An abstract of the thesis of Jay Nave for the degree of Master of Science in Computer Science presented on August 27, 1985.

Title: Computational Geometry Package with Fast Voronoi Diagram Algorithm

Abstract approved: Redacted for Privacy

Alan Coppola

An interactive Computational geometry package was developed for the purpose of experimenting with geometry problems in the Euclidean plane. The package also contains computer graphics functions to display the result. Application independent functions were developed that are both flexible and general enough for creating new geometry experiments as well as being portable to other hardware facilities.

A specific goal was to develop the algorithm and functions needed to construct the Voronoi diagram, a geometric construct of particular importance, within $O(N \log N)$ time, an improvement over previous $O(N^2)$ methods. The faster solution for this construction of the Voronoi diagram enables faster solutions to a wide range of Computational geometry problems.
Computational Geometry Package With Fast Voronoi Diagram Algorithm

by

Jay Nave

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed October 1, 1985
Commencement June 1986
Approved:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented August 27, 1985
TABLE OF CONTENTS

1 Introduction 1
2 Formulation of the Merge Algorithm 5
3 Data Structures 14
4 Computational Geometry Functions 25
5 Experiment Using Computational Geometry Package 33
6 Next Experiment Planned Using Thesis Software 35
7 Simple Graphics Package 36
8 Bibliography 37
9 Appendix 39
LIST OF FIGURES

2.1 Merge process ................................................. 8
2.2 Convex hull boundary ......................................... 10
2.3 Line \((c_0, L)\) .................................................. 11

3.1 Voronoi points array .......................................... 16
3.2 Voronoi polygon vertex node ................................. 16
3.3 Vertex list array .............................................. 17
3.4 Voronoi diagram I ............................................. 18
3.5 Merge intersection ........................................... 19
3.6 Voronoi polygon(I) ........................................... 20
3.7 Data structure for Voronoi diagram I ......................... 21
3.8 Data structure for Voronoi polygon vertices ................. 22
3.9 Merge curve edge intersection pointers ...................... 23
3.10 Merge curve structure ....................................... 24

4.1 Graham scan data .............................................. 28
4.2 Locate new edges ............................................. 30
4.3 Locate new edges 4 point case ............................... 31
4.4 Line segment - polygon intersection ......................... 32

5.1 Table 5.1 ......................................................... 34
Preface

Pages i. - vi. are sample output from the thesis software. Pages i. and ii. are each Voronoi diagrams of 50 points. These two sets are then merged to form the diagram on page iii. Likewise pages iv. and v., diagrams of 150 points each are merged to form the Voronoi diagram on page vi. Output was obtained by using a TEKTRONIX 4113 terminal and a TEKTRONIX 4662 plotter.
Choose a function by typing its first letter

- p = DISP_POINTS
- h = DISP_HULL
- u = DISP_VOR_DGRM
- d = DONE
Choose a function by typing its first letter:

p = DISP_POINTS
h = DISP_HULL
u = DISP_UOR_DGRM
d = DONE
Choose a function by typing its first letter:

- p = DISP_POINTS
- h = DISP_HULL
- u = DISP_VOR_DGRM
- d = DONE
CHAPTER 1

INTRODUCTION

The purpose of this thesis is to describe the implementation of the computational geometry algorithms needed to create a robust geometric graphics system. Functions to compute the Voronoi diagram, a geometric structure of special importance, are implemented in $O(N \log N)$ time, an improvement over the previous $O(N^2)$ bound for this computation. This faster algorithm was proposed by Michael Shamos (PhD. Thesis Yale 1978[SHAMOS(11c)]).

The software is implemented in the C programming language within the Berkeley 4.2 Unix operating system on a VAX - 11 model 750. It is currently compatible with any Tektronix 4100 series graphic display device.

Experimental results show that the developed prototype software performs in $\frac{1}{35} N \log N$ time.

Written in C for portability, it also implements what is called the Simple Graphics Package(SGP) software as prescribed by [FOLEY(3a)] according to Core Graphics System standards[SNGSC(12)][BERGERON(2)]. These Core system standards were developed to promote portability of graphics software. The entire program is comprised of about 3000 lines of C code but to move this package to another environment supporting the same language and operating system would require changing only about 200 lines of code (code that generates the display device processor instructions). This portability was an important goal as we hope to share this prototype with interested users.

Previous work to implement Shamos’s Voronoi diagram algorithm was done by Younsu Kim as part of her Master’s Thesis work in 1983 at Oregon State University in Corvallis(working with Dr. Alan Coppola). Due to the difficulty of keeping polygon edge lists updated through the merging process as well as some mistakes in details of the theoretical analysis by Shamos(see Chapter 4) the implementation was never effective for more than a few points. However, Younsu Kim’s work was motivational, both in providing a starting reference, as well as in helping to develop new approaches that avoided the difficulties in her implementation.

This software is part of a contemplated, larger more comprehensive in-
teractive geometric graphics system. This final system will consist of three parts: an already developed simple graphics package for displaying results, a mostly developed geometric graphics package, and a planned but not implemented, heuristic geometry package. The overall goal of this geometric graphics system is to prove and disprove conjectures through interactive experimentation; i.e. design an environment where an expert user can graphically display and interact with problems and their solutions.

Computational geometry is a subject concerned with the algorithmic aspects of geometrical problems as well as the associated data structures. The usual objects of concern (for planar models) are points, lines, line segments, polygons, and planar subdivisions such as bisectors and collections of these objects. We are often concerned with the intersection of these objects (line - line, polygon - polygon, general clipping), point location (point - point, point - polygon...) and more. It is the increasing use of geometric objects within computer graphics applications and computer aided design that has generated an increased interest in computational geometry.

D.T. Lee and F.P. Preparata have recently surveyed the state of the art of Computational Geometry\(^1\). Areas of this subject within this thesis include convex hulls, intersection of geometric objects and proximity problems.

Geometric models can simulate physical realities. Often we want to study the spatial relationship between objects particularly in terms of their proximity to one another.

A number of such problems of planar geometry can be solved by construction of the Voronoi diagram[\textsc{Voronoi}(13)]. This geometrical structure can be constructed in \(O(N \log N)\) time and can then be used to solve other problems such as Euclidean minimal spanning tree \(^2\) within the same time order.

In other words the development of the Voronoi diagram enables the solutions to many basic problems of computational geometry within this new, faster time bound of \(O(N \log N)\) where \(N\) equals the number of points considered. A summarised list of some of the basic problems that can now be more quickly solved includes: closest pair of points, all nearest neighbors of a point, Euclidean minimal spanning tree, Delauney triangulation of \(N\) points, nearest neighbor and \(k\) nearest neighbor searching.

Therefore construction of the Voronoi diagram is the key geometrical algorithm for computing proximity problems and as such is included as a function in the computational geometry package.

This thesis implements some computational tools that can substitute for classical Euclidean geometry constructions. \(O(N \log N)\) algorithms are available for a creating computational models of the convex hull and the Voronoi diagram


\(^2\) Given \(N\) points in the plane, the Euclidean minimal spanning tree (Emst) is an interconnected tree of minimum total length using the Euclidean Norm as defined in Chapter 2, page 7.
for a finite set of points in the $R^2$ plane. These tools are the main goal of this work since many pieces of the geometry package easily follow. These algorithms are developed in a functional manner with global data structures to allow their extended application to a wide variety of theoretical and applied problems.

