3 \pmod{8} \]
\[ a > m/100 \]
\[ a < m - \sqrt{m}, \]

the multiplier \( a \) is set equal to 1867, and the increment \( c \) to zero, that is, the multiplicative variant of the method is used.

The notorious effect that the triples of numbers generated with the linear congruential method lie on parallel planes of the unit cube is avoided by using the algorithm of MacLaren-Marsaglia. The latter uses a 'tank' array of 130 elements from which the elements are fetched and replaced using Lehmer's generator [M. D. MacLaren and O. Marsaglia 1965].

The configurations are based on a set of neighbouring polygons. If there are several such sets, each is called a layer. The configurations consisting of squares are called class 1, those consisting of rectangles class 2, and those consisting of random polygons class 3.

The \texttt{Squares} command prompts for the number \( f \) of layers and a number \( m \). Each layer will contain \( m \) columns of \( m \) squares. The configuration is generated according to the following scheme:

```pascal
TYPE POINT = RECORD x, y, z: REAL END;
VAR polygon: ARRAY [1..4] OF POINT;

FOR i := 1 TO f DO
  FOR j := 1 TO m DO
    FOR k := 1 TO m DO
      X := RMLM(); Y := RMLM(); Z := RMLM();
      WITH polygon[1] DO
        x := X + j; y := Y + k; z := Z
      END;
      WITH polygon[2] DO
        x := X + j + 1.0; y := Y + k; z := Z
      END;
      WITH polygon[3] DO
        x := X + j + 1.0; y := Y + k + 1.0; z := Z
      END;
      WITH polygon[4] DO
        x := X + j; y := Y + k + 1.0; z := Z
      END;
    END;
  END;
END
Output polygon to the graphical kernel
END
```

Similarly, the command \texttt{Rectangles} prompts for the number \( f \) of layers and the number \( m \) of rectangle pairs. The configuration is generated as follows:

```pascal
FOR i := 1 TO f DO (* Vertical strips *)
```

```pascal
END
END
```
FOR j := 1 TO n DO
  X := RMLM(); Y := RMLM(); Z := RMLM();
  WITH polygon[1] DO
    x := X + j; y := Y; z := Z
  END;
  WITH polygon[2] DO
    x := X + j + 1.0; y := Y; z := Z
  END;
  WITH polygon[3] DO
    x := X + j + 1.0; y := Y + m; z := Z
  END;
  WITH polygon[4] DO
    x := X + j; y := Y + m; z := Z
  END;
END;
Output polygon to the graphical kernel
END;
FOR i := 1 TO f DO
  Generate the horizontal strips
END.

These two configuration classes have been proposed by [A. Schmitt 1981] and other authors [K. Gremminger 1982]. They have been used for the efficiency analysis of sweep algorithms.

The Polygons command generates fully random configurations. Upon activation, the user can select between a coplanar or a layered set of polygons. In the first case, all polygons have the same random z coordinate. In the second case, each polygon has a different random z coordinate but the planes on which they lie are all parallel. Then the user is prompted for the following parameters:

- the number sample-size of polygons to be generated;
- the number max-edges (the number of edges of each polygon is a random number 3 < edges < max-edges);
- the maximal width b of the polygons in percent of the width of the configuration;
- the maximal height h of the polygons in percent of the height of the configuration.

The program then generates the polygon in three stages: computation of the number of edges; generation of a random polygon in a rectangle of base length b and height h; translation of the polygon by a random vector.

4.2.2. Experiments and results

The experiments are grouped in classes. A larger number of experiments is of class 3 (random polygons), such that this class gets a greater weight in the estimation of the constant factor of the program's running time by the mean. A few configurations produced with an interactive drawing editor have been included in the experiments and are classified as drawings. A complete description of experiments appears in the appendix.

The following occur in the experiments:
- not all start values allow to achieve the maximal length of the cyclic period of the pseudo-random number generator
- in some configurations, the order could not be preserved in the x-queue (see section 3.4)
- the version with the heap organization for x-queue requires more storage space both for the code and for the data

The results are shown in figure 10 (heap organization) and in figure 11 (bucket organization), where the running time is plotted against \((n + s) \log n\). The mean values of \(t / [(n + s) \log n]\) in the experiments for the heap organization are:

- drawings: 0.00287 seconds
- squares (class 1): 0.00285 seconds
- rectangles (class 2): 0.00335 seconds
- polygons (class 3): 0.00265 seconds
- all classes: 0.00272 seconds.

For the version with the x-queue in a bucket organization the results are:

- drawings: 0.00272 seconds
- squares (class 1): 0.00273 seconds
- rectangles (class 2): 0.00322 seconds
- polygons (class 3): 0.00259 seconds
- all classes: 0.00270 seconds.

4.2.3. Discussion

Both figures 10 and 11 prove the main objective of the implementation, namely that the growth rate of the implementation of the plane-sweep algorithm is \(O((n + s) \log n)\).

There are some outliers:

- the configuration of experiment number 3 (see appendix A or B, respectively lower left of figure 17) contains a large number of segments \((n = 1024)\), but only a few intersection points \((s = 192)\). This is due to the nature of the representation of the B-splines;
- the right half of configuration 4 contains a cluster of intersection points. See the upper right part of figure 17;
- configuration 72 is sparse \((n = 66)\) and contains very few intersection points \((s = 3)\) making the disk I/O during the initialization of the sweep process dominant

The constant factor is significantly higher for the configurations of the type class 2. This is related to the density of the intersection points. However, all points are on the same line. The asymptotic
\(O((n+s) \log n)\) behaviour is not affected. Only the constant factor is larger.

To judge the quality of the mean values of \(t/((n+s) \log n)\), scatter diagrams have been plotted. In figure 12 the values of \(t/((n+s) \log n)\) are plotted with the ratio \(s/n\) for the data with the x-queue organized as a heap. Figure 13 is the corresponding plot for the bucket organization.

The latter figure, where large values of \(s/n\) are included, shows that the quality of the estimation of the constant factor by the mean is good. The cluster of class 3 values in the interval (0.0023, 0.0025) seconds is due to the fact that there are few intersection points. Hence there are almost no collisions in the bucket table and the x-queue elements can be assessed in linear time. As expected, the slope of the same cluster is steep in the case of the heap x-queue organization (figure 12).

The interpretation of the scatter diagrams is confirmed by the following consideration. Experiments have shown that the clock of the particular Lilith personal computer used for the efficiency analysis has oscillations of \(\pm 3\%\). The standard deviation, which is a measure of the spread or dispersion, of the values of \(t/((n+s) \log n)\) is

- heap organization: 0.00030 seconds
- bucket organization: 0.00029 seconds

The errors caused by the running time measurements are

- 0.00010 seconds

In the worst case, as is verified by introducing the \(\pm 3\%\) error into the results of experiment 29 with the bucket data organization of the x-queue. Thus the standard deviation is of the same order of magnitude as the instrument error and the observations can be qualified as accurate.

According the same argument it can be stated that there is no significant difference in the constant factor between the two organizations of the x-queue. However there is a difference in the physical storage space requirements of the two variants. Hence it may be recommended that the bucket organization be used for the implementation of the x-queue.

### 4.3. An example of optimization

A good strategy in program optimization is to optimize the inner loops. This does not affect the growth rate of the program because the statements in loops are executed in constant time. However, it does reduce the constant factor. A frequent operation in the sweep algorithm is the intersection of segments.

At each transition, the neighbourhood of the event is determined and intersected with the segments of the transition. During a sweep, \(2(n+s)\) segment intersections are computed. However there are only \(s\) intersection points. Is there a way to avoid computing the \(2n+s\) non existant intersection points?

The traditional method to find the intersection point of two segments is to intersect the straightlines on which the segments lie. If the intersection point lies on both segments the segments are said to intersect and the intersection point of the segments is that of the straightlines.
What is needed is a method that checks whether or not the two segments intersect all $2(n+s)$ times, but that computes the intersection point only the $s$ times they actually intersect. In other words the two operations should be reversed. This problem can be resolved by transforming the configurations. However, transforming costs must be taken into account.

4.3.1. Parametric representation of segments

In the procedure `Update-Y-Structure` of the plane-sweep skeleton, a transformation may be used. The idea is to consider the parametric representation of segments. For a segment from $(x_{s1}, y_{s1})$ to $(x_{s2}, y_{s2})$ it is

$$
\begin{align*}
    x &= (1 - \alpha) x_{s1} + \alpha x_{s2} \\
    y &= (1 - \alpha) y_{s1} + \alpha y_{s2},
\end{align*}
$$

where the parameter $\alpha$ has the values $0 \leq \alpha \leq 1$.

Let $(x_{11}, y_{11})$ and $(x_{12}, y_{12})$ be the endpoints of a second segment, and $(x, y)$ the intersection point of the two segments.

In the traditional method the equations for the straightlines

$$
\begin{align*}
    (y - y_{s1}) / (x - x_{s1}) &= (y_{s2} - y_{s1}) / (x_{s2} - x_{s1}) \\
    (y - y_{11}) / (x - x_{11}) &= (y_{12} - y_{11}) / (x_{12} - x_{11})
\end{align*}
$$

are built and the system is solved for $x$ and $y$.

The segments intersect if and only if

$$
\begin{align*}
    \min \left[ \min (x_{s1}, x_{s2}), \min (x_{11}, x_{12}) \right] \leq x &\leq \max \left[ \max (x_{s1}, x_{s2}), \max (x_{11}, x_{12}) \right] \\
    \min \left[ \min (y_{s1}, y_{s2}), \min (y_{11}, y_{12}) \right] \leq y &\leq \max \left[ \max (y_{s1}, y_{s2}), \max (y_{11}, y_{12}) \right].
\end{align*}
$$

With the parametric method there are two systems of equations, one for each segment:

$$
\begin{align*}
    x &= (1 - \alpha) x_{s1} + \alpha x_{s2} & x &= (1 - \beta) x_{11} + \beta x_{12} \\
    y &= (1 - \alpha) y_{s1} + \alpha y_{s2} & y &= (1 - \beta) y_{11} + \beta y_{12},
\end{align*}
$$

The variable $x$ can be eliminated by combining the first equation of the first system with the first equation of the second system. In the same way $y$ is eliminated and a linear system of two equations with two unknowns $\alpha$ and $\beta$ remains.

