1. The first step is to identify the problem or question that needs to be answered. This involves understanding the context and the specific requirements of the task.

2

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Implementation of Algorithms in a Computational Geometry Workbench

by
Alan Knight

**A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science**

**School of Computer Science
Carleton University, Ottawa, Ontario**

April 9, 1990

© copyright 1990, Alan Knight

Government of Canada

Gouvernement du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60480-8

**The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis**

**Implementation of Algorithms in a
Computational Geometry Workbench**

**submitted by Alan Knight B.C.S.
in partial fulfillment of the requirements for
the degree of Master Of Computer Science**

J. R. Sack
Thesis Supervisor

J. Fiala
Director, School of Computer Science

Carleton University

April 9, 1990

ABSTRACT

In this thesis we describe the design and implementation of a computational geometry workbench and discuss issues arising from this work. The workbench is implemented in an object-oriented style and provides a set of classes to represent geometric objects; an interactive user-interface for creating, editing, and manipulating geometric objects; animation facilities; geometric and non-geometric data types; and a representative set of implemented geometric algorithms, including the previously unimplemented Tarjan and van Wyk $O(n \log \log n)$ algorithm for triangulating an n -vertex simple polygon.

We consider implementation issues for algorithms in computational geometry, generation of "random" polygons and Jordan sequences, and direct comparison and examination of constant factors for different algorithms and data structures. The comparisons include algorithms for triangulation and finding the convex hull of a simple polygon, and various data structures, including finger and splay trees. We investigate empirically the dynamic optimality conjecture of Sleator and Tarjan for splay trees.

ACKNOWLEDGEMENTS

The geometric workbench is a large system, and substantial contributions have been made by many different people. I would like to thank specifically the following people:

- Jörg Sack, my thesis supervisor, who initiated and continues to oversee the computational geometry workbench project, and whose knowledge and support have been essential to its success.
- Kirsten Knight, my wife, for her support and patience throughout.
- Joe May, Thong Nguyen, and Jeff McAffer, who were responsible for the original work on the workbench, including many of the initial design concepts and the basic user interface. Joe May is also responsible for an earlier implementation of the edge-visibility to trapezoidation section of the Tarjan and van Wyk triangulation algorithm which was very helpful.
- Una-May O'Reilly, who implemented many of the basic structures and operations for finger trees.
- Students in the computational geometry course at Carleton whose projects have been included in the workbench. These include:
 - James Stringham, 1988/89: Voronoi diagram
 - Alex Smith, 1988-89: Line segment visibility
- A number of students have work currently in progress on fourth-year projects involving the workbench. These include:
 - Jean-François Morneau: Weighted Voronoi diagrams
 - Peter Epstein: Polygon shortest path algorithms and their applications
- I would particularly like to thank Peter Epstein, whose contributions have been outstanding, including the shortest-path algorithms mentioned above, implementation of the

DCEL representation for graphs, an excellent implementation of splay trees done on very short notice, and many lesser contributions and perceptive observations.

- John Pugh and Wilf Lalonde, whose advice on matters of object-oriented style has been very helpful, and whose implementation of balanced search trees was adapted for the workbench.
- T. Doe and S. Edwards, whose Pascal implementation of "random" polygon generation was used as a basis for the test case generator.
- Mike Wilson, whose detailed knowledge of the Smalltalk/V Macintosh implementation has rescued me on more than one occasion.
- Nick Edgar, who wrote the code which allows pasting both pictures and text representations to the Macintosh clipboard.
- Andrew Rau-Chaplin and Una-May O'Reilly, for many enlightening conversations.

TABLE OF CONTENTS

Acceptance Sheet.....	ii
Abstract.....	iii
Acknowledgements	iv
Table of Contents	vi
List of Figures.....	xi
1 Introduction	1
2 The Computational Geometry workbench.....	4
2.1 Motivation.....	4
2.2 Design goals for a workbench	5
2.3 Environment	6
2.3.1 Graphical Support and User Interface.....	6
2.3.1.1 User interface concepts	8
2.3.1.2 Animation.....	9
2.3.1.3 Debugging.....	9
2.3.1.4 Portability.....	10
2.3.1.5 Uniform behaviour of geometric objects.....	11
2.3.1.6 High-level support.....	11
2.4 Algorithm support.....	12
2.4.1 Basic features of object-oriented programming.....	12
2.4.2 Object-oriented languages.....	13
2.4.3 Geometric Objects.....	14
2.4.4 Geometric data structures	17
2.4.5 Non-geometric data types and data structures	17
2.4.5.1 Data types.....	18
2.4.6 Standard Algorithmic Tools.....	19
2.4.7 Algorithmic Correctness.....	19
2.5 Important workbench features	20
2.5.1 Labels.....	20
2.5.2 Plane Sweep	21
2.5.3 Numeric enhancements.....	21
2.5.4 An example algorithm	22
2.6 Chapter Summary	23

3	Triangulation of a Simple Polygon	24
3.1	Background	24
3.2	Triangulation Algorithms	27
3.2.1	Algorithm concepts	27
3.2.2	Stage 3: Trapezoidation - Triangulation	30
3.2.2.1	Triangulating Unimonotone Polygons	31
3.2.2.2	Decomposition into Unimonotone Pieces	33
3.2.2	The Fournier and Montuno Algorithm	35
3.2.3	Stage 2: Edge-Vertex Visibility - Trapezoidation	39
3.2.4	Stage 1: Polygon - Edge-Vertex Visibility	41
3.2.4.1	Divide and Conquer	43
3.2.4.2	Lower Bounds	45
3.2.4.3	An $O(n \log n)$ algorithm	46
3.2.4.4	Jordan sorting	48
3.2.4.4.1	Amortized Computational Complexity	49
3.2.4.4.2	Homogeneous Finger Trees	51
3.2.4.4.3	Applicability of Bracket Sequences	53
3.2.4.4.4	Sorting	55
3.2.4.4.5	Representation of Family Lists	57
3.2.4.4.6	Analysis of Jordan Sorting	58
3.2.4.4.7	Summary	59
3.2.4.5	Representation of the Polygon Boundary	60
3.2.4.5.1	Boundary Segments and Visibility Regions	60
3.2.4.5.2	Heterogeneous Finger Trees	63
3.2.4.5.2.1	Representing boundary segments	63
3.2.4.5.2.2	Representing lists of boundary segments	64
3.2.4.6	Splitting sub-polygons	67
3.2.4.6.1	Deep and Shallow Sections	67
3.2.4.6.2	Essential and Non-Essential Intersections	68
3.2.4.6.3	Finding Intersections and Splitting Visibility Regions	70
3.2.4.6.4	Re-assembling Visibility Regions	72
3.2.4.7	Remaining Issues	73
3.2.4.7.1	Error Correction	73
3.2.4.7.2	Balanced Divide and Conquer	75

3.2.4.7.3	Overall Time Complexity	76
3.3	Implementation of the Algorithms.....	78
3.3.1	Implementation Goals	78
3.3.2	Implementation Issues.....	79
3.3.2.1	Algorithmic issues.....	80
3.3.2.1.1	Tangency	81
3.3.2.1.2	Symmetry.....	82
3.3.2.1.3	Finger trees.....	82
3.3.2.1.4	Representation of output.....	84
3.3.2.1.4	Trapezoidation	87
3.3.2.1.5	Heterogeneous finger trees	89
3.3.2.2	Translation Issues	90
3.3.2.2.1	Iterative Development.....	90
3.3.2.2.2	Destructive operations.....	91
3.3.2.2.3	The object-oriented model	92
3.4	Chapter Summary	94
4	Generating Jordan sequences and polygons.....	95
4.1	Random Jordan Sequences.....	95
4.1.1	Motivation.....	95
4.1.2	An $O(n \log n)$ algorithm for Jordan sequence generation	97
4.1.2.1	Algorithm Concepts	97
4.1.2.2	Selecting an arc	100
4.1.2.3	Inserting a new arc	100
4.1.2.4	Finding visible arcs.....	101
4.1.2.5	Modifying a visible node	104
4.1.2.6	Updating.....	105
4.1.3	A nearly linear-time algorithm for generating Jordan sequences	106
4.1.3.1	Random generation of forests.....	108
4.1.3.1.1	Lexical generation of random forests.....	108
4.1.3.1.2	Adding random children	110
4.1.3.1.3	Forest generation by unranking.....	110
4.1.3.2	Generating a random Jordan sequence.....	113
4.1.3.2.1	Analysis of the algorithm	114
4.1.4	Conclusions.....	116

4.2	"Random" Polygons	116
4.2.1	Generating polygons by a "random walk"	117
4.2.2	Generating restricted classes of polygons.....	118
4.2.2.1	Monotone.....	118
4.2.2.2	Star-shaped.....	119
4.2.3	Generating Random Polygons Using Visibility Information	120
4.2.3.1	Algorithm Description.....	120
4.2.3.2	Implementation.....	122
4.2.3.3	Evaluation	123
4.2.3.4	Polygon generation using edge visibility	124
4.2.3.5	Generation using triangulation	125
4.3	Chapter Summary	127
5	Comparison of Algorithms	128
5.1	General notes	128
5.2	Implementability and lines of code	128
5.3	convex hull algorithms.....	129
5.4	Comparing Jordan sorting.....	133
5.5	comparing triangulations	138
5.6	Other Comparisons.....	141
5.7	Chapter Summary	142
6	Conclusion	143
6.1	Workbench extensions.....	143
6.1.1	Improved user interface	144
6.1.2	Three-dimensional geometry.....	144
6.1.3	Additional geometric objects.....	144
6.1.4	More general algorithm structure.....	145
6.1.5	Better source code control.....	145
6.1.6	Porting.....	146
6.1.7	Organization	146
6.2	Future research	146
6.2.1	Counting Jordan sequences.....	146
6.2.2	Algorithms for dynamic property maintenance	147
6.2.3	Modifications to the Tarjan and van Wyk algorithm.....	147
	References	148

Appendix A: Detailed description of the workbench user interface	152
A.1 Using the Workbench.....	152
A.1.1 Geometric Browser Concepts	153
A.1.2 Geometric Browser Panes	153
A.1.2.1 Status Pane.....	154
A.1.2.2 Button Pane	154
A.1.2.3 Geometric Object Pane.....	156
A.1.3 Geometric Browser Menus	156

LIST OF FIGURES

Fig. 2.1	A polygon and its triangulation	7
Fig. 2.2	A geometric object browser	8
Fig. 2.3	An inspector.....	10
Fig. 2.4	Points, line segments and polygons.....	15
Fig. 2.5	An example planar straight line graph.....	16
Fig. 2.6	Voronoi diagram.....	17
Fig. 2.7	The support lines of a convex polygon	22
Fig. 2.8	A simple algorithm	23
Fig. 3.1	A triangulation of a 400 vertex polygon	25
Fig. 3.2	Stage 3 of the Tarjan and van Wyk algorithm.....	27
Fig. 3.3	Stage 2 of the Tarjan and van Wyk algorithm.....	28
Fig. 3.4	Stage 1 of the Tarjan and van Wyk algorithm.....	28
Fig. 3.5	Two Unimonotone polygons.....	30
Fig. 3.6	Several steps of the Graham scan algorithm.....	32
Fig. 3.7	Triangulation of a dent.....	32
Fig. 3.8	Pseudocode for the triangulation of unimonotone polygons	33
Fig. 3.9	Cases for trapezoids.....	34
Fig. 3.10	Representation of trapezoids	35
Fig. 3.11	Characterizations of vertices.....	36
Fig. 3.12	Pseudocode for the Fournier and Montuno algorithm.....	36
Fig. 3.13	Pseudocode for the Fournier and Montuno algorithm.....	37
Fig. 3.14	Pseudocode for the Fournier and Montuno algorithm.....	37
Fig. 3.15	Pseudocode for Fournier and Montuno's algorithm.....	40
Fig. 3.16	Intersections of a polygon with a horizontal line.....	41
Fig. 3.17	A vertex with only one visible edge	42
Fig. 3.18	Determining visibility based on turn direction.....	43
Fig. 3.19	A polygon split into three pieces.....	44
Fig. 3.20	A polygon with $O(n^2)$ intersections	45
Fig. 3.21	Regular and irregular vertices	46
Fig. 3.22	A simple polygon and its decomposition into monotone pieces.....	47
Fig. 3.23	A sample Jordan sequence.....	48
Fig. 3.24	A homogeneous finger tree.....	52
Fig. 3.25	Three-way split	53

Fig. 3.26	Examples of well-formed and non-well-formed bracket sequences.....	54
Fig. 3.27	A well-formed bracket sequence.....	54
Fig. 3.28	A family list and its representation as a homogeneous finger tree.	57
Fig. 3.29	A recurrence for the time spent in split-and-insert operations.	59
Fig. 3.30	A simpler recurrence in one variable.	59
Fig. 3.31	Example visibility regions.....	61
Fig. 3.32	Example boundary segments.....	61
Fig. 3.34	The general form of a visibility region.....	62
Fig. 3.35	A portion of the boundary of P and its representation.....	64
Fig. 3.36	The top part of a visibility region and its representation.....	66
Fig. 3.37	Deep, shallow and mixed sections.....	68
Fig. 2.38	Non-essential intersections.....	69
Fig. 3.39	Pseudocode for processing a visibility region.....	70
Fig. 3.40	A visibility region cut into 9 subregions.....	73
Fig. 3.41	Processing a deep section.....	74
Fig. 3.42	Error correction.....	74
Fig. 3.43	The three-way split operation for homogeneous finger trees.....	84
Fig. 3.44	A triangulation with $O(n)$ edges incident on vertex A.....	85
Fig. 3.45	A unimonotone polygon and its triangulation.....	86
Fig. 3.46	A difficult case for the Fournier and Montuno algorithm.....	88
Fig. 3.47	Another difficult case.....	88
Fig. 3.48	Three ways of organizing verices in a circular doubly-linked list.....	93
Fig. 4.1	Intersections of a star-shaped polygon with a horizontal line.	96
Fig. 4.2	A "random" polygon of 120 vertices.....	97
Fig. 4.3	A Jordan sequence defined by a simple closed curve and a line L.....	98
Fig. 4.4	Node Q is split into Q' and Q'' and joined to node m.....	99
Fig. 4.5	Pseudocode for insertion of a new arc.....	100
Fig. 4.6	Adding a new node A' as the right sibling of A.....	100
Fig. 4.7	Visibility from an arc.....	101
Fig. 4.8	Cases for selecting the parent node Q.....	102
Fig. 4.9	Modifying a visible arc.....	105
Fig. 4.10	Node R is split into R' and R'' and joined to node m.....	105
Fig. 4.11	Connected and disconnected intersections.....	107
Fig. 4.12	Algorithm to produce a random bracket sequence.....	109

Fig. 4.13 A bracket sequence with the first well-formed sub-sequence highlighted.....	111
Fig. 4.14 Pseudocode for the ranking operation.....	112
Fig. 4.15 Equivalence classes.....	113
Fig. 4.16 Algorithm to generate a random Jordan sequence	114
Fig 4.17 Generation using a random walk.	117
Fig. 4.18 Generation of a monotone polygon from a random set of points.	119
Fig. 4.19 Star-shaped polygons produced by the algorithms described above.	120
Fig. 4.20 Generation of a random polygon using visibility information.....	121
Fig. 4.21 An example step of the algorithm in Fig. 4.21	121
Fig. 4.22 A heuristic improvement to the algorithm of Fig. 4.20.	122
Fig. 4.23 A heuristic improvement to the algorithm of Fig. 4.20	124
Fig. 4.24 Algorithm for generation of a random polygon	125
Fig. 4.25 Generation of a random polygon using triangulation.....	126
Fig. 5.1 Gift-wrapping a polygon to find the convex hull	130
Fig. 5.2 A plot of time vs. number of vertices for convex hull algorithms on a set of points.....	131
Fig. 5.3 An enlarged plot of the Melkman and Graham Scan algorithms	132
Fig. 5.4 Four methods of sorting Jordan sequences.....	134
Fig. 5.5 Time / number of points in sorting Jordan sequences.....	136
Fig. 5.6 A function $F(n)$ which is linear in n	137
Fig. 5.7 The definition of the function plotted in Fig. 5.6.	137
Fig. 5.8 Graphs for the triangulation of simple polygons.....	139
Fig. 5.9 Triangulation of star-shaped polygons.....	140
Fig. 5.10 Triangulation of monotone polygons.....	141
Fig. A.1 An example geometric object browser	152
Fig. A.2 A summary of the different browser modes.....	155
Fig. A.3 A geometric object browser in the process of creating a polygon.....	156
Fig. A.4 The entries in the browser Edit menu	157
Fig. A.5 The entries in the browser Workbench menu	157
Fig. A.6 A geometric object browser showing several copies of an object at different stages of magnification and the workbench menu.....	158
Fig. A.7 The entries in the workbench debug menu	158

1 INTRODUCTION

In this thesis we discuss the design and implementation of a geometric computing environment called the Workbench On Computational Geometry (WOCG). This provides a suite of tools for the implementation, application, and presentation of algorithms in computational geometry. The system includes a graphical user interface which allows the creation and manipulation of geometric objects. The features of the system and its implementation are discussed in Chapter 2. The system is written in Smalltalk/V using an Apple Macintosh II computer.

The validity of the approach taken is demonstrated by showing that the Tarjan and van Wyk algorithm for triangulation of a simple polygon [63], which has not been previously implemented, can be implemented using this system. A number of interesting issues arise from this implementation. Among these is the question of how efficient this $O(n \log \log n)$ algorithm is compared to a related, but asymptotically less efficient $O(n \log n)$ algorithm due to Fournier and Montuno. [23]

Tarjan and van Wyk also restate the dynamic optimality conjecture of Sleator and Tarjan [59, 66], saying "the...dynamic optimality conjecture...suggests that the $O(n \log \log n)$ time bound is still valid if splay trees...are used in place of finger search trees." They further state that "the use of splay trees might lead to a practical implementation of our algorithm, although this must be verified by experiment."

This thesis addresses the empirical aspect of this conjecture, providing timing results for these and other algorithms in Chapter 5.

Resulting from this implementation-oriented work a problem occurs in generating appropriate test instances for the algorithms. For example, the problem of generating a Jordan sequence of a specific length arises. This thesis gives two algorithms for solving this problem, an $O(n \log n)$ time algorithm and a second, nearly linear time algorithm.

Several algorithms for generating simple polygons with a given number of vertices are also presented. Previous algorithms generate highly structured polygons, deemed

insufficiently demanding for testing algorithms. The algorithms developed here are presented in Chapter 4 of this thesis.

The major emphasis of the thesis is on the creation of a framework for implementation-oriented work in computational geometry, and on the implementation of the Tarjan and van Wyk algorithm and the issues that have arisen from that implementation. Because of the complexity of the Tarjan and van Wyk algorithm, a great deal of space has been required for the exposition of that algorithm. This algorithm, as well as the Fournier and Montuno algorithm, and some of the issues which arose during their implementation, are discussed in Chapter 3.

2 THE COMPUTATIONAL GEOMETRY WORKBENCH

2.1 MOTIVATION

During the past decade, algorithms have been developed for solving a wide spectrum of problems in computational geometry. Many of these algorithms have never been implemented. When implemented, they have often been treated in isolation. Constant factors, numerical stability, conversion between representations, and other implementation concerns are sometimes discussed in algorithm descriptions but the results of actual implementations are seldom published.

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

There is a need in the computational geometry community for more direct comparisons of algorithms in terms not only of worst-case complexity, but of execution time and space (including constant factors), ease of implementation, robustness, handling of special cases, numerical stability and average-case performance. Information on these characteristics is important in assessing an algorithm, particularly to the implementation-oriented researcher.

Geometric algorithms can be very complex, and even elementary operations such as line segment intersection can present considerable difficulty in implementation, particularly when issues of finite-precision arithmetic and degenerate cases are involved. Far more complex operations, such as finding the convex hull, the Voronoi diagram, or a triangulation, are fundamental to many algorithms. An implementation should not be

viewed as a finished product, but as a "building block" from which higher-level algorithms and applications may be constructed.

Forrest emphasizes the need for a "geometric computing environment" [21] which can provide a unified framework and a suite of tools to reduce difficulties in the implementation, application and evaluation of geometric algorithms. These tools are particularly necessary for processing large examples, but this is not the only role for a workbench. A graphical environment which allows creation and manipulation of small examples is a valuable tool for generating examples, debugging, explaining, and documenting algorithms.

2.2 DESIGN GOALS FOR A WORKBENCH

The computational geometry workbench should provide more than a collection of geometric algorithms and data structures. It should provide a library of operations, both basic and advanced, implemented in a general, flexible, and re-usable manner. It should provide a framework in which these operations can be assembled to form more complex algorithms. Finally, it should provide an environment which makes the implementation and use of these operations as simple as possible.

We list the following design criteria as being of fundamental importance, dividing them into the categories of Environment and Algorithm Support:

Environment

- an interactive graphical environment in which geometric objects can be created, modified, and operated on by algorithms
- support for algorithm animation
- high-level graphical and symbolic debugging facilities
- portability

- high-level support for basic operations such as device-independent graphics and storage management.

Algorithm support

- support for high-level, representation-independent geometric objects (points, lines, polygons, planar subdivision, polyhedra,...)
- support for geometric data structures (segment trees, range trees,...)
- support for non-geometric data types and data structures relevant to geometric problems (finger trees, splay trees, heaps,...)
- support for a variety of "standard" algorithmic tools (convex hull, triangulation, etc.) to serve as "building blocks" for more complex algorithms
- algorithmic correctness

2.3 ENVIRONMENT

The computational geometry workbench is implemented in an object-oriented style. This is a style of programming which has proven particularly appropriate for graphics-oriented applications [27, 68] and was judged to be appropriate for algorithms which operate on geometric objects.

2.3.1 Graphical Support and User Interface

A geometric computing environment should provide visual feedback on its operations. A system whose input and output are in textual form could be useful for larger computations, with inputs which come from non-human sources (e.g. satellite data, robot sensors, etc). It would not, however, aid the user in creating examples and test cases, illustrating the algorithm, or verifying results.

Fig. 2.1 shows a polygon P and a triangulation of P , in graphical and textual formats. In most circumstances, the graphical presentation is preferable.

Fig. 2.1 A polygon and its triangulation, with equivalent text representations.

The geometric workbench provides an interactive graphical interface to allow the user, under most circumstances, to deal with images of the objects under consideration, rather than their underlying representation. This interface can be divided into features intended for the algorithm implementor, and features intended for the user of the system. For users, the most important features is the *geometric object browser*, also called the *browser*. This provides an interactive facility for creating, editing, storing, retrieving, and applying algorithms to geometric objects. Fig. 2.2 shows an example browser.

Fig. 2.2 A geometric object browser, illustrating some of the algorithms applicable to polygons.

2.3.1.1 User interface concepts

We briefly explain a few fundamental concepts used in the geometric object browser. The most fundamental is that of the entities called *geometric objects*, which the browser manipulates. These correspond directly to the conceptual geometric objects on which geometric algorithms operate. Examples are points, line segments, polygons, and planar subdivisions.

The browser maintains a collection of geometric objects, and all algorithms invoked in the browser refer to one or more of these objects. Normally, one object is *selected*, meaning that it may be moved, changed, or have an algorithm applied to it. The object which is selected is called the *current object*, and is normally drawn with a thicker line than the other objects in the browser.

The current object may be moved or edited using the mouse. Buttons along the bottom of the browser window determine the *mode* of the browser. The mode may affect the response to mouse operations. For example, in *Select* mode, clicking with the mouse in the area of a vertex will cause the object which contains that vertex to be selected. In

Delete Object mode the same action will cause the object to be deleted, and in *Edit* mode it will allow the vertex to be moved.