Euclidean geometry constructions use straight edge and compass as tools. Cognition, imagination and perception are used to discover construction algorithms that yield succinct solutions to problems. These constructions are not available to digital computer models.

It is through techniques of algebra and calculus that one can construct algorithms to replace classical methods of construction so that again succinct and computable solutions are available to the user. The growth of real analysis has resulted in metric geometry and convexity theory which are exploited to create computational geometry tools.

Construction using compass and straight edge of the Voronoi diagram for say one hundred distinct points would be time consuming and wearing on the geometer (i.e., computer solutions are most suited for these kinds of repetitious calculations). However, once we desire computerized solutions, we are faced with a long list of decisions as to how we can replace such classical methods.

Let us consider the problem of finding the convex hull of a set of N points say $S = \{a_0, a_1, ..., a_{N-1}\}, a_i \in R^2$. We can usually eyeball the solution for small sets of points but it is not readily apparent as to how to set up this problem for computerized solution. What is the computational version of the problem we are trying to solve? How do we represent points, lines, convex polygons within the computer? How do we analyze the efficiency of our algorithm?

Since we are searching for the convex hull we might try directly employing definitions for the convex hull. One such definition states that a set is convex if and only if each line segment joining any pair of points in S lies wholly within S. We also know that $S'$, the convex hull, is the smallest convex set out of all convex sets that contain S and we can imagine the shortest path that surrounds all the points of S. We can even create an algorithm based on this perception of shortest path, the Rubber Band Algorithm. However should we try to use these definitions directly we would have an infinite number of possible calculations to perform which is clearly unacceptable for actual algorithms.

Here we see, at least as far as is currently known, that computational geometry cannot effectively utilize this particular cognitive method.

An approach that moves us closer to a solution is based on the fact that the hull boundary of a finite set of points can be represented by a set of successive edges that are the boundary of the given point set. Since this boundary contains the convex hull which is the smallest convex set containing S, finding the vertices, called extreme points, representing the edge list will solve our problem.

This approach will eventually succeed as in the actual program imple-

---

mentation although there are still more problems we must face before we can actually formulate a machine do-able solution. We must often reconceive ideas that a child might employ to solve certain problems.

In the remaining pages of this document we discuss in chapter 2: a formulation of the O(N) merge algorithm that makes the O(NlogN) algorithm for constructing the Voronoi diagram realizable and some related definitions; chapter 3: those data structures created to model the Voronoi diagram and its related functions; chapter 4: a discussion of those functions whose creation was unique to the theory being implemented as well as a discussion of related software implementation problems; chapter 5: Experiment to verify O(NlogN) nature of implemented algorithm to find the Voronoi diagram; chapter 6: a currently planned experiment; chapter 7: an overview of the simple graphics package as designed for and implemented in the thesis software.
CHAPTER 2

FORMULATION OF THE MERGE ALGORITHM

Basic Definitions

We begin the problem formulation by establishing some definitions with which we will develop the divide and conquer algorithm for creating the Voronoi diagram, a key data structure which solves many computational geometry problems in linear time. We wish to construct an algorithm with $O(N \log N)$ running time to compute the Voronoi diagram.

We shall develop this algorithm in the Euclidean Norm of $\mathbb{R}^2$ or $E_2$ where $a_i = (x_i, y_i)$ and $a_j = (x_j, y_j)$ and $a_i, a_j \in \mathbb{R}^2$. In this metric space the distance function $(d_2)$ is defined as follows:

Definition 1

$$d_2(a_i, a_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

(2.1)

It should be noted that distance functions for other metric spaces can be formulated by replacing 2 with $p$ in all of these definitions. Other metrics of particular interest are the $R^1(L_1)$ metric and the $R^3(E_3)$ metric. See [LEE(6)(7)] for discussion of geometric algorithms in these metrics.

Definition 2 the bisector of distinct points $a_i, a_j$, the locus of the points equidistant from $a_i$ and $a_j$ is defined as:

$$B_2(a_i, a_j) = \{r \mid r \in \mathbb{R}^2, d_2(r, a_i) = d_2(r, a_j)\}$$

(2.2)

where $a_i = (x_i, y_i)$ and $a_j = (x_j, y_j)$

Let $A$ be a set of $N$ points in $\mathbb{R}^2$. Say $A = \{a_0, a_1, ..., a_{N-1}\}$ for $i, 0 \leq i \leq N - 1, a_i \in \mathbb{R}^2$
Definition 3

\[ VOR(a_i) = \{b \mid d_2(a_i, b) \leq d_2(a_j, b) \ \forall j \} \]  

(2.3)

\[ VOR(a_i) \text{ is called the Voronoi region of point } a_i \text{ with respect to } A. \]

Let \( a_i, a_j \in \mathbb{R}^2 \) then

Definition 4

\[ HP(i, j) = \{b \in \mathbb{R}^2 \mid d_2(a_i, b) \leq d_2(a_j, b)\} \]  

(2.4)

For integers \( i, j, \ 0 \leq i, j \leq N - 1 \). \( HP(i, j) \) represents the half-plane, containing \( a_i \), defined by the bisector of the line segment between \( (a_i, a_j) \), denoted \( \text{seg}(a_i, a_j) \).

A region \( S \subseteq \mathbb{R}^2 \) is convex if for all points \( p, q \in S \) the entire line segment \( \text{seg}(p, q) \) is contained in \( S \).

Clearly, \( VOR(a_i) = \cap HP(i, j) \) with \( j \neq i \). Thus \( VOR(a_i) \) is a convex polygonal region since every halfplane is convex and the intersection of convex sets is convex.

We can now define the Voronoi diagram:

Definition 5 The Voronoi diagram, \( VD(A) \) of \( A \) is the union of the set of edges and vertices of the boundaries of Voronoi regions \( VOR(a_i), 0 \leq i \leq N - 1 \). Each open set that results from taking the complement of the Voronoi diagram has a Voronoi region \( VOR(a_i) \) as its boundary. These regions can be both bounded and unbounded.

Let set \( T \subseteq \mathbb{R}^2 \). The convex hull, \( CH(T) \) is the intersection of all convex sets containing \( T \), i.e. \( CH(T) = \cap \{S \mid T \subseteq S \text{ and } S \text{ is convex}\} \)

Since the intersection of a family of convex sets is convex, the convex hull \( CH(T) \) is a convex set and is the smallest (re: set inclusion) convex set containing \( T \). If \( T \) is a finite set then the convex hull of \( T \) is a convex polygonal region.

The goal of the next section is to prove that the Voronoi diagram of \( N \) points can be computed in time \( O(N \log N) \).

Theorem 1 Let \( A \subseteq \mathbb{R}^2 \) be given, \( N = |A| \). Then the Voronoi diagram of \( A \) can be computed in time \( O(N \log N) \).