The two segments intersect if and only if the parameters $\alpha$ and $\beta$ describe a segment, that is if and only if

$$
(0 \leq \alpha \leq 1) \text{ and } (0 \leq \beta \leq 1).
$$

If there is incidence, then the Cartesian coordinates are readily obtained by inserting $\alpha$ into the equation system of the first segment or $\beta$ into the equations of the second segment.
Denoting the additions by \(a\), the divisions by \(d\), and the multiplications by \(m\), the quantities of
operations may be compared.

<table>
<thead>
<tr>
<th></th>
<th>Traditional method</th>
<th>Parametric method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set up equations</td>
<td>6a 4m 0d</td>
<td>6a 0m 0d</td>
</tr>
<tr>
<td>Solve the system</td>
<td>3a 6m 2d</td>
<td>3a 6m 2d</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9a 10m 2d</strong></td>
<td><strong>9a 6m 2d</strong></td>
</tr>
</tbody>
</table>

Thus the computation of \((x, y)\) requires four floating-point operations multiplications more than the
computation of \((\alpha, \beta)\).

In the traditional method the intersection test requires 16 comparisons while the parametric
method requires only 4. Considering a comparison to have the same cost as an addition, the
parametric method saves 12 additions and 4 multiplications.

If the segments intersect, the parametric method requires the calculation of the \(x\)-queue element
\(key\), i.e. the \(x\) coordinate of the intersection point. This costs two additions and one multiplication:

\[
key = (1 - \alpha)x_{s1} + \alpha x_{s2} = x_{s1} + \alpha(x_{s2} - x_{s1}).
\]

The total number of operations saved is

\[
2(n + s)(12a, 4m) - s(2a, 1m),
\]

that is

\[
24n + 10s \quad \text{additions}
\]

\[
8n + 3s \quad \text{multiplications}.
\]

Because \((x, y)\) respectively \((\alpha, \beta)\) may be in the overflow range of the machine, these must be
floating-point operations.

Until now it has been assumed that the system of equations is regular. In practice both algorithms
must handle the case \(D(x) = 0\) (see section 3.3), but this does not affect the difference in the
balance of costs.

4.3.2 Segment intersection

The intersection routine of the plane-sweep skeleton contemplates also the case of segments
intersection in more than one point (type infinite) and bifurcations. Tasks of these types are not
inserted into the \(x\)-queue because the endpoints at which the topology of the configuration
changes are already in the \(x\)-queue. The routine has been coded as follows (see section 0.3 for the
notation):

```
PROCEDURE Intersect (s1, s2: SEGMENT);
    CONST \(\theta = 1/4096;\) (see section 3.6)
    VAR al1, al2, a21, a22, v1, v2, det, cof1, cof2, alpha, beta: REAL;
    newXelt: XREC;
BEGIN
```

43
\[
\begin{align*}
\alpha &= \frac{\text{cof1}}{\text{det}}, \quad \beta = \frac{\text{cof2}}{\text{det}}, \\
\text{IF } \left(\left(\text{abs}(\alpha) < 1.0 \text{ AND abs}(\alpha - 1.0) < 0\right) \text{ AND } \left(\text{abs}(\beta - 1.0) < 0 \text{ AND } \text{abs}(\beta) < 1.0\right)\right) \text{ THEN }
\end{align*}
\]

(* IMPLEMENTATION NOTE: Because of integer overflow reals must be used *)

\[
\begin{align*}
a_{11} &= s_{1.b.x} - s_{1.a.x}; \quad a_{12} = s_{2.a.x} - s_{2.b.x}; \\
v_1 &= s_{1.a.x} - s_{2.a.x}; \\
a_{21} &= s_{1.b.y} - s_{1.a.y}; \quad a_{22} = s_{2.a.y} - s_{2.b.y}; \\
v_2 &= s_{1.a.y} - s_{2.a.y}; \\
\text{cof1} &= a_{22} \cdot v_2 - a_{21} \cdot v_1; \quad \text{cof2} = a_{21} \cdot v_1 - a_{11} \cdot v_2; \\
\text{det} &= a_{11} \cdot a_{22} - a_{12} \cdot a_{21}; \\
\end{align*}
\]

IF \(\text{abs}(\text{det}) < 0\) THEN

IF \(\left(\text{abs}(\text{cof1}) < 0\text{ AND }\text{abs}(\text{cof2}) < 0\right)\) THEN (* ln. dep. *)

(* Check for infinite intersection *)

IF \(\text{abs}(a_{11}) < 0\) THEN (* vertical *)

IF \(\left(\left(s_{2.a.y} \geq s_{1.a.y} \text{ AND } s_{2.a.y} \leq s_{1.b.y}\right) \text{ OR } \left(s_{2.b.y} \geq s_{1.a.y} \text{ AND } s_{2.b.y} \leq s_{1.b.y}\right)\right)\) THEN

\text{newXelt.type} := \text{infinite}

END

ELSE (* not vertical *)

IF \(\left(\left(s_{2.a.x} \geq s_{1.a.x} \text{ AND } s_{2.a.x} \leq s_{1.b.x}\right) \text{ OR } \left(s_{2.b.x} \geq s_{1.a.x} \text{ AND } s_{2.b.x} \leq s_{1.b.x}\right)\right)\) THEN

\text{newXelt.type} := \text{infinite}

END

(* ELSE no Incidence *)

END

ELSE

\[
\alpha = \frac{\text{cof1}}{\text{det}}, \quad \beta = \frac{\text{cof2}}{\text{det}}; \\
\text{IF } \left(\left(\text{abs}(\alpha) < 1.0 \text{ AND } \text{abs}(\alpha - 1.0) < 0\right) \text{ AND } \left(\text{abs}(\beta - 1.0) < 0 \text{ AND } \text{abs}(\beta) < 1.0\right)\right) \text{ THEN}
\]

(* ELSE parallel *)

\[
\begin{align*}
\text{IF } \left(\left(\text{abs}(\text{alpha}) < 0 \text{ OR } \text{abs}(\text{alpha} - 1.0) < 0\right) \text{ AND } \left(\text{abs}(\text{beta}) < 0 \text{ OR } \text{abs}(\text{beta} - 1.0) < 0\right)\right) \text{ THEN}
\end{align*}
\]

(* type = end; already is in x-queue *)

ELSIF \(\left(\left(\text{abs}(\text{alpha}) < 0 \text{ OR } \text{abs}(\text{alpha} - 1.0) < 0\right) \text{ OR } \left(\text{abs}(\text{beta}) < 0 \text{ OR } \text{abs}(\text{beta} - 1.0) < 0\right)\right)\) THEN

\text{newXelt.type} := \text{bifurcation}

ELSE

\text{newXelt.type} := \text{crossing}

END

(* ELSE no Incidence *)

END

END;

IF \text{newXelt.type} = \text{crossing} \text{ THEN } (* Insert point into x-queue *)

(* IF (newXelt.type = \text{infinite}) \text{ OR (newXelt.type = \text{bifurcation}) THEN point already is in x-queue because it is a start, end or bend point. END *)

WITH \text{newXelt} \text{ DO s := s1; t := s2; owner := oo;}

\[
\text{key} := (1.0 - \alpha) \cdot s_{1.a.x} + \alpha \cdot s_{1.b.x}; \\
\text{xKey} := \text{TRUNC}(\text{key})
\]

END;

IF \(\text{newXelt.key} \geq \text{Xrecord.key}\) \text{ AND}

(* NOT \text{(SameRecord(newXelt, lookahead) OR SameRecord(newXelt, Xrecord))} \text{ THEN InsertX (newXelt)}

END

END
END Intersect.

The x-queue elements of type crossing are not assigned to a cursor of the polygon table. In fact, the plane-sweep skeleton operates only on a set of segments. The topology of the configuration is stored in the application specific data structure if it is needed. The application program contains all the necessary information in its specific data structures because the corresponding polygon is already present in a region in the previous slice.
Fig. 8. An example of configuration of class 1.
Number of layers: \( f = 4 \), number of rows of squares: \( m = 6 \).
Fig. 9. An example of configuration of class 2.
Number of layers: $l = 4$, number of rectangle pairs: $m = 18$. 
Fig. 10. Plot of the running time versus \((n + s) \log n\). Plane-sweep skeleton with the heap version of the x-queue organization.

- DRAWINGS
- SQUARES
- RECTANGLES
- POLYGONS
Fig. 11. Plot of the running time versus \((n + s) \log n\). Plane-sweep skeleton with the bucket version of the x-queue organization.

- **DRAWINGS**
- **SQUARES**
- **RECTANGLES**
- **POLYGONS**
Fig. 12. Plot of $t / ((n + s) \log n)$ versus $s / n$. Plane-sweep skeleton with the heap version of the $x$-queue organization.

- DRAWINGS
- SQUARES
- RECTANGLES
- POLYGONS
Fig. 13. Plot of $t / [(n + s) \log n]$ versus $s / n$. Plane-sweep skeleton with the bucket version of the x-queue organization.

- DRAWINGS
- SQUARES
- RECTANGLES
- POLYGONS
5. Applications

5.1. Plane-sweep tutorial

This introductory application demonstrates the principles of the plane-sweep algorithm. It performs a sweep in single step mode and displays the state of the sweep process at each transition point. The user selects one of two sights of the state. The first sight, the default sight, is a graphical display of the segments in the y-table. The second is a table containing statistics about the operations on the data structures (see section 5.2).

The commands are:

- display statistics
- perform next transition step
- sweep to end
- select a new configuration
- start a plane-sweep.

Since this tutorial program does not produce results, it does not contain an application specific data structure. The transition procedure uses the routine Traverse-Table exported by the plane-sweep skeleton (see section 2.6.2) for presenting the graphical information. At each node of the AVL-tree, the endpoints of the segments are retrieved and passed to the graphical kernel. When the entire y-table has been traversed, the current slice is superimposed as a grey shaded area.