Additional commands are available from the menus. Most of these apply to the current object, although some will affect all objects in the browser. Menu options include editing commands (e.g. Cut, Paste, Undelete), workbench specific commands (e.g. Send to Back, Magnify Object, File In/Out Object) and development commands (e.g. Disable/Enable Animation, Inspect Object, Disable/Enable detailed checking of algorithms). Cut and paste operations work with the Macintosh clipboard, storing pictures of the objects and a text representation from which the object may be reconstructed (See Fig. 2.1 for examples of this format).

2.3.1.2 Animation

An important component of the graphical support in the workbench is algorithm animation. This is a valuable tool for demonstrating, explaining, and debugging of algorithms. By providing a graphical record execution, an animation can make an otherwise obscure error immediately apparent. A well-designed animation of an algorithm can make the operation of a difficult algorithm clear, providing a valuable aid for teaching of algorithms and presenting research work.

In general, algorithm animation is a difficult problem [8]. Many difficulties arise from questions of how to graphically represent data structures and what parts of an algorithm to animate. Many of the structures used in geometric algorithms already have graphical representations which are supported by the environment. This can be exploited in the construction of an animation which appropriately reflects the algorithm structure.

2.3.1.3 Debugging

The geometric workbench provides high-level symbolic debugging facilities, using the debugger of the host environment. The geometric object browser can be a valuable

debugging tool, allowing the rapid creation and modification of test cases and examples. Also available to the algorithm implementor are *inspectors*, (See Fig. 2.3) which allow examination of the structure of an object along with a graphical representation (for geometric objects or non-geometric structures with a geometric interpretation).

Fig. 2.3 An inspector, displaying a visibility region from the Tarjan and van Wyk algorithm

2.3.1.4 Portability

Portability is desirable in a geometric computing environment, but is also difficult to maintain. This is particularly true when one attempts to provide a graphical user interface. Graphics capabilities and windowing systems are not well-standardized, and can introduce incompatibilities between otherwise similar machines.

That part of the workbench code which relates to the graphical user interface must be machine-dependent. We have tried to preserve the portability of the system by isolating machine-dependent code from geometric and algorithmic code. This has the additional benefit that the user interface can be altered or replaced without affecting the main body of code in the workbench.

2.3.1.5 Uniform behaviour of geometric objects

In order for the machine-dependent code to be separated from the geometric and algorithmic code, and also for purposes of abstraction, it is important that all geometric objects behave in a uniform way. This is important at many levels, but particularly so at the level of communication between the user interface and the geometric objects.

In the geometry workbench, all geometric objects respond to a specific set of messages, known as the *communications protocol*. This includes all of the operations used by the browser, including operations to add, delete and change components, operations to display and move the object, and operations defining the algorithms applicable to that object.

This has the benefit of minimizing the code which is needed for the browser, since all geometric objects can be treated in the same way. New classes of geometric objects can be added by implementing this small set of well-defined messages.

2.3.1.6 High-level support

Geometric algorithms are extremely complex in and of themselves. In a geometric computing environment, the algorithm implementor should not have to be responsible for low-level operations such as storage management and graphics drivers.

As an example, in the workbench the display of geometric objects is abstracted as much as possible. Each geometric object has code to display itself. This code is written in terms of operations on a pen. The *pen* is an abstract drawing implement which supports line drawing operations independent of the display medium. The same code is used to display an object on the screen, to print an object or to store a picture of the object in a graphics description language.

2.4 ALGORITHM SUPPORT

The workbench is intended to provide an environment in which the implementation, maintenance, and testing of geometric algorithms is made as simple as possible. To do so, it should provide a set of tools to aid in these activities. Among the tools that we consider important are high-level, representation-independent geometric objects, standard data types and data structures (both geometric and non-geometric), and algorithms implemented in a general and re-usable way.

Since the workbench is written in an object-oriented style, we begin by describing some characteristic features of this approach.

2.4.1 Basic features of object-oriented programming

Object-oriented programming is a style of programming in which data and operations (often called *methods*) on that data are grouped together to form *objects*. For a given object A , the representation of A can only be accessed by methods which are defined to operate on A . For example, a vertex V might be represented by two variables, X and Y . Only those functions defined to operate on a vertex can access these variables. Other functions which need the value of this variable must request it by calling a function which is defined to operate on a vertex and returns the appropriate value (e.g. `returnXValue(V)`). This representation hiding is called *encapsulation*, and helps to isolate dependence on a particular representation in a small body of code.

A method can only be referenced within the scope of the object on which it is defined to operate. Different objects may define the same name, possibly in different ways. Thus objects representing integers, floats, fractions, and matrices may each define the addition operation $+$ in different ways. This is called *polymorphism*, referring to the different shapes which an operation may take. Polymorphism allows representation-independent

operations, since each different representation can provide its own implementation. Encapsulation and polymorphism together allow the implementation of abstract data types.

Finally, an object may be defined to *inherit* from other objects. An object *A* which inherits from another object *B* will have, by default, all the characteristics of *B*. Any of these characteristics may be changed or added to by appropriately defining new fields in the representation or new methods. Inheritance helps avoid duplication of code and allows variations on objects to be easily implemented. An example of inheritance might be a graph vertex. This could inherit normal vertex operations from a polygon vertex, but could also define a new field containing a list of edges and define appropriate new operations.

In most object-oriented systems, objects are organized into *classes*. A class defines a common structure and operations for many different objects. Some possible geometric classes are **Vertex**, **Polygon**, **BinarySearchTree**.

The concepts of computational geometry fit well into an object-oriented model. Geometric objects are clearly defined, with a common set of operations (e.g. convex hull, triangulate) which apply to many different objects. Classes of geometric objects may have subclasses which inherit from them (e.g. convex polygon, planar graph). All of these concepts map well into an object-oriented model.

2.4.2 Object-oriented languages

There exist many different object-oriented languages, implementing different degrees of object-orientation and variations of the basic features described above. The language which introduced object-oriented programming is SIMULA [49], a language intended for discrete event simulation.

Smalltalk-80 [27] provides a pure object-oriented language, in which the entire programming environment is implemented in an object-oriented style.

Other languages, representing different views of the object-oriented paradigm, have been introduced, including Eiffel [46], Actor [69], and SELF [9]. Other implementations have added object-oriented features to existing languages. These include the Common Lisp Object System (CLOS) [5], C++ [60], Objective-C [12], and object-oriented extensions to Pascal [6].

The computational geometry workbench is implemented in Smalltalk/V [14], a dialect of Smalltalk similar to, but distinct from Smalltalk-80. In addition to the advantages of object-oriented programming described above, this offers a sophisticated programming environment, a large, well-integrated library which includes source code for the windowing environment, and automatic storage management. It has disadvantages in the area of execution time and space requirements, but these judged to be less important than the advantages in programmer productivity for a large research project with relatively few programmers.

2.4.3 Geometric Objects

Any geometric computing environment must somehow represent geometric objects. Although "standard" representations for many geometric objects exist, they are by no means universal, and many algorithms exploit different representations for efficiency. The environment should support alternative representations of the same object, and standard operations should be independent of the representation.

Consider, for instance, a simple polygon, normally represented by an array of vertices in order of occurrence around the polygon boundary [51]. This is sufficient for many algorithms, but others may require the boundary to be represented as a list or in a more unusual form (The Tarjan and van Wyk triangulation represents the polygon boundary using multiple heterogeneous finger trees). Other algorithms may require an explicit

representation for edges. Polygon operations should not depend on the particular representation being used.

In general, the interchangeability of components is an important principle. Different representations of a geometric structure, different data structures and different algorithms should all be interchangeable at the highest possible level. This is important for testing and debugging of algorithms, for conducting comparison studies and for improving already-implemented algorithms as more efficient techniques for solving sub-problems become available.

The workbench provides definitions for a number of classes of geometric objects. At the moment, most are from the domain of 2-dimensional straight-line geometry. The most basic elements represented are vertices, line segments, lines and rays. Polygons are one of the most common geometric objects in the workbench and a number of varieties are provided, including simple, non-simple, convex, monotone, star-shaped, and others. The standard representation is as an array of vertices, though conversion operations are provided for a variety of different representations. Fig. 2.4 shows examples of some of these basic components.

Fig. 2.4 Points, line segments and polygons

The system supports undirected graphs, planar straight line graphs, and planar subdivisions (See Fig. 2.5). Particular credit for the graph classes should be given to Peter

Epstein, who, in the course of working on his honours project [20] has improved and expanded the original implementation of planar straight line graphs, designed by the author.

Fig. 2.5 An example planar straight line graph and its dual. The planar straight line graph represents the decomposition into triangles of a simple polygon. Each triangle contains one vertex of the dual graph. If two triangles t_1, t_2 share an edge, then the vertices of the dual graph contained in t_1, t_2 are connected by an edge of the dual graph.

Groupings of geometric objects are an important structure, but are difficult to represent in general because of the wide variety of properties they may have. For example, the workbench provides a representation for a complete set of monotone chains which covers a regular planar subdivision. This is used as a query structure for planar point location [50], and its structure and operations are not generally applicable to arbitrary groupings of objects. The workbench currently supports collections of objects on an ad hoc basis, as required by particular algorithms.

Group structures of more general interest include point sets and line segment sets. These are applicable in several contexts, and are used by a number of algorithms.

Finally, the workbench includes a representation of another generally useful structure, the Voronoi diagram (See Fig. 2.6) [22, 51].

Fig. 2.6 Voronoi diagram of a point set P . Each region of the Voronoi diagram contains exactly one point $p \in P$. The line which separates two regions r_1, r_2 containing points $p_1, p_2 \in P$ is equidistant from p_1 and p_2 .

2.4.4 Geometric data structures

The separation between geometric objects and geometric data structures is blurred. Segment trees and range trees [4] are clearly data structures intended for geometric algorithms. Objects such as funnels, hourglasses, [30] monotone chain sets [50], and boundary segments [66] might be considered data structures, but might also be considered geometric objects in themselves. Within the workbench the only rigid distinction is that something which is classified as a geometric object can be manipulated by the browser, something which is not a geometric object cannot be so manipulated.

2.4.5 Non-geometric data types and data structures

A good set of data structures is an important tool for algorithm implementation. As with geometric objects, interchangeability is a primary concern, of even more importance in this

domain. Not only are the number of data structures which can implement the same abstract data type much larger, but there can be subtle differences in operation semantics which are more troublesome than for geometric representations. Since algorithms often exploit these differences they can be difficult to hide from the user of a data structure.

These semantic differences can include the issue of destructive and non-destructive operations. There can also be differences in the meaning of similar operations including different reactions to unusual situations or to failures, changes in output values, and different limitations. Many of these inconsistencies are most acute between array-based representations and "cell-based" structures such as linked lists and trees.

2.4.5.1 Data types

To deal with different data structures at a sufficiently abstract level, and to make these data structures interchangeable in algorithm code, we must work in terms of abstract data types rather than data structures. This can be accomplished by enclosing the data structure inside a data type which masks the differences in operation semantics as much as possible. Thus, data types should be interchangeable, even though the underlying data structures are not.

The same data structure may represent more than one data type. For example, a splay tree can be used to represent an ordered collection, a sorted collection, or a dictionary. This does not mean that operations for all of these data types can be applied at will. If objects have been inserted in arbitrary order, then attempting to retrieve them as from a dictionary will fail. Masking the representation using abstract data types can help prevent this sort of confusion.

Mechanisms of this kind are not uncommon in object-oriented programming. The concept of encapsulators occurs in many contexts. *Views*, a very similar concept, have been discussed in contexts such as software engineering [58].

In the context of data types and data structures this does not appear to have arisen as often. In the standard object-oriented paradigm it is common for each data structure to represent a single abstract data type. In this case, the data type and the data structure can be merged, with only slight confusion possible as to which methods are public and which private. It is only when providing a large set of general data structures that the need for this further abstraction becomes pressing.

2.4.6 Standard Algorithmic Tools

Computational geometry algorithms are frequently built up using other algorithms as "building blocks". A set of basic operations, implemented in a general way, is thus extremely important. The workbench provides implementations of many such basic operations, including convex hull, Voronoi diagram, triangulation, planar point location, and various visibility algorithms. For example, the triangulation algorithm [66], as implemented in the workbench, has been used in many other algorithms, e.g. [25, 30, 31, 61]. In particular, the visibility algorithms of [30] which are implemented in Peter Epstein's honours project [20] use this triangulation as a preprocessing step.

2.4.7 Algorithmic Correctness

In order to implement algorithms of the degree of complexity which often occurs in computational geometry the use of abstraction is of fundamental importance. Without proper abstraction the implementation of even a moderately complex algorithm is unlikely to be successful, and the likelihood that the code can be maintained and re-used with any degree of success is infinitesimal.

In using this abstraction, however, it is easy to sacrifice proper time complexity. Even when using predefined abstract data types and algorithms, the implementor must be aware of the cost of operations in the underlying representation, or possibly the cost of representation conversion. Correctly implementing an algorithm in the required time complexity requires a detailed understanding of the operations and data structures.

Because the distinction between data structures and data types is not often made clearly in the literature it is particularly important for the implementor to be able to determine the data types used in a problem and to implement them correctly.

2.5 IMPORTANT WORKBENCH FEATURES

We briefly describe some features of the workbench, not yet discussed, which have proven to be of particular use or interest.

2.5.1 Labels

It is common for an algorithm to require information to be stored in a geometric object in addition to that which is defined in the standard representation. Thus a vertex may store its numbering, the corresponding vertex in its graph representation, the trapezoids which it participates in, or other information. Although it is possible to expand the representation to accomodate these extra fields this is wasteful of space and can become extremely complex as more and more algorithms are implemented.

We define a *label* for a geometric object A to be an association between a key and a value which is specific to A . A geometric object may have any number of labels, and the value for a label can be set or retrieved in time $O(k)$ where k is the total number of labels stored in A . In practice, k is a small constant and so a label can be used as an extra field in the object representation.

This has proven to be an extremely valuable addition to the workbench, and one which has considerably simplified several algorithms.

2.5.2 Plane Sweep

One of the fundamental techniques in computational geometry is the plane sweep. In this model, a line sweeps across a geometric object, say from left to right. As the line progresses, it encounters *event points*. At each event point, some action may be taken and the status of the plane sweep updated.

A simple example is in finding the first pair of line segments which intersect from among a set of line segments S . The event points are the start and end points of the lines in S . As the beginning of each line is encountered, that line is added to a search tree T (which holds the algorithm status), and tested for intersection against its neighbours in T^* . As the end of a line is encountered, that line is removed from T and its neighbours in T are tested for intersection.

We have implemented a general mechanism which supports operations of this type, which allows, among other features, arbitrary orderings of the event points. This allows sweeps in any direction, including circularly.

2.5.3 Numeric enhancements

Smalltalk supports a variety of different numeric types, including fractional arithmetic. It has been extended in the workbench to allow for error in floating point comparisons, and to accomodate infinite numbers.

* Note that in implementing this example it is useful to label, as described above, the end points of a line with the line to which they belong.

2.5.4 An example algorithm

To illustrate the style of the geometric workbench, we include a small example algorithm (See Fig. 2.8). This algorithm operates on a convex polygon, and finds the support lines of that polygon via linear search. Fig. 2.7 shows example support lines of a convex polygon.

Fig. 2.7 The support lines of a convex polygon from a point p .

supportLines: givenPoint

| minPoint minAngle maxPoint maxAngle refPoint angle|

"First, find a point which is inside the polygon. The average of any two points on the boundary is sufficient"

**refPoint := ((self vertices first) +
(self vertices last)) / 2.**

"Loop through all points to find the ones with minimum and maximum angle with respect to the reference point"

minAngle := Infinities positiveInfinity.

maxAngle := Infinities negativeInfinity.

self vertices do: [:eachVertex |

angle := refPoint angleTo: givenPoint and: eachVertex.

(angle < minAngle) ifTrue: [

minPoint := eachVertex.

minAngle := angle].

(angle > maxAngle) ifTrue: [

maxPoint := eachVertex.

maxAngle := angle].

"Return a set of line segments containing the line from the given point to the points with minimum and maximum angle"

^LineSegmentSet with:

(LineSegment point1: givenPoint point2: minPoint).

with: (LineSegment point1: givenPoint point2: maxPoint).

Fig. 2.8 A simple algorithm written using the workbench. This finds the support lines of a convex polygon using a linear search.

2.6 CHAPTER SUMMARY

In this chapter we have described the design goals and the basic features of the computational geometry workbench. These include an interactive graphical environment (described in more detail in Appendix A), representations for geometric objects, representation of data types and data structures (both geometric and non-geometric), and algorithm animation.

3 TRIANGULATION OF A SIMPLE POLYGON

3.1 BACKGROUND

Decomposition problems are of fundamental importance in computational geometry, and have been extensively studied, both for their theoretical interest and because of their value as pre-processing steps in solving other problems [3, 37, 55]. See also [38, 51]. Decompositions into triangles are of particular interest, and are essential components of algorithms for planar point location [39], visibility and motion planning [25, 30, 62], and finite element analysis [7, 24, 35].

Our attention focuses on the triangulation of a simple polygon $P=(P_0, P_1, P_2, \dots, P_{n-1})$, defined by a list of its vertices in clockwise order around the boundary. Triangulation is performed by inserting $n-3$ non-intersecting *diagonals* (line segments joining two vertices of the polygon) which lie entirely within the interior of P and partition P into $n-2$ triangles. Fig. 3.1 shows an example triangulation of a simple polygon.

Fig. 3.1 A triangulation of a 400 vertex polygon

It has long been known that the triangulation problem for a set of n points has a lower time bound of $\Omega(n \log n)$. This follows from the lower bound on the convex hull of a set of points shown by Shamos, [57] since the convex hull of a set of points can easily be derived in linear time from a triangulation. Until fairly recently it was not known whether this bound applied to the problem of triangulating a simple polygon. A number of $O(n \log n)$ algorithms have been developed. The first of these is due to Preparata [51], using a decomposition method. Another is due to Fournier and Montuno [23], and is discussed in more detail in this chapter.

Linear time algorithms for limited classes of polygons [26, 56, 70], and algorithms of the form $O(n \log k)$ for some parameter k , based on the shape of the polygon [10, 32] existed,

but none could guarantee worst case performance better than $O(n \log n)$ for an arbitrary simple polygon.

In 1988, Tarjan and van Wyk published a triangulation algorithm for arbitrary n -vertex simple polygons with worst-case $O(n \log \log n)$ time complexity [66]. This was, until very recently, the best known algorithm for this problem*. It remains an open problem to demonstrate a linear-time algorithm for triangulation of a simple polygon or to show a lower bound greater than $O(n)$.

The algorithm described by Tarjan and van Wyk is very complex, both in terms of its data structures used and its time complexity. It makes extensive use of finger trees, a form of balanced binary search trees which have good amortized time complexity, but are difficult to implement. Tarjan and van Wyk suggest in the algorithm description that "Finger trees are sufficiently complicated that one would probably not want to use them in an actual implementation" and that splay trees would be suitable as an alternative, although the worst-case time bounds for splay trees have not been proven.

Due to the extreme complexity of the algorithm and to doubts about the constant factors involved, the algorithm has not previously been implemented. In this thesis we discuss our implementation of this algorithm. We investigate the performance and implementability concerns via an empirical comparison of this algorithm to the Fournier and Montuno $O(n \log n)$ algorithm, and we conduct tests on the suitability of splay trees.

This chapter describes both the Tarjan and van Wyk and the Fournier and Montuno algorithms, and the issues that arose during their implementation.

* There have been substantial developments in this area very recently with the announcement of two new algorithms. The first of these [Kirkpatrick, 1990 #54] is also $O(n \log \log n)$ but uses simpler data structures and may be less difficult to implement. The second, due to Chazelle, is a $O(n \log^*(n))$, where $\log^*(n)$ is defined to be the number of iterations of the log operation applied to n , which are needed to produce a result less than or equal to 1 (e.g. for $\log^*(16)$, $\log \log \log(16)=1$, $\therefore \log^*(16)=3$). Details of these algorithms have not yet been published.

3. TRIANGULATION ALGORITHMS

We describe both the Tarjan and van Wyk and the Fournier and Montuno algorithms in some detail. Since the Tarjan and van Wyk algorithm is much more complex and makes use of significant portions of the Fournier and Montuno algorithm, we primarily describe the Tarjan and van Wyk algorithm, with portions of the Fournier and Montuno algorithm being discussed as they arise in this context.

We begin by presenting an overview of the three stages of the Tarjan and van Wyk algorithm, followed by a naive approach to implementation which is successively refined to achieve the final $O(n \log \log n)$ time complexity.

3.2.1 Algorithm concepts

The Tarjan and van Wyk algorithm has three distinct stages. We briefly describe each stage below, followed by a more detailed exposition in a later section. To show the motivation for each stage, the stages are described in reverse order of execution.

The first of these stages is due to Tarjan and van Wyk, while the last two are based on the previously published $O(n \log n)$ algorithm of Fournier and Montuno [23].

Fournier and Montuno showed that, given a decomposition into trapezoids (called a *trapezoidation* for brevity) of a polygon P , a triangulation could be produced in linear time (See Fig. 3.2).

Fig. 3.2 Stage 3 of the Tarjan and van Wyk algorithm. A trapezoidation of a simple polygon can be transformed into a triangulation in linear time.

Fournier and Montuno's algorithm is based on an $O(n \log n)$ plane sweep which decomposes the polygon into trapezoids followed by the linear-time transformation to produce a triangulation. In another result, published in the same paper, they showed that given the horizontal visibility from vertices to edges, a trapezoidation can also be derived in linear time (see Fig. 3.3).

Fig. 3.3 Stage 2 of the Tarjan and van Wyk algorithm. Given all edge-vertex horizontal visibility information a trapezoidation can be computed in linear time.

Given the previous results, it is sufficient for triangulation to generate the horizontal edge-vertex visibility information. Tarjan and van Wyk present an $O(n \log \log n)$ algorithm which produces this information, relying on the previously described steps as the last two stages of the algorithm (See Fig. 3.4).

Fig. 3.4 Stage 1 of the Tarjan and van Wyk algorithm. Tarjan and van Wyk showed that horizontal edge-vertex visibility information can be produced in $O(n \log \log n)$ time.

While these last two stages of the algorithm are relatively simple, the section due to Tarjan and van Wyk presents considerable difficulty. Despite the use of this algorithm in a large number of papers [25, 30, 31, 61] it has not previously been implemented.

3.2.2 Stage 3: Trapezoidation - Triangulation

Fournier and Montuno [23] showed that a trapezoidation of a simple polygon P could be transformed in linear time into a triangulation of P . Their algorithm is based on partitioning P into unimonotone pieces (see definition below) by inserting diagonals of P . The unimonotone pieces can then be triangulated in linear time.

A polygon $P = (P_1, P_2, \dots, P_n)$ is defined to be *unimonotone* if for some starting vertex P_s , the sequence $(P_s, P_{s+1}, \dots, P_n, P_1, \dots, P_{s-1})$ is monotonically increasing or decreasing in y-coordinate. This is equivalent to saying that it is a polygon which is monotone in the y-direction, in which one of the monotone chains is a single edge. This is called the *unimonotone edge*. See Fig. 3.5.

Fig. 3.5 Two Unimonotone polygons

3.2.2.1 Triangulating Unimonotone Polygons

There exist several linear-time algorithms for triangulating monotone polygons [26, 67]. Since any unimonotone polygon is monotone, any of these techniques could be used. We describe a very simple method specific to unimonotone polygons, which is based on the Graham scan convex hull algorithm [28] and is also described by Fournier and Montuno [23].