The algorithm is based on the divide-and-conquer strategy[AHO(1)]. Since our goal is to show an \( O(N \log N) \) algorithm we will assume the input set \( A \) is sorted lexicographically(by x coord and if necessary by y coord). Now let \( A_L \) be the first half of sorted set \( A \), \( |A_L| = \lfloor N/2 \rfloor \), and let \( A_R \) be the second half of set \( A \), \( |A_R| = \lceil N/2 \rceil \). Next assume inductively, that we have been able to construct the Voronoi diagrams \( VD(A_L) \) and \( VD(A_R) \) of sets \( A_L \) and \( A_R \) by applying our algorithm recursively. This will take time
\[ T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) \] (2.5)

Note that \( T(1) = O(1) \) as the Voronoi diagram of one point is trivial. \( T(2) = O(1) \) as it requires the construction of a single bisector. We would like to be able to construct \( \text{VD}(A) \) from \( \text{VD}(A_L) \) and \( \text{VD}(A_R) \) by "merging" the data in time \( O(N) \) where \( N = \text{number of set points in VD(A)} \). This will take time

\[ T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N) \] (2.6)

and will therefore demonstrate that our algorithm performs in time \( O(N\log N) \).

Figure 2.1 is an example of such a merging process.

In the illustration \( \text{VD}(A) \) is shown as a SOLID line, parts of \( \text{VD}(A_L) \) and \( \text{VD}(A_R) \) which do not become part of \( \text{VD}(A) \) are shown as dotted lines. Merge curve MC (defined below) represents a sequence of bisectors which must be constructed in order to merge the two disjoint subsets \( A_L \) and \( A_R \). It belongs to \( \text{VD}(A) \) but is part neither of \( \text{VD}(A_L) \) nor \( \text{VD}(A_R) \). MC is drawn with a thick line. \( A_L = \{0, 1, 2, 3\} \) and \( A_R = \{4, 5, 6, 7\} \).

Creating line MC is the critical construction of the merging process.

Definition 6 \( MC = \{ b \in \mathbb{R}^2, d_1^*(b, A_L) = d_2^*(b, A_R) \text{ where } d_1^*(b, T) = \text{min}(d_2(a, b)), a \in T \} \)

Lemma 1

1. \( MC = \{ b \mid b \text{ lies on an edge of } \text{VD}(A) \text{ which is a bisector of some } a_i \in A_L \text{ and some } a_j \in A_R \} \). In particular, \( MC \) consists of two infinite rays(halflines) and some number of straight line segments.

2. \( MC \) is monotone, ie. \( MC \) can be directed such that no line segment or half line runs upward. That is the \( MC \) consists of a sequence of line segments and half lines spanning the sequence of points \( \{x_i, y_i\} \subset \mathbb{R}^2 \) with \( 0 \leq i \leq \infty \) where \( y_{i-1} \leq y_i \).

Proof of 1. and 2.:

Let \( C \) be the set defined in part 1) of the Lemma. Then clearly \( C \subseteq MC \). Now we show \( MC \subseteq C \).

Let \( b \in MC \) be arbitrary. Then \( \exists a_i \in A_L \) and \( a_j \in A_R \) such that \( d_2(b, a_i) = d_2(b, a_j) \leq d_2(b, a) \forall a \in A \). Thus \( b \) lies on the edge which separates the Voronoi regions \( VOR(a_i) \) and \( VOR(a_j) \) and hence \( b \in C \).

Therefore \( MC = C \).
Figure 2.1: Merge process
In particular, $MC$ consists of a set of line segments. Every line segment is a bisector $B_2(a_i, a_j)$ for some $a_i \in A_L$ and $a_j \in A_R$. Direct the line segment such that $a_i$ is to the right of the line segment ($x_{a_i} \leq x_{a_j}$). Then no line segment is directed upwards, because this would imply that the $x$ coordinate of $a_i$ is larger than the $x$ coordinate of $a_j$, a contradiction.

Since $A$ is lexicographically ordered, we may even conclude that there is at most one horizontal line segment (which must then be directed left to right). We claim there can be no more than one horizontal line segment in the merge curve, and we prove this by contradiction. Assume there is more than one horizontal line segment composing the merge curve $MC$ for the current merge process. Then there must exist at least two different sets of points where each set contains two vertices: $a_i$ from subset $A_L$ and $a_j$ from $A_R$, those subsets being merged. But this means that there are at least these four vertices whose order in all cases is inter mixed between the subsets $A_L$ & $A_R$ contradicting that $A_L \cap A_R = \emptyset$.\[\square\]

Now since the merge curve is monotone decreasing, it cannot be a closed curve. Therefore it must consist of two infinite rays and some number of (finite) line segments.

Lemma 1 characterizes the merge curve $MC$. However the significance of $MC$ follows from Lemma 2.

**Lemma 2** Let $MC$ be as defined above. Direct $MC$ in order of decreasing $y$ coordinate values and let $MC_L$ be the region of the plane to the left of $MC$. Likewise let $MC_R$ be the region of the plane to the right of $MC$. Then

\[VD(A) = ((VD(A_L) \cap MC_L) \cup MC \cup (VD(A_R) \cap MC_R))\] (2.7)

Proof: Let $VD$ be the set defined by the expression on the right hand side of the equation. We prove $VD(A) = VD$ by showing $VD(A) \subseteq VD$ and $VD \subseteq VD(A)$.

To Show $VD(A) \subseteq VD$:

Let $b$ be an element of $VD(A)$, i.e. $b$ lies on an edge of $VD(A)$. Then there exists $i, j$ such that $d_2(b, a_i) = d_2(b, a_j) \leq d_2(b, a) \forall a \in A$. If $i, j \in A_L$ then $b \in VD(A_L) \cap MC_L$, if $i \in A_L, j \in A_R$ or vice-versa then $b \in MC$ and if $i, j \in A_R$ then $b \in VD(A_R) \cap MC_R$.

To Show $VD \subseteq VD(A)$:

Let $b \in VD$. If $b \in MC$ then $b \in VD(A)$ by Lemma A. Next assume that $b \in VD(A_L) \cap MC_L$. Since $b \in MC_L$, $d_2(b, A_L) < d_2(b, A_R)$ and since $b \in VD(A_L)$ there are $i, j \in A_L \ni d_2(b, A_L) = d_2(b, a_i) = d_2(b, a_j)$. Therefore $b \in VD(A)$ by definition. similarly if $b \in VD(A_R) \cup MC_R$.\[\square\]

Now using Lemma 2 we infer that the construction of merge curve $MC$ basically solves the problem of merging diagrams $VD( A_L)$ and $VD(A_R)$.

Now we have described the merge curve but we have yet to give an actual algorithm for its construction, to which we next proceed.
We wish to construct MC in order of decreasing y coordinate values. Our first task then is to construct the upper infinite ray $U$ of merge curve MC. First we consider the convex hull of each subset $A_L$ and $A_R$ the current sets being merged. Then $CH(A) = CH(A_L \cup A_R)$.

We denote the boundary of the convex hull $CH(A)$ as $BCH(A)$ and note also that these edges of the boundary can be defined as an ordered list of vertices. These vertices are called extreme points of the convex hull. $BCH(A_L)$ and $BCH(A_R)$ have some edges (vertices) in $BCH(A_L \cup A_R)$ but also there are two "new" or additional edges in $BCH(A_L \cup A_R)$ which we can name the "upper" and "lower" new BCH edges in relation to the monotonic decreasing merge curve MC. This is depicted in figure 2.2.

As discussed in chapter 5, we can locate these new hull edges of $BCH(A_L \cup A_R)$ and using the upper edge we can form its bisector $B_2(a_i, a_j)$ where $a_i \in A_L$ and $a_j \in A_R$ and $a_i \in BCH(A_L)$ and $a_j \in BCH(A_R)$. The upper halfline or ray of this bisector is the upper infinite ray of those rays and segments composing MC. We create variables $upperhull$ and $lowerhull$ to hold those pairs of vertices ($a_i \in A_L$ and $a_j \in A_R$) that represent these two new hull edges.