5.2. Statistics

This program has been implemented to perform the efficiency analysis described in chapter 4. This program generates the following information:

- the frequency count of the operations on the x-queue
- the frequency count of the operations on the y-table
- the number \( n \) of points in the original configuration
- the number \( s \) of intersection points inserted into the x-queue
- the complexity \( (n + s) \log n \) of the configuration
- the running time of the initialization routine
- the total running time of the sweep process.
The transition procedure of this program is empty. This is because the instrumentation is supported directly by the geometrical workbench which contains a counter for every type of data structure operation as well as one for the intersection points (see section 2.2.3). The frequencies are obtained from the geometrical workbench by calling its procedure, Frequency, for every data structure operation. The number \( n \) is calculated by subtracting from the number of insertions into the \( x \)-queue the number of intersection points.

The execution time of the initialization routine is obtained by calling the procedure Lap-Time of the geometrical workbench in the application specific initialization procedure. The running time of the sweep also contains the time required to execute the initialization routine, because the filling of the originally given \( n \) events into the \( x \)-queue is part of the plane-sweep algorithm.

The evaluation of these execution times for different configurations is discussed in the previous chapter.

5.3. Raster-scan conversion

5.3.1. Introduction

The process of mapping the regions which compose a picture to a raster image is called raster-scan conversion (see figure 14). After sketching the problems of conventional raster-scan conversion algorithms, an area-oriented algorithm based on a plane-sweep is presented.

The computer graphics literature describes raster-scan conversion algorithms primarily for unstructured sets of straight line segments or vectors. The problem of drawing vectors in a raster-scan mode is straightforward from a practical as well as a theoretical point of view. It has been solved adequately by Bresenham's algorithm [Bresenham 1965].

When raster-scan conversion is applied to structured objects that include areas or regions, problems occur which are not fully resolved. Important concepts such as coherence are left at an intuitive level which makes it hard to express rigorous quantitative measures of the work that has been performed.

The problem of processing areas has traditionally been handled by bitmap techniques of limited generality. Most bitmap techniques for identifying areas are based on parity arguments. In the simplest version, every change of colour encountered during the left-to-right processing of the scan line is interpreted as a boundary point between two regions. For this technique to result in correct region identification, severe restrictions on the relative positions of vectors and on their digitization must be imposed. If several vectors intersect at one point, or close to the same point as measured by the resolution of the digitized figure, even bitmap techniques more sophisticated than parity techniques fail. Horizontal streaks across the screen, emanating toward the right from the intersection points, attest to the unsuitability of bitmap techniques for handling complex figures.

Since the trend in computer graphics away from line-oriented diagrams toward area-oriented pictures continues unabated, it is important that the applicability of raster-scan conversion algorithms to geometrical and topological problems involving regions in two and three dimensions be understood. To this date area oriented raster-scan conversion algorithms have not been studied from a complexity theory point of view.
Recent developments in computational geometry have shown that plane-sweep algorithms can handle efficiently all the geometrical and topological questions that area-oriented scan conversion algorithms need for two and three-dimensional problems. To produce pictures with removed hidden surfaces, it is sufficient to sweep a plane because the raster-scan conversion process can be performed on the projected image, the output of raster-scan conversion always being two-dimensional.

At transition points, a raster-scan conversion algorithm must make the same decisions as a plane-sweep algorithm. Conventional raster-scan conversion techniques, however, do not use the data structures that have made the $O((n+s) \log n)$ breakthrough possible. Whereas presorting of vectors is considered a replacement for the $x$-queue, sorting in $y$-order is usually not done, and there is no analogous to the application specific data structures at this time. Instead, the information that accumulated in the application specific data structures must be recomputed at each scan line using time consuming interval containment tests.

Conventional raster-scan conversion techniques do not take advantage of the fact that all scan lines in a slice are topologically equivalent. Thus the time complexity of these raster-scan conversion algorithms also depends on the size of the raster, and not only on the number of segments. Coherence in area oriented raster-scan conversion means exactly the same as it does...
in unstructured sets of vectors: the term applies only to slices, within which no topological decision are made.

A problem that is especially important for plotters is that of the required storage size. For example in document processing, personal computers are usually used, which do not have sufficient memory to store the raster for even a single page. Also in VLSI design, the plot size does not fit into the physical memory of many computers.

The conventional method of performing the raster-scan conversion in bands consisting of a fixed number of scan lines has the disadvantage that complex algorithms must be used to first produce a 'band conversion' and then to pass the information across the bands.

In summary, the unidirectional nature of plane-sweep which needs no backtracking and maintains the information about the topology of the configuration, makes it a natural algorithm for the raster-scan conversion. It requires a buffer of a single scan line, as opposed to conventional algorithms which require buffers of many scan lines for the bands or buffers of the size of the bitmap.

5.3.2. Model of the implemented algorithm

Most conventional raster-scan algorithms process a configuration polygon by polygon according to their z coordinates by overpainting them. In opposition, the implemented plane-sweep algorithm processes the configuration in a single left-to-right sweep. The conceptual model of the algorithm consists of the intersection of each polygon with the set of scan lines. Each polygon is assumed to lie on a different plane parallel to the x-y plane, and a set of scan lines is assumed for each polygon.

Consider a configuration consisting of a single polygon. In the first stage of the raster-scan conversion, a family of parallel straightlines is introduced (see figure 15). This family represents the scan lines [Nef 1978]. In the second stage the family of straightlines is intersected with the polygon. The result is the raster-scan converted image of the configuration.

For the algorithm based on plane-sweep, the coordinate system is selected such that the sweep direction is perpendicular to the family of straightlines. The scan lines are then represented by a family of sweep lines.

Note. In the applications of the project described in this work, the necessary coordinate transformations are performed by the input programs using the transformation capabilities of the graphical kernel (section 2.2.2).

Now consider a configuration consisting of an arbitrary finite number of polygons. In general these polygons are distinct. Their union is of no interest. This means that each polygon has an attribute which distinguishes it from the other polygons. In the implemented program this attribute is called colour.

The devices available for this project have binary rasters, that is, each pixel is represented by a binary digit. The scheme that has been adopted to distinguish the polygons is: For each colour, there are 16 types of straightlines. In the entire family the types equal modulo 16. Such a family is defined as a pattern. Because of the repetitive nature of the patterns along a straightline, each straightline is represented physically by a bit pattern stored in a single storage word. Thus a
Fig. 15. The raster-scan conversion of a polygon is obtained by intersecting it with a family of parallel straight lines representing the scan lines.

Pattern is represented physically by a $16 \times 16$ binary matrix. Where it exists, the pattern corresponding to a colour has been designed according to heraldic black and white colour representation. Where such an interpretation does not exist, an arbitrary pattern has been designed.

Note. Sixteen is the word size of the target machine. By limiting the size of the variables to the word size of the target machine, standard data types can be used instead of data structures. This simplifies the exposition of the algorithm and of the management of an application specific data structure.

The power of the plane-sweep algorithm lies in its ability to analyse configurations of intersecting polygons of arbitrary topology. An application in which this can be shown effectively is hidden surface elimination. The following assumptions are made:

- each polygon is assumed to lie in a distinct plane perpendicular to the $z$-axis;
- the $z$ coordinate is set to $z = -(\text{polygon number})$;
- the colours are arbitrarily ordered from red to white, where red is assigned to the polygon with $z = -1$, blue to $z = -2$, etc.

Note. Because of the problem of limited memory space, the plane-sweep skeleton cannot access the data of the
graphical kernel. Therefore, the data is transferred via a two-dimensional plot file. The polygon number is the ordinal number in which the polygon is found in the plot file by the plane-sweep skeleton.

The raster-scan conversion of the entire configuration is described as follows. The projection of the configuration on the x-y plane can be thought of as being looking from \( z = +\infty \). On each of the planes \( z = 0, z = -1, z = -2, \ldots \), there is a polygon and a family of parallel straight lines. In the next stage each polygon is intersected with its family of parallel straight lines. What is visible from the eyepoint at \((0, 0, +\infty)\) is the raster-scan conversion of the configuration with hidden surfaces removed (figures 16 and 17)

According to the model, the program must accomplish the following tasks for each interval on a sweep line:

- calculate the endpoints of the interval
- determine the top (i.e. visible) family of parallel straight lines
- determine the current (ax) straight line in the family of parallel straight lines
- intersect the straight line with the interval and place it into the scan line

Note. For didactical reasons, the number of polygons and z coordinates is limited to 16. If a configuration contains more than 16 polygons, the z coordinate is modulo 16. If a plane contains more than one polygon, then the non-connected polygon is considered, which results from the union of its interiors.

5.3.3. Data structures and initialization

A depth-queue is associated with each interval in the y-table. It contains the attributes or colours of all the polygons whose projection on the x-y plane intersect in the corresponding slice. Since the planes on which the polygons lie are all parallel, the depth-queue can be implemented by a bit vector. A bit in the bit vector is associated with each plane. If the projection of a polygon on the x-y plane is within a region the corresponding bit is turned on. Otherwise it is turned off. The depth-queue itself is physically hooked into the eyelet of the segment with largest y coordinate value forming its boundary.

In the case of polygons with holes, the application specific data structure also has a field which indicates whether the segment is below or above the polygon with respect to the polygon's interior.

The data structure declarations are:

```plaintext
TYPE COLSET = SET OF COLOUR;
ORDER = (above, below); (* with respect to interior of polygon *)
HOOK = POINTER TO REGION;
REGION = RECORD
    depth: CARDINAL; order: ORDER;
    colour: COLOUR; depth-Queue: COLSET
END.
```

The application specific data structure consists of a variable of type REGION that is associated
with each segment in the $y$-table.

The initialization routine hooks the application's data structure into the sentinel's eyelets (see subsection 2.6.2) and initializes the disk file into which the raster-scan converted image is written.

```plaintext
PROCEDURE Init-Sweep (next-gamma: XREC; Y: TABLE);
  VAR sentinel: SEGMENT; characteristic: HOOK;
BEGIN
  NEW (characteristic);
  WITH characteristic DO
    depth := -\infty; order := above;
    colour := ground; depth-Queue := { }
  END;
  WITH sentinel DO (* upper sentinel *)
    WITH a DO x := -\infty; y := +\infty END;
    WITH b DO x := +\infty; y := +\infty END
  END;
  Hook-In-Attributes (sentinel, Y, characteristic);
  WITH characteristic DO
    depth := +\infty; order := below;
    colour := ground; depth-Queue := { }
  END;
  WITH sentinel DO (* lower sentinel *)
    WITH a DO x := -\infty; y := -\infty END;
    WITH b DO x := +\infty; y := -\infty END
  END;
  Hook-In-Attributes (sentinel, Y, characteristic);
END Init-Sweep;
```

5.3.4. Processing of a slice

At each scan line, the $y$-table is traversed from the bottom up. For each interval, the procedure Scan-Conversion produces an incremental piece of raster image.