The Graham scan is a linear time algorithm for finding the convex hull of a limited class of polygons (See Fig. 3.6). It works by repeatedly flattening “dents” in the polygon, where a *dent* (also called a *convex deficiency*) D of a polygon P is defined to be a chain of vertices whose start and end vertices lie on the convex hull of P , and are the only vertices in D which do so. The line joining the start and end points is defined to be the *lid* of D .

The polygons for which this algorithm correctly finds the convex hull are those belonging to the class weakly externally visible. A polygon P is said to be *weakly externally visible* if for each dent D with lid L , the interior of D is weakly visible from L . A region R is defined to be *weakly visible* from an edge L if for each point $r \in R$, there exists a point l on L such that r is visible from L . This does not imply that the entire region R is visible from any single point on L .

The Graham scan algorithm correctly removes any dent which is weakly visible from its lid, and in the process triangulates the dent (See Fig. 3.7). Any unimonotone polygon P can occur as a dent of some polygon P' , implying that the Graham scan algorithm can be adapted to triangulate unimonotone polygons.

Fig. 3.6 Several steps of the Graham scan algorithm. The leftmost figure shows the initial dent D and its lid L .

Fig. 3.7 Triangulation of a dent D from the polygon shown in Fig. 3.6

For any unimontone polygon P with n vertices, a polygon P' with $O(n)$ vertices can be constructed, containing P as a dent whose lid is the unimontone edge of P . The Graham scan algorithm will compute the convex hull of P' in $O(n)$ time, constructing a triangulation of P in the process. Thus a unimontone polygon with n vertices can be triangulated in $O(n)$ time. Detailed pseudocode for the algorithm is shown in Fig. 3.8.


```

PROCEDURE TriangulateUnimonotone

INPUT:   A Unimonotone Polygon  $P = (P_1, P_1, P_2, \dots, P_n)$ 

IF the degenerate chain is on the right THEN
     $V_{\text{start}} :=$  Vertex with maximum y-coordinate
ELSE
     $V_{\text{start}} :=$  Vertex with minimum y-coordinate
ENDIF
CurrentVertex :=  $V_{\text{start}}$  next.
WHILE the number of vertices in  $P \geq 3$  DO
BEGIN
     $V_i :=$  CurrentVertex
     $V_{i-1} :=$  CurrentVertex previous
     $V_{i+1} :=$  CurrentVertex next.
    IF the angle between  $V_{i-1}, V_i, V_{i+1}$  is convex THEN
        BEGIN
            Create a triangle from these three vertices.
            SavedVertex :=  $V_{i-1}$ 
            Delete  $V_i$  from  $P$ 
            IF  $V_{i-1} = V_{\text{start}}$  THEN
                CurrentVertex :=  $V_{\text{start}}$  next
            ELSE
                CurrentVertex := SavedVertex.
            END
        ELSE
            CurrentVertex := Current next.
        ENDIF
    ENDWHILE
END

```

Fig. 3.8 Pseudocode for the triangulation of unimonotone polygons

3.2.2.2 Decomposition into Unimonotone Pieces

We have described a procedure for triangulating unimonotone polygons in time linear in the number of vertices. It remains to show that an n -vertex polygon and its trapezoidation can be decomposed into $O(n)$ unimonotone pieces in $O(n)$ time. A trapezoid is considered, for this purpose, to be defined by the two vertices and two non-horizontal edges which belong to the original polygon. These trapezoids fall into two categories, those in which both vertices are connected by an edge of the polygon, and those in which they are not. (See Fig. 3.9)

Fig. 3.9 Cases for trapezoids. In trapezoid a , the defining vertices d_1 and d_2 share an edge. In trapezoid b they do not.

Using this characterization, it is possible to construct a simple algorithm for partitioning a trapezoidal polygon into unimonotone pieces. In any trapezoid of type (b), insert a diagonal between the defining vertices. This partitions the polygon P into sub-polygons.

Any vertex v (except the top and bottom) in any sub-polygon must have an edge leading up and an edge leading down. If v did not have an edge leading up, then v would be the bottom vertex in a trapezoid V with some other vertex v' . Since v' is above v and v has no edge leading up, the defining vertices of V would not share an edge and a diagonal would have been inserted between v and v' . Similarly, v must have an edge leading down.

Since any vertex v (except the uppermost and lowermost) in a sub-polygon P' has edges leading up and down, P' is monotone. Further, all vertices of P' lie on one monotone chain of P' . If this were not the case, then some pair of vertices v and v' on opposite chains would be adjacent in the sorted list of vertices of P' by y -coordinate, thus defining a trapezoid. But then v and v' , defining a trapezoid, would not share an edge and would have been joined by a diagonal, partitioning P' .

Thus, every sub-polygon P' of P is unimonotone.

3.2.2 The Fournier and Montuno Algorithm

It is worth considering at this point how the decomposition into trapezoids is performed by the Fournier and Montuno algorithm. As mentioned above, the algorithm is based on the plane sweep paradigm.

To construct a triangulation, it is sufficient as a representation of the trapezoids that the top defining vertex in each trapezoid store the lowermost vertex and that any vertex of the polygon be able to find the edges incident on it. (See Fig. 3.10)

Fig. 3.10 Representation of trapezoids. In each case, vertex a stores a pointer to vertex b.

During the plane sweep the algorithm maintains a balanced search tree of *active trapezoids*, where an active trapezoid is a vertex which has been encountered in the plane sweep but not yet assigned defined as those trapezoids which are intersected by the current sweep line. Within the search tree, the sort order is based on the left and right edges of the trapezoid. As long as the trapezoid is active, any vertex which lies on the sweep line will be contained in the trapezoid if and only if it lies between the left and right trapezoid edges.

We can divide vertices into three types based on their relation to the incident edges. If an edge E is incident on a vertex V and the other endpoint of E is above V then E is said to lie above V . Otherwise, E is said to lie below V . This gives rise to the three characterizations of vertices shown in Fig. 3.11.

Fig. 3.11 Characterizations of vertices used in the Fournier and Montuno algorithm. These are based on the relationship with the incident edges.

When we process a vertex V we remove any active trapezoids which contain edges incident on V from above. Similarly we add new active trapezoids for those edges incident on V from below. This can be broken down into cases as shown in Fig. 3.12 through 3.14. In each figure the balanced search tree of active trapezoids is referred to as S .

Fig. 3.12 Pseudocode for the Fournier and Montuno algorithm, processing a vertex of type 1 in the trapezoidation step.

```

IF the current vertex V is of type 2 THEN
BEGIN
  Find, in S, the active trapezoid R which contains V (if any);
  IF R exists THEN      {Case 1}
    Complete T using V;
    Remove T from S;
    insert two new active trapezoids using V and the edges
    incident on it, and also using the edges of T;
  ELSE                  {Case 2}
    insert into S a new trapezoid with V and its incident edges
END

```

Processing a vertex of type 2. In case 1 we remove the existing active trapezoid a and insert b and c. In case 2, we add the new trapezoid a.

Fig. 3.13 Pseudocode for the Fournier and Montuno algorithm, processing a vertex of type 2 in the trapezoidation step.

```

IF the current vertex V is of type 3 THEN
BEGIN
  Find, in S, the active trapezoid(s) T which contain V;
  /* there will be either one or two trapezoids */
  IF there is only one trapezoid THEN      {Case 1}
    complete and remove it;
  ELSE                                    {Case 2}
    complete and remove both trapezoids;
    insert a new trapezoid formed by merging the edges
    of the removed trapezoids which were not incident on v;
  END
END

```

Processing a vertex of type 3. In case 1, we complete and remove the active trapezoid a. In case 2, we complete and remove a and b. The edges of a and b which do not terminate at v form the new trapezoid c.

Fig. 3.14 Pseudocode for the Fournier and Montuno algorithm, processing a vertex of type 3 in the trapezoidation step.

This algorithm is conceptually simple and is not difficult to implement. The problem arises in attempting to create an algorithm which runs in time less than $O(n \log n)$. This time is inherent in both the plane sweep and the search tree mechanisms, making it unlikely that the order of this kind of algorithm can be improved. In order to bring the complexity of the algorithm below $O(n \log n)$ we will need to consider a subsidiary result from the same paper.

3.2.3 Stage 2: Edge-Vertex Visibility - Trapezoidation

Fournier and Montuno also describe a method of transforming all edge-vertex horizontal visibility into a trapezoidation in linear time (Another reduction was developed independently by Chazelle and Incerpi [10]). Given the previous linear-time algorithm for triangulation from a trapezoidation, this reduces the problem of triangulation to that of finding horizontal visibility information. Pseudocode for transforming edge-vertex horizontal visibility into a trapezoidation method is given in Fig. 3.15.

PROCEDURE visibility_to_trapezoids**INPUT:** P : a polygon;**OUTPUT:** a trapezoidation of P ;**IF** P is not a triangle or trapezoid **THEN** take the first vertex v of P with at least one visible edge; mark v done if it has only one visible edge; select a visible edge E ; split P by a horizontal line from v to the E , producing P' and P'' ; visibility_to_trapezoids(P'); visibility_to_trapezoids(P'');**ELSE** add trapezoidation information to the uppermost vertex of P ;**ENDIF**

Fig. 3.15 Pseudocode for Fournier and Montuno's algorithm to transform edge-vertex horizontal visibility information for a polygon P into a trapezoidation of P .

This description is deceptively simple, as its implementation turned out to be much less straightforward than one might have been expected from this formulation.

3.2.4 Stage 1: Polygon - Edge-Vertex Visibility

This operation is the heart of the Tarjan and van Wyk algorithm, and considerably more complex than those described in the preceding sections. We begin by adopting a naive approach to finding the visible pairs, temporarily disregarding the time complexity, and show how this can be adapted and improved to provide an efficient algorithm.

Given a simple polygon P , consider a single vertex V and a horizontal line L through it. Compute the set of all intersections with the polygon, storing the edge on which each intersection falls.

Fig. 3.16 Intersections of a polygon with a horizontal line L which passes through the vertex v .

In Fig. 3.16 the only intersections, and thus the only edges, which could be visible from V are those immediately to its left and right, in this case p and s . These vertices lie immediately before and after V when sorted by x -coordinate. It is not always the case that both these vertices will be visible, since we are interested only in lines of visibility which lie inside the polygon P . In Fig. 3.17 the intersection q is not visible through the interior of P , even though it is adjacent to V .

Fig. 3.17 A vertex with only one visible edge

The intersections which lie on either side of V can be found in time linear in their number. Determining if the line of visibility is inside the polygon will require some additional calculations when finding the intersections. The intersections of L with $P=(P_1, \dots, P_n)$ are found by finding the intersection of the line segment P_i, P_{i+1} with L . If P_i, P_{i+1} intersects L at X_i , then, given a point X on L , where no other edge of P intersects L between X and X_i , X is visible along L through the interior of P if and only if the angle P_i, P_{i+1}, X is a right turn (See Fig. 3.18). We define a point to be a *right turn* from a line segment if the point lies in the right of half-plane defined by the corresponding line.

Fig. 3.18 Determining visibility based on turn direction. In this example, X is visible to X_i because P_i, P_{i+1}, X is a right turn.

Thus, in constant time per intersection, it can be determined which side of the intersection is visible through the polygon interior. By storing this information, V can determine if an adjacent intersection is visible in $O(1)$ time. This allows the edge-visibility information for a single vertex to be found in $O(n)$ time overall, as the time to initially find intersections dominates the total cost. By applying this procedure to each vertex the all-edge-vertex visibility information can be found in $O(n^2)$.

3.2.4.1 Divide and Conquer

In the process of finding visibility information for one vertex a great deal of information is produced, some of which could be applied to reduce the amount of work required by other subproblems. The algorithm can be improved by making use of this information.

One piece of information which can be exploited is the partition of the polygon P into pieces by the horizontal line L (See Fig. 3.19). The horizontal visibility from a vertex V is being computed, and no intersection can be horizontally visible from V if it lies on or below a horizontal line which is below V . Similarly, intersections which lie above a horizontal line above V can be discarded. Thus, P can be partitioned into pieces which are separated

by a horizontal line L without losing any visibility information. These pieces are called *visibility regions*, and will be discussed in more detail below.

If, as each vertex is processed, the polygon is split into pieces, then subsequent processing steps will have fewer vertices to process and thus less work to do.

Fig. 3.19 A polygon split into three pieces along the horizontal line L

It remains to be considered how much additional work will be required to split the polygon at each stage and how much of a saving can be expected from performing this split. Since the intersections of P with L are already computed, in $O(n)$ additional time the polygon can be split into pieces, providing that the polygon is represented as a linked list. This represents only a constant additional cost per vertex.

One would expect that repeatedly reducing the size of the subproblems would improve the algorithm substantially. Although this does reduce the constant factor, in the worst case this algorithm is still $O(n^2)$. Consider for example, a unimonotone polygon (See Fig. 3.5). If we process the vertices of this polygon in order, we repeatedly split a polygon of size m into two pieces of size 1 and $m-1$ respectively. Since it requires $O(m)$ time to find all the intersections, the total time will be $O(n^2)$.

3.2.4.2 Lower Bounds

Before proceeding further in attempting to reduce the time complexity of the algorithm it will be helpful to establish lower bounds.

In a unimonotone polygon, each of the non-extremal vertices sees exactly one edge. Since there are $n-2$ such vertices, a lower bound of $\Omega(n)$ on the total number of vertex-edge visible pairs is apparent. Since a vertex can only see one edge in a direction, and horizontal visibility admits only two directions, an upper bound of $2n$ visible pairs can be established.

A problem with the divide and conquer approach, as described above, arises in splitting the polygon. In order to compute horizontal visibility from a vertex v the polygon (or a sub-polygon) must be intersected with a horizontal line through v . To compute visibility from all vertices, intersections must be computed with a horizontal line through each vertex. There exist polygons (as in Fig. 3.20) which, intersected with a horizontal lines through each vertex, will produce $O(n^2)$ intersections in total.

Fig. 3.20 A polygon with $O(n^2)$ intersections with the horizontal lines through its vertices.

These lower bounds provide some good news and some bad. Although the output size is $O(n)$, which does not rule out the possibility of a linear time algorithm, the approach of

computing all intersections cannot do better than $\Omega(n^2)$. If this is to be improved a new approach is required.

3.2.4.3 An $O(n \log n)$ algorithm

Given this $\Omega(n^2)$ bound for computing intersections, we would like to know if there exists a method for computing the visibility information in time less than $\Omega(n^2)$.

Preparata [50] describes a method of decomposing a simple polygon into monotone chains. The method is based on four plane sweeps along the y-axis, alternating between up and down sweeps. The first two sweeps transform the polygon into a regular planar subdivision by adding edges. A *regular planar subdivision* is defined as a subdivision in which every vertex has at least one incident edge leading up and one incident edge leading down (See Fig. 3.21). There are no more than $O(n)$ edges added in regularizing a polygon.

Fig. 3.21 Regular and irregular vertices, each with two incident edges.

Preparata goes on to define a decomposition into monotone chains, using two more plane sweeps, which can be used for planar point location in a two-stage binary search. This suggests two different methods of finding horizontal visibility information in $O(n \log n)$ time. The first is to use the planar point location technique to find visibility information from all vertices. Although the point location normally requires $O(\log^2 n)$ time, the first binary search can be avoided by processing each vertex in turn, where a

vertex knows the monotone chains of which it is a member. Thus, the horizontal visibility information can be found in $O(n \log n)$ time overall. The polygon can then be triangulated as in the Tarjan and van Wyk algorithm.

The regularization process also implicitly partitions the polygon into y -monotone pieces, which suggests a simpler approach to triangulation. If the regularization algorithm is modified so as not to add edges which lie outside the original polygon boundary, then the resulting planar straight line graph is a tiling of the original polygon with y -monotone pieces (See Fig. 3.22). Each y -monotone piece can be triangulated in time linear in the number of its vertices, giving an overall time bound of $O(n \log n)$.

Fig. 3.22 A simple polygon and its decomposition into monotone pieces, formed by adding regularizing edges.

Since this approach can find the horizontal visibility information in less than $\Omega(n^2)$ it is apparent that $\Omega(n^2)$ cannot form a lower bound. This implies that the work done in finding $O(n^2)$ intersections is in some way unnecessary. By refining the intersection finding technique we may be able to avoid this work and reduce the time bound.

does not represent an improvement over the previous approaches, but this technique has two other benefits. We will be able to retain information from the sorting process to use in splitting the polygon, and to modify the technique to avoid computing “unnecessary” intersections.

As the technique is quite complex, we begin by introducing some of the data structures and subsidiary ideas as background.

3.2.4.4.1 Amortized Computational Complexity

Both the Jordan sorting and triangulation algorithms will rely heavily on the amortized analysis of time complexity [65], so a brief discussion of this concept is in order.

Most algorithm analysis is done using worst-case analysis. Further, the worst case analysis of a complex operation is often defined merely as the composition of the worst case of each component piece. Thus, an operation which may need to be repeated $O(n)$ times, with each step requiring $O(\log n)$ time, is said to require $O(n \log n)$ time overall.

This analysis is sometimes unnecessarily pessimistic. It may be that the worst case occurs only in a small number of instances, and that this number can be bounded, often by a constant. Consider, as a simple example, a growable array data type. This behaves exactly as an array, but also supports an append operation. The representation of a growable array of size n uses an array of size $n+k$ s.t. $0 \leq k \leq n$ and maintains a count of the number of elements in the collection.

The append operation checks to see if there are spare elements in the underlying array ($k > 0$). If there are, then the element count is incremented, k is decremented and the new element added at the $n+1^{\text{st}}$ position. This requires no more than $O(1)$ time. The problem arises if $k=0$. In this case, a new array will be allocated with $k > 0$, and all the elements of the current array are copied into the new array.

If we assume a model of computation, possibly unrealistic, in which an array of size j can be allocated in $O(j)$ time, then the allocation of a new array at each growing step will require $O(n)$ work.

A simple worst-case analysis would say that since any individual append can cost $O(n)$ work, constructing a collection of n elements in this way requires $O(n^2)$ work. However, when the array is forced to grow, we can insist that the new size be proportional to the old size, for example, doubling the size of the array at each grow step. If this is done, then the array will need to grow less frequently and the total time required can be shown to be $O(n)$.

Suppose that the array is initially of size p . Then, as it grows we successively allocate arrays of size $p, 2p, 4p, 8p, \dots, 2^j p$ until $2^j p \geq n$, where n is the number of elements in the final array. The total work performed is:

$$\sum_{i=1}^j 2^i n = n \sum_{i=1}^j 2^i = n(2^{j+1} - 1)$$

We know that $2^j p \geq n$, so assume without loss of generality that they are strictly equal. Then $k = \log_2(\frac{n}{p})$ and the previous expression becomes:

$$n(2^{\log_2 \frac{n}{p}} - 1) = O(p \frac{n}{p}) = O(n)$$

It is important not to confuse amortized analysis with average-case analysis. The amortized analysis is still worst-case, but amortized over a sequence of operations, only a small number of which can be expensive. The average-case argument is that the expensive operations will normally occur infrequently and that it is unlikely (but still possible) that an expensive series of operations will occur.

Another example, particularly important for this case, applies to balanced search trees [34, 42]. In a B-tree (e.g. a 2-3 tree) split and merge operations at a node are expensive relative to searches at a node. Although the number of splits and merges is normally small,

it can be shown that a sequence of m insertions and deletions can cause $O(m \log n)$ splits and merges. Further, there are methods (similar to the finger trees we describe below) for performing fast searches which involve locality of reference. These methods do not, however, limit the number of merges and splits.

The worst case occurs when every node in the tree is full. A single insertion at this point will cause the entire tree to be grown by one level, requiring $O(\log n)$ work. Immediate deletion of the same element will cause the tree to collapse the added level, again requiring $O(\log n)$ work. The solution to this problem lies in allowing the tree to maintain more levels than is strictly necessary. With this modification, it is not possible for a collapse to immediately follow a grow, as the tree will tolerate the extra level (costing only a constant additional amount in a search) and avoid the split operations. In order to accomplish this it is only necessary to change the relation between the minimum and maximum number of children at a node slightly, and for our purposes it is sufficient to note that a 2-4 tree satisfies these conditions.

3.2.4.4.2 Homogeneous Finger Trees

Fundamental to the algorithm for sorting Jordan sequences are level-linked, red-black, balanced finger search trees, also called *homogeneous finger trees*. These are balanced search trees (a red-black tree [43, 64] is equivalent to a 2-4 tree) in which all siblings on a level are linked bi-directionally, and children are linked bi-directionally to their parents. All data is stored at the external nodes, and an internal node which is not the root stores the key of its smallest child. The root stores only pointers to its children. Homogeneous finger trees will allow searches, insertions, deletions and splitting of trees in better amortized time than is possible in normal balanced search trees.

Fig. 3.24 A homogeneous finger tree.

The *fingers* of such a tree are defined to be the leftmost and rightmost external nodes. (See Fig. 3.24). All access to the tree is through the fingers. The *left* and *right path* in the finger tree are defined as the nodes along the left and right hand sides of the tree.

In a normal balanced search tree (e.g. a 2-3 tree) the operations of searching, insertion and deletion are all $O(\log n)$. Searching dominates the cost, and all searches begin at the root and travel down through the tree to the appropriate leaf. The finger tree achieves its performance by starting searches at either the left or right finger. A query travels up the appropriate path in the tree until it can be sure that the element being searched for is part of its sub-tree. After this the query proceeds as for a normal search tree.

The advantage of this approach is that a query travels up the tree only as far as is necessary. The greatest height reached is proportional to the log of the distance from the finger at which the search started. Since the closest finger can be determined in constant

time before starting the search, the time required to search for an element d positions from either finger is $O(\log(\min\{d, n-d\}))$.

In an arbitrary pattern of searches this is of little advantage, but when we perform operations close to the fingers this will be very beneficial. It is also possible to generalize this type of tree into one which may have more fingers, which may be placed anywhere in the tree. If searches often take place in the same area, adding a finger in that area can save considerable time.

Since the underlying representation of a finger tree is a red-black tree, the time to re-balance is amortized $O(1)$ for any sequence of insertions and deletions. Thus one can construct a homogeneous tree in linear time from a sorted collection of its elements by inserting these elements in order at the right finger. Each insertion, including re-balancing, requires only $O(1)$ amortized time. Finally, homogeneous finger search trees support several efficient splitting operations. We will be particularly interested in the operation of *three-way splitting* which removes a subrange from a tree, and can be done in time $O(\log(\min\{d, n-d\}))$ for a subrange of size d in a sequence of size n . (See Fig. 3.25)

ABCDEFGHIJKLMN	
ABCLMN	DEFGHIJK

Fig 3.25 Three-way split. A sequence is split between values D and K to produce two sequences.

3.2.4.4.3 Applicability of Bracket Sequences

Rosenstiehl [53] has shown that any Jordan sequence of size $2n$ is equivalent to a matched pair of well-formed bracket sequences of length $2n$. A *well-formed bracket sequence* is defined as a sequence of opening and closing brackets in which the following conditions hold:

- 1) The number of opening brackets equals the number of closing brackets.
- 2) In reading the sequence from left to right, the number of opening brackets is at all times greater than or equal to the number of closing brackets.

$()(())$	$((()()))()$
$()(())()$	$()()()$
$((())())$	$)()()$
Well-Formed Sequences	Non-Well-Formed Sequences

Fig. 3.26 Examples of well-formed and non-well-formed bracket sequences.

There is a one-to-one correspondence between well-formed bracket sequences, forests, and the portion of a closed curve C which lies either above or below a horizontal line which intersects C . (See Fig. Fig. 3.27).

Fig. 3.27 A well-formed bracket sequence represented as a forest and as the upper portion of a curve intersected with a horizontal line.

The algorithm for sorting these sequences will construct two trees, corresponding to the portions of the curve above and below the cutting line, and this information will be of use in assembling subsections of the original polygon.