Before proceeding further let us characterize MC more carefully with $c_0, c_1$ representing the upper infinite ray of MC, $c_1, ..., c_{i-1}$ being the vertices of the line segments from the upper ray to the lower ray in decreasing order and $c_{i-1}, c_i$ representing the lower infinite ray. Therefore $MC = \{c_0, c_1, c_2, ..., c_{i-1}, c_i\}$ with $i \geq 1$ ($\{c_1, c_2, ..., c_{i-1}\}$ can be the empty set).

We have so far upper ray coordinates $c_0$ and $L = lowerhull$ coordinates of upper hull edge bisector $B_2(a_i, a_j)$ as in figure 2.3.

We introduce variables $curedgeleft$ and $curedgeright$ for storage of those
set indices of those $a_i \in A_L$ and $a_j \in A_R$ whose bisectors are used to form the successive rays and segments of MC. Variable $cindex$ is used to index merge curve vertices $c_0, c_1, c_2, ..., c_{i-1}, c_i$, as described above. Variable $curray$ represents the lower halfline or ray computed as the bisector of $\text{seg}(\text{curedgeleft}, \text{curedgeright})$. This ray will intersect with either a left or right Voronoi polygon.

Now we are ready to state the algorithm.
cindex ← 1;
curredgeleft ← upperhull a_i ∈ A_L;
curredgeright ← upperhull a_j ∈ A_R;
compute (c_0, L) ← Bi(curredgeleft, curedgeright);
curray ← c_0 L;

while not ((curredgeleft = lowerhull a_i) and (curredgeright = lowerhull a_j)) do begin
  find the intersections between curray and
  VOR(a_i) ∈ A_L and VOR(a_j) ∈ A_R;

  keep pointers to polygon edges of VOR(a_i)
  and VOR(a_j) where these intersections occurred.

  from all the discovered intersections of VOR(A_i) and VOR(A_j) by curray
  calculate where curray forms intersection points with these polygon edges;

  from among these candidate intersections choose the intersection
  point having the highest y coordinate.

  if(polygon edge of highest intersection point ⊆ VOR(a_i)) then begin
    a_k ← a_i;
    curedgeleft ← a_k;
  end
  else if(polygon edge ⊆ VOR(a_j)) then
    a_k ← a_j;
    curedgeright ← a_k;
  end

  c [cindex] ← found point of intersection;
  L ← point at negative infinity of Bi(curredgeleft, curedgeright);
  curray ← (c [cindex], L);
  cindex ← cindex + 1;
endwhile;

Compute the bisector of the lower hull a_i and lower hull a_j;
c_i = point of lower halfline located at negative infinity;
Now we proceed to show that this merging process can be accomplished in $O(N)$ time ($N$ equal number of points considered), as then it follows by the recurrence relation $T(N) = 2T(n/2) + O(N)$ that $T(N) = O(N \log N)$.

We claim the number of Voronoi polygon edges visited by this merging process is $O(N)$.

Proof lies in the fact that the merge curve is monotone decreasing in the y coordinate (by Lemma 1) and therefore once some current edge index $a_i$ or $a_j$ is replaced that $VOR(a_i)$ or $VOR(a_j)$ will never again be visited during the current merge process.

For each construction of a merge curve segment there will occur up to four polygon intersections, a maximum of two for a particular $VOR(a_i)$ or $VOR(a_j)$ for each merge curve segment constructed. But since there is no backtracking to any such vertex (once the merged curve has passed by it) there will be no more than $O(N)$ such vertices considered in the worst case and therefore the merge process is $O(N)$. $\square$
An important area of concern for constructing a robust geometry package is that of choosing the appropriate data structures. We represent our computational geometry data structures schematically by use of spatial diagrams. Only those data structures judged to need illustration and relevant to geometric modeling are discussed, the rest can be examined within the software.

We begin with discussion of a basic geometry concept, the point. We choose to represent points using homogeneous coordinates for reasons we discuss in the next section.

**Homogeneous Coordinates**

The implemented computational geometry package utilizes what are termed homogeneous coordinates[FOLEY(3b)]. There are extensive benefits derived from the use of this data representation. We give some definitions and a short discussion of this type of coordinate system.

**Definition 7** Homogeneous coordinates for the point \( p \in \mathbb{R}^2 \) or \( \mathbb{E}^2 \) where \( p = (x^*, y^*) \) is represented by the triple \((x_p, y_p, w_p)\) where if \( w_p = 0 \) then \((x_p, y_p, 0)\) represents a point \( \in \mathbb{E}^2 \) located at infinity in the direction \((x_p, y_p)\) but in the common case \( w_p \neq 0 \).

Except in this infinite case the original coordinates \((x^*, y^*)\) can be recovered as \(x^* = x_p/w_p\) and \(y^* = y_p/w_p\). We say \((x^*, y^*)\) is congruent to \((x_p, y_p, w_p)\).

Clearly there exist an infinite number of homogeneous coordinates for a given unique Euclidean point such as \((x^*, y^*)\) as expressed by \(\alpha(x, y, w) = (\alpha x, \alpha y, \alpha w)\). Usually one chooses the \(w\) value to be equal to one since then \((x, y, 1) = (x/1, y/1, 1) = (x^*, y^*)\) and clearly we can just read the \(x^*\) and \(y^*\) values.

One of the immediate benefits of choosing this homogeneous method of representing points is its compositional compatibility with general two dimensional transformation matrices to rotate, scale or translate objects. In fact composition of such transformations is simply the product of the homogeneous point
and each transformation multiplied in the sequence desired (see [FOLEY(3b)] or [HARRINGTON(5)]).

A second benefit occurs when we must determine the intersection of two line segments including the infinite segments. As previously stated in the introduction, we are often concerned with the intersection of two geometric objects. One such intersection problem consists of intersecting the modeled object (a collection of points, lines and polygons) with the user defined display region known as the display "window" (see [FOLEY(3c)]). Usually this "window" is rectangular in shape although any polygonal area or volume could also be used. Cutting up the original model so it will fit into this display region is referred to as clipping. As detailed by Theo Pavlidis [PAVLIDIS(9)], such clipping can be partly performed using determinants of three by three matrices wherein combinations of three homogeneous points become the row vectors of the determinant matrix. The main idea is that two segments (and or rays) intersect if their determinants satisfy certain sign requirements. Since we use three points for each determinant it is desirable for each point to have three dimensions.

**Lines and Polygons**

Line segments and lines are represented as pairs of points. Polygons are also represented as ordered lists of points. In order to represent the polygons that would be needed by the Voronoi diagram algorithm it was clear that flexibility was important because of the dynamic and rapidly changing nature of these lists.

As the merging process is called recursively, each level of recursion requires the reconstruction and clipping of those involved Voronoi polygons. These polygons are continually reassociated with left and right subsets over the course of the process. Thus structures are required that are both flexible for insertion and deletion of polygon edges, as well as capable of supporting searches for merge curve and Voronoi polygon intersections from both left and right orientations.

The natural choice for the Voronoi diagram is the doubly linked circular list with each node representing a Voronoi polygon vertex record. This record will contain an index to some Voronoi edge vertex. Two doubly linked nodes will be used to represent an edge. This Voronoi point array is represented in figure 3.1. Thus a polygon is represented by a set of vertices which are linked in successive clockwise and counterclockwise order. Each node record also contains a clockwise link and a counterclockwise link (c1 and ccl) as illustrated in figure 3.2.