The procedure for processing a slice is straightforward for all cases except at the cross points of self-intersecting polygons. At these cross points, the following arbitrary artifact has been introduced to produce the same result as in raster-scan conversion with the parity method:

If a cross point occurs as a self intersection point of a polygon, then the polygon is supposed to be folded at this point. The 'lower face' or negative of the folded part that overlaps an 'upper face' or positive creates a transparent region.

This convention can easily be replaced in the procedure Scan-Conversion without changing the remainder of the algorithm. This is necessary if the polygons represent a material, as for example in integrated circuit mask specification. In this case a convention known as non-zero winding has to be used. A detailed discussion of the conventions to determine the inside and outside of polygons can be found in [Newell and Sequin 1980].

In the case of points of the type start and crossing, the routine processing a slice determines the
position of each segment with respect to the interior of the polygon. If this were not the case it would not be possible to process 'holes' (see figure 17).

The procedure Transition uses the local variables $rs, rt, rl$ of type $HOOK$ and $ord$ of type $ORDER$.

**CASE start:** (*$s$ is the upper segment*)

NEW ($rs$); NEW ($rt$);  
hook $rl$ into eyelet of successor;  
WITH $rs$  
  depth := owner; colour := Colour (owner);  
  depth-Queue := $rt$.depth-Queue;  
  (* Check whether it is an end point (hole) *)  
  IF (colour IN depth-Queue) THEN  
    order := above; EXCL (depth-Queue, colour)  
  ELSE order := below; INCL (depth-Queue, colour)  
END;  
END;  
WITH $rt$  
  depth := owner; colour := Colour (owner);  
  depth-Queue := $rl$.depth-Queue;  
  IF (colour IN depth-Queue) THEN order := below  
  ELSE order := above  
END;  
SetX (future);  
Hook-In-Attributes ($s, Y, rs$); Hook-In-Attributes ($t, Y, rt$)

**CASE crossing:** (*Attributes already permuted by the plane-sweep skeleton.*)

hook $rs$ into eyelet of upper segment prior to intersection;  
hook $rs$ into eyelet of lower segment prior to intersection;  
IF $rs$.depth $= rt$.depth THEN  
  ord := $rs$.order; $rs$.order := $rt$.order; $rt$.order := ord;  
  IF $rt$.order $= above$ THEN  
    INCL ($rs$.depth-Queue, $rs$.colour);  
    EXCL ($rt$.depth-Queue, $rt$.colour)  
  ELSE  
    INCL ($rt$.depth-Queue, $rt$.colour);  
    EXCL ($rs$.depth-Queue, $rs$.colour)  
END;  
ELSE  
  IF $rs$.order $= below$ THEN  
    IF $rt$.order $= below$ THEN  
      INCL ($rt$.depth-Queue, $rs$.colour);  
      EXCL ($rs$.depth-Queue, $rt$.colour)  
    ELSE (*$rt$.order $= above*)  
      INCL ($rs$.depth-Queue, $rt$.colour);  
      INCL ($rt$.depth-Queue, $rs$.colour)  
    END  
  ELSE (*$rs$.order $= above*)  
    IF $rt$.order $= below$ THEN  
      EXCL ($rt$.depth-Queue, $rs$.colour);
EXCL (rst.depth-Queue, rtt.colour)
ELSE (* rtt.order = above *)
INCL (rst.depth-Queue, rtt.colour);
EXCL (rtt.depth-Queue, rst.colour)
END
END
END

At bends no action is required, and at endpoints the two priority queues must be removed. At this stage the procedure Traverse-Table is called with the parameter Scan-Conversion for each scan line in the slice.

Note. As noted in section 5.3.2, the configurations are scaled such that the x coordinate is the number of the current scan line.

5.3.5. Implementation of the raster-scan conversion

According to the model, the visible polygon for each region can be inferred by the following procedure:

PROCEDURE Top-Region (eyelet: ADDRESS): COLOUR;
VAR attributes: HOOK; i: COLOUR;
BEGIN
attributes := HOOK (eyelet);
i := red;
WHILE (1 < white) DO
IF (i IN attributes.depth-Queue) THEN RETURN (1) END;
INC (i)
END;
RETURN (white)
END Top-Region.

The procedure performing the raster-scan conversion is called for each interval in the slice. Therefore the interval limits are stored in global variables:

VAR ly1, lyf, hy1, hyf: CARDINAL;

(where l stands for low, h for high, i for entire, f for fraction).

PROCEDURE Scan-Conversion (t: TABLE);
VAR mask, pattern: BITSET; w, l: CARDINAL; region: COLOUR;
BEGIN
WITH t.data DO
IF (a.y + b.y = -oo) THEN (* lower sentinel *)
hy1 := bol; hyf := 0; scan-line[bol] := {0 ... 15}
ELSE
ly1 := hy1; lyf := hyf;
IF (a.y + b.y = +oo) THEN (* upper sentinel *)
hy1 := eol; hyf := 15
ELSE
hy1 := y (t.data, ax); hyf := hy1 MOD 16;
hy1 := hy1 DIV 16 + 1
END;
region := Top-Region (tt.eyelet);
pattern := colour-pattern [region, ax MOD 16];
(= ax contains the actual x coordinate =)
FOR w := lyi + 1 TO hyi - 1 DO
  scan-line[w] := pattern
END;
mask := {9 ... 15};
IF lyl <> hyi THEN
  l := 15 ELSE
  l := hyf
END;
FOR w := lyl + 1 TO l DO
  IF NOT (w IN pattern) THEN EXCL (mask, w) END
END;
scan-line[ly1] := scan-line[ly1] AND mask;
IF lyl <> hyi THEN
  mask := {0 ... 15};
  FOR w := 0 TO hyf DO
    IF NOT (w IN pattern) THEN EXCL (mask, w) END
  END;
  scan-line[hy1] := mask
END
END
END Scan-Conversion.

After the production of a scan line, the procedure Transition augments it by a margin and writes
the scan line to disk.

Note. The last stage contains a complication which is that the size of the raster (2112 scan lines of 1712 pixels
each) is larger than the maximum file size allowed on the target machine. The strategy adopted consists of
writing the raster image into three files in PDP-11 format, transferring these files to a PDP-11, concatenating them
with the standard software and printing the raster image on this machine.

Figures 17 and 18 show two examples of raster-scan converted configurations.

5.4. Area computation

The area computation program incrementally computes the area of the polygons in a
configuration. The unit of measure is chosen so that the area corresponds to a plot generated with
the raster-scan conversion program. At each transition the area of the polygons in the current
slice is updated on the screen.

The slices are decomposed into a set of trapezia and triangles by the segments in the y-table. A
triangle is considered a trapezium whose shorter base has length zero, and whose area can be
computed incrementally according to the following simple scheme:

- initialize the variables containing the area of the polygons

- traverse the y-table at the left cross section and calculate the left base length of the
  trapezia which is the length of the intervals

- repeat the procedure at the right cross section to calculate the right base length
- for each polygon, calculate the mean of the two base lengths and multiply it by the height, i.e. the distance between the two cross sections

- increment the area accumulator of each polygon and go to the next transition

The application specific data structure is the same as for the raster-scan conversion. The transition routine is:

PROCEDURE Transition (gamma, next-gamma: XREC; Y: TABLE);
    update application specific data structure;
    IF gamma.xKey <> next-gamma.xKey THEN
        FOR i := red TO white DO cross-section[i] := 0.0 END;
        updated := { };  (* ax is a global variable into which the x-coordinate is stored *)
        ax := gamma.xKey; Traverse-Table (Y, Compute-Xsection);
        ax := next-gamma.xKey; Traverse-Table (Y, Compute-Xsection);
        Compute-Area (next-gamma.xKey - gamma.xKey);
        Update display surface
    END
END Transition.

The cross-section array is declared as

VAR cross-section: ARRAY COLOUR OF REAL.

Because the polygons need not to be connected, the procedure Compute-Xsection updates this array several times at each transition:

VAR last-y: REAL;

PROCEDURE Compute-Xsection (t: TABLE);
    VAR region: HOOK; new-y, h: REAL;
    BEGIN
        WITH t.data DO
            IF (a.y + b.y = -∞) THEN last-y := 1.0
            ELSE
                IF (a.y + b.y = +∞) THEN new-y := maximum-y
                ELSE new-y := y (t.data, ax)
            END;
            h := new-y - last-y; last-y := new-y;
            region := HOOK (t.eyelet);
            FOR i := red TO white DO
                IF i IN region.depth-Queue THEN
                    INCL (updated, i);
                    cross-section[i] := cross-section[i] + h
                END
            END
        END
    END
END Compute-Xsection.
Because of the additive property of the area function the non-connectedness needs no special handling:

```pascal
PROCEDURE Compute-Area (d: CARDINAL);
  FOR i := red TO white DO
    IF i IN updated THEN
      area[i] := area[i] + cross-section[i] * d / 2
    END
  END
END Compute-Area,
```

where `area` is of the same type as `cross-section`.

5.5. Maximal height of polygons

This application incrementally computes the maximal height of the polygons in a configuration. At each transition, the height of the polygons in the current slice is updated on the display surface.

Note. The unit of measure is chosen such that the height corresponds to that in a plot generated with the raster-scan conversion program.

Because there are no constraints on the connectedness of the polygons, an accumulator called `cross-section` is used:

```pascal
VAR max-cross-section, cross-section: ARRAY COLOUR OF REAL.
```

The transition routine processing a slice has the following simple structure:

```pascal
PROCEDURE Transition (gamma, next-gamma: XREC; Y: TABLE);
  update application specific data structure;
  IF xKey <> next-gamma.xKey THEN
    FOR i := red TO white DO cross-section[i] := 0.0 END;
    ax := gamma.xKey; Traverse-Table (Y, Compute-Xsection);
    updated := { }
    FOR i := red TO white DO
      IF cross-section[i] > max-cross-section[i] THEN
        INCL (updated, i);
        max-cross-section[i] := cross-section[i]
      END;
    END
    Update display surface
  END
END Transition,
```

where

```pascal
PROCEDURE Compute-Xsection (t: TABLE);
  VAR region: HOOK; new-y, h: REAL;
BEGIN
  WITH t+.data DO
```
IF (a.y + b.y = -\infty) \text{ THEN }\ last-y := 1.0\
ELSE
    IF (a.y + b.y = +\infty) \text{ THEN }\ new-y := \text{ maximum-y}
    ELSE \ new-y := y (t.\ data, ax)
END;
\ h := new-y - last-y;\ last-y := new-y;
\ region := HOOK (t.\ eyelet);
\ FOR \ i := \text{ red TO white DO}
    IF \ i \ IN \ region.\ depthQueue \ THEN
        \ accumulator[i] := accumulator[i] + h
    END
END
END Compute-Xsection.