3.2.4.4.4 Sorting

The algorithm proceeds by incrementally processing the intersections $(x_0, x_1, \dots, x_m)$ in the order in which they occur around the polygon boundary. The algorithm maintains two trees: the *upper tree*, representing the section of the polygon above the line, and the *lower tree*, representing the section below the line. In addition, it stores a linked list of the points in sorted order. To maintain consistency in the algorithm and to allow the upper and lower trees to be represented as trees rather than forests, each has a single root which represents the arc $(-\infty, +\infty)$. The end points of this arc are imaginary intersections with x-coordinates at negative and positive infinity.

As a vertex x_i is processed, the arc (x_{i-1}, x_i) (which we will from time to time also refer to as a *node* in the upper or lower tree) is added to the appropriate tree. If i is odd, then the new arc is added to the upper tree. If i is even the arc is added to the lower tree. Assume for the moment that it is to be added to the upper tree, and further assume $x_{i-1} < x_i$. (Note that all comparisons are with respect to x-coordinates).

It can be shown that whenever a new intersection is processed it will always fall into one of four cases, as shown in the following diagram.

Case	Description	Illustration
A	New intersection cannot be reached without violating simplicity	
B	New arc falls at the end of the list of children of an existing arc	
C	The new arc falls in the middle of the list of children of an existing arc.	
D	The new arc falls in the middle of the list of children of an existing arc and encompasses some or all of those children.	

It can be shown that each of these cases, and their symmetric variations, can be handled. Case A is clearly an error. If this is encountered, the algorithm terminates, reporting a non-simple input. Case B is handled by adding the new arc as the last child of its parent arc. Case C can be handled by inserting the new arc into the middle of the list of children of an existing arc.

Case D is by far the most complex. In some cases it may be an error similar to case A, but in general it requires removing a sublist from the list of children, replacing it with the new arc, and adding the removed sublist as children of the new arc. Although we will not go into details of the Jordan sorting algorithm here, it is this splitting operation which is one of the major factors in the analysis of the algorithm.

3.2.4.4.5 Representation of Family Lists

In order for this algorithm to run in linear time, the lists of children must be represented appropriately. In particular, the representation of the children of an arc, known as a *family list*, is important. The operations required are adding children to the end of a list, and several variations of splitting operations.

These operations are supported by the homogeneous finger tree as described above. The algorithm can store in these trees the arcs (represented by pairs of vertices) which are the members of the family list. There are also bidirectional pointers between the parent and the first and last members of the family list. (See Fig. 3.28)

Fig. 3.28 A family list and its representation as a homogeneous finger tree.

Given the parent arc, any child at position d in a list of size s can be found in in $O(1 + \log(\min\{d, s-d\} + 1))$. Similarly, given the tree node containing a child c_2 any

other node in the tree c_2 (including the parent) can be reached in $O(1 + \log(\min\{d, s-d\} + 1))$ where d is the distance between c_1 and c_2 .

More importantly, given the tree node containing a child, another child can be inserted next to it in amortized $O(1)$ time and it is possible to remove a sublist of size d from a total list of size s in time $O(1 + \log(\min\{d, s-d\} + 1))$ using the three-way splitting operation on homogeneous finger trees described above.

3.2.4.4.6 Analysis of Jordan Sorting

Most of the operations described require only amortized $O(1)$ time and so permit an $O(n)$ overall running time for the algorithm. The splitting of the family lists, however, requires a more detailed analysis.

Whenever the Jordan sorting algorithm executes case D, a sublist A of some family list B is removed and replaced by a single pair P . P becomes the parent of A . This operation is called *split-and-insert*, and it dominates the processing time. Recall that insertion at a known position is amortized $O(1)$ in a finger tree. Removing a sublist of size d from a list of size s requires amortized time $O(1 + \log(\min\{d, s-d\} + 1))$.

To determine the overall time taken to execute all split-and-insert operations, a recurrence $T(p, s)$ is defined as the maximum time taken to execute p split-and-insert operations on a list initially of size s .

As the base case $T(0, x) = O(1)$ since no split-and-insert operations are performed.

If $p > 0$ a sublist of size d will be removed and a pair inserted in its place. This requires time $O(1 + \log(\min\{d, s-d\} + 1))$ and generates two new lists A, B of size $d+1$ and $s-d+1$, respectively. (The parent of a family list is counted as a member, so the newly inserted pair P is counted twice). There are now $p-1$ splitting operations left to be performed. Assume that i of these are allocated to the removed sublist A and $p-1-i$ of them to B .

This is sufficient to establish a recurrence, as given in Fig. 3.29, since the time required for splits on A is $T(i, d+1)$ and for B is $T(p-1-i, s-d+1)$. In attempting to determine the worst-case time complexity of the algorithm, a pair of values for i, d must be found to maximize the value of $T(p, s)$.

$$T(p, s) = \begin{cases} 0 & \text{if } p=0; \\ \max_{\substack{0 \leq i \leq p \\ 0 \leq d \leq s}} \left\{ T(i, d+1) + T(p-1-i, s-d+1) \right. \\ \quad \left. + O(1 + \log(\min\{d, s-d\} + 1)) \right\} & \text{if } p > 0; \end{cases}$$

Fig. 3.29 A recurrence for the time spent in split-and-insert operations.

The above recurrence can be reduced to a known recurrence in one variable (See Fig. 3.30).

$$T'(n) = \begin{cases} O(1) & \text{if } n=1; \\ \max_{1 \leq i \leq n-1} \left\{ T'(i) + T'(n-i) + O(\log \min\{i, n-i\}) \right\} & \text{if } n > 1; \end{cases}$$

Fig. 3.30 A simpler recurrence in one variable.

using the relation that $T(p, s) \leq T'(4p + s)$ and $T'(n)$ is $O(n)$ [44]. Thus the Jordan sorting algorithm takes $O(n)$ time overall.

3.2.4.4.7 Summary

Using the Jordan sorting algorithm the n intersections of a polygon with a horizontal line can be sorted in $O(n)$ time. By placing the horizontal line to intersect a vertex v of the original polygon, horizontal edge-vertex visibility information for v can be calculated in $O(n)$ time.

More significantly, the algorithm constructs representations of the upper and lower forests as it proceeds. In combination with a more complex representation of the polygon, using a variation of the finger search trees used in the sorting algorithm, this allows a divide and conquer approach to efficiently calculating the horizontal edge-vertex visibility information for all vertices.

3.2.4.5 Representation of the Polygon Boundary

So far we have not discussed the representation of the polygon, implicitly assuming that it is stored as an array of vertices. This turns out to be inadequate for several reasons.

First, finding intersections with any given line using this representation will require $O(n)$ time. Repeated for each of the $O(n)$ cutting lines, this would lead to an $O(n^2)$ running time for the algorithm. A structure which supports fast searching for intersections is needed.

Second, if a divide and conquer technique is to be applied, then a representation is needed which can support operations on the sub-polygons which result from repeatedly splitting the polygon along the horizontal cutting lines.

3.2.4.5.1 Boundary Segments and Visibility Regions

In order to represent the sub-polygons produced by a divide and conquer approach Tarjan and van Wyk define a *visibility region*. This represents pieces of the boundary of the original polygon P , joined by horizontal line segments where P has been split by horizontal lines (See Fig. 3.31).

Fig. 3.31 Example visibility regions.

A visibility region is defined by several *boundary segments*. Each boundary segment defines a chain consisting of zero or more adjacent edges of P and up to two partial edges of P . A *partial edge* E of P is defined to be a line segment collinear with an edge F of P , with length less than that of F . A partial edge is formed when an edge F of P is cut by a horizontal line at a point which is not an endpoint of F . It will be important to distinguish between vertices of P and *intersections*, end points of partial edges which are vertices of a boundary segment B but not of P . Fig. 3.32 shows some example boundary segments.

Fig. 3.32 Example boundary segments. Partial edges are shown dotted

In a visibility region, boundary segments are joined together by horizontal line segments whose y-coordinate is equal to the minimum or maximum y-coordinate of the visibility region. A visibility region always contains *left* and *right* boundary segments, which form

the left and right boundary of the sub-polygon, respectively. A visibility region also contains two lists (possibly empty) of boundary segments which form the top and bottom. A boundary segment b is a *top boundary segment* of a visibility region V if b is part of V and if both ends of the chain defined by b have y -coordinate equal to the maximum y -coordinate of V . Similarly, b is a *bottom boundary segment* of V if both ends of the chain which b defines have y -coordinate equal to the minimum y -coordinate of V . Fig. 3.34 shows the general form of a visibility region.

Fig. 3.34 The general form of a visibility region. Each curve represents a boundary segment. Any visibility region has both left and right side segments. The top and/or bottom lists of segments may be empty. All top boundary segments and the left and right boundary segment have the same maximum y -coordinate. All bottom boundary segments and the left and right boundary segment have the same minimum y -coordinate. Segments are joined by horizontal line segments at these minimum and maximum y -coordinates

The algorithm for computing horizontal edge-visibility begins by partitioning the boundary of P into two chains A , B . If v_1 is the vertex of minimum y -coordinate in P and v_2 is the vertex of maximum y -coordinate in P then the chain A contains the vertices of P from v_1 to v_2 inclusive and the chain B contains the vertices of P from v_2 to v_1 inclusive. A and B will become the left and right segments, respectively, of a visibility region V representing P .

3.2.4.5.2 Heterogeneous Finger Trees

Both boundary segments and lists of boundary segments (also called *sections*) will be represented using heterogeneous finger search trees. It is this two-level tree structure that reduces the cost of intersection operations with a horizontal line.

A *heterogeneous finger tree* is similar to a homogeneous finger tree, but differs in that it lacks sibling pointers and that pointers are not bidirectional. In particular, nodes on the left and right path maintain pointers to their parent and to their child which is not a path node, while interior nodes maintain pointers to both of their children and not to their parents. These trees are called heterogeneous (in contrast to the homogeneous trees described above) because different nodes contain different sets of pointers and because it is not possible to efficiently search from any starting element, but only from the left and right fingers.

The top and bottom sections of a visibility region will be represented as a heterogeneous finger tree containing boundary segments which are themselves represented as heterogeneous finger trees.

3.2.4.5.2.1 Representing boundary segments

Representing a boundary segment by a heterogeneous finger tree allows several different kinds of searches to be conducted. The primary organization of the tree is by position on the boundary of P . Boundary segments will be constructed by repeatedly appending vertices, so the ordering within the tree will be the same as the vertex ordering in P .

This allows vertices in a boundary segment to be efficiently located by position on the polygon boundary. Secondary keys which are stored at a node p include the number of external nodes reachable from p and the minimum and maximum y-coordinate of the

vertices stored in these external nodes. Fig. 3.35 shows the tree representation of a boundary segment.

Fig. 3.35 A portion of the boundary of P and its representation as a heterogeneous finger search tree. The coordinate system has $(0,0)$ in the upper left hand corner.

Using the information on how many leaves are reachable from any given node we can find a node with a given index in time $O(\log d)$ where d is the distance from the closest finger. The leftmost (or rightmost) vertex whose y -coordinate is less than (or greater than) a particular value can also be found. (Note that in this context leftmost and rightmost refer to the ordering of vertices around the boundary of P and not to x -coordinate ordering).

3.2.4.5.2.2 Representing lists of boundary segments

In using a heterogeneous finger search tree to represent a *section*, or list of boundary segments, slightly different information will be required than that used in representing a single boundary segment. In order to describe this information, we define a *section* of a boundary segment list as a number of consecutive boundary segments which form part of

the top or bottom of a visibility region. If the top/bottom list is represented by a tree, then the leaves reachable from an internal node constitute a section.

Each internal node stores the first and last edges of the section which it represents (in most cases these will be partial edges), the number of external nodes reachable, and the minimum and maximum y-coordinates of vertices in the section. Note that in determining minimum and maximum y-coordinates only those vertices which were present in the original polygon are considered, not the end-points of partial edges. For the boundary segment shown in Fig. 3.35 the minimum and maximum would be 4 and 12 respectively.

Fig. 3.36 shows a small example list of boundary segments and its representation as a heterogeneous finger tree.

Fig. 3.36 The top part of a visibility region and its representation as a heterogeneous finger search tree. The internal nodes are shown with the leftmost and rightmost partial edges of the section which they represent. In a tree of this size, each section contains only one boundary segment, but in general the left and right partial edges will come from different segments. The nodes also hold the number of reachable leaves (not shown), and minimum/maximum vertex coordinates (shown only for leaves).

The operations on the list of boundary segments are similar to those for a boundary segment. These include finding the k^{th} element for some integer k , and finding the leftmost (or rightmost) boundary segment containing a vertex with y -coordinate greater than (or less than) a given value.

Given that all vertices within this list of boundary segments lie strictly above (or strictly below) the horizontal line L and that the partial edges of every segment in the list cross L (this corresponds to a deep section, as described below), find the leftmost (or rightmost)

segment whose intersection with L has x coordinate less than (or greater than) a given value.

As with the boundary segments, all these operations can be done in $O(\log d)$ where the resulting segment s is d positions removed from the closest finger.

3.2.4.6 Splitting sub-polygons

The representation described above will be useful in finding intersections and in splitting the polygon. To realize this benefit, two additional concepts must be introduced: characterizations of boundary sections with respect to a line, and non-essential intersections.

3.2.4.6.1 Deep and Shallow Sections

Given a boundary segment B , there are three possible characterizations of B with respect to a horizontal line L . (Recall that a partial edge of a boundary segment does not terminate in a vertex of the original polygon P and that a boundary segment contains at most two partial edges).

- *Shallow*: No part of the boundary of B intersects L . All vertices and partial edges of B lie completely above or below L .
- *Deep*: The boundary of B intersects L at two partial edges. All vertices of B lie completely above or below L .
- *Mixed*: The boundary of B intersects L at least once in a non-partial edge. Vertices of B lie both above and below L .

Fig. 3.37 shows illustrates these relations, showing several boundary segments and their characterization with respect to different cutting lines.

Sections of boundary segments may be characterized in the same way. Thus, a *shallow section* is a group of adjacent boundary segments, all of which are shallow. *Deep sections* and *mixed sections* are similarly defined.

A line L does not intersect a section S which is shallow with respect to L , so a section which has been characterized as shallow need not be processed when computing intersections. A section which is deep with respect to L may have a number of intersections, but as described below, only a constant number of these need normally be processed.

Fig. 3.37 Deep, shallow and mixed sections. The boundary segments shown represent a section which is deep with respect to the line $L1$ and shallow with respect to the line $L3$. With respect to the line $L2$, segments a and b are deep. Segments c and d are mixed, since vertices of these segments lie both above and below $L2$.

3.2.4.6.2 Essential and Non-Essential Intersections

We referred earlier to a problem with the divide and conquer approach. Repeatedly cutting the polygon along horizontal lines through vertices may create as many as $O(n^2)$ visibility regions containing no vertices. An example is shown in Fig. 3.20 To avoid processing these degenerate regions the concept of essential and non-essential intersections is introduced.

Consider a section S which is deep with respect to a horizontal line L . All vertices in S lie on one side of L . When split along L , half of the sub-regions which are produced will

contain no vertices and produce no new visibility information. We prefer not to have to process these degenerate regions.

Fig. 2.38 Non-essential intersections. Although each boundary segment intersects the line L , only the intersections a and b are relevant to visibility or splitting. The others can be ignored.

The leftmost and rightmost intersection with the segment must be calculated, since they may affect horizontal visibility. The remaining intersections do not appear to affect the calculation. Suppose that, instead of splitting each deep segment into two new visibility regions we simply shorten the partial edges of the segment. The unneeded portions of the partial edge can be discarded, since they will not affect processing*.

A stronger statement is possible. If these unneeded sections do not affect processing then it is not even necessary to shorten the partial edges. If the interior deep segments are left unmodified, the excess length of the partial edges will also have no effect on processing. An illustration of this is given in Fig. 3.38.

This has two advantages. First, it avoids creating unnecessary visibility regions, limiting the total number to $O(n)$. Second, because the interior deep segments are not altered in any way, a deep section of any size can be processed in $O(1)$ time.

The leftmost and rightmost intersections of a line L with a deep section S are called *essential*, and all others are called *non-essential*.

* Visibility information from this region may become important if another segment intrudes into the area. This can be dealt with without violating the time complexity and is discussed under the heading of error correction.

3.2.4.6.3 Finding Intersections and Splitting Visibility Regions

The algorithm for processing a visibility region is summarized in Fig. 3.39.

PROCEDURE ProcessVisibilityRegion**INPUT:** V : a visibility region;**OUTPUT:** horizontal visibility information for V ;**BEGIN**

Choose a vertex $v \in V$ whose y-coordinate is not the minimum or maximum in V . If no such vertex exists then discard the region and return;

Construct the horizontal line L through v ;

Find the essential intersections of L with the boundary of V ;

Sort these intersections using the Jordan sorting algorithm described above;

Using the sorted list of intersections, compute horizontal visibility from v ;

Divide V into subregions along L ;

Recursively process each subregion of V ;

END

Fig. 3.39 Pseudocode for processing a visibility region to find edge-vertex horizontal visibility information.

This section describes in more detail how the essential intersections of V with L may be found and how V may be split efficiently into sub-regions.

Recall that, in the representation of a boundary segment each internal node t stores the number of external nodes reachable from t and the minimum/maximum y-coordinates of vertices stored in the nodes reachable from t . Since all nodes are reachable from either the left or the right finger, the minimum and maximum y-coordinates of vertices in a segment can be found immediately, characterizing it with respect to a line L .

We have already described how the intersections with a shallow or deep section can be found in constant time. To find the intersections with a mixed segment S , recall that a heterogeneous finger tree can be split in the same amortized time that is required to search. We proceed by repeatedly searching for the leftmost (or rightmost) intersection of S with L and splitting S at that point.

The time required to find the leftmost (or rightmost) intersection $s \in S$ is $O(\log d)$ where d is the distance of s from the closest finger. The time required to find all intersections of S with L depends on both the number of intersections and on the average distance between them. These quantities are inversely related. If there are k intersections in a segment of n vertices then the average distance between intersections is $\frac{n}{k}$. This is exploited to achieve an $O(n)$ overall time bound.

If the vertices are stored in a normal balanced search tree and portions which have already been considered are not split off, then $O(k \log n)$ time is required. If k is $O(n)$ then this is $O(n \log n)$.

The advantage of the heterogeneous finger tree is that the search time is proportional only to the distance from the closest finger rather than the total number of nodes. Any single intersection can be at a distance $O(n)$ from a finger but this cannot be true for more than a constant number. Total time in this case is $O(k \log \frac{n}{k})$. This also has a maximum when $k = O(n)$ but since

$$\log\left(\frac{n}{O(n)}\right) = \log(O(1)) = 1$$

the overall time complexity is $O(n)$.

A similar result holds for the top and bottom lists of boundary segments. The process begins by splitting the list into maximal deep and shallow sections. Because the minimum and maximum coordinates of vertices are stored at internal nodes, a deep or shallow section can be determined based on values at an internal node. These sections can be determined without having to examine every segment in the section.

To summarize the results for splitting visibility regions:

- Lists of boundary segments are split into maximal deep and shallow sections.
- Shallow sections never contain intersections.
- Deep sections have only two essential intersections, which can be found in $O(1)$ time.

- Only mixed sections need to be considered individually.
- A mixed section of size n with k intersections can be processed in time $O(k \log \frac{n}{k})$, which in the worst case is $O(n)$.

3.2.4.6.4 Re-assembling Visibility Regions

The boundary of a visibility region has been split at its essential intersections, and these intersections have been sorted to produce horizontal visibility information. It remains to re-assemble the split boundary segments to form new visibility regions.

This process is relatively simple. Each intersection C_i has associated with it the two boundary segments B, A which lie before and after it (and possibly a number of top/bottom sections). Each arc C has associated with it two intersections C_{left} and C_{right} .

Consider the tree T_{upper} (T_{lower} is treated in the same way). Each arc in T_{upper} has either odd or even depth. Each arc whose depth is odd defines part of the top boundary of a new visibility region, and each arc whose depth is even describes part of the bottom boundary of a new visibility region.

More specifically, a new visibility region V is formed for each arc R of odd depth, and the bottom boundary of this visibility region is formed by the children of R (who have even depth). Fig. 3.40 illustrates this process.

Fig. 3.40 A visibility region cut into 9 subregions (shaded) by a horizontal line. The numbers indicate depths of arcs in the upper and lower trees. Each arc with odd depth and its children define a new visibility region.

3.2.4.7 Remaining Issues

There are three issues which have not yet been dealt with: the possibility of errors in determining which vertices are non-essential, choosing the vertex at which a visibility region is split to ensure a balanced division of boundary segments, and the source of the $O(n \log \log n)$ time bound.

3.2.4.7.1 Error Correction

When a deep section D is encountered, only the leftmost and rightmost intersections a, b of D with the line L have been taken to be essential. This is equivalent to transforming the polygon so that the other segments of the deep section do not intersect L (See Fig. 3.41). Normally, this transformation does not affect the computation.

Fig. 3.41 Processing a deep section, such as the segment shown above left, by disregarding all intersections except a and b is equivalent to processing the shallow section on the right. Under most circumstances the transformation does not affect the result.

It is possible for another segment T , which is not part of D , to intersect L between a and b . The assumption that no vertices lie in this area will become false and the Jordan sorting algorithm will find the visibility region to be non-simple. An example is shown in Fig. 3.42.

Fig. 3.42 Error correction. The deep section between a and b has already been processed, disregarding all "non-essential" intersections. The mixed segment T intrudes and will cause the algorithm to fail. The problem is corrected by computing the additional essential intersections c and d .

To correct this problem, two additional intersections must be computed. The deep section can be split into two new sections, new essential intersections c , d calculated, and the upper and lower forests retroactively updated. This can be done without violating the required time complexity. Since each new pair of essential intersections is created if and

only if at least one vertex intrudes into D , this does not cause the total number of intersections to exceed $O(n)$.

3.2.4.7.2 Balanced Divide and Conquer

There are $O(n)$ distinct boundary segments created in the course of computing horizontal edge-vertex visibility. Each boundary segment appears in only one visibility region at a time. It is, however, possible for a single top or bottom segment to be undisturbed through many iterations of the splitting process. For example, when a visibility region V is split into new regions $\{V_1, V_2, V_3, \dots\}$ any shallow segments in V will not be split and will be present in some region $V' \in \{V_1, V_2, V_3, \dots\}$. It is possible for there to be $O(n^2)$ occurrences of boundary segments in visibility regions even though the total number of distinct boundary segments is only $O(n)$.

A boundary segment B is processed exactly once, since a segment is split into new segments whenever it is processed. It does, however, contribute to the cost of searching in the trees which represent lists of segments. Thus, $O(n^2)$ different occurrences would imply an $O(n \log n)$ time bound rather than $O(n^2)$. To reduce the overall time complexity of the algorithm to $O(n \log \log n)$ the number of different occurrences must be reduced to $O(n \log n)$.

This can be done by selecting more carefully the vertex v at which a visibility region V is split. The goal is to bound the number of boundary segments that lie in an individual visibility region after splitting. Recall that the size of a boundary segment or list of boundary segments stored as a heterogeneous finger tree can be determined in $O(1)$ time.

If V has less than three boundary segments in the top and bottom list of boundary segments, select any vertex $v \in V$ which is a vertex of the original polygon P . In such a

region the total number of segments and thus the number of segments in any sub-region is bounded by a constant.

If V has more than three boundary segments in either the top or bottom list of boundary segments, select a list L to be the larger of the two. If L is the top list of boundary segments in V then let v be the vertex of P in L with minimum y -coordinate, otherwise let v be the vertex of P in L with maximum y -coordinate.