Before we illustrate a schematic diagram for an example Voronoi diagram we need to discuss the geometric data that such a structure represents.

Each Voronoi polygon is associated with some \( a_i \in A \) the set of points whose bisectors create the Voronoi diagram. Since the \( a_i \) are presented in a lexicographically increasing order we will use array vertexlist as an array of ordered

---

1 for a definition of a standard polygon order see [SHAMOS(11a)]
ARRAY VORONOI POINTS

\[
\begin{bmatrix}
0 & X_0 & Y_0 & W_0 \\
1 & X_1 & Y_1 & W_1 \\
\vdots \\
m-1 & X_{m-1} & Y_{m-1} & W_{m-1}
\end{bmatrix}
\]

Figure 3.1: Voronoi points array

VORONOI POLYGON NODE:

\[
\begin{bmatrix}
CV & CCV & UPINDX & CL & CCL \\
\end{bmatrix}
\]

WHERE
- CV = CLOCKWISE VERTEX
- CCV = COUNTER CLOCKWISE VERTEX
- UPINDX = VORONOI POINTS ARRAY INDEX
- CL = CLOCKWISE LINK
- CCL = COUNTER CLOCKWISE LINK

Figure 3.2: Voronoi polygon vertex node
vertices, a logical record containing the x, y, and w coordinates (see homogeneous coordinates) of the indexed \( a_i \) as well as a pointer to the polygon list for that particular \( a_i \). This is illustrated in figure 3.3.

We now discuss figure 3.4 as an instance of the Voronoi diagram geometric data structure. Notice that the Voronoi polygon for point three contains bisectors formed between points three and zero, three and one, three and two, three and four, three and five, and three and six.

Notice also that since we store our polygon edges as polygon vertex points (Voronoi points) that we have a family of two point sets associated with each Voronoi point (except infinite points) besides the set index of the given Voronoi polygon. In our figure this is illustrated by \( x \) which belongs to both bisector \( y, x \) and bisector \( x, w \), where \( y, x \in B_i(2, 3) \) and \( x, w \in B_i(3, 5) \).

We note that \( x, w \) moves clockwise from \( x \) to \( w \) whereas \( x, y \) moves counterclockwise from \( x \) to \( y \). We represent this information in our node record by recording the set point \( a_j \) used to construct the particular \( B_i(a_i, a_j) \) used to formulate the edge of the intersected Voronoi polygon \( VOR(a_i) \). If the edge was directed clockwise as is the case for \( seg(x, w) \) then we would record set point \( a_j = 5 \) as the clockwise vertex or (cv) for node record \( x \) whereas the counterclockwise vertex (ccv) for node \( x \) related to edge \( x, y \) is \( a_j = 3 \).

Let us proceed to show how this neighbor information is relevant to our algorithm.

Analysis of the merging process shows that as we intersect some bisector, either in the left subset or in the right we will switch the current edge vertex to the left or right index as given by the vertex associated with that portion of the bisector intersected (some \( a_i \in S, a_j \in S, S = A_L \) or \( A_R \)).

This situation is illustrated by figure 3.5. In this figure we have a typical
Figure 3.4: Voronoi diagram I
case where $B_i(a_0, a_2)$ intersects $B_i(a_0, a_1)$ which was formed during a previous recursive call to the function Voronoi diagram.

In this case vertex $a_0$ is switched to become vertex $a_1$ as $B_i(a_0, a_1)$ was the intersected edge. The next merge curve segment will be a subset of $B_i(a_1, a_2)$.

Therefore we want to keep information about these adjacent set vertices which taken together represent some portion of their mutual bisector. We give an example in figure 3.6 where the data structure for Voronoi polygon(1) is illustrated.

Now we are prepared to schematically represent the Voronoi diagrams data structure in figure 3.7 and 3.8.

After recursively forming the left and right Voronoi diagrams for subsets $A_L$ and $A_R$ we must next construct the merge curve i.e. the two halflines and the line segments(if any) between them (as in Lemma 1). However, we also need to store more than just these $c_i$'s of the merge curve.

The data tables must store the merge curve's intersection points with previously formed bisectors as well as the Voronoi polygon edges wherein these intersections occurred, as these data are later needed by the clipping process.

We've previously chosen to represent the Voronoi polygons as circularly linked lists with every node containing an index to a table of the Voronoi points created. Note however that polygons share edges which are some portion of a mutual bisector between some $a_i, a_j \in A$. Edge duplication is represented by duplicate addresses to polygon vertices.

Both the intersected edge found during the merge curve construction and also the adjacent and duplicated bisector must be saved during the merging process for the particular clipping of these intersected polygons. This is illustrated
THE POLYGON VOR(1) WOULD BE REPRESENTED AS

![Diagram]

Figure 3.6: Voronoi polygon(1)
Figure 3.7: Data structure for Voronoi diagram I
<table>
<thead>
<tr>
<th></th>
<th>X_a</th>
<th>Y_a</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>X_b</td>
<td>Y_b</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>X_c</td>
<td>Y_c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>X_d</td>
<td>Y_d</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>X_e</td>
<td>Y_e</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>X_f</td>
<td>Y_f</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>X_u</td>
<td>Y_u</td>
<td>1</td>
</tr>
<tr>
<td>v</td>
<td>X_v</td>
<td>Y_v</td>
<td>1</td>
</tr>
<tr>
<td>W</td>
<td>X_W</td>
<td>Y_W</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>X_x</td>
<td>Y_x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>X_y</td>
<td>Y_y</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.8: Data structure for Voronoi polygon vertices
by figure 3.9. Examining this figure we note that the merge curve segment $i$: $(c[i-1], c[i])$ intersects with seg(A,B) of Voronoi polygon(1). This is denoted the current intersect for the merge curve segment $i$. Likewise Voronoi polygon(2) is intersected and denoted the adjacent intersect for merge curve segment $i$.

This clipping consists of finding the intersection between the portion of the merge curve related to the polygons being clipped.

The portion of the merge curve related to the clipping of some Voronoi polygon is delimited by the time during which $VOR(a_i)$ is one of the vertices representing the current edge $a_i, a_j$ for which $Bi(a_i, a_j)$ is an element of the merge curve (see chapter 2).

The upper hull edge and the lower hull edge are also special cases. These special cases are easily flagged utilizing some of merge curve[i]'s storage.

The merge curve table may be schematically represented as in figure 3.10.
Figure 3.10: Merge curve structure
CHAPTER 4

COMPUTATIONAL GEOMETRY FUNCTIONS

Construction of the Voronoi diagram requires the use of all the computational geometry functions that have been so far encoded. Therefore in this chapter we concentrate on those geometric functions needed to construct the Voronoi diagram. In particular we discuss those functions that required further analysis and development than what is given by the more theoretical characterizations of the divide and conquer algorithm for the Voronoi diagram (as discussed in Chapter 2).

For example, during the construction of the mergecurve (as described in Chapter 2) we cannot simply observe and locate the new upper hull edge and the new lower hull edge. Instead we must somehow compute those set points that describe the new upper hull edge and the new lower hull edge. To help us proceed, we summarize some relevant statements:

- With our recursive partitioning of A we are dealing with disjoint sets L and R at each stage of recursion.