5.6. Further applications

Other geometrical applications are implemented by modifying the application specific data structure and the integration procedure of the area or the height program:

- compute the perimeter of the polygons or regions
- compute the area of the regions belonging to more than one polygon
- compute the winding number of a point
- find the maximal width of the polygons
- determine the maximal width and height of the regions
- check whether minimal distance requirements are fulfilled
- perform a triangulation

The winding number of a boundary with respect to a given point is defined as the number of times a point on the boundary wraps around the given point while the boundary point makes one complete traversal of the boundary [Newell and Sequin 1980].

The maximal width of a polygon is the sum of the projections along the y-axis of its connection components.

The application performing the minimal distance check is of relevance in VLSI design rule checking.

The triangulation application is easily implemented because in a slice all regions are triangles or trapezia. Each transition point is a triangulation point for the region on whose boundary it lies. When a triangle is complete (the sum of the angles is \pi), it is output. A different approach based on plane-sweep is discussed in [Greene 1981].
The program for the computation of the area of the regions, or of the regions belonging to more than one polygon, can be obtained by making the following changes in the area computation program presented in section 5.4. The variables area and cross-section are declared as arrays whose number of elements is the maximum number of regions in a configuration. The application specific data structure should maintain a cursor into this array. A simple scheme is to hash this cursor from the variable depth-Queue.

The changes which apply to area computation also apply to the application which determines the maximum height of the regions.

Other applications, such as hidden line elimination, construction of a minimal spanning tree (MST) or of a partially ordered set (poset) describing the regions, require different application specific data structures.
Fig. 16. Raster-scan conversion of figure 8. 
Number of originally given points: \( n = 576 \), intersection points found during the sweep: \( s = 1776 \), running time (bucket version) \( T(576, 1776) = 58.28 \) seconds.
Fig. 17. Lower left: raster-scan conversion of the drawing (B-splines) used in experiment 3. Number of originally given points: \( n = 1024 \), intersection points found during the sweep: \( s = 192 \), running time (bucket version) \( T(1024, 192) = 21.92 \) seconds.

Upper right: raster-scan conversion of the drawing used in experiment 4. It shows how self-intersecting polygons are processed. Number of originally given points: \( n = 62 \), intersection points found during the sweep: \( s = 111 \), running time (bucket version) \( T(62, 111) = 3.38 \) seconds.
6. Conclusions

An important class of algorithms in computational geometry - the sweep algorithms - trade a space dimension for a time dimension in order to apply efficient logarithmic search techniques. Recent developments in computational geometry have shown that sweep algorithms can be used to solve a large class of geometrical and topological problems. Possible applications include raster-scan conversion in computer graphics, and layout of integrated circuits in computer-aided design.

This paper extracts an application independent skeleton from the original plane-sweep algorithm for polygons and implements it on the Lilith personal computer. The skeleton advances the sweep line and maintains the general data structures. The application programs supply the code to process a single region. They usually require an additional specific data structure for this purpose which may be attached to one of the general data structures of the skeleton.

It is found that the finite arithmetic of computers causes access problems. Assuming integer coordinates for the vertices of the polygons, the worst case of the numerical problems is identified and a criterion for the number of bits required to store the fraction part of floating-point numbers is established.

The implementation demonstrates the practical value of one class of plane-sweep algorithms, by Nievergelt and Preparata. Its elegance and simplicity is evidenced in its low memory requirements and short running time, which make it attractive in an interactive environment. It has also been demonstrated that application programs are easily implemented.

The implementation of a raster-scan conversion program is valuable both from a theoretical and a practical point of view. The fact that all cross sections in a slice are topologically equivalent explains the notion of 'coherence' which previously was known only at an intuitive level. Of practical importance is the reliable, but simple and fast raster-scan program.

The implementation of other applications is suggested for further study, in particular a hidden-surface elimination algorithm.
Appendix A. Results of the experiments with the heap data structure

The results of the experiments for the performance analysis are presented here without further comments. A discussion can be found in subsection 4.2.3.

The columns contain:

- no.: experiment number
- n: number of originally given points
- s: number of intersection points inserted into the x-queue
- time: running time in seconds
- \((n+s)\log n\): time complexity of the configuration
- s/n: ratio of intersection points
- \(t/[(n+s)\log n]\): estimator of the constant factor of the growth rate

Drawings produced with an interactive editor:

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>((n+s)\log n)</th>
<th>s/n</th>
<th>(t/[(n+s)\log n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95</td>
<td>111</td>
<td>4.42</td>
<td>1353.398</td>
<td>1.168</td>
<td>0.00327</td>
</tr>
<tr>
<td>3</td>
<td>1024</td>
<td>192</td>
<td>23.84</td>
<td>12160.801</td>
<td>0.187</td>
<td>0.00198</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>111</td>
<td>3.48</td>
<td>1030.076</td>
<td>1.790</td>
<td>0.00338</td>
</tr>
</tbody>
</table>

Mean value of \(t/[(n+s)\log n]\): 0.00287

Configurations consisting of squares (class 1). The experiments have been performed varying the parameters \(f\) (number of layers) and \(m\) (number of columns of \(m\) squares each). The experiments have been repeated with different start values for the pseudo-random number generator.

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>((n+s)\log n)</th>
<th>s/n</th>
<th>(t/[(n+s)\log n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>200</td>
<td>282</td>
<td>10.30</td>
<td>3684.339</td>
<td>1.410</td>
<td>0.00280</td>
</tr>
<tr>
<td>6</td>
<td>288</td>
<td>392</td>
<td>15.44</td>
<td>5555.549</td>
<td>1.361</td>
<td>0.00278</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
<td>684</td>
<td>23.50</td>
<td>8097.158</td>
<td>2.280</td>
<td>0.00290</td>
</tr>
<tr>
<td>8</td>
<td>400</td>
<td>1256</td>
<td>42.22</td>
<td>14314.226</td>
<td>3.140</td>
<td>0.00295</td>
</tr>
<tr>
<td>12</td>
<td>200</td>
<td>266</td>
<td>9.78</td>
<td>3562.037</td>
<td>1.330</td>
<td>0.00275</td>
</tr>
<tr>
<td>13</td>
<td>288</td>
<td>420</td>
<td>15.88</td>
<td>5784.387</td>
<td>1.468</td>
<td>0.00275</td>
</tr>
<tr>
<td>14</td>
<td>392</td>
<td>592</td>
<td>24.00</td>
<td>8476.874</td>
<td>1.510</td>
<td>0.00283</td>
</tr>
<tr>
<td>15</td>
<td>300</td>
<td>636</td>
<td>22.48</td>
<td>7702.174</td>
<td>2.120</td>
<td>0.00292</td>
</tr>
<tr>
<td>16</td>
<td>400</td>
<td>1160</td>
<td>39.86</td>
<td>13464.410</td>
<td>2.900</td>
<td>0.00296</td>
</tr>
</tbody>
</table>

Mean value of \(t/[(n+s)\log n]\): 0.00285

Configurations consisting of rectangle pairs (class 2). The experiments have been obtained by varying the parameters \(f\) (number of layers) and \(m\) (number of rectangle pairs). The same start value for the pseudo-random number generator has been used for all experiments.

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>((n+s)\log n)</th>
<th>s/n</th>
<th>(t/[(n+s)\log n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>144</td>
<td>1084</td>
<td>29.72</td>
<td>8804.668</td>
<td>7.528</td>
<td>0.00338</td>
</tr>
</tbody>
</table>
mean value of $t/[(n+s) \log n]$: 0.00335

Configurations consisting of random polygons (class 3). The experiments have been obtained by varying the size and number of polygons as well as the start value for the pseudo-random number generator.