Tarjan and van Wyk show that this approach guarantees that no more than $\frac{15}{16}$ of the boundary segments from the original region will occur in any one subregion. This gives a bound of $O(n \log n)$ occurrences of boundary segments overall. It is worth noting that the log in this expression is base $\frac{15}{16}$, which introduces a constant factor of approximately 10 compared to the more common log base 2.

3.2.4.7.3 Overall Time Complexity

As described, the total number of essential intersections and of visibility regions produced in processing a polygon with n vertices is bounded by $O(n)$, and most operations on visibility regions of size m take $O(m)$ time. The source of the $O(n \log \log n)$ time bound for the algorithm has not yet been described. This bound arises from the $O(n \log n)$ occurrences of boundary segments described in the previous section, manifested in the time required to perform repeated splits and merges on the heterogeneous finger trees which represent the top and bottom lists of boundary segments in a visibility region.

Merge operations arise because, in finding intersections, the top and/or bottom boundary segment list is split into maximal deep and shallow sections. This allows these sections to be processed in constant time, but these splits may have to be undone during the assembly phase. Consider, for example, a top list D in some visibility region V such the segments of D alternate between being deep and shallow with respect to a line L . D will be split into

deep and shallow sections of size 1 for processing. When the pieces of V are assembled to form new visibility regions, all the segments of D will fall in the same region V' . The different segments cannot be left as separate sections because their characterization is dependent on L . To form the top section of V' , these sections must be merged back together.

The analysis which shows a linear time bound for finger tree operations applies only to sequences of split-and-insert operations. When merge operations are permitted, this bound does not apply. Tarjan and van Wyk show that the time, including merge operations, is bounded by $O(n \log \log n)$. This dominates the processing time for the algorithm, so the overall time complexity is also $O(n \log \log n)$.

3.3 IMPLEMENTATION OF THE ALGORITHMS

Both the Tarjan/van Wyk and the Fournier/Montuno algorithms have been implemented using the computational geometry workbench. This section discusses the goals of the implementation, their realization, and issues which arose during algorithm implementation.

3.3.1 Implementation Goals

Our goals in implementing the Tarjan/van Wyk triangulation algorithm relate both to algorithmic issues and to the computational geometry workbench.

Triangulation of a simple polygon is a fundamental operation in computational geometry, and is used as part of many other algorithms [30, 39]. It is important to have such a basic operation available in the workbench.

The algorithm is extremely complex and difficult to implement. Although it is asymptotically good, the constant factors involved appear to be large. An implementation of the algorithm allows us to estimate constant factors. The Fournier and Montuno algorithm is similar to that of Tarjan and van Wyk but much simpler and takes $O(n \log n)$ time. By implementing both of these algorithms in the same environment we introduce the possibility of direct comparisons, in terms.

The linear-time sorting of Jordan sequences is a complex algorithm in its own right. We can empirically test this algorithm separately to compare its performance against simpler $O(n \log n)$ sorts.

Finally, implementation of this algorithm provides an opportunity to test the dynamic optimality conjecture of Sleator and Tarjan for splay trees [59]. By substituting splay trees for finger trees in all or part of the Tarjan and van Wyk algorithm we can examine the validity of this conjecture empirically.

A *splay tree* is a self-adjusting form of a non-balanced binary search tree. As any element X is accessed it is moved to the root of the tree. All elements which lie along the path from X to the previous root are moved closer to the root. Sleator and Tarjan conjecture but do not prove that splay trees perform as well as any balanced tree structure over a sufficiently long sequence of accesses. This implies that splay trees could be used in place of finger trees without affecting the algorithm complexity.

In terms of the computational geometry workbench there are several reasons to consider implementing the Tarjan and van Wyk algorithm. This is one of the most complex algorithms known, and an implementation serves as a demonstration of the utility of the tools available in the workbench. It also gives insight into the difficulties and the implementation issues involved in complex geometric algorithms, suggesting areas in which these tools need to be further refined. Finally, it allows us to compare algorithms from the point of view of implementability as well as running time.

Results of the direct comparison studies are detailed in Chapter 5. The remainder of this chapter outlines some of the issues which arose during the implementation of these triangulation algorithms and others. Some are specific to these algorithms, while others are more general in nature.

3.3.2 Implementation Issues

Implementation issues can be divided into those directly concerning the algorithm and those concerning its translation into the workbench. Algorithmic issues are those which relate directly to the algorithm, including ambiguities in description, non-operational definitions and unspecified assumptions. Translation issues relate to the mapping of the

algorithm into an executable program and are primarily issues of software engineering and abstraction.

3.3.2.1 Algorithmic issues

Algorithm design and description normally proceeds at a very abstract level. Papers are published for an audience which already possesses a background in the field, and the emphasis is on economically communicating algorithm concepts and on complexity analysis.

This style is appropriate and necessary in communicating amongst the algorithms community. Abstraction of the important concepts is vital in describing complex algorithms. A fully detailed exposition at all levels would occupy many pages and often obscure the significant concepts in minutiae. Complexity analysis can be extremely complicated, and may require a great deal of space for even an overview of the analysis.

This style, however, can cause considerable difficulty when attempting to implement algorithms described in a paper. Details which are considered insignificant may be omitted, and minor variations from the techniques cited in a reference may be left to the reader.

The complexity analysis of an algorithm is only tangentially relevant to its implementation. While the analysis may dictate a particular representation or method of performing an operation, this fact is not always made clear.

Operations may be phrased in a manner convenient for description without regard to implementability. Consider the example of searching in a finger tree, as described by Tarjan and van Wyk: "In an n -item heterogeneous search tree we can search for an item ... searching up along the left and right paths concurrently until we find a subtree or two subtrees guaranteed to contain the desired item, and then searching down in this subtree..."

Conducting these concurrent searches in an implementation would be substantially more complex than a normal tree searching operation. Although the given description is correct, it is needlessly complex. In the definition of a finger search tree given by Tarjan and van Wyk, the right child of the root maintains the smallest key k accessible from it, which is always the smallest key in the right subtree. If the search key is less than k then the search should begin from the left finger, otherwise it should begin from the right finger. Thus by examination of the right child of the root, the appropriate finger can be determined in $O(1)$ time.

The standard model of computation for geometric algorithms includes real arithmetic. This is an invalid assumption, but the complications introduced by finite precision arithmetic are so onerous that this is a valuable simplifying assumption. It is important to bear in mind, however, that implementations do not have access to real arithmetic, and operations which rely on this assumption to excess should be avoided.

We describe in more detail some specific examples.

3.3.2.1.1 Tangency

Many algorithms assume that no two vertices have the same x or y coordinate, sometimes with the caveat that this can always be achieved by perturbing the coordinates or by rotating the coordinate system appropriately. Forrest [21] points out that these techniques are not normally acceptable in an implementation, and the implementor of an algorithm must often develop the treatment of these special cases independently.

For instance, in the Jordan sorting portion of the Tarjan and van Wyk algorithm it is assumed that no vertex lies directly on the line of intersection. This is clearly false, since the line is chosen so as to intersect a vertex. Only one paragraph is devoted to explaining how the algorithm can be modified to accommodate such vertices.

This is a common special case in geometric algorithms, and failure to consider it can lead to difficulties in implementing even simple operations.

3.3.2.1.2 Symmetry

The Jordan sorting portion of the Tarjan and van Wyk algorithm was difficult to implement, both because of the number of cases and because of the difficulty of resolving symmetry across several cases. The algorithm is described in terms of the assumptions that $x_{i-1} < x_i$ and that the vertex under consideration is to be inserted into T_{upper} .

Description of the procedure makes use of the terms *before*, *after*, *left*, *right*, *less than*, and *greater than*. It is not clear from the description which of these terms are to be inverted under the different symmetric conditions.

With some interpretation it is possible to resolve most of these ambiguities, but in one case an extremely subtle error in the implementation was caused by misinterpreting a symmetric condition.

3.3.2.1.3 Finger trees

As a data structure, finger trees are appealing. Many useful operations are supported with good amortized complexity. As a result, they are used, with minor variations, in a number of algorithms [30, 62].

The fact that many variations exist causes, in itself, difficulties in implementing a general form. More important, finger trees are extremely difficult to implement. Tarjan and van Wyk say "Finger search trees are sufficiently complicated that one would probably not want to use them in an actual implementation" [66].

This is correct.

The complications of finger trees arise not just from their structure, which in itself induces numerous special cases and exceptions, but from the operations which are to be

1

2

performed on them. The most difficult of these is the three-way split operation used in Jordan sorting. (See Jordan sorting).

One case of this operation occurs when the two nodes between which the tree is to be split have ancestors who are siblings. In this case two subtrees A, B must be removed from the tree C , split at the appropriate locations, some of the pieces must be merged back into C , and C must be re-balanced (See Fig. 3.43). The difficulty arises in merging the split portions of A and B back into C and re-balancing. The tree re-balancing operations assume that at most one subtree is removed from the tree at a time and that this removal is the one being re-balanced. Simultaneous removal of two subtrees causes these operations to fail.

The only description given for this case in the Jordan sorting paper is “split the subtree with root e at x , split the subtree with root f at y , and combine the pieces in the appropriate way” [33].

Fig. 3.43 The three-way split operation for homogeneous finger trees. The original tree *C* is to be split at *x* and *y*. At some level, *x*, *y* have ancestors *e*, *f* which are siblings and root the subtrees *A*, *B*. These subtrees are removed from *C* and split at *x* and *y* to form the new subtrees *a*, *b*, *c*, and *d*. Of these trees, *b*, *c* are merged to form a new tree and *a*, *d* are re-inserted into *C* and *C* is re-balanced.

3.3.2.1.4 Representation of output

Many algorithms assume a “standard” representation for their input and output. Even if an unusual data structure (such as the heterogeneous finger trees of the Tarjan and van Wyk algorithm) is to be used, a transformation to and/or from a standard representation is described or assumed.

The output of a triangulation is a planar straight line graph, commonly represented as a doubly connected edge list (DCEL) [51]. An important characteristic of this representation is that the edges incident on a vertex V are sorted by angle around V .

The sorting of incident edges is often assumed by algorithms in visibility and motion planning which use triangulation as a preprocessing step [30]. Since there can be $O(n)$ edges incident on a single vertex in a triangulation (See Fig. 3.44) this would appear to require $O(n \log n)$ time to produce, greater than the time required by triangulation using the Tarjan and van Wyk algorithm.

Fig. 3.44 A triangulation with $O(n)$ edges incident on vertex A .

The transformation from trapezoidation to triangulation described by Fournier and Montuno does not address this concern. While Fournier and Montuno claim that the transformation takes $O(n)$ time for an n -vertex polygon, if this were shown to require $O(n \log n)$ it still would not affect the overall time bound for their algorithm.

The Tarjan and van Wyk algorithm also does not address this concern. This is surprising, since, at first glance it appears to place a lower bound on triangulation, or at least on triangulation to the standard representation.

In fact, it is possible to add the diagonals which form the triangulation in sorted order while avoiding the $O(n \log n)$ cost. Any vertex v participates in at most three trapezoids,

so at most three diagonals of trapezoids will be incident on v . The remaining edges are added by triangulation of a unimonotone polygon (a vertex also participates in at most three unimonotone polygons). The nature of the algorithm for triangulating unimonotone polygons is such that new edges are always added at one or the other end of a sorted list (See Fig. 3.45).

Fig. 3.45 A unimonotone polygon and its triangulation. The additional edges are numbered in the order in which they are added. Consider vertex A, whose initial list of edges includes a and b. In triangulating we add edges 2, 3, 4, and 6. If the sorted list of edges is considered to begin with the first edge counterclockwise from the polygon interior then a new edge is always added at the beginning or end of this list.

These edges can thus be added in sorted order, and the three (or fewer) sorted lists merged. This produces a planar straight line graph with edges incident on any vertex v sorted by angle around v while requiring only $O(n)$ time.

It is unclear why this concern was not addressed in either the Fournier and Montuno or the Tarjan and van Wyk papers. It may have been considered as too trivial for inclusion.

3.3.2.1.4 Trapezoidation

Significant problems arose from the Fournier and Montuno algorithm for transforming horizontal edge-vertex visibility information into a trapezoidation. This algorithm is presented by Fournier and Montuno, and used by Tarjan and van Wyk as one stage of their triangulation algorithm.

The algorithm is conceptually very simple, and is discussed in only 23 lines in the original paper, including pseudocode [23]. In implementation, difficulties arose with several aspects of this algorithm. Some of these difficulties were originally encountered by Joe May while attempting to implement this algorithm as part of a course project.

The algorithm repeatedly splits a polygon P into two pieces A, B along a horizontal line from a vertex v to an edge e which is horizontally visible from v . For the algorithm to run in $O(n)$ time overall this operation must be performed in $O(1)$ time, which is not possible using the standard representation of a polygon as an array of vertices.

Instead, P may be represented using a circular doubly-linked list structure. In this representation, given two vertices v_1, v_2 , split the first sub-polygon A from the remainder (using a constant number of pointer operations), perform the first recursive step on A , re-assemble P , split the second sub-polygon B , and perform the second recursive step on B . The time to split and re-assemble the list structure at each step is only $O(1)$.

It is also important in this algorithm, although left unstated, that the recursive splitting be done in a depth-first rather than a breadth-first manner. It would not be possible, for example, to evaluate the recursive decomposition in parallel. This is true not only because of the sequential nature of the splitting operations described above, but because the sharing of references to edges requires direct modification of edges, and thus sequential processing. Consider the unimonotone polygon shown in Fig. 3.46.

Fig. 3.46 A difficult case for the Fournier and Montuno algorithm. The unimodotone edge A will be referred to be $O(n)$ vertices. Whenever the polygon is split, A must be modified accordingly.

Several other issues related to determining when an object was or was not part of the current sub-polygon being processed. This problem arose in the context of vertices which belonged to edges when only part of the edge was present in the current sub-polygon, in the context of trapezoids which may or may not be part of the current sub-polygon, and in the context of horizontal visibility edges. We discuss here only the horizontal visibility edges, as illustrated by Fig. 3.47.

Fig. 3.47 Another difficult case. Splitting at v can cause problems for the Fournier and Montuno algorithm.

We wish to split this polygon at vertex v . Since v has more than one visible edge it cannot yet be ruled out as a candidate for future splits. Assume that we select edge b to split along, producing the sub-polygons A and B . The algorithm is then applied recursively to each sub-polygon, selecting another vertex with unprocessed visible edges. Vertex v appears in both sub-polygons, so we may again select it and begin to process edge a . Is this permissible?

Processing a may or may not be an error. It depends on which of the two sub-polygons is processed first. If we first consider A , then a is a valid visible edge, and not processing it will lead to an incorrect result. If we first consider B , line a represents a line of visibility to an edge which is not in the current sub-polygon, and processing it would lead to an incorrect result.

This can be resolved by considering the conditions under which this situation arises. The vertex v must be a local maxima or minima which has two visible edges. Further, when we split the polygon along one visible edge, we choose to process first the sub-polygon which does NOT contain the second visible edge.

Suppose that v is a local minima. In this case, if we split along the leftmost visible edge and process the leftmost sub-polygon first then we avoid the problem. Conversely, if v is a local maxima then by choosing the rightmost visible edge and the rightmost polygon we can avoid the problem.

3.3.2.1.5 Heterogeneous finger trees

The Tarjan and van Wyk algorithm makes use of two variations on finger trees, heterogeneous and homogeneous. The major difference between the two types are, in principle, that in a homogeneous finger tree, all pointers are bi-directional and pointers to siblings are maintained. In practice, the heterogeneous finger trees are also used to support a number of different search operations from the homogeneous trees.

The sibling pointers and the bi-directional pointers of the homogeneous finger tree do require only $O(1)$ additional time to update than the pointers of a heterogeneous tree. In fact, all operations which are performed using heterogeneous trees could be performed using a variation of homogeneous trees. This would also make many operations simpler by allowing the additional pointers to be exploited.

The Tarjan and van Wyk algorithm is implemented in the workbench using both heterogeneous and homogeneous finger trees, even though all operations could have been done with homogeneous trees. Implementing all operations using homogeneous trees would have required implementing at least two varieties of homogeneous trees, almost as much effort as implementing heterogeneous trees, and it was considered safer to stay as close to the algorithm description as possible.

3.3.2.2 Translation Issues

The difficulties encountered in translating an algorithm into code depend on the language of implementation, the approach adopted, and on early design decisions.

3.3.2.2.1 Iterative Development

An important part of the object-oriented paradigm is rapid prototyping. A version which in some manner “works” is created as early in the process as possible and successively refined. If necessary, entire sections can be discarded and rewritten with the benefit of the understanding which has been gained from the previous versions.

This approach allows subsections to be tested as they are developed, helps identify difficulties early in the process, encourages modularity of code (since sections may be replaced at any time) and aids in understanding the problem. In our case, the problem (triangulation) had already been solved, and the difficulty lay in understanding the solution.

This approach also has disadvantages. One substantial disadvantage that became apparent during the implementation of the Tarjan and van Wyk algorithm was that the

different semantics of different data structures can be misleading, as described under the heading of destructive operations.

3.3.2.2.2 Destructive operations

The initial approach to implementation of the algorithm was to ignore the finger trees and to proceed with an implementation using array-based data types. The differences between arrays and finger trees were encapsulated by the algorithm data types, so the algorithm could easily be converted to use the correct data structures when they became available.

This proved to be more difficult in practice. The semantics of arrays differ in important ways from those of tree-based data structures, particularly in the area of destructive operations. While it is possible to encapsulate these differences, the initial implementation did not do so sufficiently, and some design decisions were made that proved to be inappropriate when implemented using finger trees.

This approach was not entirely without benefit, however. Data structures with destructive operations can be extremely difficult to debug. An error in the algorithm implemented using non-destructive operations produces a walkback which contains a useful history of the calculation to date. Destructive operations destroy this history and make it necessary to follow the progress of the algorithm directly.

It also proved extremely useful to have more than one implementation of the algorithm. Once an array-based version was working substantially correctly, errors in a similar tree-based version could be followed by tracing both algorithms and watching for divergences.

To summarize, this substitution of simple data types has value, but care must be taken to encapsulate all of the different properties between the data structures used.

3.3.2.2.3 The object-oriented model

Some issues arose relating to the object-oriented model in which the workbench is implemented. One such issue is how to organize the classes which are required for a particular algorithm.

Consider, for example, the linear-time algorithm for sorting Jordan sequences. An obvious rule is that an algorithm should be implemented in the class of the geometric object on which it operates. This is certainly appropriate for algorithms operating on geometric data structures such as polygons, point sets, etc.

The Jordan sorting algorithm operates on the list of intersections of a polygon with a horizontal line. It seems inappropriate to create a geometric object corresponding to this set of intersections, but there are no obvious alternatives for the algorithm to operate on.

In the workbench, this algorithm is implemented using a class called **JordanSorter**, a procedural class which takes a list of points as input and sorts them. This might be considered bad object-oriented style, but the class does clearly indicate its function and groups together a substantial body of code. It is worth considering other styles of implementation for future algorithms and comparing the results in terms of re-usability and encapsulation.

Another important issue that arose is the difficulty that can arise in maintaining object identity. Algorithms often exploit object identity (usually expressed as maintaining a pointer to a structure) and make use of destructive operations. It is important to maintain object identity where necessary while not permitting destructive operations to unintentionally alter other structures.

One area where this issue becomes important is in cell-based data structures (i.e. trees and lists) and the distinction between the contained data and the container. For instance, in the trapezoidation portion of the Tarjan and van Wyk algorithm the vertices of a polygon

are stored in a circular doubly-linked list. Numerous structures, including the trapezoids, maintain pointers to vertices and must be able to find the next and previous vertices. How is this to be implemented?

There are three basic organizations that are immediately apparent, as shown in Fig. 3.48:

Fig. 3.48 Three ways of organizing vertices in a circular doubly-linked list. Option 1: (upper left) Vertices store a reference to the list cell which contains them. External references are to vertices. Option 2: (upper right) Vertices implement or inherit list cell behaviour. Option 3: (lower left) Vertices are stored in a list cell but do not store a reference to it. All external references are to the list cell, which delegates vertex operations to the vertex.

Each of these basic methods might be applied in different ways. If a vertex is to have list cell behaviour, as in case 2 of Fig. 3.48, it might get this behaviour via direct implementation or via multiple inheritance. In case 3 of Fig. 3.48, the system for delegating messages to the vertex might vary. A list cell might implement all vertex operations by passing them to the vertex. It might also re-implement `doesNotUnderstand`: so as to pass any misunderstood messages to the vertex.

None of the above implementations provides a really satisfactory solution. In the workbench, option 3 has been implemented for circular lists, and option 1 has been implemented for tree nodes.

If references to list cells and to vertices are distinct (i.e. no external reference expects to see both) then the most appropriate interface would be to provide two different views of the object (with whichever representation option is preferred): a vertex view and a list view.

3.4 CHAPTER SUMMARY

In this chapter we have described the Tarjan and van Wyk and Fournier and Montuno algorithm for triangulation of a simple polygon in some detail. We have also discussed issues that arose during the implementation of these algorithms.

4 GENERATING JORDAN SEQUENCES AND POLYGONS

4.1 RANDOM JORDAN SEQUENCES

4.1.1 Motivation

The algorithm for sorting Jordan sequences in linear time is a crucial component of the Tarjan and van Wyk algorithm and is applicable to other problems. It therefore merits our attention for the next chapters, which focus on empirical testing and algorithms for generation of Jordan sequences. More specifically, in Chapter 5 we give timing results for the Jordan sorting algorithm as described by Tarjan and van Wyk, conventional sorting techniques, and an alternate implementation using splay trees. This alternate implementation is motivated by the dynamic optimality conjecture of Sleator and Tarjan [59]. Tarjan and van Wyk conjecture that "...the $O(n \log \log n)$ time bound is still valid if splay trees ... are used in place of finger search trees. The use of splay trees might lead to a practical implementation of our algorithm, although this must be verified by experiment" [66]. We provide this experiment.

To obtain empirical timing results for the Jordan sorting algorithm it is necessary to create Jordan sequences. The most direct approach is to create a polygon with a large number of edges, find its intersections with some horizontal line and test the Jordan sorting algorithm on this set of intersections. This has two serious drawbacks. First, it appears difficult to estimate the number of intersections that will be created. Second, the number of intersections generated for "random" polygons is much smaller than the total number of vertices in the polygon.

Restricted classes of polygons may admit a large number of intersections. For example, a star-shaped polygon with n vertices can be intersected with a horizontal line so as to produce a number of intersections which is a significant fraction of n (See Fig. 4.1). However, these intersections occur in sorted order along the line of intersection. This can be exploited by the sorting algorithm and does not provide a very interesting test case.

Fig. 4.1 Intersections of a star-shaped polygon with a horizontal line.

In general, the intersections of a line with a “random” polygon will not be pre-sorted but the number will tend to be a small fraction of the total number of vertices (See Fig. 4.2). For example, a polygon of 8000 vertices may have only 100 intersections with a given line. Generating large Jordan sequences in this way would be very expensive.

Fig. 4.2 A “random” polygon of 120 vertices. No horizontal line intersects at more than 20 places.

This motivates our study on generating arbitrary Jordan sequences directly. Note that, given a sequence S , it can be determined if S is a Jordan sequence in $O(n)$ time by several methods. Given an implementation of the Jordan sorting algorithm [66] as described above, the algorithm may be run on S . Lacking such an implementation, a simpler method is described by Rosenstiehl [53].

4.1.2 An $O(n \log n)$ algorithm for Jordan sequence generation

4.1.2.1 Algorithm Concepts

We present an algorithm which generates an arbitrary Jordan sequence of length $2n$ in $O(n \log n)$ time and show that it can generate any Jordan sequence, although uniformity of the distribution of sequences so produced is not guaranteed. Determining the number of Jordan sequences of a given size is an open problem [53], which makes assessment of the uniformity of any generation technique difficult.