- \( \text{Hull}(A_L \cup A_R) = \text{Hull}(A_L) \cup \text{Hull}(A_R) \cup \text{two additional edges are added at each mergestep}. \)

We also note Theorem 6.16 from Shamos[SHAMOS(11d)]:

Given the Voronoi diagram on N points in the plane, their convex hull can be found in linear time.

One other concept, related to the convex hull, which we will use to help construct the algorithm to find the new hull edges is the following Lemma:

Lemma 3 The first point of a given point set in $\mathbb{R}^2$ we encounter when approaching this set from an infinite distance will be a point contained on the convex hull boundary.

Obtain Hull Points

As Theorem 6.16 from Shamos states, we should be able to obtain the hull points of a set whose Voronoi diagram we have already calculated, in $O(N)$ time. Looking at figure 2.1 or 3.4 we notice that all infinite rays of a completed Voronoi diagram say $VD(A)$ are bisectors of certain pairs of hull points. We can use the data structure for the Voronoi polygons to discover the extreme points of the hull for set $A$ in the following way. Using Lemma 3 we locate the least $x$ coordinate point in the set of points for which we wish to find the hull. This point will be an extreme point of the set $A$. Now we can search this point’s Voronoi polygon list in a counterclockwise direction until we discover that the next counterclockwise vertex (ccv) is -1 denoting an infinite ray (refer to figure 3.4). Then the accompanying cv field will indicate which other set point was used to construct the ray. Continuing in this manner we can search only around the outside of the Voronoi diagram (of set $A$) linking successive hull points sharing bisectors obtained from the Voronoi polygon information thus finding the convex hull of set $A$ in $O(N)$ time. Thus for figure 3.4 we would obtain extreme points $\{0, 2, 5, 6, 4, 1\}$ in that order. The actually implemented function obtain hull points uses this method to find the extreme points of each subset currently being merged.

In this way we obtain those candidates points from which the hull of the merged sets $A_L$ and $A_R$ will be formed.

Graham Scan

Now we turn to the problem of computing the convex hull from these obtained points.

Since our goal is to stay within the $O(N\log N)$ bound for the Voronoi diagram, we must be sure to not take more than $O(N\log N)$ time to find the convex hull for the merged set. One method that meets this requirement is the Graham Scan [GRAHAM(4)].

Theorem 2 The convex hull of $N$ points in the plane can be found in $O(N\log N)$ time and $O(N)$ space using only arithmetic operations and comparisons.

Notice that for $N$ distinct points ($N > 1$) the found number of extreme points can be as few as two (all points collinear) or three (entire set contained within encompassing triangle) or as great as $N$ (every point on convex hull).

---

2 as discussed in chapter 3 and illustrated in figure 3.7 and 3.8.
Our input will be a set of points say $A$, obtained as the union of the convex hulls of $A_L$ and $A_R$ and our output shall be some subset of these $a_i \in A$ ordered in such a way as to represent a polygon in standard order [SHAMOS(11a)] whose vertices represent the set of extreme points characterizing the hull of $A$.

The Graham scan rearranges the points of the input list so that the convex hull appears in the first $M$ positions of this list in the desired order.

Convex hull algorithms involve trigonometric operations whereas sorting algorithms use key comparisons, but the similarity of sorting and hull finding as well as the $O(N \log N)$ shared lower bound seems to demonstrate more than a chance connection between the two. \(^3\) In fact, convex hulls can be used to perform sorting. [SHAMOS(11e)].

The Graham Scan was created by R.L. Graham in 1972 and performs by separating those points on the hull (in order) from those that are not.

We first locate a point that we know is an element of the hull boundary using Lemma 3 to choose say $a_0$ (the least $x$). Then we sort the remaining points (those obtained from each subset $a_L$ and $a_R$) by computing an angular value theta for each $a_i$ pair $(1 \leq i \leq N - 1)$ as depicted in figure 4.1.

There is a notable mistake in Michael Shamos' PhD. Thesis [SHAMOS(11e)] wherein he suggests using an interior (non boundary) point to sort the candidate points for the convex hull. The illustration that accompanies this discussion uses what appears to be the centroid of the considered example set as an origin from which to determine an angular value theta for each potential hull point. Analysis reveals this may cause these hull points to be in the wrong order for the subsequent Graham Scan.

We use fewer calculations than for standard trigonometric angle by calculating a pseudo theta. The function for this theta [SEDGEWICK(10a)] is both monotonic and continuous (mod 2 $\pi$) just as the standard angles are and can be thought of in conceptually the same way as standard trigonometric angles but this theta computation takes less processor operations.

Now we are ready to scan the points in order and output the hull points in the first $M$ positions of the hull array.

We distinguish which points are accepted and which are not by examining successive subsets of four points (if less than four return all points in angular order).

We now discuss figure 4.1. We note that we have a sorted list of $a_i$ given as $A = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$ where theta was used as the sort key. Notice that $a_0$ will always be placed first.

We use set notation to describe the Graham scan algorithm.

We start with our first subset $= \{a_0, a_1, a_2, a_3\}$. Next we assign point $i$ to set element $i-1$ ($pt1 \leftarrow a_0$ etc.)

\(^3\) $O(N \log N)$ time is both necessary and sufficient however for certain distributions in the $E_2$ plane. Jarvis's March algorithm will run in $O(HN) = O(N)$ with $H = $ to the number of hull points and $H$ much less than $N$ for large sets.
Figure 4.1: Graham scan data
The test for eliminating a point from the set of possible extreme points is as follows: When we examine point 4, we can eliminate point 3 from the hull if point 3 and point 2 lie on different sides of the line formed between point 4 and point 1. If point 1 and point 4 are on the same side of the line formed by points 2 and 3, then point 3 is still a candidate for the hull set and is not eliminated. If point 3 is not eliminated we next consider a new subset of set points created by subtracting the first indexed point in the set and by adding the next sequential set point from the sorted list \( a_0 \rightarrow a_N \). If point 3 is eliminated then we subtract the corresponding \( a_i \) from the set and add the next sequential set point from the sorted list and then repeat the process. This is continued until all set points have been considered. In the figure 3.9 we would proceed as follows:

- Start with \( \{a_0, a_1, a_2, a_3\} \) (point 3 not eliminated)
- \( \{a_1, a_2, a_3, a_4\} \) (\(-a_0 + a_4, a_3\) eliminated)
- \( \{a_1, a_2, a_4, a_5\} \) (\(-a_3 + a_5, \) point 3 not eliminated)
- \( \{a_2, a_4, a_5, a_6\} \) (\(-a_1 + a_6, a_5\) eliminated)
- \( \{a_2, a_4, a_6, a_0\} \) (\(-a_6 + a_0, \) point 3 not eliminated)

For further discussion see \([\text{SEDGEWICK}(10b)]\).

**Locate New Edges**

Once the Graham scan has correctly calculated the new extreme points of the hull it is a simple matter to scan the new hull points in order until we locate the two pairs of vertices, comprising the upper and lower new hull edges. We compare vertices in the new hull to the left and right subsets \( A_L \) and \( A_R \). Thus we can partition the new hull points into three sets: the old left, the old right and the two new pairs of vertices.

The problem of identifying which pair of vertices is the upper hull edge and which the lower hull edge is slightly more complicated but analysis reveals clear cases that can be solved using already created functions of the package.