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>$(n+s) \log n$</th>
<th>s/n</th>
<th>$t/[(n+s) \log n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>178</td>
<td>48.92</td>
<td>14575.729</td>
<td>10.102</td>
<td>0.00336</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>96</td>
<td>11.72</td>
<td>3642.710</td>
<td>4.604</td>
<td>0.00331</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>$(n+s) \log n$</th>
<th>s/n</th>
<th>$t/[(n+s) \log n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>96</td>
<td>153</td>
<td>5.00</td>
<td>1639.656</td>
<td>1.594</td>
<td>0.00305</td>
</tr>
<tr>
<td>35</td>
<td>148</td>
<td>284</td>
<td>8.98</td>
<td>3114.484</td>
<td>1.919</td>
<td>0.00268</td>
</tr>
<tr>
<td>36</td>
<td>204</td>
<td>552</td>
<td>17.20</td>
<td>6800.354</td>
<td>2.706</td>
<td>0.00297</td>
</tr>
<tr>
<td>37</td>
<td>258</td>
<td>813</td>
<td>25.36</td>
<td>8580.824</td>
<td>3.161</td>
<td>0.00296</td>
</tr>
<tr>
<td>38</td>
<td>309</td>
<td>1092</td>
<td>35.22</td>
<td>11454.516</td>
<td>3.640</td>
<td>0.00307</td>
</tr>
<tr>
<td>39</td>
<td>78</td>
<td>46</td>
<td>2.46</td>
<td>779.390</td>
<td>0.590</td>
<td>0.00313</td>
</tr>
<tr>
<td>40</td>
<td>119</td>
<td>135</td>
<td>5.60</td>
<td>1751.284</td>
<td>1.134</td>
<td>0.00320</td>
</tr>
<tr>
<td>41</td>
<td>256</td>
<td>810</td>
<td>26.20</td>
<td>8528.000</td>
<td>3.164</td>
<td>0.00307</td>
</tr>
<tr>
<td>42</td>
<td>63</td>
<td>37</td>
<td>2.28</td>
<td>765.005</td>
<td>0.446</td>
<td>0.00298</td>
</tr>
<tr>
<td>43</td>
<td>126</td>
<td>70</td>
<td>3.64</td>
<td>1387.547</td>
<td>0.556</td>
<td>0.00266</td>
</tr>
<tr>
<td>44</td>
<td>169</td>
<td>147</td>
<td>6.14</td>
<td>2338.878</td>
<td>0.870</td>
<td>0.00263</td>
</tr>
<tr>
<td>45</td>
<td>206</td>
<td>191</td>
<td>8.12</td>
<td>3051.541</td>
<td>0.927</td>
<td>0.00268</td>
</tr>
<tr>
<td>46</td>
<td>246</td>
<td>254</td>
<td>10.84</td>
<td>3971.257</td>
<td>1.033</td>
<td>0.00273</td>
</tr>
<tr>
<td>47</td>
<td>283</td>
<td>372</td>
<td>14.76</td>
<td>5334.751</td>
<td>1.314</td>
<td>0.00277</td>
</tr>
<tr>
<td>48</td>
<td>326</td>
<td>508</td>
<td>18.98</td>
<td>6962.839</td>
<td>1.558</td>
<td>0.00273</td>
</tr>
<tr>
<td>49</td>
<td>366</td>
<td>686</td>
<td>24.88</td>
<td>8958.516</td>
<td>1.874</td>
<td>0.00278</td>
</tr>
<tr>
<td>50</td>
<td>484</td>
<td>1197</td>
<td>43.46</td>
<td>14992.609</td>
<td>2.473</td>
<td>0.00290</td>
</tr>
<tr>
<td>51</td>
<td>403</td>
<td>844</td>
<td>30.64</td>
<td>10792.331</td>
<td>2.094</td>
<td>0.00284</td>
</tr>
<tr>
<td>52</td>
<td>320</td>
<td>560</td>
<td>18.76</td>
<td>6823.981</td>
<td>1.562</td>
<td>0.00275</td>
</tr>
<tr>
<td>53</td>
<td>244</td>
<td>275</td>
<td>11.55</td>
<td>4116.053</td>
<td>1.127</td>
<td>0.00279</td>
</tr>
<tr>
<td>54</td>
<td>165</td>
<td>147</td>
<td>6.22</td>
<td>2298.293</td>
<td>0.891</td>
<td>0.00271</td>
</tr>
<tr>
<td>55</td>
<td>691</td>
<td>533</td>
<td>29.64</td>
<td>11545.431</td>
<td>0.771</td>
<td>0.00257</td>
</tr>
<tr>
<td>56</td>
<td>662</td>
<td>433</td>
<td>24.50</td>
<td>10260.903</td>
<td>0.654</td>
<td>0.00239</td>
</tr>
<tr>
<td>57</td>
<td>590</td>
<td>380</td>
<td>22.42</td>
<td>8926.434</td>
<td>0.644</td>
<td>0.00251</td>
</tr>
<tr>
<td>58</td>
<td>656</td>
<td>478</td>
<td>26.92</td>
<td>10611.464</td>
<td>0.729</td>
<td>0.00254</td>
</tr>
<tr>
<td>59</td>
<td>556</td>
<td>323</td>
<td>20.12</td>
<td>8015.549</td>
<td>0.561</td>
<td>0.00251</td>
</tr>
<tr>
<td>60</td>
<td>488</td>
<td>253</td>
<td>16.32</td>
<td>6617.676</td>
<td>0.518</td>
<td>0.00247</td>
</tr>
<tr>
<td>61</td>
<td>421</td>
<td>183</td>
<td>12.76</td>
<td>5265.477</td>
<td>0.435</td>
<td>0.00242</td>
</tr>
<tr>
<td>62</td>
<td>349</td>
<td>117</td>
<td>9.44</td>
<td>3936.341</td>
<td>0.335</td>
<td>0.00240</td>
</tr>
<tr>
<td>63</td>
<td>274</td>
<td>78</td>
<td>6.84</td>
<td>2650.507</td>
<td>0.285</td>
<td>0.00240</td>
</tr>
<tr>
<td>64</td>
<td>207</td>
<td>36</td>
<td>4.52</td>
<td>1869.517</td>
<td>0.174</td>
<td>0.00242</td>
</tr>
<tr>
<td>65</td>
<td>137</td>
<td>18</td>
<td>2.70</td>
<td>1100.195</td>
<td>0.131</td>
<td>0.00245</td>
</tr>
<tr>
<td>66</td>
<td>66</td>
<td>3</td>
<td>1.14</td>
<td>417.063</td>
<td>0.045</td>
<td>0.00273</td>
</tr>
<tr>
<td>67</td>
<td>601</td>
<td>295</td>
<td>20.34</td>
<td>8271.174</td>
<td>0.491</td>
<td>0.00246</td>
</tr>
<tr>
<td>68</td>
<td>571</td>
<td>273</td>
<td>19.80</td>
<td>7728.801</td>
<td>0.478</td>
<td>0.00256</td>
</tr>
<tr>
<td>69</td>
<td>541</td>
<td>261</td>
<td>17.52</td>
<td>7199.952</td>
<td>0.464</td>
<td>0.00244</td>
</tr>
<tr>
<td>70</td>
<td>511</td>
<td>215</td>
<td>19.12</td>
<td>6531.952</td>
<td>0.421</td>
<td>0.00293</td>
</tr>
<tr>
<td>71</td>
<td>481</td>
<td>199</td>
<td>14.80</td>
<td>6058.727</td>
<td>0.414</td>
<td>0.00244</td>
</tr>
<tr>
<td>72</td>
<td>451</td>
<td>171</td>
<td>13.38</td>
<td>6484.164</td>
<td>0.379</td>
<td>0.00244</td>
</tr>
<tr>
<td>73</td>
<td>421</td>
<td>145</td>
<td>11.84</td>
<td>4934.265</td>
<td>0.344</td>
<td>0.00240</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td>361</td>
<td>301</td>
<td>240</td>
<td>180</td>
<td>120</td>
<td>631</td>
</tr>
<tr>
<td></td>
<td>117</td>
<td>87</td>
<td>66</td>
<td>32</td>
<td>12</td>
<td>331</td>
</tr>
<tr>
<td></td>
<td>9.86</td>
<td>7.70</td>
<td>5.74</td>
<td>3.86</td>
<td>2.38</td>
<td>22.14</td>
</tr>
<tr>
<td></td>
<td>4061.019</td>
<td>3194.644</td>
<td>2419.509</td>
<td>1588.273</td>
<td>911.710</td>
<td>8948.039</td>
</tr>
<tr>
<td></td>
<td>0.324</td>
<td>0.289</td>
<td>0.275</td>
<td>0.178</td>
<td>0.100</td>
<td>0.525</td>
</tr>
<tr>
<td></td>
<td>0.00243</td>
<td>0.00241</td>
<td>0.00237</td>
<td>0.00243</td>
<td>0.00237</td>
<td>0.00247</td>
</tr>
</tbody>
</table>

Mean value of $t/(n + s \log n)$: 0.00265

Mean value of $t/(n + s \log n)$ (all experiments): 0.00272

Standard deviation: 0.2958D-03
Appendix B. Results of the experiments with the bucket data structure

The results of the experiments for the performance analysis are presented here without further comments. A discussion can be found in subsection 4.2.3.

The columns contain:

<table>
<thead>
<tr>
<th>no.</th>
<th>experiment number</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number of originally given points</td>
</tr>
<tr>
<td>s</td>
<td>number of intersection points inserted into the x-queue</td>
</tr>
<tr>
<td>time</td>
<td>running time in seconds</td>
</tr>
<tr>
<td>(n+s) log n</td>
<td>time complexity of the configuration</td>
</tr>
<tr>
<td>s/n</td>
<td>ratio of intersection points</td>
</tr>
<tr>
<td>t/[(n+s) log n]</td>
<td>estimator of the constant factor of the growth rate</td>
</tr>
</tbody>
</table>

Drawings produced with an interactive editor:

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>(n+s) log n</th>
<th>s/n</th>
<th>t/[(n+s) log n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95</td>
<td>111</td>
<td>4.20</td>
<td>1353.390</td>
<td>1.168</td>
<td>0.00310</td>
</tr>
<tr>
<td>2</td>
<td>106</td>
<td>30</td>
<td>2.46</td>
<td>914.997</td>
<td>0.283</td>
<td>0.00269</td>
</tr>
<tr>
<td>3</td>
<td>1024</td>
<td>192</td>
<td>21.92</td>
<td>12160.000</td>
<td>0.187</td>
<td>0.00180</td>
</tr>
<tr>
<td>4</td>
<td>62</td>
<td>111</td>
<td>3.38</td>
<td>1030.076</td>
<td>1.790</td>
<td>0.00328</td>
</tr>
</tbody>
</table>

mean value of t/[(n+s) log n]: 0.00272

Configurations consisting of squares (class 1). The experiments have been performed varying the parameters \( I \) (number of layers) and \( m \) (number of columns of \( m \) squares each). The experiments have been repeated with different start values for the pseudo-random number generator.