For the purpose of solving the generation problem we represent a Jordan sequence as the set of $2n$ intersection points of a closed curve with a horizontal line L (See Fig. 4.3). Each intersection point is the endpoint of two *arcs*, portions of the original closed curve.

The arcs are grouped into two k -ary forests, T_{upper} and T_{lower} , representing the arcs above and below the horizontal cutting line respectively. The forests are converted into rooted trees by adding an arc joining imaginary intersections with x -coordinates of $-\infty$ and $+\infty$. We define the *twin tree* $\text{twin}(T_{upper})$ to be T_{lower} and $\text{twin}(T_{lower})$ to be T_{upper} .

Fig. 4.3 A Jordan sequence defined by a simple closed curve and a line L .

Any node m in a tree $T \in \{T_{upper}, T_{lower}\}$ represents a single arc which connects two intersection points. The node m is linked to a node $p \in \text{twin}(T)$ if m, p share one or more endpoints. Each node maintains a list of its children as a balanced search tree with parent pointers, sorted by x -coordinate. A node also maintains pointers to the node(s) with which it is linked.

The algorithm for generating Jordan sequences is described inductively. Given a Jordan sequence of size $2n-2$ we show how to incorporate, in $O(\log n)$ time, a new pair of intersections so that the result is a Jordan sequence of size $2n$.

We define a pair of intersections $m = (m_{left}, m_{right})$, $m_{left} < m_{right}$ to be *adjacent* in the Jordan sequence J if for any intersection $j \in J$ for which $j \neq m_{left}, j \neq m_{right}$ it holds that $j < m_{left}$ or $j > m_{right}$ (comparison based on x -coordinates).

Two points p, q are defined to be *mutually visible* in the Jordan sequence J if a path can be drawn between them which does not cross any other arc $k \in J$ or the line L . Two arcs P, R are defined to be *mutually visible* if there exist points $p \in P, r \in R$ s.t. p, r are mutually visible. The set of arcs visible from an arc or a point A is expressed as $\text{Visible}(A)$.

The algorithm exploits the fact that, given a pair of adjacent intersections m , any arc Q which is visible from m can be split into two new arcs Q' , Q'' which connect to m and that the sequence so formed remains a Jordan sequence. See Fig. 4.4.

Fig. 4.4 Node Q is split into Q' and Q'' and joined to node m .

The following operations, discussed in detail below, are used:

Operation	Description
Select	Select an arc at random from T_{upper} or from T_{lower} .
Insert	Insert a new arc as a child or sibling of an existing arc.
Find visible arcs	Find all arcs visible from a given intersection and select one at random.
Modify	Split an arc into two new arcs, incorporating two new intersections.
Update	Update the required data structures.

4.1.2.2 Selecting an arc

Two arrays are maintained, containing all nodes (except the root) of T_{upper} and T_{lower} , respectively. Each node maintains its index in these arrays. By selecting an array and an index at random, we can select a random arc in $O(1)$ time.

4.1.2.3 Inserting a new arc**PROCEDURE InsertArc**

INPUT: a tree T containing n arcs
an arc $A \in T$

OUTPUT: a modified tree containing $n+1$ arcs

BEGIN

IF A is a leaf node AND Random(50%) THEN
Add a new arc A' as a child of A

ELSE

Add a new arc A' as the right sibling of A ;

END;

Fig. 4.5 Pseudocode for insertion of a new arc.

A new arc A' is added as a child or right sibling of the selected node A (See Fig. 4.5, 4.6). Since A may be any non-root node in T , A' may be added as a child of any node in T . Since any tree may be generated by repeated insertion of children, this is sufficient to generate any possible tree.

Fig. 4.6 Adding a new node A' as the right sibling of A .

4.1.2.4 Finding visible arcs

The twin tree of T must be modified to incorporate the intersections. Any arc visible from an endpoint of A' is a candidate for modification. We define the *left neighbour* of an intersection point $a \in J$ to be the intersection $b \in J$ s.t. $b < a$ and for all other intersections $i \in J$ either $i < b$ or $i > a$. The *right neighbour* is defined symmetrically. We define the *left twin neighbour* of $A' \in T$ as the arc in $\text{twin}(T)$ which has the left neighbour of A' as an endpoint. The *right twin neighbour* is again defined symmetrically.

The left and right neighbours and twin neighbours of a pair of intersections or an arc are defined similarly. The left neighbour of $A = (A_{\text{left}}, A_{\text{right}})$ is defined to be the left neighbour of A_{left} , and the right neighbour of A is defined to be the right neighbour of A_{right} . Left and right twin neighbours of pairs of intersections are defined as above.

We show that, given the left and right twin neighbours L, R of a pair of intersections A' , if $L=R$ then $\text{Visible}(A') = \{L\}$. If $L \neq R$ then $\text{Visible}(A')$ is the intersection of the sets $\text{Visible}(L)$ and $\text{Visible}(R)$. The set $\text{Visible}(A')$ can also be expressed, more informally, as the smallest arc Q in $\text{twin}(T)$ whose interval encloses A' , and all of Q 's children. Fig. 4.7 illustrates the visible arcs.

Fig. 4.7 Visibility from an arc. From the endpoints of A' , the "parent" P of A' and all of P 's children are visible.

The "parent" node Q of A' can be found in $O(\log n)$. Let L, R be the left and right neighbours of A' respectively. Let u, v be left and right twin neighbours of A'

respectively. Using the cases outlined in Fig. 4.8 we can determine Q to be either u , v , or the parent of u , v . The balanced tree structure which maintains lists of children is bidirectional, so the parent node of a given node can be determined in $O(\log n)$ time in the worst case.

Case	Action	Illustration
$u=v$	Let $Q = u$ or v
u, v are siblings	Let $Q = \text{parent of } u, v$
u is the parent of v v is the parent of u	Let $Q = u$ Let $Q = v$

Fig. 4.8 Cases for selecting the parent node Q .

Since L and R are neighbours of A' , and no intersections lie in the interval $(A'_{\text{left}}, A'_{\text{right}})$, L and R must have been neighbours of each other before the intersections in A' were added. The depth in T of arcs whose endpoints are neighbours differ by at most one. Thus the table in Fig. 4.8 covers all possible relationships between L , R .

LEMMA 1: The interval defined by the left and right end points of Q completely encloses the interval defined by A' .

PROOF: There are three cases to be analyzed in the relation between Q and A' .

Case 1: P 's left endpoint = L , whose intersection point is by definition to the left of A' . Similarly Q 's right endpoint is to the right of A' 's right endpoint. Thus the interval defined by Q encloses the interval defined by A' .

Case 2: L, R are linked to two nodes $u, v \in \text{twin}(T)$ which are siblings. Assume without loss of generality that u is to the left of v . Then u contains the point L , and v contains the point R . The interval defined by (u, v) thus encloses the interval defined by A' . The interval defined by Q encloses the interval defined by any of its children and thus encloses the interval defined by A' .

Case 3: L, R are connected to two nodes (u, v) one of which is the parent of the other. Without loss of generality assume that u is the parent and that v connects to A_{right} . The left end point of u is L , and the right end point of u cannot be further to the left than R . Thus the interval is at least (L, R) and therefore encloses the interval defined by A' .

LEMMA 2: The set of nodes visible in $\text{twin}(T)$ from the endpoints of A' , a new child of $A \in T$, consists of the node Q and all children of Q as defined above.

PROOF: The interval defined by Q encloses the interval defined by A' (Lemma 1). The arc Q , combined with the horizontal line l , forms a closed curve which encloses the endpoints of A' . Nothing outside this closed curve is visible from A' . Within this closed curve lie only Q and descendants of Q . Thus, no arc not a descendant of Q can be visible.

Similarly, consider an arc q which is a child of Q . Descendants of q lie inside a closed curve defined by q and the horizontal line l . This curve excludes the endpoints of A' . Thus, any node not a child of Q is not visible.

It remains to show that Q and its children are visible. Since an arbitrary curve connecting the points is sufficient for visibility, any blockage of visibility must be caused by a closed curve which separates the endpoints of A' from Q and its children.

Consider the intersection L , the left neighbour of A' . L is the left neighbour of A'_{left} and is also the endpoint of Q or a child of Q (see Fig. 4.10). Since L is the left neighbour of A' , no arc intersects l between L and A'_{left} . Thus, any arc with endpoint L is visible to A' by a straight line along l connecting L to A'_{left} .

A node Y is connected to L and thus visible from A' . If A' lies outside Y then Y and its parent and siblings are visible from A' . If A' lies inside Y then Y and its children are visible from A' . If A' lies outside Y then Y is a child of Q . If A' lies inside Y then $Y=Q$. In either case, Q and all of its children are visible from A' .

Since Q can be found in $O(\log n)$ time, the set of nodes visible from the endpoints of the new node A' can be found in the same time. The set consists of Q and all children of Q , represented as a balanced search tree. A node q can be selected at random from this set in $O(\log n)$ time.

4.1.2.5 Modifying a visible node

A node R , visible from A' , is split into two new arcs R' and R'' , each of which connects to one endpoint of A' . The procedure is described in Fig. 4.9 and illustrated in Fig. 4.10.

PROCEDURE ModifyVisibleArc

INPUT: R : an arc
 A' : a pair of intersections
 C : a balanced search tree of arcs
 B : a boolean, indicating if R is the parent of C or a member of C

OUTPUT: R', R'' : arcs
 (C is modified)

BEGIN
 IF B **THEN**
 Let $R' = (R_{\text{left}}, A'_{\text{left}})$
 Let $R'' = (A'_{\text{right}}, R_{\text{right}})$
 Remove R
 Split C into two lists, those left of m and those right of m
 Assign those left of m as children of R'
 Assign those right of m as children of R''
 ELSE
 Let $R' = (R_{\text{left}}, A'_{\text{right}})$
 Let $R'' = (R_{\text{right}}, A'_{\text{left}})$
 Remove R
 ENDIF

Fig. 4.9 Modifying a visible arc.

Fig. 4.10 Node R is split into R' and R'' and joined to node m .

Removing R , inserting R' and R'' , and splitting the list of children takes $O(\log n)$ time.

4.1.2.6 Updating

It remains to update the data structures used in the algorithm. We can append the new arc A' to the array of all arcs in T in $O(1)$ time. R can be replaced with R' and R'' appended to the array of all arcs in $\text{twin}(T)$ in $O(1)$ time.

Each of the preceding operations requires no more than $O(\log n)$ time, and executing $O(n)$ such operations is sufficient to construct a Jordan sequence of size $2n$ from a sequence of size $2n-2$. Thus we have established:

THEOREM 1

A Jordan sequence of size $2n$ can be generated in $O(n \log n)$ time. Any Jordan sequence may be generated by this method.

4.1.3 A nearly linear-time algorithm for generating Jordan sequences

Since a Jordan sequence can be sorted in linear time [33], a lower bound of $\Omega(n \log n)$ can be established on constructing a Jordan sequence from a given set of $2n$ points. It is not immediately apparent if this bound is applicable to the methods described here, since these are generated without reference to a specific set of points.

As a topic of investigation, we had posed ourselves the question of whether $\Omega(n \log n)$ is a lower bound on the problem of Jordan sequences or whether an $o(n \log n)$ algorithm can be designed.

In this section, we show that the latter is true by presenting an algorithm whose running time is $O(n\alpha(n))$ where $\alpha(n)$ is related to a functional inverse of Ackerman's function and is extremely slow-growing. Ackerman's function and $\alpha(n)$ [1, 2, 63] can be defined by the expression:

Initially, each arc in T_{upper} is an equivalence class. The algorithm proceeds by successively connecting adjacent disconnected intersections and merging the equivalence classes. The goal is to produce a tree T_{lower} of size $2n+2$, leaving T_{upper} with no disconnected intersections and only one equivalence class.

4.1.3.1 Random generation of forests

The first step of the algorithm is to generate a tree T_{upper} of size $2n+2$. There exist several methods to accomplish this in linear time, of which we consider three: lexical generation, generation by insertion, and generation by unranking.

Note that the size of $2n+2$ includes the root of T_{upper} , which represents imaginary intersections at $+\infty$ and $-\infty$. In this section we disregard this root and discuss generating k -ary forests of size $2n$.

4.1.3.1.1 Lexical generation of random forests

There is, as previously discussed, a one-to-one correspondence between forests and well-formed bracket sequences. Using well-known techniques for parsing expressions it is possible to transform a bracket sequence into a forest in linear time. Thus, to generate a forest of size $2n$ it is sufficient to generate a well-formed bracket sequence of size $2n$. One method of producing well-formed bracket sequences is given in Fig. 4.12.

```

PROCEDURE RandomForestFromBrackets;

INPUT:      n: an integer
OUTPUT:     a well-formed bracket sequence of size n.
Variables:  B: a list of brackets;
            level: Integer;
            choice: Boolean;

level := 0;
FOR i := 1 TO n DO
    choice := Random (true, false);
    CASE
        (level = 0) OR choice:
            Append('(' , B); increment(level);
        ((level - (n - i)) >= 0) OR NOT(choice):
            Append(')' , B); decrement(level);
    ENDCASE
ENDFOR

```

Fig. 4.12 Algorithm to produce a random bracket sequence

This algorithm appends an opening or closing bracket to the sequence at each step. This is chosen at random unless the choice is forced by one of two conditions:

- 1) The level of the sequence is 0. Adding a closing bracket would cause the sequence not to be well-formed.
- 2) The level of the sequence is greater than or equal to $2n$ - the current sequence size. Adding an opening bracket would cause the level of nesting to be so deep that the sequence could not be completed in $2n$ steps.

These restrictions prevent all non-well-formed sequences and allow all well-formed sequences to be produced. However, it is also desirable that all possible sequences should be produced with equal probability, which is not the case.

Consider the sequence $((((()))$. This will be produced by the algorithm if and only if the first four brackets to be inserted are chosen to be opening brackets. The remaining four are forced by the algorithm to be closing brackets. Since the first bracket must always be an opening bracket, this sequence will occur with a probability of $\frac{1}{2^3}$. In producing the sequence $((()))$, only three choices (the first, and the last two) are forced. This sequence

is thus produced with a probability of $\frac{1}{2^3}$. Thus, not all well-formed sequences have equal probability.

4.1.3.1.2 Adding random children

A random forest of size $2n$ can be generated by generating a random tree of size $2n+2$ and removing the root. In our Jordan sequence generation algorithm it would not be necessary to remove the root, since we intend to transform the forest into a tree.

A simple algorithm to generate a random tree is to begin with a tree T of only one node. Select a node $p \in T$ at random and add a new node as a child of p . By repeating this process n times, any tree of size $2n+2$ can be generated.

This algorithm tends to produce bushy trees. The expected number of children of a node p is proportional to the number of steps of the generation algorithm for which it has existed in the tree. If a node p is at level k in the tree, it cannot have been added before step k of the algorithm and will have a lower expected number of children than any node at level $k-1$ or higher.

4.1.3.1.3 Forest generation by unranking

Unranking is a common technique for uniform generation of random instances of some class of structures. It can be applied to generation of random forests as follows:

Define for each possible forest a *rank*, which is a unique integer between 1 and the number of forests. If we can define an *unranking* procedure which generate a forest given its rank, then generating a uniform distribution of integers between 1 and the number of forests is sufficient to ensure a uniform distribution of forests. We define a ranking and the corresponding unranking procedure for k -ary forests, again making use of the one-to-one correspondence with well-formed bracket sequences.

The number of possible forests, and thus the number of well-formed bracket sequences, of size $2n$ is given by the n^{th} *Catalan* number [40, 52]. These numbers form the sequence:

1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786,...

according to the formula

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Now consider a sequence S . It is composed of two parts: the first well-formed sub-sequence S' and the remainder R . Note that R may be empty.

Assume that S' is of length $2k$. S' must begin with an opening bracket b and end with a closing bracket c and these brackets must form a matched pair. If this were not the case, then some other bracket d , preceding c , would match against b . This would mean that the sub-sequence from b to d would be the first well-formed sub-sequence, a contradiction.

The brackets b and c are constrained, but the remaining $2k-2$ brackets of S' may combine in any well-formed sequence, of which there are C_{k-1} . Fig. 4.13 shows an example with an initial well-formed sub-sequence of length 5 and a remainder of length 5. There are $C_4=14$ ways in which the inner 4 bracket pairs can be arranged and $C_5=42$ ways in which the five pairs which make up the remainder can be arranged.

$$\begin{array}{c} \downarrow \quad \downarrow \\ (O(00))00(00) \end{array}$$

Fig. 4.13 A bracket sequence with the first well-formed sub-sequence highlighted.

We define the *rank* of a sequence by defining the number of sequences with smaller rank. If, in a sequence of length $2n$, the first well-formed sub-sequence S' is of length $2k$, then define the rank of this sequence to be greater than the rank of all those sequences of length $2n$ whose first well-formed sub-sequence is of length less than $2k$. These can be

enumerated by determining the number of initial well-formed subsequences and multiplying by the number of possible remainders.

$$\text{rank} \geq \sum_{i=1}^{k-1} C_{i-1} * C_{n-i}$$

The rank of the sequence '()' is defined to be 1.

This definition may be recursively applied to the first well-formed sub-sequence (omitting the initial and final brackets) and to the remainder, to give the following formalization (See Fig. 4.14).

PROCEDURE Rank

INPUT: S: a well-formed bracket sequence

OUTPUT: the rank of S

BEGIN

 first := first well-formed sub-sequence of S;

 rest := the remainder of S;

 rank := 1;

 FOR i := 1 to size(S)-1 DO

 rank := rank + (C_{size(first)-1} * C_{size(rest)});

 firstRank := rank(first);

 rank := rank + ((firstRank-1) * C_{size(rest)});

 rank := rank + rank(rest) - 1;

END

Fig. 4.14 Pseudocode for the ranking operation.

This process can be reversed to create any sequence S given the Rank(S) by recursively determining portions of S and subtracting appropriately from the rank. This allows generation a forest of size 2n in O(n) time.

There is an important implementation caveat relating to this algorithm, since it involves many arithmetic operations on large integers. These integers grow exponentially (e.g. in generating a forest of only 25 nodes the algorithm may deal with numbers as large as C₂₅ = 4861946401452). Under these circumstances, the assumption that integer

arithmetic operations require only $O(1)$ time is no longer valid. In practical terms, operations on very large numbers are much slower than those on the limited range of integers (usually approximately 32 bits) which are normally supported by the underlying hardware, and this can cause the algorithm to run considerably more slowly than would be expected.

4.1.3.2 Generating a random Jordan sequence

Recall the definition of an equivalence class above. Each equivalence class contains exactly two disconnected intersections, as long as any disconnected intersections remain. We define these to be the *endpoints* of the equivalence class. Each disconnected intersection is an endpoint of some equivalence class, and stores a pointer to this class and to its position in a sorted list of all disconnected intersections.

We define the *left* and *right neighbour* of a disconnected intersection d as the disconnected intersection which is closest to d and lies to the left or right of d , respectively. A disconnected intersection is *extreme* if it has only one neighbour. (See Fig. 4.15)

Fig. 4.15 Equivalence classes. The disconnected intersections a, b are the left and right endpoints, respectively, of the equivalence class A . The right neighbour of a is b and the left neighbour is e . The extreme intersections are c and d .

If we select any equivalence class E , it is normally possible to find a neighbour of one endpoint e of E which belongs to an equivalence class F s.t. $F \neq E$. If this is not possible,

then E is the only remaining equivalence class and we may join its endpoints and terminate the algorithm. Pseudocode is shown in Fig. 4.16.

```

PROCEDURE RandomJordanSequence

INPUT:   $T_{upper}$ : a  $k$ -ary tree
         $n$ : an integer
OUTPUT:  $T_{lower}$ : a  $k$ -ary tree
VARIABLES:  $E$ : an array of equivalence classes;
            $D$ : a sorted list of disconnected edges

BEGIN
  REPEAT  $n$  TIMES
    Select at random an equivalence class  $e$  from  $E$ ;
    Select an endpoint  $p$  of  $e$  and a neighbour  $q$  of  $p$  s.t.  $q \in e$ ;
    Connect  $p, q$ ;
    Merge the equivalence classes of  $p, q$ ;
    Remove  $p, q$  from  $D$ ;
  END REPEAT
END

```

Fig. 4.16 Algorithm to generate a random Jordan sequence

4.1.3.2.1 Analysis of the algorithm

An array of all equivalence classes is maintained, and a class can be selected from this at random in $O(1)$. Each class has at most two endpoints, each of which has at most two neighbours, which can be found in $O(1)$ in the sorted list D . The information stored at a disconnected intersection D can be updated to indicate a new equivalence class, and the intersection removed from the sorted list in $O(1)$ time.

The limiting factor in this analysis is merging the equivalence classes. In particular, references are maintained to these classes in the array E , from which they cannot quickly be deleted.

These references can be maintained as sets with the MERGE and FIND operations. Whenever two equivalence classes are merged, the MERGE operation is performed to update these references. Whenever a reference to an equivalence class stored in E is made, a FIND operation is performed to determine the equivalence class.

An easily implementable representation of sets with MERGE and FIND operations is described in Knuth and in Aho, Hopcroft, Ullman [2, 40]. This representation has been analyzed by Tarjan, who showed it to have a worst-case time complexity of $O(n \alpha(n))$ for a sequence of n or more MERGE and FIND operations [63].

This complexity dominates the overall running time of the algorithm, allowing the generation of a Jordan sequence of length $2n$ in $O(n \alpha(n))$ time.

It remains to show that any Jordan sequence can be generated by this algorithm. In the absence of the constraint that adjacent intersections are not connected if they belong to the same equivalence class, any forest can be constructed repeatedly connecting adjacent disconnected intersections. This can most clearly be seen by considering the reverse process, in which an arbitrary forest is decomposed into disconnected intersections by repeatedly disconnecting the endpoints of arcs.

The constraint that adjacent disconnected intersections are not to be connected if they belong to the same equivalence class does not exclude any Jordan sequence of size $2n$. If two intersections which belong to the same equivalence class are connected, they form a closed curve. This is permitted in the last step of the algorithm. If performed earlier, this results in a closed curve of size less than $2n$. This does not produce a Jordan sequence of size $2n$.

Thus we have shown:

THEOREM 2

A Jordan sequence of size $2n$ can be generated in $O(n \alpha(n))$ time and any Jordan sequence may be generated by this method.

4.1.4 Conclusions

We have demonstrated two very different approaches to generating random Jordan sequences, one operating in $O(n \log n)$, the other in $O(n \alpha(n))$. There remain several open problems.

The first of these is the demonstration of a linear-time algorithm to generate Jordan sequences, or the proof of a lower bound greater than linear. The second is an algorithm which can guarantee a uniform distribution of the generated sequences. This is closely related to the third problem, which is determining the number of Jordan sequences of a given size. This last is an extremely difficult problem, on which work is still underway.

4.2 "RANDOM" POLYGONS

Another interesting problem is the generation of random polygons. Many geometric algorithms operate on polygons, and the ability to generate random instances efficiently would be extremely valuable.

We have found very little literature on this subject. Many approaches to generating polygons from arbitrary point sets have tried to generate "nice" polygons which can more easily be processed [16, 17, 48, 54]. Our area of interest is in the generation of "difficult" polygons which can provide demanding test cases for algorithms.

More specifically, we are interested in generating polygons which do not possess properties which are exploited by the algorithms under consideration. Particular emphasis is placed on the generation of difficult cases, including polygons with a high degree of spiralling.

It appears unlikely that a rigorous definition of what constitutes a random polygon is possible, and even the criteria to be used to measure randomness are unclear. A simple

definition requires the vertices of the polygon to follow a uniform random distribution. One might, however, require that the lengths of edges, the number of lines per unit area, the total area enclosed by the polygon, or the number of intersections with an arbitrary line also follow a uniform distribution. It is not clear that all these criteria can be satisfied, or that they do not contradict one another.