We proceed to locate the new upper and lower hull edges as follows: There are two distinct cases that occur although they have similar solutions. One case is that of three points as given in figure 4.2.

We see that the new edges share some vertex in common:

However in each of these three illustrative cases we can identify the leftmost point of the current set of pairs. Next we construct an x-y coordinate axis using the common point as the origin. This axis is rotated \( \pi \) radians from the standard orientation in the clockwise direction. We then find the angular rotation of the remaining two points relative to this origin. The least angular point is part of the lower hull edge while the greatest angle signifies the upper hull edge.
Usually locating the new edges will produce four new edge vertices for the upper and lower hull edges, although the three point case can occur in very large points sets as in the case of the encompassing large triangle (two point hulls are avoided by not allowing more than two initial set points to be collinear).

Let us examine the four point cases more closely. An example is illustrated in figure 4.3.

Once again we determine the four vertices determining the new hull edges. Again we locate the leftmost newpoint and proceed similarly as done in the case of three points. Now the cases differ slightly from the three point cases but once again rotational arguments correctly identify the new upper hull edge and the new lower hull edge.

**Polygon Clipping**

As explained in chapter 3, page 18, each polygon lying along the right side of the left subset and the left side of the right subset being merged must be clipped against the merge curve. Clipping is also performed in the world coordinate space to window to viewport mapping as well as being specifically needed by a future experiment (discussed in Chapter 8).

Therefore the implementation uses functions which are generalizable to provide flexibility and adaptability to various clipping applications. Our discussion centers around the clipping of some \( \text{seg}(v_i, v_j) \) and some particular polygon represented as a collection of vertices arranged in some standard order (see [SHAMOS(11a)]) usually successively counterclockwise say \( P = \{a_0, a_1, ..., a_{N-1}\} \) where \( N = \) number of vertices.
The basic function needed is one that can search the polygon edges (window) in the desired order (clockwise or counterclockwise) and check the vertices of $seg(v_i, v_j)$ to see how they relate to the edge being considered.

Having implemented homogeneous coordinates we can use the determinants of $3 \times 3$ matrices to decide whether a given polygon edge $a_i, a_j$ intersects $seg(v_i, v_j)$ the segment being clipped.

Looking at figure 4.4 we see that $seg(v_i, v_j)$ intersecting polygon A in edge $a_m, a_n$.

We compute

- $S_1 = det_3(v_i, a_m, a_n)$
- $S_2 = det_3(v_j, a_m, a_n)$
- $S_3 = det_3(a_m, v_i, v_j)$
- $S_4 = det_3(a_n, v_i, v_j)$

Now $seg(v_i, v_j)$ intersects $seg(a_m, a_n)$ if $f (S_1 * S_2 <= 0)$ and $(S_3 * S_4 <= 0)$ as is the case in figure 4.4. For an in depth discussion of clipping techniques using homogeneous coordinates see [PAVLIDIS(9)].

This determinant method to discover if two line segments (or seg, ray or ray, ray) intersect is adapted for use in the thesis software function poly-seg. We know that the next segment or ray of the merge curve will intersect an edge in VOR (current edge left) or in VOR (current edge right). We can use the determinant methods as just described above to find those candidates for the next merge curve intersection.
Figure 4.4: Line segment - polygon intersection
CHAPTER 5

EXPERIMENT USING COMPUTATIONAL GEOMETRY PACKAGE

Table 5.1 represents the data gathered by experimental running of the thesis software on various points sets to construct the Voronoi diagram with the $O(N \log N)$ algorithm.

The experiment on set of $N$ points was performed on a Vax-11 model 750 with software written in C programming language. We used the Unix system function `random()` to generate point sets for size $N$ ranging from fifty points to three hundred points. Timings in seconds were obtained using the Unix function `time(0)` to measure the processing time between the call to the function Voronoi diagram and completion of the internal computations of this Voronoi structure. All i/o was suppressed and timings were averaged over 100 runs to eliminate system variations.

As seen in the table the timings support that the algorithm executes in $\frac{1}{35} N \log N$. 
<table>
<thead>
<tr>
<th>N points</th>
<th>Timings in seconds</th>
<th>approximate $N \log N$</th>
<th>$\frac{1}{35} N \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>71</td>
<td>2468</td>
<td>70.5</td>
</tr>
<tr>
<td>150</td>
<td>32</td>
<td>1084</td>
<td>31.0</td>
</tr>
<tr>
<td>150</td>
<td>29</td>
<td>1084</td>
<td>31.0</td>
</tr>
<tr>
<td>100</td>
<td>16</td>
<td>664</td>
<td>19.0</td>
</tr>
<tr>
<td>75</td>
<td>14</td>
<td>467</td>
<td>13.3</td>
</tr>
<tr>
<td>75</td>
<td>14</td>
<td>467</td>
<td>13.3</td>
</tr>
<tr>
<td>50</td>
<td>8</td>
<td>282</td>
<td>8.1</td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>282</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Figure 5.1: Table 5.1
The experiment will be performed to demonstrate the basic computational package as applied to a simple experiment on a given point set \( A = \{a_0, a_1, ..., a_{N-1}\} \) with \(|A| = N\).

We first calculate the Voronoi diagram \( \text{VD}(A) \) of the point set \( A \). Additionally we have surrounded our point set with a window prescribed simply as \( \text{xmin}, \text{xmax} \) and \( \text{ymin}, \text{ymax} \). The Voronoi diagram is clipped against this window boundary thus enclosing the polygons of infinite area. All points of \( A \) are contained within this window boundary.

The program performs the following loop until the experiments convergence criterion (see 4 below) is satisfied or until some given number of loops have been executed.

1. compute the Voronoi diagram for the given point set \( A \). This is initially provided by user using the file functions.
2. redefine all polygons whose initial set points are the vertices forming \( \text{BCH}(A) \).
   These infinite region polygons are clipped against the experimental window boundary (\( \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax} \)).
3. \( \forall \ VOR(a_i) \ 0 <= i <= N - 1 \) compute the centroid of that polygon using \( VOR(a_i)'s \) polygon vertices. Denote this centroid for \( a_i \) as \( a'_i \).
4. If \( (|a_i - a'_i| < \epsilon) \ \forall \ i, 0 <= i <= N - 1 \) then stop; else if counter > number of desired iterations then stop; else set \( A = (a'_0, a'_1, ..., a'_{N-1}) \) and go to 1.

The main point in implementing this experiment is to study point packing problems in the plane. We hope to show that given a generic point set \( A \), the iterates of \( A \), under the procedure outlined in 1 - 4 above will converge to a maximal packing of \( N \) points inside the original window. The convex hull was needed to determine those set points needing clipping against the window boundary.
CHAPTER 7

SIMPLE GRAPHICS PACKAGE

The Simple Graphics Package (SGP - see introduction) consists of a small but functionally complete set of application-independent programs for creating arbitrary views of two-dimensional objects and for supporting interaction between the applications program and its user. This facilitates the generalization of the package to multiple applications as well as display device independence of the graphics display routines. This display device independence allows us to replace only the actual device code generator which converts the high level functions of the SGP primitives to actual display device code (analogous to translation of high level code to machine code).

A drawback to this methodology is that some device speed and efficiency is wasted but this is less important in our package than the user's time and program portability.

The actual functional facilities of SGP available to the applications programmer can be divided into six classes: Graphics output primitives, Attribute setting, Segment Control, Viewing operation, Input and Control (see [FOLEY(3a)] for details).