<table>
<thead>
<tr>
<th>no.</th>
<th>n</th>
<th>s</th>
<th>time</th>
<th>(n+s) log n</th>
<th>s/n</th>
<th>t/[(n+s) log n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>200</td>
<td>282</td>
<td>9.92</td>
<td>3684.339</td>
<td>1.410</td>
<td>0.00289</td>
</tr>
<tr>
<td>6</td>
<td>288</td>
<td>392</td>
<td>14.64</td>
<td>5555.549</td>
<td>1.361</td>
<td>0.00264</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
<td>684</td>
<td>22.16</td>
<td>8997.158</td>
<td>2.289</td>
<td>0.00274</td>
</tr>
<tr>
<td>8</td>
<td>400</td>
<td>1256</td>
<td>39.30</td>
<td>14314.226</td>
<td>3.140</td>
<td>0.00275</td>
</tr>
<tr>
<td>9</td>
<td>600</td>
<td>2842</td>
<td>89.94</td>
<td>31765.594</td>
<td>4.737</td>
<td>0.00280</td>
</tr>
<tr>
<td>10</td>
<td>700</td>
<td>3966</td>
<td>121.64</td>
<td>43154.230</td>
<td>6.523</td>
<td>0.00282</td>
</tr>
<tr>
<td>11</td>
<td>800</td>
<td>5060</td>
<td>160.14</td>
<td>50512.997</td>
<td>6.326</td>
<td>0.00283</td>
</tr>
<tr>
<td>12</td>
<td>200</td>
<td>266</td>
<td>6.58</td>
<td>3562.037</td>
<td>1.330</td>
<td>0.00269</td>
</tr>
<tr>
<td>13</td>
<td>288</td>
<td>420</td>
<td>15.18</td>
<td>5784.307</td>
<td>1.458</td>
<td>0.00282</td>
</tr>
<tr>
<td>14</td>
<td>392</td>
<td>592</td>
<td>22.34</td>
<td>8476.874</td>
<td>1.510</td>
<td>0.00264</td>
</tr>
<tr>
<td>15</td>
<td>300</td>
<td>636</td>
<td>21.30</td>
<td>7702.174</td>
<td>2.212</td>
<td>0.00267</td>
</tr>
<tr>
<td>16</td>
<td>400</td>
<td>1160</td>
<td>37.20</td>
<td>13484.416</td>
<td>2.900</td>
<td>0.00278</td>
</tr>
<tr>
<td>17</td>
<td>576</td>
<td>1778</td>
<td>58.26</td>
<td>21566.003</td>
<td>3.067</td>
<td>0.00279</td>
</tr>
<tr>
<td>18</td>
<td>500</td>
<td>1980</td>
<td>59.54</td>
<td>21517.082</td>
<td>3.800</td>
<td>0.00278</td>
</tr>
<tr>
<td>19</td>
<td>720</td>
<td>2829</td>
<td>89.68</td>
<td>33801.160</td>
<td>3.917</td>
<td>0.00267</td>
</tr>
<tr>
<td>20</td>
<td>600</td>
<td>2764</td>
<td>87.74</td>
<td>31045.746</td>
<td>4.607</td>
<td>0.00283</td>
</tr>
<tr>
<td>21</td>
<td>700</td>
<td>3778</td>
<td>118.94</td>
<td>42322.523</td>
<td>5.397</td>
<td>0.00281</td>
</tr>
<tr>
<td>22</td>
<td>720</td>
<td>2918</td>
<td>93.26</td>
<td>34531.362</td>
<td>4.053</td>
<td>0.00270</td>
</tr>
</tbody>
</table>
mean value of $t/[(n+s) \log n]$: 0.00273

Configurations consisting of rectangle pairs (class 2). The experiments have been obtained by varying the parameters $I$ (number of layers) and $m$ (number of rectangle pairs). The experiments have been repeated with different start values for the pseudo-random number generator.

<table>
<thead>
<tr>
<th>no.</th>
<th>$n$</th>
<th>$s$</th>
<th>time</th>
<th>$(n+s) \log n$</th>
<th>$s/n$</th>
<th>$t/[(n+s) \log n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>144</td>
<td>108</td>
<td>28.22</td>
<td>8804.666</td>
<td>7.528</td>
<td>0.00321</td>
</tr>
<tr>
<td>24</td>
<td>176</td>
<td>177</td>
<td>46.40</td>
<td>14575.729</td>
<td>10.102</td>
<td>0.00318</td>
</tr>
<tr>
<td>25</td>
<td>208</td>
<td>241</td>
<td>64.80</td>
<td>20190.553</td>
<td>11.606</td>
<td>0.00321</td>
</tr>
<tr>
<td>26</td>
<td>96</td>
<td>442</td>
<td>11.32</td>
<td>3542.710</td>
<td>4.004</td>
<td>0.00320</td>
</tr>
<tr>
<td>27</td>
<td>224</td>
<td>288</td>
<td>78.86</td>
<td>24249.644</td>
<td>12.866</td>
<td>0.00317</td>
</tr>
<tr>
<td>28</td>
<td>320</td>
<td>568</td>
<td>166.34</td>
<td>51662.530</td>
<td>18.400</td>
<td>0.00322</td>
</tr>
<tr>
<td>29</td>
<td>352</td>
<td>7156</td>
<td>264.96</td>
<td>63513.413</td>
<td>20.330</td>
<td>0.00323</td>
</tr>
<tr>
<td>30</td>
<td>320</td>
<td>5730</td>
<td>163.52</td>
<td>50347.665</td>
<td>17.906</td>
<td>0.00325</td>
</tr>
<tr>
<td>31</td>
<td>352</td>
<td>6956</td>
<td>200.30</td>
<td>61821.526</td>
<td>19.761</td>
<td>0.00324</td>
</tr>
<tr>
<td>32</td>
<td>320</td>
<td>5830</td>
<td>166.08</td>
<td>51179.858</td>
<td>18.219</td>
<td>0.00325</td>
</tr>
<tr>
<td>33</td>
<td>352</td>
<td>7002</td>
<td>203.60</td>
<td>62210.660</td>
<td>19.892</td>
<td>0.00327</td>
</tr>
</tbody>
</table>

mean value of $t/[(n+s) \log n]$: 0.00322

Configurations consisting of random polygons (class 3). The experiments have been obtained by varying the size and number of polygons as well as the start value for the pseudo-random number generator.