4.2.1 Generating polygons by a "random walk"

One may approach the problem by defining a randomized method for creating a polygon without attempting to predetermine its properties. One simple algorithm is given in Fig. 4.17.

PROCEDURE RandomWalk

INPUT - N : the number of points desired

OUTPUT - P : a simple polygon containing at least N vertices

VARIABLES - q : a point

v : the visibility polygon from q to the current
simple chain of P

P .First := Random point;

q := a random point;

WHILE (P .Size < N) OR (P .Start NOT visible from q) DO

v := the visibility polygon from q to P ;

q := a random point inside v ;

Append q to P ;

ENDWHILE

Join q to P .first;

Return(P);

Fig 4.17 Generation using a random walk.

This algorithm will use a "random walk" approach to construct a polygon with at least n vertices. Since the visibility polygon can be produced in $O(n)$ time [13, 18, 41] this can potentially produce an n -vertex simple polygon in $O(n^2)$ time. The disadvantage of this algorithm is that termination occurs only after n or more vertices have been added and the

two ends of the chain are visible. The algorithm provides no guarantee when this will occur, and so the total number of vertices produced and the total running time are unbounded.

A polygon produced in this manner might reasonably be described as random, but the inability to guarantee termination and output size makes this ineffective for practical purposes.

4.2.2 Generating restricted classes of polygons

Another approach, rather than seeking an efficient algorithm for generating arbitrary simple polygons is to restrict our attention to particular classes of polygons. There are well-known algorithms to produce a monotone or star-shaped polygon from an arbitrary set of points in $O(n \log n)$ time, which are described below.

4.2.2.1 Monotone

A monotone polygon can be generated from an arbitrary a set of n points P in $O(n \log n)$ time. The algorithm begins by defining a total ordering on the points as the length of their projection on a line L . We then proceed as shown in Fig. 4.18.

```

PROCEDURE RandomMonotonePolygon;

INPUT:    $P$ : a random set of points and their projections on  $L$ ;
            $L$ : a line of monotonicity;
OUTPUT:  $P'$ : a polygon monotone

Find  $p_{\min}, p_{\max}$ , the points with minimum and maximum projection on  $L$ ;
Construct two chains,  $C_l = (p_{\min})$  and  $C_r = (p_{\max})$ ;
FOR each point  $p$  in  $P$  not  $p_{\min}$  or  $p_{\max}$  DO
    IF  $p_{\min}, p_{\max}, p$  is a left turn THEN
        Append( $C_l, p$ )
    ELSE
        Precede( $C_r, p$ );
ENDFOR
Append( $C_l, p_{\max}$ );
Append( $C_r, p_{\min}$ );
 $P' :=$  the concatenation of  $C_l, C_r$ ;

```

Fig. 4.18 Generation of a monotone polygon from a random set of points.

Upon completion, C_r and C_l will be chains monotone with respect to the line from $p_{\min}$ to $p_{\max}$. The concatenation of these chains will be a polygon monotone with respect to the same line.

4.2.2.2 Star-shaped

Star-shaped polygons may be created from a random set of points using a similar procedure, sorting by angle around a point rather than by projection on a line of monotonicity.

Given a random point set P , containing n points, select a point q which lies inside the convex hull of P . An efficient method of selecting such a point is to select three points from P at random and to define q as lying inside the triangle which these points define. Sort the points in P by angle around q and by distance from q . The sorted order of these points defines a star-shaped polygon whose kernel includes q .

A variation of this method constructs a star-shaped polygon without being given the initial point set. A point q is selected and n points generated at random in a polar coordinate system with origin at q . These points are sorted as in the preceding version.

These two algorithms are equivalent in that any polygon generated with one could be generated by the other. They differ only in the distribution of the point sets from which they are generated. The first algorithm operates on a uniform distribution within a square, while the second are uniformly distributed within a circle. Both the above algorithms have been implemented, and sample output polygons from each algorithm are shown in Fig. 4.19).

Fig. 4.19 Star-shaped polygons produced by the algorithms described above.

4.2.3 Generating Random Polygons Using Visibility Information

4.2.3.1 Algorithm Description

Even though a rigorous definition of randomness in a polygon is unavailable, methods can be developed which do not create polygons with obvious properties that can be exploited by an algorithm. We describe one method which constructs a polygon from a point set using visibility information.

The algorithm is described inductively in Fig. 4.20, generating at each step a polygon of size $n+1$ from a polygon of size n and a randomly selected point. An illustration of the process is shown in Fig. 4.21.

PROCEDURE RandomPolygonFromVisibility

INPUT: P: a polygon, initially of three vertices

N: the desired final size of P

OUTPUT: P: a polygon of size N

BEGIN

Select a random point q ;

Find an edge $e=P_i, P_{i+1}$ s.t. e is completely visible from q ;

Remove e from P and replace it with P_i, q and q, P_{i+1}

END

Fig. 4.20 Generation of a random polygon using visibility information

Fig. 4.21 An example step of the algorithm in Fig. 4.21. The edge e is completely visible from the point p . The edge is removed and replaced with e' and e'' .

Although this technique appears to generate polygons without any obvious characterizations it is expensive. A brute-force implementation takes $O(n^2)$ time to find a visible edge, making the total running time of $O(n^3)$. Methods exist to find the visibility polygon from an edge in $O(n)$ time [13, 41], which can reduce the running time to $O(n^2)$.

Even with this improvement, the generation of test cases may be considerably more costly than execution of the algorithms to be tested.

4.2.3.2 Implementation

The high time complexity of this algorithm suggests that if it is to be useful it should be implemented with as small a run-time constant factor as possible. A brute-force version of this algorithm had previously been coded in Pascal [15]. It was decided to use this previous implementation, both to avoid duplication of effort and to allow faster execution of the algorithm than would be possible in Smalltalk.

The Pascal $O(n^3)$ implementation was ported to an IBM-compatible microcomputer and it was discovered that the running times were prohibitively large for examples of sufficient size to be interesting. This prompted a heuristic refinement of the algorithm as shown in Fig. 4.22.

```
PROCEDURE ImprovedRandomPolygonGeneration

INPUT:  P: a polygon, initially of size 3
        N: the desired size of P
OUTPUT: P: a polygon of size N

BEGIN
    Pick a point Q at random;
    Pick an edge at random;
    done := false;
    WHILE not done DO
        FOR each edge E' in P DO
            IF the line from Q to either end point of E intersects E' THEN
                E := E';
                Abort and restart the FOR loop;
            ENDIF
        ENDFOR
        done := true; { We reach this point if no line obscures E }
    ENDWHILE
    Remove E;
    Add new edge from start(E) to Q;
    Add new edge from Q to end(E);
END
```

Fig. 4.22 A heuristic improvement to the algorithm of Fig. 4.20.

In this improved version, a candidate edge E is only considered until it is obscured by some other edge E' . Once this occurs, E' immediately becomes the candidate edge.

This forms a sequence of candidate edges, each of which is closer to Q than the previous. We define the length of this sequence to be $f(n)$, and the running time of the algorithm is $O(n^2 f(n))$. In the worst case, $f(n)$ is $O(n)$ and the running time of the algorithm has not been improved, but for many polygons it is much smaller.

Since we have already described a $O(n^2)$ algorithm based on finding visibility polygons, this heuristic seems unnecessary. It has the advantage, however, that it is extremely simple to implement, and this can be done by small modifications to the existing Pascal code.

By making these modifications, the running time of the generation program was dramatically improved. By translating to the C language and running the program on a Sun SparcStation, the performance was sufficient to generate large test cases in a reasonable time.

4.2.3.3 Evaluation

As described above, the performance of the improved algorithm is $O(n^3)$ worst-case, which is far from optimal. We are, however, less concerned with the asymptotic running time than with the "randomness" of the created polygons. The degree of randomness has can only be evaluated subjectively, but inspection of the polygons produced by this algorithm has not revealed many properties which can be exploited by an algorithm.

One property which can be exploited is a lack of spiralling in the polygons produced by the algorithm. The reasons for this can be seen by considering the conditions necessary to produce a spiral. At each stage of the algorithm, a point p is selected at random. If adding p to the polygon is to increase the degree of spiralling in some sub-section, p must lie in some region S . As the degree of spiralling increases, the size of the region S decreases,

and the probability of a new point q lying in that region decreases. Although any degree of spiralling is theoretically possible, high degrees become increasingly improbable. Fig. 4.23 illustrates this point.

Fig. 4.23 A heuristic improvement to the algorithm of Fig. 4.20. In the algorithms of Fig. 4.20 or 4.22 the point q must be selected to lie within S if the degree of spiralling is to be increased. As the degree of spiralling increases, the size of S decreases.

This property could be exploited by an algorithm. The Chazelle and Incerpi [11] triangulation algorithm, for example, runs in time $O(n \log s)$ where s is defined as the *sinuosity* of the polygon, which corresponds to the degree of spiralling.

4.2.3.4 Polygon generation using edge visibility

A number of variations on the technique described above might help produce polygons with a higher degree of spiralling. One such variation is described in Fig. 4.24.

PROCEDURE RandomPolygonFromEdgeVisibility**INPUT:** P : a polygon, initially of size 3 N : the desired size of P **OUTPUT:** P : a polygon of size N **BEGIN****REPEAT** $N-3$ **TIMES** Select an edge $E \in P$ at random; Find the visibility polygon from E in P ; Select a point q inside P ; Replace the edge E with E_{start}, q and q, E_{end} **END REPEAT****END**

Fig. 4.24 Algorithm for generation of a random polygon by selecting an edge and finding the visibility polygon.

In this algorithm, an edge E to be modified is first selected and the point q is chosen to lie inside the visibility polygon from E . This ensures that each edge will be modified with equal probability, increasing the likelihood of spiralling.

4.2.3.5 Generation using triangulation

Given a triangulation of an n -vertex polygon P , a point p which is visible from an edge E can be determined in $O(\log n + k)$ [30], where k is the number of diagonals of the triangulation which are crossed by a path from p to E .

In removing the edge $E=(e_1, e_2)$ and replacing it with the edges $E'=(e_1, p)$ and $E''=(p, e_2)$, $O(k)$ triangles will be altered. The area defined by these triangles forms a polygon which is edge-visible (from E) and can thus be re-triangulated in time $O(k)$. Fig. 4.25 shows an example of this technique.

Fig. 4.25 Generation of a random polygon using triangulation. A point p which is visible from an edge E is selected and the E replaced by two new edges. On the left, triangulation edges which are crossed by the path from E to p are shown shaded. On the right, these triangulation edges have been removed and new edges, also shown shaded, have been inserted.

This method offers two advantages. First, If the vertices of the polygon are stored in a balanced search tree, then the edge E can be removed and E' and E'' substituted in $O(\log n)$ time. The dominant factor in the time complexity is then k , the length of the path from E to p . Since k is in the worst case $O(n)$, the overall time complexity is $O(n^2)$. If p is restricted to lie within $O(\log n)$ of E , then the running time of the algorithm is $O(n \log n)$. If this restriction is applied, then the edges of the resulting polygon will be shorter than with the other methods described. This gives us, however, a scalable algorithm, which runs in time $O(nm)$, where m can be chosen arbitrarily within the range $O(\log n) \leq m \leq O(n)$.

Second, the method of selecting the point p from amongst the candidate triangles can be selected so as to select p from amongst all candidate triangles with equal probability. The number of triangles in a region is proportional to the number of vertices in that region rather than to the space occupied by that region. Thus, polygons with a high degree of spiralling are more likely to be produced.

One modification to the algorithm is necessary. As described, the point p must always be chosen to lie within the boundary of the polygon P , which is an undesirable limitation. This can be overcome by also triangulating the area between P and another polygon P' .

which contains P . Most simply, P' can be chosen to be the convex hull of P , but it might also be an arbitrary polygon bounding the region from which points are to be selected.

This method has not been implemented in the workbench, but is a promising technique for generation of "random" polygons. Implementation and empirical evaluation of the "randomness" of the resulting polygons must be conducted before

4.3 CHAPTER SUMMARY

This chapter has described algorithms for the random generation of k -ary trees, Jordan sequences, and polygons. Each of these generation problems is motivated by the need to generate test cases for algorithms. The emphasis is therefore on producing difficult cases which may cause problems for algorithms, as opposed to the more common emphasis on producing "nice" cases which can be more easily processed.

Several open problems remain. Neither of the two algorithms for generation of Jordan sequences produce all sequences with equal probability. It remains an open problem to produce such an algorithm, and to enumerate the number of Jordan sequences of a given size.

It is a difficult problem even to define a random polygon. Objective measures of randomness and more efficient algorithms for producing "random" polygons might be possible.

5 COMPARISON OF ALGORITHMS

5.1 GENERAL NOTES

A major design goal of the computational geometry workbench is to provide an environment in which the constant factors and implementation concerns of algorithms can be compared.

The algorithms described below have all been implemented by the same person (the author), in the same environment, using the same primitive operations. Thus, we can expect the resulting comparisons to reasonably reflect the efficiency of the algorithms themselves.

None of these implementations have been optimized for speed to any significant degree, and are as close as possible to the original algorithmic description. The degree to which an algorithm admits to optimization is also an important concern, but one which has not been addressed here. We note in passing that the time required for the Fournier and Montuno algorithm is normally dominated by comparison operations on line segments and active trapezoids. By optimizing this comparison operation, it is likely that the constant factors of this algorithm could be greatly reduced with small effort.

5.2 IMPLEMENTABILITY AND LINES OF CODE

For the purposes of comparing implementability, we list here a few statistics regarding the implementation of different algorithms. These figures are approximate, and have some

margin of error in including irrelevant or obsolete code, but should be substantively correct.

The geometric workbench contains approximately 20,000 lines of Smalltalk code, in 133 classes. Of this amount, approximately 2000 lines in 10 classes are user interface code. The Smalltalk/V Mac system also contains approximately 20,000 lines of code.

8000 lines of code are devoted to the Tarjan and van Wyk triangulation algorithm, of which 3200 lines are used for homogeneous and heterogeneous finger trees. Splay trees are implemented using 625 lines of code, and the Fournier and Montuno algorithm involves less than 1000 lines of code. Jordan sorting uses approximately 2700 lines of code.

None of the convex hull algorithms tested required more than 100 lines of code to implement.

Under most circumstances, the Fournier and Montuno algorithm is preferable to the Tarjan and van Wyk algorithm. This is not so much because of its relatively small speed advantages, but due to its much simpler implementation.

5.3 CONVEX HULL ALGORITHMS

The computational geometry workbench implements three different convex hull algorithms for point sets. The gift-wrapping algorithm [36] works incrementally constructing the convex hull. At each step, the next vertex on the hull is found by determining the vertex whose angle with respect to the previous hull vertex is a minimum. The disadvantage of this algorithm is that, for a set of n points, it requires $O(n)$ time at each step. The total time is thus $O(nm)$ where m is the number of vertices in the convex hull. In the worst case, $m=n$ and the running time is $O(n^2)$, but for a uniform distribution of points, the expected value of m is $O(\log n)$, giving a running time of $O(n \log n)$. The

algorithm is illustrated operating on a set of points which form a simple polygon in Fig. 5.1.

Fig. 5.1 Gift-wrapping a polygon to find the convex hull. At each step, a ray is computed from the last hull vertex found to the point with the minimum angle from the last hull vertex. Since the method does not exploit the ordering of the points it can be applied to arbitrary point sets.

The Graham scan algorithm [28] operates by constructing, in $O(n \log n)$ time, a polygon whose convex hull can be found in $O(n)$ time, using the "dent-flattening" method which is described as part of the Tarjan and van Wyk triangulation. In this implementation, we construct a monotone polygon.

The Melkman algorithm [45] is a dynamic variation of another convex hull algorithm for polygons due to Graham and Yao [29]. Using a two-ended queue, this algorithm computes the convex hull of any simple polygon in linear time. This algorithm also requires the creation, in $O(n \log n)$, of a simple polygon.

If a uniform distribution of points is assumed, then all of these algorithms should run in $O(n \log n)$ time. By plotting the time required to actually run each algorithm, we can see that there are great differences in the constant factors involved. Fig. 5.2 shows such a plot.

Fig. 5.2 A plot of time vs. number of vertices for convex hull algorithms on a set of points.

The “gift-wrapping” algorithm is much slower than either the Melkman or Graham scan. There are several possible reasons for this. The first is that the “gift-wrapping” algorithm may be requiring more time because the number of vertices on the hull is a larger constant than that on the sorting step required by the Melkman and Graham scan algorithms. Both

of the latter are simple linear-time algorithms once the initial sort to produce a monotone polygon is complete, and the constant involved in sorting may be much lower.

Another reason is that the primitive operations involved in "gift-wrapping" involve computing angles, an expensive operation. The point sets used in these examples have coordinates which are specified in floating point numbers. One might expect the difference to be even larger with integer coordinates, since the Melkman and Graham Scan algorithms could then use integer arithmetic exclusively.

Fig. 5.3 shows more detail of the Melkman and Graham scan results. In this plot, the difference in the constant factors of the two algorithms is evident.

Fig. 5.3 An enlarged plot of the Melkman and Graham Scan algorithms

5.4 COMPARING JORDAN SORTING

The algorithm for sorting Jordan sequences in linear time is an important part of the Tarjan and van Wyk triangulation algorithm, and an important algorithm in its own right. We conduct empirical tests on this algorithm in two ways. First, we empirically test the dynamic optimality conjecture of Sleator and Tarjan [59] by using both splay trees and finger trees in the algorithm. Second, we compare this algorithm against conventional sorting algorithms, specifically heapsort and quicksort. Plots of the time required for all four methods is shown in Fig. 5.4.

Sorting Jordan Sequences

Fig. 5.4 Four methods of sorting Jordan sequences. Note that the time values for quicksort and heapsort are multiplied by 10 in this figure.

In order to test the performance of splay trees, implementations of the Jordan sorting algorithm were done with both homogeneous finger trees and with splay trees. The algorithms are identical in all other respects, and were run on identical inputs.

The results appear to support the dynamic optimality conjecture. For almost every example, the implementation using splay trees was at least as efficient as that using finger trees. The two results are virtually identical for small input sizes, so this advantage of splay trees does not appear to be due to a smaller constant factor mitigating a larger asymptotic running time.

Results are shown here for sequences ranging in size from 10 to 250. These are not extremely large, but it is worth noting that they are larger than the great majority of Jordan sequences encountered in the Tarjan and van Wyk algorithm. Because the number of intersections of a polygon with a horizontal line is, in general, quite small (See Chapter 4), the size of the Jordan sequences normally encountered is also very limited. Even for classes of polygons which may admit a large number of intersections (e.g. star-shaped), many of these intersections will be non-essential, and will not participate in the Jordan sequence.

Both implementations of the Jordan sorting algorithm were much slower than conventional sorting algorithms with worse asymptotic complexity.

The other question which arises regarding the Jordan sorting algorithm, is whether the algorithm requires linear time in practice. From the previous graph this is difficult to determine, although the function does appear close to linear.

For a more detailed look at the way the time required for Jordan sorting grows with increasing number of points, we plot the time divided by the number of points. For a perfectly linear function (i.e. $y = kx$ for some constant k) this should appear as a horizontal line. The results of plotting the Jordan sorting times are shown in Fig. 5.5.

Time / Number of Points for Sorting Jordan Sequences

Fig. 5.5 Time / number of points in sorting Jordan sequences. For an algorithm which is perfectly linear this should form a horizontal line.

From this figure the plots for heapsort and quicksort appear as expected. We would also expect, however, that the plot of times for the Jordan sorting algorithm should approximate a horizontal line, since the algorithm is $O(n)$. Instead, both versions fairly consistently increase with respect to n . This would seem to imply that either the analysis or the implementation of the algorithm must be wrong.

This is, however, not necessarily true, since the statement that the time taken by an algorithm is $O(n)$ is by no means the same as the statement that it conforms to a function of the form $y=kn$. Specifically, we claim that it is possible to construct a function $F(n)$ which

is $O(n)$, which depends on no hidden constants larger than 1, and for which the plot of $F(n)/n$ vs. n increases with respect to n . Such a function is plotted in Fig. 5.6.

Fig. 5.6 A function $F(n)$ which is linear in n may nevertheless exhibit behaviour which is apparently non-linear.

The function $F(n)$ is defined as shown in Fig. 5.7:

$$F(n) = \left\{ \begin{array}{l} O(1) \text{ if } n=1; \\ i = \text{random}(1, n-1) \\ T'(i) + T'(n-i) + \log \min\{i, n-i\} \text{ if } n>1; \end{array} \right\}$$

Fig. 5.7 The definition of the function plotted in Fig. 5.6.

This is the same recurrence relation which is defined in Fig. 3.30, from which the $O(n)$ time bound on Jordan sorting is derived. Since this recurrence is known to be $O(n)$, it implies that this behaviour, and thus the behaviour of the Jordan sorting algorithm is also consistent with an $O(n)$ running time.

This raises an important issue in empirical experiments of this nature. If such apparently non-linear characteristics are, in fact, consistent with an $O(n)$ running time, how can it be determined if an algorithm exceeds $O(n)$? This is a difficult question, and one to which a answer may not be possible without reference to the structure of the algorithm and its implementation.

5.5 COMPARING TRIANGULATIONS

A primary goal of implementing both the Tarjan and van Wyk and the Fournier and Montuno algorithms for triangulation of a simple polygon was to compare constant factors. The Tarjan and van Wyk algorithm is asymptotically the better of the two, but its extreme complexity seems to indicate that constant factors may be significantly higher.

The following graphs (Fig. 5.8 - 5.10) show the time required for triangulation using each of these algorithms.

Triangulation of Simple Polygons

Fig. 5.8 Graphs for the triangulation of simple polygons.

Fig. 5.8 shows the time required for triangulation of “random” simple polygons. These polygons were generated using the visibility-based random generation technique described in Chapter 4. For all ranges tested, the Fournier and Montuno algorithm was slightly faster than the Tarjan and van Wyk algorithm. One might expect to see the Tarjan and van Wyk algorithm improving its relative performance as the numbers become very large, but this is not observed here. One possible reason for this failing to occur is the previously discussed characteristic of test polygons, in that they have relatively few intersections with a horizontal line. If we conjecture that, for these test instances, the number of intersections is $O(\log n)$, then the Fournier and Montuno algorithm becomes $O(n \log \log n)$ if the initial sorting step is disregarded as insignificant.

In order to test this conjecture, we ran similar tests on star-shaped polygons, which normally have many more intersections with an arbitrary horizontal line. Fig. 5.9 shows these results.

Triangulation of Star-Shaped Polygons

Fig. 5.9 Triangulation of star-shaped polygons

For star-shaped polygons, as expected, the Tarjan and van Wyk algorithm is more efficient for large input sizes than the Fournier and Montuno algorithm.

One would expect the opposite to be true for Y-monotone polygons, for which the Fournier and Montuno algorithm, disregarding the initial sorting step, should be $O(n)$. These results are shown in Fig. 5.10.

Triangulation of Y-Monotone Polygons

Fig. 5.10 Triangulation of monotone polygons

For monotone polygons the Fournier and Montuno algorithm is much faster than the Tarjan and van Wyk algorithm, as the size of the balanced tree structure maintained by the Fournier and Montuno algorithm becomes $O(1)$.

5.6 OTHER COMPARISONS

Comparison based on time, while valuable, and often a major concern, can make it difficult to discover the details of operation costs. An alternative is counting specific operations. For example, in the case of a tree structure, we may count comparison operations.