These various functions (see Appendix) serve as functional building blocks that can be joined together to create the display code necessary to convert the applications model into a two-dimensional graphics display with clipping. In the thesis software such functional collections include functions display points, display hull and display Voronoi diagram.
BIBLIOGRAPHY


[SHAMOS(11a)] Shamos Michael I., Computational Geometry, Yale University PhD., 1978, pp. 15-16.

[SHAMOS(11b)] Shamos Michael I., Computational Geometry, Yale University PhD., 1978, Chapter 3.

[SHAMOS(11c)] Shamos Michael I., Computational Geometry, Yale University PhD., 1978, pp. 135, 136, 179.


[SHAMOS(11e)] Shamos Michael I., Computational Geometry, Yale University PhD., 1978, pp. 40-42.


APPENDIX
This appendix contains a listing of the Simple Graphics Package functions encoded within the thesis software.

FUNCTION: xtransinv(x), x integer, xtransinv returns a value of type float. This function converts screen coordinate x to world coordinate x. It uses values assigned in setworld ans calctrans.

FUNCTION: ytransinv(y), y integer, ytransinv returns a value of type float. This function converts screen coordinate y to world coordinate y. It uses values assigned in setworld ans calctrans.

FUNCTION: xtrans, x float, xtrans returns an integer. This function converts the world coordinate x to a screen coordinate x. It uses the values assigned in set-world, set-viewport and calctrans to perform the transformation.

FUNCTION: ytrans, y float, ytrans returns an integer. This function converts the world coordinate y to a screen coordinate y. It uses the values assigned in set-world, set-viewport and calctrans to perform the transformation.

FUNCTION: calctrans(). This function calculates the parameters of the world-screen and screen world transformations. It is called whenever a window or a viewport is changed.

FUNCTION: clip1(x,y) x and y are float, clip1(x,y) is a boolean of type integer. This function checks to see if the world coordinate pair (x,y) lies within the limits of the defined world(it then returns as TRUE).

FUNCTION: get-pos(xpos,ypos), xpos and ypos are integer. This function turns on the terminal's locator device and returns the screen coordinates of the locator for its current positioning.

FUNCTION: set-visibility(name,visibility), name and visibility are integer. This procedure allows the user to turn off or on the visibility flag for a defined segment. Words ON and OFF have been defined for use as function parameters.

FUNCTION: set-segment-highlighting(name,highlight) , name and highlight are of type integer. This function allows the user to highlight a particular segment. Use predefined parameters ON and OFF. ON produces highlighting as a flashing segment.
FUNCTION: set-segment-detectability(name,detectability), name and detectability are of type integer. This function determines whether a segment may be picked using a locator. Use predefined parameters ON and OFF.

FUNCTION: clear-screen(). This function clears the screen. All defined segments are cleared.

FUNCTION: close-segment(). This function closes or ends the definition of the current segment as started by a previous open-segment() command.

FUNCTION: delete-segment(name), name is integer. This function erases the primitives of the segment name. The segment is erased from the screen. ALL is defined as -1 so that delete-segment(ALL), deletes all currently defined segments.

FUNCTION: rename-segment(old-name,new-name), old-name and new-name are integer. This procedure allows the user to redefine the integer name of an already named segment.

FUNCTION: set-background-color(color), where color is of type integer. This function allows the user to set the color of the background on the screen. Those defined colors (see set-line-color) can be used for the color parameter.

FUNCTION: set-line-style(style), where style is an integer name. This function allows the user to set the line style for the line-abs and line-rel commands.

FUNCTION: set-marker-style(style), where style is an integer name. This function allows the user to set the marker style for the draw-marker command. Markers range from 0 to 9.

FUNCTION: set-line-color(color), where color is an integer name. This function allows the user to set the color for the marker and line commands. Valid colors are ERASER, WHITE, RED, GREEN, BLUE, CYAN, MAGNETA and YELLOW.

FUNCTION: set-text-color(color), where color is an integer name. This function allows the user to set the color for the text commands. Valid colors are ERASER, WHITE, RED, GREEN, BLUE, CYAN, MAGNETA and

FUNCTION: begin-panel(). This function is used in conjunction with an end-panel() command. begin-panel begin the definition of a polygon at
the current pen position. Succeeding calls to line commands define the polygon edges. Finally end-panel is called to close the polygon. end-panel draws a line from the current pen position to the pen position where the polygon definition began. It then fills the polygon with the parameters defined in set-fill-pattern.

FUNCTION: end-panel(). This function closes a polygon previously started with begin-panel(). It draws a line from the current pen position to the pen position where the polygon definition began. It then fills the polygon with the parameters defined in set-fill-pattern.

FUNCTION: set-fill-pattern(pattern), where pattern is an integer. This function allows the user to set the fill pattern (i.e., solid color) with which a polygon defined by begin-panel & end-panel is filled. Solid colors are obtained by preceding the color name with a dash (-) as for example (-RED).

FUNCTION: set-text-size(width,height,spacing), where width, height and space are integer. This function allows the user to set the size of the text drawn by the text command. The parameter units are measured in device pixels.

FUNCTION: open-segment(name), name is integer name. This function allows the user to begin definition of a graphics segment. Names must be in the range 0..32000.

FUNCTION: text (line), where line is an array of character (string). This function draws the text line beginning at the current pen position. The lower left hand corner of the first character of array line will be at this pen position. Size and spacing may be set with set-text-size and color set with set-text-color. Note that upon completion of function the current pen position remains unchanged as before function call.

FUNCTION: draw-marker(x,y), where x & y are float. This procedure draws a marker at the world coordinate (x,y). The current pen position is set to (x,y). The marker is clipped with clip1. See set-marker-style & set-line-style. Note markers cannot be enlarged or shrunk by window, viewport changes.

FUNCTION: move-rel(dx,dy), dx and dy are float. This procedure moves the current pen position to the new pen position cpx + dx , cpy + dy. It then calls move-abs(cpx+dx,cpy+dy) to reset the current pen position.

FUNCTION: line-rel(dx,dy), dx and dy are float. This procedure draws a line from the current pen position to the new pen position cpx + dx , cpy + dy. Clipping against window is performed.
FUNCTION: line-abs(x,y), x and y are float. This procedure draws a line from the current pen position to the new pen position x,y. Clipping against window is performed.

FUNCTION: terminate(). This procedure deletes all segments and clears the screen.

FUNCTION: init-graphics(). This procedure initializes the SGP. It must be called before any graphics commands are performed. It sets the current pen position to (0,0). Line and text colors are set to WHITE. Text size is set at a standard size, and line style is set to solid. The marker style is set to a diamond with an x inside. It also defines the world to map onto the entire screen, deletes all segments and clears the screen.

FUNCTION: move-abs(x,y), x & y are float. This function moves the current pen position to the world coordinates (x,y). It also calls plot to position the pen.

FUNCTION: segment-exists(name), name of type integer. This function returns a boolean value (TRUE or FALSE) depending on whether the queried name already exists as a segment.

FUNCTION: set-window(xmin,ymin,xmax,ymax), xmin, ymin, xmax, ymax are float. Allows user to define world coordinates. Should not be called if a segment is currently open.

FUNCTION: set-viewport(xmin,ymin,xmax,ymax), xmin, ymin, xmax, ymax are float. Allows user to define screen viewport.