<table>
<thead>
<tr>
<th>no.</th>
<th>$n$</th>
<th>$s$</th>
<th>time</th>
<th>$(n+s) \log n$</th>
<th>$s/n$</th>
<th>$t/[(n+s) \log n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>96</td>
<td>153</td>
<td>5.54</td>
<td>1639.666</td>
<td>1.594</td>
<td>0.00301</td>
</tr>
<tr>
<td>35</td>
<td>148</td>
<td>244</td>
<td>6.54</td>
<td>3141.484</td>
<td>1.919</td>
<td>0.00283</td>
</tr>
<tr>
<td>36</td>
<td>204</td>
<td>552</td>
<td>16.42</td>
<td>5800.354</td>
<td>2.706</td>
<td>0.00283</td>
</tr>
<tr>
<td>37</td>
<td>258</td>
<td>813</td>
<td>24.20</td>
<td>8580.024</td>
<td>3.151</td>
<td>0.00282</td>
</tr>
<tr>
<td>38</td>
<td>300</td>
<td>192</td>
<td>32.86</td>
<td>11454.516</td>
<td>3.640</td>
<td>0.00287</td>
</tr>
<tr>
<td>39</td>
<td>354</td>
<td>1534</td>
<td>45.56</td>
<td>15986.839</td>
<td>4.333</td>
<td>0.00285</td>
</tr>
<tr>
<td>40</td>
<td>413</td>
<td>2153</td>
<td>64.20</td>
<td>22298.535</td>
<td>5.213</td>
<td>0.00288</td>
</tr>
<tr>
<td>41</td>
<td>78</td>
<td>46</td>
<td>2.48</td>
<td>779.390</td>
<td>0.590</td>
<td>0.00318</td>
</tr>
<tr>
<td>42</td>
<td>119</td>
<td>135</td>
<td>5.22</td>
<td>1751.284</td>
<td>1.134</td>
<td>0.00298</td>
</tr>
<tr>
<td>43</td>
<td>256</td>
<td>810</td>
<td>24.60</td>
<td>8528.000</td>
<td>3.164</td>
<td>0.00288</td>
</tr>
<tr>
<td>44</td>
<td>83</td>
<td>37</td>
<td>2.16</td>
<td>765.005</td>
<td>0.546</td>
<td>0.00282</td>
</tr>
<tr>
<td>45</td>
<td>126</td>
<td>70</td>
<td>3.64</td>
<td>1367.547</td>
<td>0.556</td>
<td>0.00286</td>
</tr>
<tr>
<td>46</td>
<td>159</td>
<td>147</td>
<td>6.10</td>
<td>2338.678</td>
<td>0.878</td>
<td>0.00281</td>
</tr>
<tr>
<td>47</td>
<td>266</td>
<td>191</td>
<td>7.90</td>
<td>3051.541</td>
<td>0.927</td>
<td>0.00259</td>
</tr>
<tr>
<td>48</td>
<td>246</td>
<td>254</td>
<td>10.48</td>
<td>3971.257</td>
<td>1.033</td>
<td>0.00264</td>
</tr>
<tr>
<td>49</td>
<td>283</td>
<td>372</td>
<td>14.40</td>
<td>5334.751</td>
<td>1.314</td>
<td>0.00279</td>
</tr>
<tr>
<td>50</td>
<td>326</td>
<td>508</td>
<td>18.40</td>
<td>6962.839</td>
<td>1.558</td>
<td>0.00264</td>
</tr>
<tr>
<td>51</td>
<td>366</td>
<td>686</td>
<td>23.22</td>
<td>8958.516</td>
<td>1.874</td>
<td>0.00259</td>
</tr>
<tr>
<td>52</td>
<td>798</td>
<td>3264</td>
<td>105.46</td>
<td>39158.675</td>
<td>4.098</td>
<td>0.00269</td>
</tr>
<tr>
<td>53</td>
<td>725</td>
<td>2711</td>
<td>86.94</td>
<td>32648.313</td>
<td>3.739</td>
<td>0.00266</td>
</tr>
<tr>
<td>54</td>
<td>646</td>
<td>2154</td>
<td>69.64</td>
<td>26139.093</td>
<td>3.334</td>
<td>0.00266</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>568</td>
<td>1598</td>
<td>62.68</td>
<td>19770.751</td>
<td>2.820</td>
<td>0.00266</td>
</tr>
<tr>
<td>56</td>
<td>484</td>
<td>1197</td>
<td>39.82</td>
<td>14992.609</td>
<td>2.473</td>
<td>0.00266</td>
</tr>
<tr>
<td>57</td>
<td>403</td>
<td>844</td>
<td>26.76</td>
<td>10792.331</td>
<td>2.094</td>
<td>0.00266</td>
</tr>
<tr>
<td>58</td>
<td>320</td>
<td>600</td>
<td>17.72</td>
<td>6623.981</td>
<td>1.562</td>
<td>0.00266</td>
</tr>
<tr>
<td>59</td>
<td>244</td>
<td>275</td>
<td>10.74</td>
<td>4116.053</td>
<td>1.127</td>
<td>0.00261</td>
</tr>
<tr>
<td>60</td>
<td>165</td>
<td>147</td>
<td>6.18</td>
<td>2298.293</td>
<td>0.891</td>
<td>0.00269</td>
</tr>
<tr>
<td>61</td>
<td>691</td>
<td>533</td>
<td>27.88</td>
<td>11545.431</td>
<td>0.771</td>
<td>0.00241</td>
</tr>
<tr>
<td>62</td>
<td>662</td>
<td>433</td>
<td>23.64</td>
<td>10260.903</td>
<td>0.664</td>
<td>0.00238</td>
</tr>
<tr>
<td>63</td>
<td>590</td>
<td>360</td>
<td>21.38</td>
<td>8924.434</td>
<td>0.644</td>
<td>0.00239</td>
</tr>
<tr>
<td>64</td>
<td>566</td>
<td>478</td>
<td>25.48</td>
<td>10611.464</td>
<td>0.729</td>
<td>0.00240</td>
</tr>
<tr>
<td>65</td>
<td>558</td>
<td>323</td>
<td>19.24</td>
<td>8015.649</td>
<td>0.581</td>
<td>0.00240</td>
</tr>
<tr>
<td>66</td>
<td>488</td>
<td>253</td>
<td>15.72</td>
<td>6617.678</td>
<td>0.518</td>
<td>0.00238</td>
</tr>
<tr>
<td>67</td>
<td>421</td>
<td>183</td>
<td>12.42</td>
<td>5265.477</td>
<td>0.435</td>
<td>0.00238</td>
</tr>
<tr>
<td>68</td>
<td>349</td>
<td>117</td>
<td>9.38</td>
<td>3936.341</td>
<td>0.335</td>
<td>0.00238</td>
</tr>
<tr>
<td>69</td>
<td>274</td>
<td>78</td>
<td>6.78</td>
<td>2859.517</td>
<td>0.265</td>
<td>0.00238</td>
</tr>
<tr>
<td>70</td>
<td>207</td>
<td>36</td>
<td>4.64</td>
<td>1869.517</td>
<td>0.174</td>
<td>0.00248</td>
</tr>
<tr>
<td>71</td>
<td>137</td>
<td>18</td>
<td>2.76</td>
<td>1100.195</td>
<td>0.131</td>
<td>0.00251</td>
</tr>
<tr>
<td>72</td>
<td>66</td>
<td>3</td>
<td>1.26</td>
<td>417.063</td>
<td>0.045</td>
<td>0.00262</td>
</tr>
<tr>
<td>73</td>
<td>601</td>
<td>295</td>
<td>19.60</td>
<td>8271.174</td>
<td>0.491</td>
<td>0.00237</td>
</tr>
<tr>
<td>74</td>
<td>571</td>
<td>273</td>
<td>18.40</td>
<td>7728.801</td>
<td>0.478</td>
<td>0.00238</td>
</tr>
<tr>
<td>75</td>
<td>541</td>
<td>251</td>
<td>17.00</td>
<td>7199.052</td>
<td>0.464</td>
<td>0.00238</td>
</tr>
<tr>
<td>76</td>
<td>511</td>
<td>215</td>
<td>15.62</td>
<td>6531.952</td>
<td>0.421</td>
<td>0.00239</td>
</tr>
<tr>
<td>77</td>
<td>481</td>
<td>199</td>
<td>14.46</td>
<td>6056.727</td>
<td>0.414</td>
<td>0.00239</td>
</tr>
<tr>
<td>78</td>
<td>451</td>
<td>171</td>
<td>13.12</td>
<td>5484.164</td>
<td>0.379</td>
<td>0.00239</td>
</tr>
<tr>
<td>79</td>
<td>421</td>
<td>145</td>
<td>11.72</td>
<td>4934.265</td>
<td>0.344</td>
<td>0.00238</td>
</tr>
<tr>
<td>80</td>
<td>361</td>
<td>117</td>
<td>9.72</td>
<td>4061.919</td>
<td>0.324</td>
<td>0.00239</td>
</tr>
<tr>
<td>81</td>
<td>301</td>
<td>87</td>
<td>7.68</td>
<td>3194.644</td>
<td>0.269</td>
<td>0.00240</td>
</tr>
<tr>
<td>82</td>
<td>240</td>
<td>66</td>
<td>5.76</td>
<td>2419.509</td>
<td>0.275</td>
<td>0.00238</td>
</tr>
<tr>
<td>83</td>
<td>180</td>
<td>32</td>
<td>3.88</td>
<td>1588.273</td>
<td>0.178</td>
<td>0.00244</td>
</tr>
<tr>
<td>84</td>
<td>120</td>
<td>12</td>
<td>2.44</td>
<td>911.710</td>
<td>0.100</td>
<td>0.00268</td>
</tr>
<tr>
<td>85</td>
<td>631</td>
<td>331</td>
<td>21.48</td>
<td>8948.039</td>
<td>0.525</td>
<td>0.00239</td>
</tr>
<tr>
<td>86</td>
<td>661</td>
<td>375</td>
<td>23.12</td>
<td>9705.773</td>
<td>0.567</td>
<td>0.00238</td>
</tr>
<tr>
<td>87</td>
<td>691</td>
<td>405</td>
<td>24.84</td>
<td>10338.066</td>
<td>0.566</td>
<td>0.00240</td>
</tr>
<tr>
<td>88</td>
<td>721</td>
<td>431</td>
<td>26.06</td>
<td>10938.921</td>
<td>0.598</td>
<td>0.00238</td>
</tr>
</tbody>
</table>

mean value of $t/[(n + s) \log n]$: 0.00259

mean value of $t/[(n + s) \log n]$ (all experiments): 0.00270

standard deviation: 0.2944D-03
Literature

[Adelson-Velskii and Landis 1962]
G. M. Adelson-Velskii and Y. M. Landis:
An Algorithm for the Organization of Information.

[Aho et al. 1974]
A. V. Aho, J. E. Hopcroft and J. D. Ullman:
The Design and Analysis of Computer Algorithms.

[Aho et al. 1983]
A. V. Aho, J. E. Hopcroft and J. D. Ullman:
Data Structures and Algorithms.

[Bentley and Ottmann 1979]
J. L. Bentley, T. A. Ottmann:
Algorithms for Reporting and Counting Geometric Intersections.

[Bieri and Nef 1982]
H. Bieri, W. Nef:
A Recursive Sweep-Plane Algorithm, Determining All Cells of a Finite Division of $\mathbb{R}^d$.

[Bieri and Nef 1983]
H. Bieri, W. Nef:
A Sweep-Plane Algorithm for Computing the Volume of Polyhedra Represented in Boolean Form.

[Bieri and Nef 1984]
H. Bieri, W. Nef:
A Sweep-Plane Algorithm for Computing the Euler-Characteristic of Polyhedra Represented in Boolean Form.
To be published.

[Bresenham 1965]
J. E. Bresenham:
Algorithm for Computer Control of a Digital Plotter.

[Brown 1981]
K. Q. Brown:
Comments on "Algorithms for Reporting and Counting Geometric Intersections".

[Fink 1984]
P. Fink:
Personal communication.

[Greene 1981]
D. H. Greene:
Triangulating a Self-Intersecting Polygon.
Xerox PARC, 81C5LH-0063.press, 1981.

[Gremminger 1982]
K. Gremminger:
Implementierung und Analyse des Hidden-Line bzw. Visible-Surface-Algorithmus
auf der Basis der Plane-Sweep-Technik.

[Henrici 1980]
P. Henrici:
A Model for the Propagation of Rounding Error in Floating Point Arithmetic.
Seminar fuer angewandte Mathematik, Federal Institute of Technology Zurich,

[Knudsen 1983]
S. E. Knudsen:
ETH PhD Dissertation 7346, 1983.

[MacLaren and Marsaglia 1965]
M. D. MacLaren, G. Marsaglia:
Uniform Random Generators.
JACM, 12, 83-89, 1966.

[Nef 1978]
W. Nef:
Beitâge zur Theorie der Polyeder.
Herbert Lang, Bern, 1978.

[Newell and Sequin 1980]
M. E. Newell, C. H. Sequin:
The Inside Story on Self-Intersecting Polygons.

[Nievergelt and Preparata 1982]
J. Nievergelt, F. P. Preparata:

[Rutishauser 1976]
H. Rutishauser:
Vorlesungen über numerische Mathematik, Bd. 2.

[Shamos 1978]
M. I. Shamos:
Computational Geometry.
Yale University PhD Dissertation,
University Microfilms International 7819047, 1978.

[Shamos and Hoey 1976]
M. I. Shamos, D. Hoey:
Geometric Intersection Problems.

[Schmitt 1981]
A. Schmitt:
Time and Space Bounds for Hidden Line and Hidden Surface Algorithms.
Proc. Eurographics '81, 43-56.

[Sugaya 1982]
H. Sugaya:
Tree File: A Data Organization for Interactive Programs.
ETH PhD Dissertation 6944, 1983.

[Wirth 1981]
N. Wirth:
The Personal Computer Lilith.
Software Development Environments.
Acknowledgements

I wish to thank Prof. J. Nievergelt for supervising this thesis, and Prof. W. Nef for his encouragement and for accepting the job of being co-examiner. Thank you also to Dr. H. Bieri, P. Fink, and Dr. A. Meier for their comments on drafts of this thesis, and A. Müller and Nancy Van Natta for checking my English formulations. A special thank you to M. Stone for convincing me of the relevance of the numerical problems.
Curriculum vitae

I was born on April 14th 1961 in Brugg, Switzerland. After attending primary school in Bellinzona and Lugano, I entered the Ginnasio in Lugano. In 1971, I obtained the Maturita' Tipo C from the Liceo Cantonale in Lugano.

In autumn 1971, I began my studies at the Faculty of Mathematics and Physics of the Swiss Federal Institute of Technology (ETH), where I earned a Diploma in Mathematics in 1977. In my Diploma Thesis, I studied the stability of a class of fast algorithms under the supervision of Prof. P. Henrici.

From 1977 to the spring of 1980, I worked as a Systems Representative for Burroughs Switzerland in Zurich.

In April 1980, I joined the research group of Prof. J. Nievergelt at the Institut für Informatik of the ETH. There I worked in a project on man-machine communication. My primary contribution in this effort consisted of the development of graphic software.