This has several advantages. It allows the relative cost of operations to be disregarded. This is useful, for example, when running floating point calculations on a machine which is not equipped with a floating point processor. It can also help eliminate side-effects which are particular to the machine and not to the operation in question (e.g. garbage collection, swapping to disk).

5.7 CHAPTER SUMMARY

This chapter describes several experiments in comparing algorithms for triangulation, convex hull, and sorting Jordan Sequences. The results of these experiments appear to support the dynamic optimality conjecture of Sleator and Tarjan, which states that Splay trees may be substituted for finger trees in the Jordan sorting algorithm without violating the time complexity. These results also show that the performance of the Tarjan and van Wyk algorithm is comparable to that of the Fournier and Montuno algorithm, with each having advantages in particular circumstances. In general, however, the Fournier and Montuno algorithm is to be preferred for its simplicity of implementation.

We also raise the issue of the difficulty of confirming asymptotic complexity from empirical observation, and discuss measures of comparison other than time.

6 CONCLUSION

Although much has been accomplished, the computational geometry workbench is by no means a complete product. There remain many areas in which both the framework and the number of available algorithms should be extended. Correct implementation of geometric algorithms can be extremely difficult, even more so than we had anticipated, but the framework and the base of knowledge which has been accumulated so far has lessened the difficulty. As the suite of tools that compose the workbench becomes larger and more mature, we can only expect further improvement.

We emphasize the importance, in implementation, of good, abstract mechanisms. The separation of abstract data types from data structures is vital for the production of maintainable and re-usable code. Subtle differences between data structures (as with array and tree-based representations) can cause great difficulties if not properly abstracted. Finally, general mechanisms such as labels help implement minor variations on algorithms with a minimum of difficulty.

We close this thesis with a discussion of possible extensions to the workbench, and suggestions for future research which have been inspired by this work.

6.1 WORKBENCH EXTENSIONS

There are many possible extensions to the current workbench system. This section lists some of the more significant.

6.1.1 Improved user interface

The current user interface is simple and serves many useful purposes. As the workbench becomes more sophisticated, however, it is in need of some extensions. One serious deficiency is that all browser operations are done in terms of screen coordinates. Providing coordinate transformations and the ability to work in object coordinates is an important extension.

Improved support for user preferences is necessary. Options might include different levels of algorithm checking, specifying floating point vs. rational arithmetic for operations, and colour preferences.

Other extensions include a better Undo facility, more interactive creation, constrained objects (e.g. rectilinear polygons), support for dynamic algorithms, use of colour, and a help facility.

6.1.2 Three-dimensional geometry

Up to this point, our main concentration has been on implementing a number of similar algorithms for purposes of direct comparison. As this has included the extremely complex Tarjan and van Wyk triangulation algorithm, little time has remained for expanding the breadth of the workbench. Three-dimensional computational geometry is an area into which the workbench should be expanded. Work is already in progress on this, and course projects by Duong Nguyen [47] and Danny Epstein [19] have begun to address three-dimensional issues.

6.1.3 Additional geometric objects

The geometric objects supported by the workbench are not complete. Polygons with holes should be supported, the planar subdivision should be generalized to be able to

represent a Voronoi diagram, and arbitrary groups of geometric objects should be supported.

6.1.4 More general algorithm structure

One of the difficulties encountered in the workbench is that the concept of an algorithm is insufficiently abstracted. The Smalltalk classes do not effectively capture all properties of an object which can be exploited, nor is it desirable that they should. A major improvement to the system would be the construction of abstract operations which can decide, based on the properties of an object, which implementation of that operation is most appropriate.

With checking of object properties, it is also possible to implement constraints on objects. For example, a polygon might be constrained to be monotone in some direction. As long as this constraint was in effect, no modification could be made to the polygon which would destroy this property.

6.1.5 Better source code control

Many different people have contributed to the workbench, often working simultaneously. A limited form of source code control has been implemented, but this is inadequate for the large scale conflicts which are likely to arise as the project expands. This is a general weakness of Smalltalk, and one which needs to be addressed.

There are other areas in which extensions to Smalltalk are desirable. Smalltalk/V does not support class categories or method protocols. Both of these, or some similar mechanism, are necessary for organizing large numbers of messages. Private classes, or limited scopes of some kind would also help improve the organization of the system. Finally, a facility which would allow adding arbitrary code to be executed immediately before or after the execution of a specific method would be useful. This would allow

implementation of pre and post-conditions on methods, and provide a valuable debugging aid.

6.1.6 Porting

The workbench currently runs only in Smalltalk/V on an Apple Macintosh. In order to be available to a larger group of users it should be ported to other platforms, including Smalltalk-80.

6.1.7 Organization

The design of the workbench becomes increasingly important as its size and scope expand. Generality and modularity of the tools implemented so far will be vital if we are to take advantage of the work done so far in expanding into new areas. As the number and complexity of the algorithms and data structures exceeds the knowledge of any one person, coherent high-level views of the system become vital.

6.2 FUTURE RESEARCH

Several topics of future research have been suggested by this work. This section describes some of the more important.

6.2.1 Counting Jordan sequences

The number of Jordan sequences of a given size is, as described above, an open problem. Although some progress has been made in this area, there are no firm results at this time. This problem is of substantial interest as a theoretical question, and is also

important in attempting to construct an algorithm for the generation of Jordan sequences which constructs all sequences with equal probability.

6.2.2 Algorithms for dynamic property maintenance

Constraints are a desirable feature in editing geometric objects. We might wish to know, given a polygon P , with some set of properties R , and a vertex $V \in P$ which is to be modified, the region in which V may be moved without violating any of the properties of R .

Current algorithms address some of these issues, but not all, and discovering asymptotically efficient algorithms which are fast enough to be used in an interactive editor is a promising area of research.

6.2.3 Modifications to the Tarjan and van Wyk algorithm

The Jordan sorting step in the Tarjan and van Wyk algorithm is very slow. Suppose that we were to substitute another sorting algorithm in its place. After the list of intersections has been sorted, it is possible to construct bracket sequences, and thus the upper and lower trees in linear time. Given a method which can sort Jordan sequences in linear time (the dynamic optimality conjecture suggests that insertion into a splay tree is sufficient), this could provide a fast method of Jordan sorting. It remains to be seen if such a method could also handle the error correction required by the Tarjan and van Wyk algorithm.

REFERENCES

- [1] W. Ackerman, "Zum Hilbertschen Aufbau der reellen Zahlen", *Math. Ann.*, Vol. 99, 1928, pp. 118-133.
- [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1987.
- [3] D. Avis and G. T. Toussaint, "An efficient algorithm for decomposing a polygon into star-shaped polygons", *Pattern Recognition*, Vol. 13, No. 6, 1981, pp. 395-398.
- [4] J. L. Bentley and M. I. Shamos, "A problem in multivariate statistics: Algorithms, data structure, and applications", in *Proc. 15th Annual Allerton Conference on Communication, Control, and Computing*, 1977, pp. 193-201.
- [5] D. G. Bobrow, L. G. De Michiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon, "Common LISP object system specification", Tech. Report, ANSI X3J13 (ANSI Common LISP), 1988.
- [6] Turbo Pascal 5.5 reference guide, Borland International, 1989.
- [7] M. E. Botkin and J. A. Bennett, "The application of adaptive mesh refinement to shape optimization of plate structures", Tech. Report No. GMR-4745, General Motors Research Laboratories, Warren, Michigan, 1984.
- [8] M. H. Brown, "Algorithm Animation", PhD Thesis, MIT, 1987.
- [9] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of SELF: A dynamically typed object-oriented language based on prototypes", in *Proc. OOPSLA '89*, 1989, pp. 49-70.
- [10] B. Chazelle and J. Incerpi, "Triangulating a polygon by divide-and-conquer", in *Proc. 21st Allerton Conference on Communication, Control and Computing*, Monticello, Illinois, 1983, pp. 339-349.
- [11] B. Chazelle and J. Incerpi, "Triangulation and shape complexity", *ACM Trans. on Graphics*, Vol. 3, 1984, pp. 135-152.
- [12] B. Cox, *Object-oriented programming: An evolutionary approach*, Addison-Wesley, 1986.
- [13] J. Dean and J.-R. Sack, "Efficient hidden-line elimination by capturing winding information", in *Proc. 23rd Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Illinois, 1985, pp. 496-505.
- [14] Smalltalk/V Mac Reference Manual, Digital Inc., 1988.
- [15] T. Doe and S. Edwards, *Course Project: Random generation of polygons*, 95.508, Carleton University, 1984.
- [16] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, W. Brauer, G. Rozenberg, and A. Salomaa (Ed.), EATCS Monographs on theoretical computer science, 10, Springer-Verlag, 1987.
- [17] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel, "Constructing arrangements of lines and hyperplanes with applications", *SIAM J. Comput.*, Vol. 15, 1986, pp. 341-363.

- [18] H. ElGindy and D. Avis, "A linear algorithm for computing the visibility polygon from a point", *Journal of Algorithms*, Vol. 2, No. 2, 1981, pp. 186-197.
- [19] D. Epstein, *Course Project on three-dimensional convex hull algorithms, Course 95.508*, Carleton University, 1990.
- [20] P. Epstein, "Polygon shortest path algorithms and applications", 4th Year Honours Thesis, Carleton University, 1990.
- [21] A. R. Forrest, "Computational geometry and software engineering: Towards a geometric computing environment", in D. F. Rogers and R. A. Earnshaw (Ed.), *Techniques for computer graphics*, Springer-Verlag, 1987, pp. 23-37.
- [22] S. Fortune, "A sweepline algorithm for voronoi diagrams", *Algorithmica*, Vol. 2, 1987, pp. 153-174.
- [23] A. Fournier and D. Y. Montuno, "Triangulating simple polygons and equivalent problems", *ACM Trans. on Graphics*, Vol. 3, No. 2, 1984, pp. 153-174.
- [24] W. H. Frey and J. C. Cavendish, "Fast planar mesh generation using the Delauney triangulation", Tech. Report No. GMR-4555, General Motors Research Labs, Michigan, 1983.
- [25] J. Friedman, J. Hershberger, and J. Snoeyink, "Compliant Motion in a simple polygon", in *Proc. 5th Annual Symposium on Computational Geometry*, Saarbruchen, West Germany, 1989, ACM Press, pp. 175-185.
- [26] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a simple polygon", *Inform. Process. Lett.*, Vol. 7, No. 4, 1978, pp. 175-180.
- [27] A. Goldberg and D. Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley, 1983.
- [28] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters*, Vol. 1, 1972, pp. 132-133.
- [29] R. L. Graham and F. F. Yao, "Finding the convex hull of a simple polygon", *Journal of Algorithms*, Vol. 4, 1983, pp. 324-331.
- [30] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, "Linear time algorithms for visibility and shortest path problems inside simple polygons", *Algorithmica*, Vol. 2, No. 2, 1987, pp. 209-233.
- [31] L. J. Guibas and J. Hershberger, "Optimal shortest path queries in a simple polygon", in *Proc. 3rd Annual Symposium on Computational Geometry*, Waterloo, Ontario, 1987, ACM Press, pp. 50-63.
- [32] S. Hertel and K. Mehlhorn, "Fast triangulation of a simple polygon", in *Proc. Conf. Found. of Comput. Theory*, New York, 1983, Springer-Verlag, pp. 207-218.
- [33] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R. Tarjan, "Sorting Jordan sequences in linear time using level-linked search trees", *Inform. and Control*, No. 68, 1986, pp. 170-184.
- [34] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists", *Acta Informatica*, Vol. 17, 1982, pp. 157-184.
- [35] I. Imafuku, Y. Kodera, M. Sayawaki, and M. Kono, "A generalized automatic mesh generation scheme for finite element method", *International Journal for Numerical Methods in Engineering*, Vol. 15, 1980, pp. 713-731.

- [36] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane", *Information Processing Letters*, Vol. 2, 1973, pp. 18-21.
- [37] J. M. Keil, "Decomposing polygons into simpler components", *SIAM J. Comput.*, Vol. 14, 1985, pp. 799-817.
- [38] J. M. Keil and J.-R. Sack, "Minimum decompositions of polygonal objects", in *Computational Geometry*, North Holland, 1985, pp. 197-216.
- [39] D. Kirkpatrick, "Optimal search in planar subdivisions", *Siam J. Comput.*, Vol. 12, 1983, pp. 28-35.
- [40] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 1: Fundamental algorithms, Addison-Wesley, 1973.
- [41] D. T. Lee, "Visibility of a simple polygon", *Computer Vision, Graphics, and Image Processing*, Vol. 22, No. 2, 1983, pp. 207-221.
- [42] D. Maier and S. C. Salveter, "Hysterical B-Trees", *Information Processing Letters*, Vol. 12, No. 4, 1981, pp. 199-202.
- [43] K. Mehlhorn, *Data structures and efficient algorithms, Volume 1: sorting and searching*, Springer-Verlag, 1984.
- [44] K. Mehlhorn, *Data structures and efficient algorithms, Volume 1: sorting and searching*, Springer-Verlag, 1984, pp. 185.
- [45] A. A. Melkman, "On-line construction of the convex hull of a simple polygon", *JPL*, Vol. 25, 1987, pp. 11-12.
- [46] B. Meyer, *Object-oriented software construction*, Prentice Hall, 1988.
- [47] D. Nguyen, *Course project on a three-dimensional editor for the computational geometry workbench*, Course 95.508, Carleton University, 1990.
- [48] J. O'Rourke, H. Booth, and R. Washington, "Connect-the-dots: a new heuristic", *Comput. Vision, Graphics, Image Process.*, Vol. 39, No. 2, 1987, pp. 258-266.
- [49] R. J. Pooley, *An introduction to programming in SIMULA*, Blackwell Scientific, Oxford, 1986.
- [50] F. P. Preparata, "A new approach to planar point location", *SIAM J. Comput.*, Vol. 10, No. 3, 1981, pp. 473-482.
- [51] F. P. Preparata and M. I. Shamos, *Computational Geometry: An introduction*, D. Gries (Ed.), Texts and monographs in computer science, Springer-Verlag, 1985.
- [52] F. S. Roberts, *Applied Combinatorics*, Prentice-Hall, 1984.
- [53] P. Rosenstiehl, "Planar Permutations Defined by Two Intersecting Jordan Curves", in *Proc. Graph Theory and Combinatorics: Proceedings of the Cambridge Combinatorial Conference in Honour of Paul Erdős*, Cambridge, England, 1984, B. Bollóbas (Ed.), Academic Press, pp. 259-271.
- [54] J.-R. Sack, "Rectilinear computational geometry", PhD Thesis, McGill, 1984.
- [55] B. Schachter, "Decomposition of polygons into convex sets", *IEEE Trans. Comput.*, Vol. 27, No. 11, 1978, pp. 1078-1082.
- [56] A. A. Schoone and J. van Leeuwen, "Triangulating a star-shaped polygon", Tech. Report No. RUU-CS-80-3, University of Utrecht, 1980.
- [57] M. I. Shamos, "Computational geometry", PhD Thesis, Yale, 1978.

- [58] J. J. Shilling and P. F. Sweeney, "Three steps to views: extending the object-oriented paradigm", in Proc. *OOPSLA '89*, New Orleans, 1989, N. Meyrowitz (Ed.), pp. 353-361.
- [59] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees", *Journal of the ACM*, No. 32, 1985, pp. 652-686.
- [60] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1986.
- [61] S. Suri, "A linear time algorithm for minimum link paths inside a simple polygon", *Computer Vision, Graphics and Image Processing*, Vol. 35, 1986.
- [62] S. Suri, "Minimum link paths in polygons and related problems", PhD Thesis, The John Hopkins University, 1987.
- [63] R. E. Tarjan, "On the efficiency of a good but not linear set merging algorithm", *Journal of the ACM*, Vol. 22, No. 2, 1972, pp. 215-225.
- [64] R. E. Tarjan, *Data structures and network algorithms*, CBMS-NSF regional conference series in applied mathematics, 44, SIAM, 1983, pp. 49-53.
- [65] R. E. Tarjan, "Amortized computational complexity", *SIAM J. Algebraic and Discrete Methods*, Vol. 6, 1985, pp. 306-218.
- [66] R. E. Tarjan and C. J. van Wyk, "An $O(n \log \log n)$ -time algorithm for triangulating a simple polygon", *SIAM J. Computing*, 1988, pp. 143-178.
- [67] G. T. Toussaint, "A new linear algorithm for triangulating monotone polygons", Tech. Report No. SOCS-83.9, McGill University, 1983.
- [68] J. M. Vlissides and M. A. Linton, "Applying object-oriented design to structured graphics", in Proc. *USENIX 1988 C++ Conference*, 1988, pp. 81-94.
- [69] Actor language manual, The WhiteWater Group, 1987.
- [70] T. C. Woo and S. Y. Shin, "A linear time algorithm for triangulating a point-visible polygon", *ACM Trans. on Graphics*, Vol. 4, 1985, pp. 60-69.

APPENDIX A

DETAILED DESCRIPTION OF THE WORKBENCH USER INTERFACE

A.1 USING THE WORKBENCH

The main tool for user interaction with the geometric workbench is through the Geometric Object Browser. This is a window in which the user may create, edit and remove small-scale geometric object. Algorithms can be applied to particular objects and the results viewed (possibly in animated form). Fig. A.1 shows an example of a browser window. An important feature of the workbench is the independence of the user interface and the geometric classes, allowing this user interface to be significantly changed without affecting the algorithmic portions of the workbench.

Fig. A.1 An example geometric object browser, showing a polygon and a planar straight line graph. The polygon (highlighted) is selected, and the workbench menu is active.

A.1.1 Geometric Browser Concepts

The browser operates using a few simple concepts which should be explained before proceeding further. The most fundamental of these is the concept of a *geometric object*. These correspond directly to the objects on which geometric algorithms operate. Examples are points, line segments, polygons and planar subdivisions.

The browser maintains a collection of these geometric objects, and all operations take place on these objects. At any given time, one of these objects may be *selected*. Most operations in the browser apply to the object which is selected, also referred to as the *current object*. Objects can be manipulated using menu operations, keystroke sequences and mouse activities. The particular effects of each action may depend on the current *mode* of the browser. The possible modes are described in Fig. A.2. Menu options do not change with the mode, but one menu varies with the current object, describing the algorithms applicable to it.

When an algorithm is executed it can produce an animation of the results if desired. The most recent animation can also be replayed or viewed frame by frame. Construction of the animation is the responsibility of the implementor of an algorithm, although tools are available.

The workbench communicates in a limited manner with other applications through the Macintosh clipboard. Objects can be stored in the Macintosh clipboard both as pictures and in a text format from which the object may be reconstructed.

A.1.2 Geometric Browser Panes

The geometric object browser window consists of three panes. At the top is a *status pane* which can display various information about the current state of the browser. At the bottom is a *button pane*, which indicates the current mode of operation and allows the user to select different modes. In the centre, occupying most of the window is a *geometric*

object pane. It is in this central pane that most browser activity takes place and that geometric objects are created, viewed and modified.

A.1.2.1 Status Pane

The status pane indicates the current status of the browser. In the current system, this pane displays the name of the current object, but the pane remains available for additional information which may be required in the future. This is a display only pane and has no associated menu or actions.

A.1.2.2 Button Pane

The button pane indicates the current browser mode. The mode may be changed by pressing the appropriate button with the mouse. The following table briefly describes each mode.

Edit:	Allows a component to be selected and translated. Note that while in this mode, Shift-Click will select a component and allow detailed examination.
Select:	This is the usual mode of operation. While in this mode an object can be selected at a vertex, moved around the browser and have algorithms applied to it. Clicking on the Select button while in this mode will select another object in the browser. This effect can also be achieved using the space bar.
Create:	On pressing this button a pop-up menu will request one of the possible objects to be created. Not all geometric objects can be created directly in the browser, as some will appear only as the output of an algorithm. While in create mode, each click in the geometric object pane will add a vertex to the new object. Clicking on any other button or selecting an algorithm from the menu will end the creation process. In creation of a polygon it is not necessary to join the first and last points as this will be done automatically upon completion. Fig. A.3 illustrates the process of polygon creation.
Add Part:	This mode allows components to be added to the current object.
Delete Object:	This will delete the current object. Confirmation is requested, and the undelete function will recover the object if necessary. The delete key acts as a keyboard shortcut for this operation.
Delete Part:	This mode allows components to be deleted from the objects in the browser.

Fig. A.2 A summary of the different browser modes

Fig. A.3 A geometric object browser in the process of creating a polygon.

A.1.2.3 Geometric Object Pane

This pane is where most of the browser activity takes place. It is here that geometric objects are created, displayed, and moved, and that the results of algorithms are displayed. Most of what is described in this section refers directly or indirectly to activities in this pane.

A.1.3 Geometric Browser Menus

There are four extra menus used in the geometric object browser, in addition to those normally available in Smalltalk. Two of these, the **Edit** (See Fig. A.4) and **Workbench** (See Fig. A.5) menus are always present and provide basic workbench operations. The **Debug** (See Fig. A.7) menu provides additional facilities for algorithm development and can be removed when the system is not being used for algorithm development. The **Object** menu provides a list of algorithms available for the currently selected object. This changes whenever a new class of object is selected and disappears if there is no selected object. An example browser is shown in Fig. A.6.

Edit Menu

Undelete:	The most recently deleted object is replaced in the browser and made current.
Cut:	Remove the current object from the browser and place it in the workbench clipboard. Copy a picture of the object to the Macintosh clipboard.
Copy:	Copy a picture of the current object to the Macintosh clipboard, and the object itself into the workbench clipboard.
Paste:	Insert into the browser and make current the contents of the workbench clipboard. Note that it is possible to cut and paste between different geometric browsers in this way, but we do not yet support pasting into a browser from another Macintosh application via the clipboard.
Clear:	Empty the workbench clipboard and delete buffer.
Copy All:	Copy pictures of all objects in the browser to the Macintosh clipboard.

Fig. A.4 The entries in the browser Edit menu

Workbench Menu

Send to Back:	Objects in the browser are ordered. An ambiguous selection will always be considered to refer to the object closest to the front. This command moves the current object to the back of the object list.
Magnify:	Makes a copy of the current object at 100% magnification. See Fig. A.6.
Shrink:	Makes a copy of the current object at a 50% reduction.
Duplicate:	Makes a copy of the current object.
Get Info:	Gets information on the current object. Currently this opens an inspector.
Preferences:	Allow the user to adjust system parameters.
FileIn...:	File in an object from a file selected by the user and make the new object current. The file must be in the correct fileout format for some geometric object class or an error will result.
FileOut...:	File out the current object onto a file selected by the user.

Fig. A.5 The entries in the browser Workbench menu

Fig. A.6 A geometric object browser showing several copies of an object at different stages of magnification and the workbench menu

Debug Menu

Check algorithms:	Toggles checking of algorithms on/off. This is a global flag.
Animate:	Toggles algorithm animation on/off for this browser.
Replay:	Replay the last animation in this browser.
Step:	Step through the last animation. Each mouse click advances one more step.
Redraw:	Redraw the main browser pane.
Graphic Inspect:	Inspect the current object.
Normal Inspect:	Open a standard Smalltalk inspector on the object (no graphical representation). This is most useful when the object is in an unstable state and an attempt to produce the graphical representation will cause an error.
Inspect Animation:	Inspect the most recent animation.

Fig. A.7 The entries in the workbench debug menu

Object Menu

The title of this menu, as it appears in the Macintosh menu bar, is variable. It is referred to as the Object menu because its contents are determined by the current object. This menu

contains a list of all algorithms available for the current object, and performs whichever of them is selected, possibly animating the result. The menu is not present if no object is selected, and its title and contents change with the current object. Note that certain items may be grayed out on this menu, indicating that while they are generally available for this class of object, they are inappropriate for this instance.

END

1 2 0 4 9 1

FIN